© ISO/IEC 2012 – All rights reserved

WG 14 N1615

ISO/IEC JTC 1/SC 22 0000

Date: yyyy-mm-dd

Reference number of document: **ISO/IEC nnn-n**

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

Secretariat: ANSI

Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 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: Binary 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 rights of which they are aware and to provide supporting documentation.

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

Copyright notice

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO's member body in the country of the requester:

ISO copyright office
Case postale 56 CH-1211 Geneva 20
Tel. +41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

 © ISO/IEC 2012 – All rights reserved

Contents

Page

© ISO/IEC 2012 – All rights reserved

Foreword

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

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

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

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

ISO nnn-n was prepared by Technical Committee ISO JTC 1, *Information Technology*, Subcommittee SC 22, *Programming languages, their environments, and system software interfaces*.

ISO nnn consists of the following parts, under the general title *Floating-point extensions for C*:

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplemental functions*

— *Part 5: Supplemental attributes*

Part 1 updates ISO/IEC 9899 11 (*Information technology — Programming languages, their environments and system software interfaces — Programming Language C*), Annex F in particular, to support all required features of ISO/IEC/IEEE 60559:2011 (*Information technology — Microprocessor Systems — Floating-point arithmetic*).

Part 2 supersedes ISO/IEC TR 24732:2008 (Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic).

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

0 Introduction

0.1 Background

0.1.1 IEC 60559 floating-point standard

The IEC 60559:1989 international standard was equivalent to the IEEE 754-1985 standard for binary floating-point arithmetic. Its stated goals were:

 © ISO/IEC 2012 – All rights reserved

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 binary floating-point formats, including representations for Not-a-Number (NaN) and signed infinities and zeros, and basic arithmetic operations that compose a well-defined, closed arithmetic system. It also specified status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact), controls for choosing different rounding methods, conversions between floating-point formats and decimal character sequences, and a few auxiliary operations.

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

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

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

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

© ISO/IEC 2012 – All rights reserved

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 includes 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.

0.1.2 C support for IEC 60559

IEC 9899:1999 (C99), in conditionally normative Annex F, introduced nearly complete support for the IEC 60559:1989 standard for binary floating-point arithmetic. Also, C99's informative Annex G offered a specification of complex arithmetic that is compatible with IEC 60559:1989.

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

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

0.2 Purpose

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

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

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

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

 © ISO/IEC 2012 – All rights reserved

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

1 Scope

Part 1 of Technical Specification 00000 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.

TS 00000-1 does not cover decimal floating-point arithmetic, nor most other optional features of IEC 60559.

TS 00000-1 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 of this Technical Specification if

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

b) It defines **__STDC_IEC_60559_BFP__** to **201**ymm**L**.

Suggested change to C11:

Replace the third sentence of 4#6:

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

with:

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

The library functions, macros, and types defined in Part 1 of this Technical Specification are defined by their respective headers if the macro **__STDC_WANT_IEC_60559_BFP__** is defined at the point in the source file where the appropriate header is first included.

© ISO/IEC 2012 – All rights reserved

3 Normative references

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

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

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

4 Terms and definitions

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

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

5 Predefined macros

The following suggested change to C11 replaces `__STDC_IEC_559__`, the conformance macro for Annex F, with `__STDC_IEC_60559_BFP__`, for consistency with other conformance macros and to distinguish its application to binary floating-point arithmetic. Note that an implementation may continue to define `__STDC_IEC_559__`, so that current strictly conforming programs that use `__STDC_IEC_559__` will remain valid under the suggested changes in Part 1 of this Technical Specification.

Suggested change to C11:

In 6.10.8.3#1, replace:

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

with:

> `__STDC_IEC_60559_BFP__` The integer constant `201ymmL`, intended to indicate conformance to Annex F (IEC 60559 binary floating-point arithmetic).

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

Suggested changes to C11:

In 6.10.8.3#1, after the new `__STDC_IEC_60559_BFP__` item, insert the item:

> `__STDC_IEC_60559_COMPLEX__` The integer constant `201ymmL`, 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.

© ISO/IEC 2012 – All rights reserved

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 suggested change to C11 below updates the introduction to Annex F to acknowledge the revision to IEC 60559.

Suggested change 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 **`201`**`ymm`**`L`** 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.

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

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

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 suggested changes to C11 below reflect the new terminology in IEC 60559; these changes are not substantive.

© ISO/IEC 2012 – All rights reserved

Suggested changes to C11:

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

— 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 **`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.

