C9X Floating-Point Proposal

WG14/N546 X3J11/96-010 (Draft 2/26/96)

Jim Thomas
517 Hendon Ct.
Sunnyvale, CA 94087
jthomas@best.com

Overview

This is a detailed proposal to integrate the "Floating-Point C Extensions" chapter of X3J11's *Numerical C Technical Report* (X3/TR-17:199x) into C9X, with modifications according to directions approved by SC22/WG14. The main sections, in addition to this overview are:

C9X Edits
Mathematics (Enhanced) <math.h>—merged with <fp.h> from TR
Floating-Point Environment <fenv.h>
Annex X: IEC 559 Floating-Point Arithmetic

Purpose

The proposal addresses two long-standing needs of programmers: (1) more predictable floating-point arithmetic (for all implementations), and (2) a standard language binding for features of IEC 559.

Vagaries of floating-point arithmetic have plagued programmers and users since its inception. And they still do, even though hardware floating-point is now largely standardized. When IEEE binary floating-point standard 754—later to become IEC 559—became an official standard in July 1985, 26 months before the radix-independent standard 854, several IEEE implementations were already shipping. Now virtually all new floating-point implementations conform to these standards—at least in format, if not to the last detail. Although these standards have been enormously successful in influencing hardware implementation, many of their features, including predictability, remain impractical or unavailable for use by programmers. IEC 559 does not include language bindings—a cost of delivering the basic standard in a timely fashion. The ANSI C committee attempted to remove conflicts with IEC 559, but did not specify IEC 559 support. Expediencies of programming language implementation and optimization can deny the features offered by modern hardware. In the meantime, particular companies have defined their own IEC 559 language extensions and libraries; not surprisingly, lack of portability has impeded programming for these interfaces.

History

The Numerical C Extensions Group, NCEG, at its initial meeting in May 1989, identified support for IEC 559 floating-point arithmetic as one of its focus areas and organized a subgroup to produce a technical report. The ensuing effort benefited from the considerable C language and IEC 559 floating-point expertise associated with NCEG

and X3J11. It included individuals with substantial experience with language extensions (albeit proprietary) for IEC 559. And, following after the standardization of C, it had a stable, well defined base for its extensions. Thus there has been a unique opportunity to solve this problem.

NCEG/X3J11.1 and X3J11 mailings have included thirteen drafts of FPCE and two drafts of the four papers constituting the specific C9X proposal. NCEG's FP/IEEE subgroup, NCEG, X3J11, and numerous other interested parties have reviewed the drafts to varying degrees. Proprietary extensions for IEEE support have provided prior art for many features. Both developmental and commercial compilers and libraries have implemented the specification.

Major milestones include

Date	Milestone
May 92	FPCE draft distributed for preliminary review to various professional organizations, including X3J11, WG14, X3J16/WG21, X3T2, and X/Open
Jan 93	FPCE draft distributed for public comment
Jun 93	FPCE draft forwarded by NCEG/X3J11.1 to X3J11 as its FP/IEEE technical report
Jun 94	FPCE draft accepted by X3J11 as part of its technical report on numerical extensions
Mar 95	Final version of FPCE, for X3J11 technical report
Aug 95	FPCE recast for C9X integration (four papers)

Substantive changes from FPCE

Widest-need is no longer one of the defined options for expression evaluation. *Rationale*: Lack of prior art.

In **`<float.h>`**, the **`FLT_EVAL_METHOD`** macro now replaces FPCE's **`_MIN_EVAL_FORMAT`** and **`_WIDEST_NEED_EVAL`** macros. *Rationale*: Two macros are no longer needed, since widest-need was removed.

The implementation can now set **`FLT_EVAL_METHOD`** to indicate that its method of expression evaluation is defined by the implementation or indeterminate, rather than one of the standard defined methods. *Rationale*: Eases acceptance while still promoting standard-defined methods.

Macros replace pragmas as the means for uttering translation directives. *Rationale*: Macros are preferred because of general limitations of pragmas and the desire not to require standard pragmas.

Translation directives can now be applied to a compound statement, as well as to the subsequent part of a translation unit. *Rationale*: Avoids unnecessary restrictions on optimization and supports locality in source code.

FPCE's directives for allowing wide representation for function parameters, function returns, and local variables have been dropped. *Rationale*: This FPCE specification is awkward and use of the directives is highly system dependent; the need seems insufficient, given availability of the **`float_t`** and **`double_t`** types.

Wide character functions are now covered. *Rationale*: Brings the proposal up-to-date with the current C standard.

The predefined constant **`__STDC_IEC_559__`** replaces **`__FPCE_IEEE`**, and **`__FPCE__`** has been dropped. *Rationale*: The C standard generally references international versions of standards. Because the proposal is for changes to the standard and not for extensions, the FPCE name is not appropriate, and **`__FPCE__`** is not needed.

Matching of translation-time and execution-time conversion of decimal numbers is now recommended instead of required. *Rationale*: This requirement would be too great a burden for some implementations.

Overloading is no longer specified as part of the main proposal for **`<math.h>`** functions. A separate proposal advocates overloading macros as a replacement for FPCE's overloading functions. Using **`#undef`** to remove a macro definition reveals a **`double`** version of the function, a la C89 **`<math.h>`**. Otherwise this change is cosmetic. *Rationale*: The C committee asked for overloading as a separate proposal. Macros allow a more natural specification and continue to support C89's unmasking scheme.

The names **`fegetexceptflag`** and **`fesetexceptflag`** replace FPCE's **`fegetexcept`** and **`fesetexcept`**. *Rationale*: This distinguishes these functions—which deal with the entire content of the flags, take an **`fexcept_t*`** argument, and are used as a pair—from the other exception functions.

Recommendations about type widths for non-IEC 559 implementations, and the recommendation for a warning when the recommendations for the widths are not met, have been dropped. *Rationale*: The needed doesn't seem sufficient, now that the diversity of implementations has loosened such expectations about types.

The requirement for a "warning" when a hexadecimal constant cannot be represented exactly has been downgraded to a recommendation (for a non-fatal diagnostic). *Rationale*: The need does not seem sufficient to introduce warning requirements into the C standard, just for this purpose.

Certain recommendations for operator and function documentation have been dropped. *Rationale*: This will be left to LIA.

Certain specification related to the floating-point environment now covers just IEC 559 implementations. (This change does not affect the header **`<fenv.h>`**.) *Rationale*: This allows localizing most of the material about the floating-point environment in **`<fenv.h>`** and the IEC 559 annex, which is appropriate since the specification in question is essentially for IEC 559 implementations anyway.

Acknowledgments

People who have made especially substantial contributions to this proposal include, in alphabetical order: Jerome Coonen, Bill Gibbons, David Hough, Rex Jaeschke,

W. Kahan, Clayton Lewis, Stuart McDonald, Colin McMaster, Rick Meyers, David Prosser, and Fred Tydeman.

Others who have provided invaluable contributions, reviews, or administrative help include, in alphabetical order: Joel Boney, Norris Boyd, Larry Breed, Walter Bright, W. J. Cody, Elizabeth Crockett, Karen Deach, James Demmel, Fred Dunlap, Frank Farance, Clive Feather, Yinsun Feng, Samuel Figueroa, Paul Finlayson, James Frankel, Scott Fraser, David Gay, Eric Grosse, Ron Guilmette, Doug Gwyn, Bill Homer, Kenton Hanson, Paul Hilfinger, Martha Jaffe, Bob Jervis, Ed Johnston, David Keaton, Earl Killian, David Knaak, John Kwan, Roger Lawrence, Tom MacDonald, Michael Meissner, Randy Meyer, Randy Meyers, Antonine Mione, Stephen Moshier, Adolfo Nemirovsky, Jon Okada, Conor O'Neill, Tom Pennello, Tim Peters, P.J. Plauger, Thomas Plum, Sanjay Poonen, Pat Ricci, Ali Sazegari, Roger Schlafly, Steve Sommars, Richard Stallman, Linda Stanberry, Gordon Sterling, Bill Torkelson, Douglas Walls, and Terrence Yee.

The author benefited from experience with numerics and languages groups at Apple Computer, Inc., developing the Standard Apple Numerics Environment (SANE) and its various language bindings, and at Taligent, Inc., implementing this specification.

C9X Edits

Feature: "Recommended practice" designation

3 Definitions and conventions

Insert the following definition:

3.xx recommended practice: Sections so entitled contain specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations.

Feature: Expression evaluation methods

5.1.2.3 Program execution

Remove the last sentence of example 3.

5.2.4.2.2 Characteristics of floating types `<float.h>`

Replace the third paragraph with:

Of the values in the **`<float.h>`** header, **`FLT_EVAL_METHOD`** and **`FLT_RADIX`** shall be constant expressions suitable for use in **`#if`** preprocessing directives; all other values need not be constant expressions. All except **`FLT_EVAL_METHOD`**, **`FLT_RADIX`**, and **`FLT_ROUNDS`** have separate names for all three floating-point types. The floating-point model representation is provided for all values except **`FLT_EVAL_METHOD`** and **`FLT_ROUNDS`**.

After the definition of FLT_ROUNDS add:

The values of operations subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the value of **`FLT_EVAL_METHOD`**:

-1 indeterminable
0 evaluate all operations and constants just to the range and precision of the type
1 evaluate operations and constants of type **`float`** and **`double`** to the range and precision of the **`double`** type, evaluate **`long double`** operations and constants to the range and precision of the **`long double`** type
2 evaluate all operations and constants to the range and precision of the **`long double`** type

All other values for **`FLT_EVAL_METHOD`** characterize implementation-defined behavior.

> [Well defined expression evaluation is essential for predictable arithmetic. Although specifying just one method would have facilitated porting code, any one method would

have been unacceptably inefficient on some important architectures. On the other hand, still other expression evaluation methods are conceivable, for example evaluating **float** operations to **float** format, and all others to **long double.** The expression evaluation methods described in this section comprise an intentionally small set with at least one method that is efficient for any of the existing or anticipated, commercially significant, floating-point architectures.

Standard C generally defines the semantic type of a floating-point operation to be the widest type of its operands, but gives explicit license to represent the operation's result in a format wider than its type.

Representation of constants in a format commensurate with expression evaluation, not a traditionally uniform practice, better meets certain expectations than would representation strictly according to semantic type—for example, **0.1f == 1.0f/10.0f**, even if **float** operations are evaluated to **double** or **long double**. Viewed as translation-time operations that convert decimal strings to internal floating representations, literal floating constants naturally follow the method of expression evaluation.

Early drafts for this specification defined a **#pragma evaluate** which allowed switching expression evaluation methods between external declarations. This facility was believed to be without sufficient utility and somewhat error prone. Implementations that support multiple expression evaluation methods can supply translation options. The evaluation of **FLT_EVAL_METHOD** is intended to correctly reflect the expression evaluation method currently in effect.

In the following description of floating-point architectures, the terms *extended*, *double*, and *single* apply to both IEC 559 and non-IEC 559 systems. Generally, extended is wider than double which is wider than single.

Extended-based. The arithmetic engine is extended. Source operands can be single, double, or extended, though generally arithmetic with single and double types is less efficient, requiring extra conversions. Examples include Intel 80x87, Cyrix 3D87, Motorola 6888x, and AT&T WE 32x06. The Motorola 88110 and Intel 960 can be used as extended-based architectures, or alternatively as single/double/extended ones (see below).

Double-based. The arithmetic engine is double. Source operands can be single or double, though generally arithmetic with the single type is less efficient, requiring extra conversions. Extended may be available, but implemented in software. Examples include IBM RISC System/6000, PDP-11 in double mode, CRAY, and CYBER 180. On CRAY and CYBER, single and double may be the same format. The CYBER provides some hardware support for extended.

Single/double. These provide orthogonal operations for single and double arithmetic. Single is typically faster than double. Extended may be available, but implemented in software. Examples include MIPS, SPARC, HP PA-RISC, Motorola 88100, Intel 860, and systems assembled with Weitek or BIT processors. The MIPS, SPARC, and HP PA-RISC architectures specify extended, though it is not yet in hardware.

Single/double/extended. These provide orthogonal operations for single, double, and extended arithmetic. Single is faster than double, which is faster than extended. Examples include Motorola 88110, Intel 960, IBM S/370, and VAX.

The early C implementations provided just the **float** and **double** floating-point types and evaluated all floating expressions to **double**. Intentional or not, some C programs have relied on extra precision for their computation with **float** operands. **FLT_EVAL_METHOD** equal 1 is a natural choice for double-based architectures.

Implementations for single/double and single/double/extended architectures may find **FLT_EVAL_METHOD** equal 0 compellingly more efficient, despite potential problems of conformity to expectations based on C's tradition of wide evaluation.

Even on a single/double or single/double/extended architecture, an implementation might have **FLT_EVAL_METHOD** equal 1, for compatibility reasons. Common statements of the form

```
f1 = f2 op f3; /* where f1, f2, f3 are of type float  */
```

can be done optimally by many such implementations, including all IEC 559 ones, where rounding the result to **double** and then to **float** is equivalent to rounding to **float** directly.