7.2 Canonical encodings

IEC 60559 refers to preferred encodings in a format as *canonical*. Some formats also contain redundant or ill-specified encodings, which are *non-canonical*. All encodings in IEC 60559 binary interchange formats are canonical; however, its extended formats may have non-canonical encodings. (IEC 60559 decimal interchange formats, covered in Part 2 of this Technical Specification, contain non-canonical redundant encodings.)

Suggested change to C11:

After 5.2.4.2.2#5, add:

[5a] The preferred encodings in a floating type are called *canonical*. A floating type may also contain *non-canonical* encodings, for example, redundant encodings of some or all of its values, or encodings

 © ISO/IEC 2012 – All rights reserved

that are extraneous to the floating-point model. Typically, floating-point operations deliver results with canonical encodings.

8 Operation binding

IEC 60559 includes several new required operations. Table 1 in the suggested 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.

Suggested 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

© ISO/IEC 2012 – All rights reserved

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.

Table 1 — Operation binding

IEC 60559 operation	C operation	Clauses - C11	Clauses - this document
roundToIntegralTiesToEven	**roundeven**		14.1.1

 © ISO/IEC 2012 – All rights reserved

roundToIntegralTiesAway	**round**	7.12.9.6, F.10.6.6	14.1.1
roundToIntegralTowardZero	**trunc**	7.12.9.8, F.10.6.8	14.1.1
roundToIntegralTowardPositive	**ceil**	7.12.9.1, F.10.6.1	14.1.1
roundToIntegralTowardNegative	**floor**	7.12.9.2, F.10.6.2	14.1.1
roundToIntegralExact	**rint**	7.12.9.4, F.10.6.4	
nextUp	**nextup**		14.4
nextDown	**nextdown**		14.4
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**		14.3
maxNumMag	**fmaxmag**		14.3
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	14.2
addition	**+**	6.5.6	
formatOf addition with narrower format	**fadd**, **faddl**, **daddl**		14.5
subtraction	**-**	6.5.6	
formatOf subtraction with narrower format	**fsub**, **fsubl**, **dsubl**		14.5
multiplication	*****	6.5.5	
formatOf multiplication with narrower format	**fmul**, **fmull**, **dmull**		14.5
division	**/**	6.5.5	
formatOf division with narrower format	**fdiv**, **fdivl**, **ddivl**		14.5
squareRoot	**sqrt**	7.12.7.5, F.10.4.5	
formatOf squareRoot with narrower format	**fsqrt**, **fsqrtl**, **dsqrtl**		14.5
fusedMultiplyAdd	**fma**	7.12.13.1, F.10.10.1	
formatOf fusedMultiplyAdd with narrower format	**ffma**, **ffmal**, **dfmal**		14.5
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4	
convertToIntegerTiesToEven	**fromfp**, **ufromfp**		14.1.2
convertToIntegerTowardZero	**fromfp**, **ufromfp**		14.1.2
convertToIntegerTowardPositive	**fromfp**, **ufromfp**		14.1.2
convertToIntegerTowardNegative	**fromfp**, **ufromfp**		14.1.2
convertToIntegerTiesToAway	**fromfp**, **ufromfp**, **lround**, **llround**	7.12.9.7, F.10.6.7	14.1.2
convertToIntegerExactTiesToEven	**fromfpx**, **ufromfpx**		14.1.2
convertToIntegerExactTowardZero	**fromfpx**, **ufromfpx**		14.1.2
convertToIntegerExactTowardPositive	**fromfpx**, **ufromfpx**		14.1.2

© ISO/IEC 2012 – All rights reserved

convertToIntegerExactTowardNegative	**fromfpx**, **ufromfpx**		14.1.2
convertToIntegerExactTiesToAway	**fromfpx**, **ufromfpx**		14.1.2
convertFormat - different floating types	cast and implicit conversions	6.3.1.5, 6.5.4	
convertFormat - same floating type	**canonicalize**		14.9
convertFromDecimalCharacter	**strtod**, **wcstod**, **scanf**, decimal floating constants	7.21.6.2, 7.22.1.3, 7.29.4.1.1, F.5	10.1
convertToDecimalCharacter	**printf**, **strfromflt**	7.21.6.1, F.5	10
convertFromHexCharacter	**strtod**, **wcstod**, **scanf**, hexadecimal floating constants	7.21.6.2, 7.22.1.3, 7.29.4.1.1	
convertToHexCharacter	**printf**, **strfromflt**	7.21.6.1	10.2
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**		14.6
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)**		14.6
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	14.7
isSignMinus	**signbit**	7.12.3.6	
isNormal	**isnormal**	7.12.3.5	
isFinite	**isfinite**	7.12.3.2	
isZero	**x == 0.0**	6.5.9	
isSubnormal	**issubnormal**		14.7
isInfinite	**isinf**	7.12.3.3	