FLT_EVAL_METHOD equal 2 is common on extended-based architectures. Programs that run under one of the other expression evaluation methods generally run at least as well when all expressions are evaluated to **long double**. Most program failures due to extra precision arise from its inconsistent use (see 5.1.2.3).

"Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report defined *widest-need* expression evaluation methods: With widest-need, the evaluation format for an operation is the widest of the semantic types appearing in a certain enclosing expression and at least as wide as an implementation-defined minimum evaluation format (**float**, **double**, or **long double**). More precisely, the evaluation format for an operation subject to the usual arithmetic conversions, or for an assignment (including the assignment of function arguments to parameters, but not cast conversions), is propagated to its operands (or arguments): if an operand is a variable or an operation not subject to the usual arithmetic conversions it is converted to the evaluation format; if the operand is an operation subject to the usual arithmetic conversions, or a floating constant, the evaluation format is imposed recursively. The definition of widest-need is based on a similar scheme for Fortran presented in [9]. It does not affect integer expression evaluation.

As computer speed and memory size increase, so will the number of problems attempted and the size of data sets. Thus, the likelihood that a program will suffer serious roundoff error for some actual data will increase. Wider precision, not for the entire computation but just for expressions containing certain variables, often will fix the problem, without unduly affecting performance, and without requiring costly error analysis. With widest-need, an expression is automatically evaluated to the format of its widest component. To achieve the same effect without widest-need expression evaluation, the programmer must add or delete casts and constant suffixes throughout the program.

Widest-need expression evaluation is a particularly attractive compromise for architectures whose wider formats are significantly slower. It offers the accuracy of wide evaluation where likely needed and also the speed of narrow evaluation where clearly intended. Note that casts can be used to inhibit widest-need widening, even within a wide expression.

This specification does not define widest-need methods because of the scarcity of prior art. Of course, an implementation could provide widest-need, as prescribed in the Technical Report, as an optional expression evaluation method.

"Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report specifies three pragmas granting license for the implementation to represent function return values, function parameters, and variables in a format wider than their declared type:

#pragma fp_wide_function_returns *on-off-switch*
#pragma fp_wide_function_parameters *on-off-switch*
#pragma fp_wide_variables *on-off-switch*

#pragma fp_wide_function_returns on—allows floating return values to be represented in a wider format than (and not narrowed to) the declared type of the function.

#pragma fp_wide_function_parameters on—allows floating parameters to be in a wider format than (and not narrowed to) the declared type of the formal parameter.

#pragma fp_wide_variables on—allows values of automatic scalar floating variables to be in a wider format than their declared type.

If the *on-off-switch* is **off** then widening is disallowed. Standard C compatibility requires that the default state for the pragmas be *off*.

These pragmas can occur outside external declarations, and allow (if *on*) or disallow (if *off*) widening from their occurrence until another pragma instructing otherwise is encountered, or until the end of the translation unit. The effect of these pragmas appearing

inside an external declaration is undefined. Widening is consistent throughout the effect of an enabling pragma: either all instances of the returns, parameters, or variables are widened or none are; the representation format for a parameter or variable does not vary. However, which, if any, of these pragmas actually cause widening is implementation-defined. The **fp_wide_function_returns** and **fp_wide_function_parameters** pragmas may affect function definitions or calls but not prototypes.

When the implementation detects an address or **sizeof** operator of a widened parameter or variable it emits a translation-time warning; execution-time behavior is then undefined.

This mechanism facilitates generating efficient code for extended-based or double-based architectures. The types **float_t** and **double_t** allow finer application than do the pragmas, but require more extensive changes to existing code.

The function writer who decides that narrowing arguments and returns to their semantic type is less desirable than efficiency can write the function under the effect of enabling **fp_wide_function_parameters** and **fp_wide_function_returns** pragmas. Similarly the programmer who uses the function can apply these pragmas to the call site. In either case widening (not narrowing) may or may not occur. These pragmas do not demand widening, nor even recommend it, but merely declare that widening would be acceptable. It is up to the implementation to determine whether widening would be both safe and also more efficient. For example, implementations that pass different type (floating) parameters in different formats can not widen parameters of external functions safely.

Even among implementations which respond to the pragmas, the location in the source code where the pragmas must be placed to be effective may vary. For example, an implementation might perform requisite narrowing of parameters at the call site, in which case the call would have to be under the effect of an enabling **fp_wide_function_parameters** pragma; or, it might narrow within the function, in which case the function definition would have to be under the effect of the pragma.

The pragmas do not affect function prototypes. Doing so might have benefited implementations that pass different floating type arguments and returns in different ways. However, maintaining consistency between the prototype and implementation seemed particularly error prone. A language extension to assure the consistency seemed unjustified. The **float_t** and **double_t** type definitions are available for such prototypes.

Casts are not affected by any of these pragmas, nor by wide expression evaluation, so can be used portably to force narrowing.

Another approach would have been to introduce **register** as a function qualifier that would have allowed widening of both the parameters and the return value. This would have been inconsistent with the **register** storage class specifier, which is unrelated to widening, and would not have helped with variables.

This specification does not include **fp_wide** pragmas, nor macro equivalents, because their use and meaning would vary so much from one implementation to the next. Of course and implementation could provide them as an extension.]

Feature: Dynamic FLT_ROUNDS

5.2.4.2.2 Characteristics of floating types <float.h>

In the fourth paragraph, add the following footnote to "rounding mode":

Evaluation of **FLT_ROUNDS** correctly reflects any execution-time change of rounding mode through the function **fesetround** in **<fenv.h>**.

Feature: Hexadecimal floating constants

3. Definitions and conventions

Add the following definition:

3.x correctly rounded result: A representation in the result format that is nearest in value, subject to the effective rounding mode, to what the result would be given unlimited range and precision.

6.1.3.1 Floating constants

Replace the syntax with the following syntax:

Syntax

floating-constant:
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

hexadecimal-floating-constant:
0x *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
0X *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
0x *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$
0X *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ **-**

digit-sequence:
digit
digit-sequence digit

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ **.** *hexadecimal-digit-sequence*
hexadecimal-digit-sequence **.**

binary-exponent-part:
p *sign*$_{opt}$ *digit-sequence*
P *sign*$_{opt}$ *digit-sequence*

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

floating-suffix: one of
f l F L

In the third sentence of the Description, replace "**e** or **E**" *with* "**e**, **E**, **p**, or **P**".

Replace the second clause of the last sentence of the Description with:

for decimal floating constants, either the period or the exponent part shall be present.

Replace the first paragraph of the Semantics with:

The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, if the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants, if **FLT_RADIX** is a power of 2 and the scaled value is in the range of representable values (for its type), then the result of a hexadecimal floating constant is correctly rounded.

Recommend practice

The implementation emits a non-fatal diagnostic if a hexadecimal constant cannot be represented exactly in its evaluation format.

6.1.3.2 Integer constants

Delete the hexadecimal-digit syntax (now in 6.1.3.1).

6.1.8 Preprocessing numbers

Before the last line of the Syntax insert:

pp-number **p** *sign*
pp-number **P** *sign*

[Hexadecimal more clearly expresses the significance of floating constants.

The binary-exponent part is required to avoid ambiguity from an **f** suffix (being mistaken as a hexadecimal digit).

Unlike integers, floating values cannot all be represented directly by hexadecimal constant syntax. A sign can be prefixed for negative numbers and -0. Infinities might be produced by hexadecimal constants that overflow. NaNs require some other mechanism. The character sequence **0x1.FFFFFEp128f** might appear to be an IEC 559 single-format NaN, which is characterized by an unbiased exponent of 128 and a nonzero significand; however, this character sequence overflows to an infinity in the single format.

An alternate approach might have been to represent bit patterns. For example

```
#define FLT_MAX         0x.7F7FFFFF
```

This would have allowed representation of NaNs and infinities. However, numerical values would have been more obscure owing to bias in the exponent and the implicit significand bit. NaN representations would not have been portable—even the determination of IEC 559 quiet NaN vs signaling NaN is implementation-defined. NaNs and infinities are provided through macros in 7.x.

Note that constants of **long double** type are not fully portable, even among IEC 559 implementations. See rationale in Annex X.2.

The straightforward approach of denoting octal constants by a **0** prefix would have been inconsistent with allowing a leading **0** digit—a moot point as the need for octal floating constants was deemed insufficient.]

7.10.1.4 The strtod function

Replace the first sentence of the second paragraph of the Description with:

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.1.3.1;
- a **0x** or **0X**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary-exponent part, where either the decimal-point character or the binary-exponent part is present;

but no floating suffix.

In the first sentence of the third paragraph of the Description, replace "if neither an exponent part nor a decimal-point character appears" *with* "if neither an exponent part, a binary-exponent part, nor a decimal-point character appears".

After the fourth paragraph of the Description insert the paragraph:

If the subject sequence has the hexadecimal form and **FLT_RADIX** is a power of 2, then the value resulting from the conversion is correctly rounded.

After the Description insert:

Recommended practice

If the subject sequence has the hexadecimal form and **FLT_RADIX** is not a power of 2, then the result is one of the two numbers in the appropriate internal format that are

adjacent to the hexadecimal floating source value, with the extra stipulation that the error have a correct sign for the current rounding direction.

7.16.4.1.1 The `wcstod` function

Replace the first sentence of the second paragraph of the Description with:

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in subclause 6.1.3.1;

- a **`0x`** or **`0X`**, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary-exponent part, where either the decimal-point character or the binary-exponent part is present;

but no floating suffix.

In the first sentence of the third paragraph of the Description, replace "if neither an exponent part nor a decimal-point wide character appears" *with* "if neither an exponent part, a binary-exponent part, nor a decimal-point wide character appears".

After the fourth paragraph of the Description insert the paragraph:

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

After the Description insert:

Recommended practice

If the subject sequence has the hexadecimal form and **`FLT_RADIX`** is not a power of 2, then the result is one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error have a correct sign for the current rounding direction.

7.9.6.1 The `fprintf` function

In the third bullet of the second paragraph of the Description, replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the fourth bullet of the second paragraph of the Description, replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the # flag specification in the fourth paragraph of the Description, replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the 0 flag specification in the fourth paragraph of the Description, replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

After the g, G conversion specifier and its meaning in the fifth paragraph, insert:

a, **A** The **double** argument is converted in the style *[-]0xh.hhhhp±d*. The number of hexadecimal digits *h* after the decimal-point character is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish values of type **double**, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point character is nonzero for normalized floating-point numbers and is otherwise unspecified; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **a** conversion specifier will produce a number with **x** and **p** and the **A** conversion specifier will produce a number with **X** and **P**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

In the second sentence of the preceding inserted text, footnote the word "distinguish" with:

The precision *p* is sufficient to *distinguish* values of the source type if

$$16^{p-1} > b^{n}$$

where *b* is **FLT_RADIX** and *n* is the number of base-*b* digits in the significand of the source type. A smaller *p* might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

In the third sentence of the preceding inserted text, footnote the word "implementation" with:

Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble boundaries.

At the end of the Description append the paragraphs:

For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

If **FLT_RADIX** is not a power of 2, the result is one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have a correct sign for the current rounding direction.

[To illustrate alignment to nibble (4-bit) boundaries, the next value greater than one in the common IEC 559 80-bit extended format should be

```
0x8.000000000000001p-3
```

The next value less than one in IEC 559 double should be

```
0x1.fffffffffffffp-1
```

Note that if the precision is missing, trailing zeros may be omitted. For example, the value positive zero might be

0x0.p+0

The more suggestive conversion specifiers for hexadecimal formatting, namely **x** and **h**, were unavailable. Since **h** was taken **H** was ruled out in favor of a lower/upper case option. Possibilities other than **a** included: **b j k m q r t v w y z**. The optional **h** to indicate hexadecimal floating, as in **%he**, was deemed a less natural fit with the established scheme for specifiers and options.

The *decimal-point character* is defined in 7.1.1. *Radix-point character* would be a better term.

Use of the **A** format specifier constitutes a minor extension to ISO/IEC 9899 : 1990 (E) which does not reserve it.]

7.16.2.1 The **fwprintf** function

In the third bullet of the third paragraph of the Description, replace "**e**, **E**," *with* "**a**, **A**, **e**, **E**,".

In the fourth bullet of the third paragraph of the Description, replace "**e**, **E**," *with* "**a**, **A**, **e**, **E**,".

In the # flag specification in the fifth paragraph of the Description, replace "**e**, **E**," *with* "**a**, **A**, **e**, **E**,".

In the 0 flag specification in the fifth paragraph of the Description, replace "**e**, **E**," *with* "**a**, **A**, **e**, **E**,".

After the g, G conversion specifier and its meaning in the sixth paragraph, insert:

a, **A** The **double** argument is converted in the style *[-]0xh.hhhhp±d*. The number of hexadecimal digits *h* after the decimal-point wide character is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and **FLT_RADIX** is not a power of 2, then the precision is sufficient to distinguish values of type double, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point wide character is nonzero for normalized floating-point numbers and is otherwise unspecified; if the precision is zero and the **#** flag is not specified, no decimal-point wide character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **a** conversion specifier will produce a number with **x** and **p** and the **A** conversion specifier will produce a number with **X** and **P**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.