© ISO/IEC 2012 – All rights reserved

isNaN	**isnan**	7.12.3.4	
isSignaling	**issignaling**		14.7
isCanonical	**iscanonical**		14.7
radix	**FLT_RADIX**	5.2.4.2.2	
totalOrder	**totalorder**		14.8
totalOrderMag	**totalordermag**		14.8
lowerFlags	**feclearexcept**	7.6.2.1	
raiseFlags	**fesetexcept**		15.1
testFlags	**fetestexcept**	7.6.2.5	
testSavedFlags	**fetestexceptflag**		15.2
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**		15.3
restoreModes	**fesetmode**		15.3
defaultModes	**fesetmode(FE_DFL_MODE)**		15.3

[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** (see 12 below).

[4] 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.

[5] 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).

[6] The C **getpayload**, **setpayload**, and **setpayloadsig** (see 14.10 below) functions provide program access to NaN payloads, defined in IEC 60559.

[7] 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.

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.

Suggested 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

© ISO/IEC 2012 – All rights reserved

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

$H \geq M + 3,$

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

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 suggested changes to C11 satisfy these requirements.

Suggested change to C11:

Rename F.5 from:

F.5 Binary-decimal conversion

to:

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

Insert after F.5#2:

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

CR_DECIMAL_DIG

which expands to an integral constant expression suitable for use in **#if** preprocessing directives whose value is a number such that conversions between all supported IEC 60559 binary types 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 IEC 60559 binary-floating types 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 IEC 60559 binary floating types 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

© ISO/IEC 2012 – All rights reserved

destination type. The "inexact" floating-point exception is raised (once) if either conversion is inexact[1]. (The second conversion may raise the "overflow" or "underflow" floating-point exception.)

10.2 Conversions to character sequences

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

Suggested changes to C11:

After 7.22.1.2, add:

7.22.1.3 The `strfromflt` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <stdlib.h>
int strfromflt(char * restrict s, size_t n, const char * restrict format,
    …);
```

Descripton

[2] The **strfromflt** function is equivalent to **snprintf** (7.21.6.5) if the **format** string contains only an optional precision, an optional **L** length modifier, and one **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** format specifier. Use of **strfromflt** with any other format string results in undefined behavior.[2]

Returns

[3] The **strfromflt** functon returns the value returned by **snprintf** (7.21.6.5), for valid input.

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 suggested 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.

Suggested 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:

---

[1] The intermediate conversion is exact only if all input digits after the first **CR_DECIMAL_DIG** digits are **0**.

[2] The **strfromflt** function may be equivalent to **snprintf** for all inputs.

© ISO/IEC 2012 – All rights reserved

[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

```
FP_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.

 © ISO/IEC 2012 – All rights reserved

with:

[9] The macro

```
FP_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, sets floating-point control modes, or establishes non-default 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

[1] The pragma defined in 7.6.1a is available to the program if the macro **__STDC_WANT_IEC_60559_BFP__** is defined at the point in the source file where the **<fenv.h>** header is first included.

Synopsis

[2]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <fenv.h>
#pragma STDC FENV_ROUND direction
```

Description

[3] The **FENV_ROUND** pragma provides a means to specify a constant rounding direction for binary 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

© ISO/IEC 2012 – All rights reserved

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

[4] ***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`**, **`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.

[5] Within the scope of an **`FENV_ROUND`** directive establishing a mode other than **`FE_DYNAMIC`**, all floating-point operators and invocations of functions indicated in Table 2 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 Table 2 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.

Table 2 — 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, 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>`	`strfromflt, strtod, strtof, strtold`
`<wchar.h>`	`wcstod, wcstof, wcstold`
`<stdio.h>`	`printf` and `scanf` families

[6] 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 ...
```

 © ISO/IEC 2012 – All rights reserved

```
        // 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 7.6.4.1 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

© ISO/IEC 2012 – All rights reserved

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:

 © ISO/IEC 2012 – All rights reserved

[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 suggested changes is to allow implementations to conditionally support signaling NaNs as specified in IEC 60559, but to require only minimal support for signaling NaNs.