In the second sentence of the preceding inserted text, footnote the word "distinguish" with:

The precision *p* is sufficient to *distinguish* values of the source type if

$$16^{p-1} > b^{n}$$

where *b* is **FLT_RADIX** and *n* is the number of base-*b* digits in the significand of the source type. A smaller *p* might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point wide character.

In the third sentence of the preceding inserted text, footnote the word "implementation" with:

Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries.

At the end of the Description append the paragraphs:

For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

If **FLT_RADIX** is not a power of 2, the result is one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have a correct sign for the current rounding direction.

7.9.5.8 The `fscanf` function

In the third bullet of the second paragraph of the Description, replace "**e**, **f**, and **g**" *with* "**a**, **e**, **f**, and **g**".

In the Description paragraph listing the conversion specifiers, replace "**e, f, g**" *with* "**a, e, f, g**".

In the fourth to last paragraph in the Description, replace "specifiers **E**," *with* "specifiers **A**, **E**,". *In the same paragraph, replace* "respectively, **e**," *with* "respectively, **a**, **e**,".

7.16.2.2 The `fwscanf` function

In the third bullet of the second paragraph of the Description, replace "**e**, **f**, and **g**" *with* "**a**, **e**, **f**, and **g**".

In the Description paragraph listing the conversion specifiers, replace "**e, f, g**" *with* "**a, e, f, g**".

In the third to last paragraph in the Description, replace "specifiers **E**," *with* "specifiers **A**, **E**,". *In the same paragraph, replace* "respectively, **e**," *with* "respectively, **a**, **e**,".

Feature: Output of wide exponents

7.9.6.1 The `fprintf` function

In the e item in the list of conversion specifiers and their meanings, in the next to last sentence, replace "at least two digits" *with* "at least two digits, and only as many more digits as necessary to represent the exponent".

7.16.2.1 The `fwprintf` function

In the e item in the list of conversion specifiers and their meanings, in the next to last sentence, replace "at least two digits" *with* "at least two digits, and only as many more digits as necessary to represent the exponent".

Feature: Support for infinities, NaNs, and -0

6.1.2.5 Types

To the sixth paragraph append :

Values of floating types might include infinities and NaNs, as well as floating-point numbers. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising an exception. A *signaling NaN* generally raises an exception when occurring as an arithmetic operand.

7.10.1.4 The `strtod` function

In the first sentence of the second paragraph of the Description, as amended by the change above for hexadecimal floating constants, after the last bullet, insert:

- one of **`INF`** or **`INFINITY`**, ignoring case
- one of **`NAN`** or **`NAN(`***n-char-sequence*$_{opt}$**`)`**, ignoring case in the **`NAN`** part, where

n-char-sequence:
digit
nondigit
n-char-sequence digit
n-char-sequence nondigit

In the first sentence of the third paragraph of the Description replace "expected form" *with* "expected form for a floating-point number".

After the first sentence of the third paragraph of the Description insert:

A character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity, if representable in the **`double`** type, else like a floating constant that is too large for the range of **`double`**. A character sequence **`NAN`** or **`NAN(`***n-char-sequence*$_{opt}$**`)`** is interpreted as a quiet NaN, if supported in the **`double`** type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequences is implementation-defined.

In the preceding insertion, footnote "implementation-defined" with:

An implementation may use the *n-char-sequence* to determine extra information to be represented in the NaN's significand.

In the next to last sentence of the third paragraph of the Description, footnote "negated" with:

The **strtod** function honors the sign of zero if the arithmetic supports signed zeros.

> [So much is implementation-defined because so little is portable. Attaching meaning to NaN significands is problematic, even for one implementation, even an IEC 559 one. For example, the IEC 559 standard does not specify the effect of format conversions on NaN significands—conversions, perhaps generated by the compiler, may alter NaN significands in obscure ways.
>
> Requiring a sign for NaN or infinity input was considered as a way of minimizing the chance of mistakenly accepting nonnumeric input. The need for this was deemed insufficient, partly on the basis of prior art.
>
> For simplicity, the infinity and NaN representations were provided through straightforward extensions, rather than through a new locale. Note also that Standard C locale categories do not affect the representations of infinities and NaNs.
>
> A proposal that **strtod** be allowed to return a NaN for invalid numeric input, as recommended by IEEE standard 854, was withdrawn because of the incompatibility with the C standard, which demands that **strtod** return 0 for invalid numeric input.]

7.16.4.1.1 The **wcstod** function

In the first sentence of the second paragraph of the Description, as amended by the change above for hexadecimal floating constants, after the last bullet, insert:

- one of **INF**, **INFINITY**, or any wide string equivalent except for case
- one of **NAN**, or **NAN(***n-wchar-sequence*$_{opt}$**)**, or any wide string equivalent except for case in the **NAN** part, where

n-wchar-sequence:
digit
nondigit
n-wchar-sequence digit
n-wchar-sequence nondigit

In the first sentence of the third paragraph of the Description replace "expected form" *with* "expected form for a floating-point number".

After the first sentence of the third paragraph of the Description insert:

A wide character sequence **INF** or **INFINITY** is interpreted as an infinity, if representable in the **double** type, else like a floating constant that is too large for the range of **double**. A wide character sequence **NAN** or **NAN(***n-wchar-sequence*$_{opt}$**)** is interpreted as a quiet NaN, if supported in the **double** type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequences is implementation-defined.

In the preceding insertion, footnote "implementation-defined" with:

An implementation may use the *n-wchar-sequence* to determine extra information to be represented in the NaN's significand.

In the next to last sentence of the third paragraph of the Description, footnote "negated" with:

The **wcstod** function honors the sign of zero if the arithmetic supports signed zeros.

7.9.6.1 The `fprintf` function

In the third bullet of the second paragraph of the Description, replace "**E**, and **f**" *with* "**E**, **f**, and **F**".

In the fourth bullet of the second paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

Footnote the + flag specification in the fourth paragraph of the Description with:

The results of all floating conversions of a negative zero include a minus sign.

In the # flag specification in the fourth paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the 0 flag specification in the fourth paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the list of conversion specifiers and their meanings, change the "**f**" *bullet to* "**f, F**".

In the f item in the list of conversion specifiers and their meanings, replace "**double** argument" *with* "**double** argument representing a floating-point number".

At the end of the f item in the list of conversion specifiers and their meanings, append the paragraph:

A **double** argument representing an infinity is converted in one of the styles *[***-***]***inf** or *[***-***]***infinity**—which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[***-***]***nan** or *[***-***]***nan(***n-char-sequence***)**—which style, and the meaning of any *n-char-sequence*, is implementation-defined. The **F** conversion specifier will produce an **INF**, **INFINITY**, or **NAN** instead of an **inf**, **infinity**, or **nan**.

Footnote the preceding inserted paragraph with:

When applied to infinite and NaN values, the **-**, **+**, and *space* flag characters have their usual meaning; the **#** and **0** flag characters have no effect .

In the e item in the list of conversion specifiers and their meanings, replace "**double** argument" *with* "**double** argument representing a floating-point number".

At the end of the e item in the list of conversion specifiers and their meanings, append the sentence:

A **double** argument representing an infinity or a NaN is converted in the style of an **f** or **F** conversion specifier.

In the newly inserted a item in the list of conversion specifiers and their meanings, replace "**double** argument" *with* "**double** argument representing a floating-point number".

In the newly inserted a item in the list of conversion specifiers and their meanings, append the sentence:

A **double** argument representing an infinity or a NaN is converted in the style of an **f** or **F** conversion specifier.

In the g item in the list of conversion specifiers and their meanings, replace "style **E**" *with* "style **E** or **F**".

> [See the rationale above for **strtod**.
>
> Use of the **F** format specifier constitutes a minor extension to ISO/IEC 9899 : 1990 (E) which does not reserve it.]

7.16.2.1 The **fwprintf** function

In the third bullet of the third paragraph of the Description, replace "**E**, and **f**" *with* "**E**, **f**, and **F**".

In the fourth bullet of the third paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

Footnote the + flag specification in the fifth paragraph of the Description with:

The results of all floating conversions of a negative zero include a minus sign.

In the # flag specification in the fifth paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the 0 flag specification in the fifth paragraph of the Description, replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the list of conversion specifiers and their meanings, change the "**f**" *bullet to* "**f, F**".

In the f item in the list of conversion specifiers and their meanings, replace "**double** argument" *with* "**double** argument representing a floating-point number".

At the end of the f item in the list of conversion specifiers and their meanings, append the paragraph:

A **double** argument representing an infinity is converted in one of the styles *[***−***]***inf** or *[***−***]***infinity**—which style is implementation-defined. A **double** argument representing a NaN is converted in one of the styles *[***−***]***nan** or *[***−***]***nan(***n-wchar-sequence***)**—which style, and the meaning of any *n-wchar-sequence*, is implementation-defined. The **F** conversion specifier will produce an **INF**, **INFINITY**, or **NAN** instead of an **inf**, **infinity**, or **nan**.

Footnote the preceding inserted paragraph with:

When applied to infinite and NaN values, the **−**, **+**, and *space* flag wide characters have their usual meaning; the **#** and **0** flag wide characters have no effect .

In the e item in the list of conversion specifiers and their meanings, replace "**double** argument" *with* "**double** argument representing a floating-point number".

At the end of the e item in the list of conversion specifiers and their meanings, append the sentence:

A **`double`** argument representing an infinity or a NaN is converted in the style of an **`f`** or **`F`** conversion specifier.

In the newly inserted a item in the list of conversion specifiers and their meanings, replace "**`double`** argument" *with* "**`double`** argument representing a floating-point number".

In the newly inserted a item in the list of conversion specifiers and their meanings, append the sentence:

A **`double`** argument representing an infinity or a NaN is converted in the style of an **`f`** or **`F`** conversion specifier.

In the g item in the list of conversion specifiers and their meanings, replace "style **`E`**" *with* "style **`E`** or **`F`**".

Feature: Strings -> float, long double

7.10.1 String conversion functions

Add two new subclauses and renumber others accordingly (note that HUGE_VALF and HUGE_VALL are defined in <math.h>):

7.10.1.4 The `strtof` function

Synopsis

```
#include <stdlib.h>
float strtof(const char *nptr, char **endptr);
```

Description

The **`strtof`** function is similar to the **`strtod`** function, except the returned value has type **`float`** and plus or minus **`HUGE_VALF`** is returned for values outside the range.

7.10.1.7 The `strtold` function

Synopsis

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr);
```

Description

The **`strtold`** function is similar to the **`strtod`** function, except the returned value has type **`long double`** and plus or minus **`HUGE_VALL`** is returned for values outside the range.

7.16.4.1 Wide-string numeric conversion functions

Add two new subclauses and renumber others accordingly (note that HUGE_VALF and HUGE_VALL are defined in <math.h>):

7.16.4.1.x The `wcstof` function

Synopsis

```
#include <wchar.h>
float wcstof(const wchar_t *nptr, wchar_t **endptr);
```

Description

The **`wcstof`** function is similar to the **`wcstod`** function, except the returned value has type **`float`** and plus or minus **`HUGE_VALF`** is returned for values outside the range.

7.16.4.1.x The `wcstold` function

Synopsis