Suggested 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 into 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 unordered macro raises no floating-point exceptions if neither argument is a signaling NaN."

Change F.2.1 from:

© ISO/IEC 2012 – All rights reserved

F.2.1 Infinities, signed zeros, and NaNs

[1] This specification does not define the behavior of signaling NaNs.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).

[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 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.

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

[4] The **fprintf** family of functions in **<stdio.h>** should behave as if floating-point operands were passed through the **canonicalize** function of the same type.[3]

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:

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

 © ISO/IEC 2012 – All rights reserved

> 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.

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 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 suggested changes

© ISO/IEC 2012 – All rights reserved

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>**.

Suggested changes to C11:

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
— width of type **intmax_t**
  INTMAX_WIDTH 64
— width of type **uintmax_t**
  UINTMAX_WIDTH 64

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*

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>**.

 © ISO/IEC 2012 – All rights reserved

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 suggested changes add a function that rounds to nearest and remove the latitude to raise the "inexact" floating-point exception.

Suggested 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

© ISO/IEC 2012 – All rights reserved

```
#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__
#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 whose least significant bit 0), regardless of the current rounding direction.

 © ISO/IEC 2012 – All rights reserved

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 suggested changes below satisfy this requirement with four new functions that take two extra arguments to represent the rounding direction and the rounding precision.

Suggested changes to C11:

After 7.12#6, add:

[7.12.6a] 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. These macros are for use with the **fromfp**, **ufromfp**, **fromfpx**, and **ufromfpx** functions.

After 7.12.9.8, add:

7.12.9.9 The fromfp and ufromfp functions

Synopsis

© ISO/IEC 2012 – All rights reserved

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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 type **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 **width** argument is 0, the functions return 0. 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 integers of the specified width, 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__
#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

© ISO/IEC 2012 – All rights reserved

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

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 integers 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 integers 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 suggested 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.

Suggested 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 either **`LONG_MIN`** or **`-LONG_MAX`.** The value of **`FP_LLOGBNAN`** shall be either **`LONG_MAX`** or **`LONG_MIN`.**

After 7.12.6.6, add:

7.12.6.6a The `llogb` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
```

Description

© ISO/IEC 2012 – All rights reserved

[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.n).

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.

Suggested changes to C11:

After 7.12.12.3, add:

7.12.12.4 The fmaxmag functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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 numeric value of their argument whose magnitude is the maximum of the magnitudes of the arguments.[4]

Returns

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

7.12.12.5 The fminmag functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
double fminmag(double x, double y);
```

[4] 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.

 © ISO/IEC 2012 – All rights reserved

```
float fminmagf(float x, float y);
long double fminmagl(long double x, long double y);
```

Description

[2] The **fminmag** functions determine the numeric value of their argument whose magnitude is the minimum of the magnitudes of the arguments.[5]

Returns

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

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[6]

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

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 suggested changes to C11 add functions for the new operations and retain the **nextafter** and **nexttoward** functions already in C11.

Suggested changes to C11:

After 7.12.11.4 add:

7.12.11.5 The nextup functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);
```

Description

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

[6] This implementation does not handle signaling NaNs as required of implementations that define **FP_SNANS_ALWAYS_SIGNAL**.

© ISO/IEC 2012 – All rights reserved

[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__
#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.

F.10.8.6 The nextdown functions

[1]
— **nextdown(+∞)** returns the largest-magnitude positive finite number in the type of the function.
— **nextdown(-∞)** returns -∞.

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 suggested changes to C11 add functions for these operations that round to formats narrower than the operand formats. The operations that round to the same and wider formats are already available by casting operands of the built-in operators (+, -, *, /) to the desired type and by calling the **fma** and **sqrt** functions of the desired type.

Suggested changes to C11:

After 7.12.13, add:

7.12.13a Functions that round result to narrower type

 © ISO/IEC 2012 – All rights reserved

7.12.13a.1 Add and round to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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__
#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__
#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

© ISO/IEC 2012 – All rights reserved

[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__
#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__
#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** 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__
#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.

© ISO/IEC 2012 – All rights reserved

Returns

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

After old F.10.10 add:

F.10.10a Functions that round result to narrower type

[1] The functions that round their result to narrower type (7.12.14) are fully specified in IEC 60559. The returned value is dependent on the current rounding direction mode.

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 suggested changes to C11 provide a function-like macro for compareSignalingEqual. The negation of the macro provides compareSignalingNotEqual. (See Table 1.)