```
#include <wchar.h>
long double wcstold(const wchar_t *nptr, wchar_t **endptr);
```

Description

The **`wcstold`** function is similar to the **`wcstod`** function, except the returned value has type **`long double`** and plus or minus **`HUGE_VALL`** is returned for values outside the range.

Feature: FP environment management

5.1.2.3 Program execution

In the first sentence of the second paragraph, footnote "side effects" with:

The IEC 559 standard for binary floating-point arithmetic requires certain status flags and control modes, with user access. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state will need to regard changes to it as side effects—see Annex X for details. The floating-point environment library **`<fenv.h>`** provides a programming facility for indicating when these side effects matter, freeing the implementation in other cases.

Feature: Correctly rounded binary-decimal conversion

6.1.3.1 Floating constants

Insert after semantics (as revised above) the paragraph:

Recommended practice

The translation-time conversion of floating constants matches the execution-time conversion of character strings by library functions, such as **`strtod`**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.

Footnote the preceding paragraph with:

The specification for the library functions recommends more accurate conversion than required for floating constants. See 7.10.1.4.

7.10.1.4 The `strtod` function

After the Description insert the paragraph:

Recommended practice

If the subject sequence has the decimal form and at most **`DECIMAL_DIG`** (defined in **`<math.h>`**) significant digits, then the value resulting from the conversion is correctly rounded. If the subject sequence D has the decimal form and more than **`DECIMAL_DIG`** significant digits, consider the two bounding, adjacent decimal strings L and U, both having **`DECIMAL_DIG`** significant digits, such that the values of L, D, and U satisfy $L \le D \le U$. The result of conversion is one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D has a correct sign for the current rounding direction.

Footnote the preceding inserted paragraph with:

`DECIMAL_DIG`, defined in **`<math.h>`**, is recommended to be sufficiently large that L and U will usually round to the same internal floating value, but if not will round to adjacent values.

7.16.4.1.1 The `wcstod` function

After the Description insert the paragraph:

Recommended practice

If the subject sequence has the decimal form and at most **`DECIMAL_DIG`** (defined in **`<math.h>`**) significant digits, then the value resulting from the conversion is correctly rounded. If the subject sequence D has the decimal form and more than **`DECIMAL_DIG`** significant digits, consider the two bounding, adjacent decimal strings L and U, both having **`DECIMAL_DIG`** significant digits, such that the values of L, D, and U satisfy $L \le D \le U$. The result of conversion is one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D has a correct sign for the current rounding direction.

Footnote the preceding inserted paragraph with:

`DECIMAL_DIG`, defined in **`<math.h>`**, is recommended to be sufficiently large that L and U will usually round to the same internal floating value, but if not will round to adjacent values.

7.9.6.1 The `fprintf` function

At the end of the Description insert the paragraph:

Recommended practice

For **`e`**, **`E`**, **`f`**, **`F`**, **`g`**, and **`G`** conversions, if the number of significant decimal digits is at most **`DECIMAL_DIG`**, then the result is correctly rounded. If the number of significant decimal digits is more than **`DECIMAL_DIG`** but the source value is exactly representable with **`DECIMAL_DIG`** digits, then the result is an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **`DECIMAL_DIG`** significant digits; the value of the result decimal string D satisfies $L \le D \le U$, with the extra stipulation that the error have a correct sign for the current rounding direction.

In the first sentence of the preceding paragraph, footnote "correctly rounded" with:

For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

7.16.2.1 The `fwprintf` function

At the end of the Description insert the paragraph:

Recommended practice

For **`e`**, **`E`**, **`f`**, **`F`**, **`g`**, and **`G`** conversions, if the number of significant decimal digits is at most **`DECIMAL_DIG`**, then the result is correctly rounded. If the number of significant decimal digits is more than **`DECIMAL_DIG`** but the source value is exactly representable with **`DECIMAL_DIG`** digits, then the result is an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having **`DECIMAL_DIG`** significant digits; the value of the result decimal string D satisfies $L \le D \le U$, with the extra stipulation that the error have a correct sign for the current rounding direction.

In the first sentence of the preceding paragraph, footnote "correctly rounded" with:

For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

Feature: Contraction control

6.3 Expressions

Append to the subclause the paragraph :

A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method. The **`FP_CONTRACT`** macros in **`<math.h>`** provide a way to

disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation defined.

Footnote the first sentence in the preceding new paragraph with:

A contracted expression might also omit side effects such as the raising of floating-point exception flags.

Footnote the preceding new paragraph with:

This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use must be well-defined and clearly documented.

> [An implementation that is able to multiply two **`double`** operands and produce a **`float`** result in just one machine instruction might contract the multiplication and assignment in:
>
> ```
> float f;
> double d1, d2;
> ...
> f = d1 * d2;
> ```
>
> Other examples of potential contractions include:
>
> | | |
> |---|---|
> | multiply-add | **`x * y + z`** |
> | compound assignments | **`+=`**, **`-=`**, etc. |
> | ternary add | **`x + y + z`** |
>
> A contraction might omit a rounding error that would have fortuitously improved accuracy, so that the more accurate evaluation of a subexpression might result in a less accurate evaluation of the containing expression.
>
> Contractions can lead to subtle anomalies even while increasing overall accuracy. The value of expressions like **`a * b + c * d`** will depend on how the translator uses a contracted multiply-add. Knowing that the implementation contracts multiply-adds, the programmer should be able to control results (and reap the benefits of contraction) through simple coding measures, for example parenthesizing **`(a * b) + c * d`**. However, the Intel 860's multiply-add is slightly more problematic. Since it keeps a wide but partial product, **`a * b + z`** may differ from **`c * d + z`** even though the exact mathematical products **`a * b`** and **`c * d`** are equal; the result depends not just on the mathematical result and the format, as ordinarily expected for error analysis, but also on the particular values of the operands.
>
> The extra accuracy of the IBM RISC System/6000 and HP PA8000's *fused* multiply-add, which produces a result with just one rounding, can be exploited for simpler and faster codes. See [19] for details.]

Feature: Optimization guidance

5.1.2.3 Program execution

Add an example after current example 3:

Implementations employing wide registers must take care to honor appropriate semantics. Values must be independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register must not alter the value. Also, an

explicit *store and load* must round to the precision of the storage type. In particular, casts and assignments must perform their specified conversion: for the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **d1** and **d2** must have been converted to **float**.

Add an example after current example 4:

Rearrangement for floating-point expressions is restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative laws for addition or multiplication, nor the distributive law, because of roundoff error, even in the absence of overflow and underflow. Likewise, the implementation cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical laws are not valid. See Annex X.8.

```
double x, y, z;
/*...*/
x = (x * y) * z;        /* not equivalent to x *= y * z; */
z = (x - y) + y;        /* not equivalent to z = x; */
z = x + x * y;          /* not equivalent to z = x * (1.0 + y); */
y = x / 5.0;            /* not equivalent to y = x * 0.2; */
```

Feature: Optional IEC 559 support

6.8.8 Predefined macro names

After the first paragraph insert the paragraph:

The following macro name is defined if and only if the implementation conforms to Appendix X:

__STDC_IEC_559__ The decimal constant 1.

Feature: Constant values in <float.h>

5.2.4.2.2 Characteristics of floating types <float.h>

Replace the first sentence of the third paragraph with:

All integral values in the **<float.h>** header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions.

[This change from the earlier C standard treats the integral values in **<float.h>** like those in **<limits.h>**, and makes the floating values available for static and aggregate initialization.]

Mathematics (Enhanced) <math.h>

This is a C9X proposal to merge the FPCE <fp.h> header into <math.h>.

7.x Mathematics <math.h>

The header **<math.h>** declares two types and several mathematical functions and defines several macros. Most synopses specify a function which takes one or two **double** arguments and returns a **double** value; for each of such function, there are functions with the same name but with **f** and **l** suffixes which are corresponding functions with **float** and **long double** arguments and return values.[1] Integer arithmetic functions and conversion functions are discussed later.

> ["Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report called this header **<fp.h>**. For integrating the features into the standard, merging them into **<math.h>** seemed to provide the least troublesome change to the standard.]

The types