Suggested changes to C11:

After 7.12.14.6, add:

7.12.14.7 The `iseqsig` macro

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
int iseqsig(floating-type x, floating-type y);
```

Description

[2] The **iseqsig** macro determines whether its arguments are equal. A domain error occurs if an argument is a NaN.

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 **iseqsiq** macro produce equivalent results, except that the **iseqsig** macro raises the “invalid” floating-point exception if an argument is a NaN.

14.7 Inquiry macros

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

Suggested change to C11:

After 7.12.3.1, add:

7.12.3.1a The `iscanonical` macro

© ISO/IEC 2012 – All rights reserved

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
int iscanonical(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 7.12.3.6 (The **isnormal** macro), add:

7.12.3.7 The issignaling macro

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <math.h>
int issignaling(real-floating x);
```

Description

[2] The **issignaling** macro determines whether its argument value is a signaling NaN, without raising a floating-point exception.

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__
#include <math.h>
int issubnormal(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.

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*:

© ISO/IEC 2012 – All rights reserved

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* ≥ the exponent of *y*
      ii) otherwise totalOrder(x, *y)* is true if and only if the exponent of *x* ≤ 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 suggested change to C11 provides these operations.

Suggested change to C11:

After F.10.11, add:

F.10.12 The total order functions

This annex specifies the total order functions required IEC 60559.[7]

F.10.12.1 The `totalorder` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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`**. The functions are fully specified in IEC 60559.

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

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

© ISO/IEC 2012 – All rights reserved

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
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**. The functions are fully specified in IEC 60559.

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**.

14.9 The 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 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 suggested change to C11 below provides the IEC 60559 convertFormat operation.

Suggested change 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__
#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**.[8] If the input ***x** is a signalling 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.

After F.10.8.6 (added above), add:

F.10.8.7 The canonicalize functions

[8] Arguments **x** and **cx** may point to the same object.

© ISO/IEC 2012 – All rights reserved

[1] The **canonicalize** functions produce[9] the canonical version of the representation in the object pointed to by the argument **x**. If the imput ***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.

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 suggested changes to C11 provide functions to get and set the NaN payloads defined in IEC 60559.

Suggested change to C11:

After F.10.12 (added above), add:

F.10.13 Payload functions

F.10.13.1 The getpayload functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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. Floating-point exceptions are not raised for signaling NaN arguments. If ***x** is not a NaN, the return result is unspecified.

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__
#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 into the object pointed to by ***res**. If **pl** is not a positive floating-point integer representing a valid payload, ***res** is set to positive zero.

[9] As if **x*1e0** were computed.

© ISO/IEC 2012 – All rights reserved

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__
#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 into the object pointed to by **`*res`**. 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 suggested change to C11 provides the raiseFlags operation.

Suggested change to C11:

After 7.6.2.3, add:

7.6.2.3a The `fesetexcept` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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.[10]

Returns

[10] Enabled traps for floating-point exceptions are not taken.

© ISO/IEC 2012 – All rights reserved

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

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 suggested change to C11 provides the testSavedFlags operation.

Suggested change to C11:

After old 7.6.2.4, add:

7.6.2.4a The `fetestexceptflag` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#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`**. The **`excepts`** argument specifies the floating-point status flags to be queried.

Returns

[3] The **`fetestexcept`** 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 suggested changes to C11 support this operation.

Suggested 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 old 7.6#8, add:

[8a] 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-

© ISO/IEC 2012 – All rights reserved

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.

After 7.6.3 insert:

7.6.3.0 The fegetmode function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP__
#include <fenv.h>
int fegetmode(femode_t *modep);
```

Description

[2] The **fegetmode** function attempts to store all the dynamic floating-point control modes into 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__
#include <fenv.h>
int fesetmode(const fenv_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.

 © ISO/IEC 2012 – All rights reserved

16 Type-generic math <tgmath.h>

The following suggested changes to C11 enhance the specification for type-generic math macros to accommodate functions and the constant rounding mode pragma in Part 1 of this Technical Specification.

Suggested 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**, **setpayloadsig, and canonicalize**, 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 (7.12.14), use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

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**

> [6b] 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).

> [6c] A type-generic macro corresponding to a function indicated in Table 2 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#7, append to the table:

© ISO/IEC 2012 – All rights reserved

`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

© ISO/IEC 2012 – All rights reserved

Bibliography

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

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

[3] ISO/IEC TR 24732:2008, *Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic*

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

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

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

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

© ISO/IEC 2012 – All rights reserved