```
float_t
double_t
```

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

> [A facility to use wider types is needed for writing portable efficient code. Previously, Standard C gave no way of asking for the most efficient floating type with at least a given width. Efficiency on different floating-point architectures required different prototypes.
>
> | architecture most efficient prototype | |
> |---|---|
> | extended-based | `long double f(long double)` |
> | double-based | `double f(double)` |
> | single/double | `float f(float)` |
> | single/double/extended | `float f(float)` |
>
> Differences may involve whether values can be kept in registers, hence are substantial. Implementations for the various floating-point architectures might use the following type definitions:

[1] Particularly on systems with wide expression evaluation, the **<math.h>** function macros might pass arguments and return values in wider format than the synopsis prototype indicates.

[2] The types **float_t** and **double_t** are intended to be the implementation's most efficient types at least as wide as **float** and **double**, repectively. For **FLT_EVAL_METHOD** equal 0, 1, or 2, the type **float_t** is the narrowest type used by the implementation to evaluate floating expressions.

architecture	**float_t**	**double_t**
extended-based	**long double**	**long double**
double-based	**double**	**double**
single/double	**float**	**double**
single/double/extended	**float**	**double**

An alternate approach would have been to modify the semantics of the **register** storage-class specifier. Applied to a floating type, **register** would have meant the associated value may be wider than the type. This was rejected as inconsistent with existing use of **register** in Standard C.]

The macro

HUGE_VAL

expands to a positive **double** constant expression, not necessarily representable as a **float** . The macros

HUGE_VALF
HUGE_VALL

are respectively **float** and **long double** analogs of **HUGE_VAL**.[3]

The macro

INFINITY

expands to a constant expression of type **float_t** representing an implementation-defined positive or unsigned infinity, if available, else to a positive constant of type **float_t** that overflows at translation time.

The macro

NAN

is defined if and only if the implementation supports quiet NaNs for the **float_t** type. It expands to a constant expression of type **float_t** representing an implementation-defined quiet NaN.

[The **HUGE_VAL**, **INFINITY**, and **NAN** macros are required to be constant expressions in order that they can be used for static and aggregate initialization.]

The macros

FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO

are for number classification. They represent the mutually exclusive kinds of floating-point values. They expand to integral constant expressions with distinct values.

[3] **HUGE_VAL**, **HUGE_VALF**, and **HUGE_VALL** can be positive infinities in an implementation that supports infinities.

[Some prior art uses a finer classification: **FP_POS_INFINITE**, **FP_NEG_INFINITE**, etc. The consensus was that those specified, in conjunction with the **signbit** macro, are generally preferable.]

The macro

DECIMAL_DIG

expands to an integral constant expression whose value is implementation-defined. It represents a number of decimal digits supported by conversion between decimal and all internal floating-point formats.[4]

[**DECIMAL_DIG** is distinct from **DBL_DIG**, which is defined in terms of conversion from decimal to **double** and back.

DECIMAL_DIG was deemed more useful than **FP_CONV_DIG**, which previous versions of this specification defined as the number of decimal digits for which the implementation guaranteed correctly rounded conversion.]

Recommended practice

Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back is the identity function.[5]

7.x.1 The FP_CONTRACT macros

Synopsis

```
#include <math.h>
FP_CONTRACT_ON;
FP_CONTRACT_OFF;
FP_CONTRACT_DEFAULT;
```

Description

The **FP_CONTRACT** macros can be used to allow (if the state is *on*) or disallow (if the state is *off*) the implementation to contract expressions (6.3). Each macro can occur either outside external declarations or preceding all declarations and statements inside a compound statement. When outside external declarations, the macro takes effect from its occurrence until another **FP_CONTRACT** macro is encountered, or until the end of the translation unit. When inside a compound statement, the macro takes effect from its occurrence until another **FP_CONTRACT** macro is encountered (within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the macros is restored to its condition just before the compound statement. The effect of one of these macros in any other context is undefined. The default state (*on* or *off*) for the macros is implementation-defined.

---

[4] **DECIMAL_DIG** is intended to give an appropriate number of digits to carry in canonical decimal representations.

[5] In order that correctly rounded conversion from an internal floating-point format with precision m to decimal with **DECIMAL_DIG** digits and back be the identity function, **DECIMAL_DIG** should be a positive integer n satisfying the inequality:

$n \geq m$	if **FLT_RADIX** is 10
$10^{n-1} >$ **FLT_RADIX**^{m}	otherwise

[Previous versions of this specifications used pragmas instead of macros for this mechanism. Macros were preferred because of general limitations with pragmas and because of the wish not to require standard pragmas.

The ability to apply the macros to compound statements was added to avoid unnecessary restrictions on optimization and to support locality in source code.]

7.x.2 Classification macros

In the synopses in this subclause, *floating-type* indicates a parameter of the same floating type as the argument. The result is undefined if an argument is not of floating type.

[Requiring the arguments to be of floating type allows efficient implementation.]

7.x.2.1 The `fpclassify` macro

Synopsis

```
#include <math.h>
int fpclassify(floating-type x);
```

Description

The **`fpclassify`** macro classifies its argument value as NaN, infinite, normal, subnormal, or zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[6]

Return

The **`fpclassify`** macro returns the value of the number classification macro appropriate to the value of its argument.

[**`fpclassify`** might be implemented as

```
#define fpclassify(x) ((sizeof(x) == sizeof(float)) ? __fpclassifyf(x) \
    : (sizeof(x) == sizeof(double)) ? __fpclassifyd(x) \
    : __fpclassifyl(x))
```

]

7.x.2.2 The `signbit` macro

Synopsis

```
#include <math.h>
int signbit(floating-type x);
```

[6] Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal **`long double`** value might become subnormal when converted to **`double`**, and zero when converted to **`float`**.

Description

The **signbit** macro determines whether the sign of its argument value is negative.[7]

Return

The **signbit** macro returns a nonzero value if and only if the sign of its argument value is negative.

7.x.2.3 The `isfinite` macro

Synopsis

```
#include <math.h>
int isfinite(floating-type x);
```

Description

The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). 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.

Return

The **isfinite** macro returns a nonzero value if and only if its argument has a finite value.

7.x.2.4 The `isnormal` macro

Synopsis

```
#include <math.h>
int isnormal(floating-type x);
```

Description

The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). 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.

Return

The **isnormal** macro returns a nonzero value if and only if its argument has a normal value.

[7] The **signbit** macro faithfully reports the sign of all values, including infinities, zeros, and NaNs.

7.x.2.5 The `isnan` macro

Synopsis

```
#include <math.h>
int isnan(floating-type x);
```

Description

The **`isnan`** macro determines whether its argument value is a NaN. 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.[8]

Return

The **`isnan`** macro returns a nonzero value if and only if its argument has a NaN value.

7.x.3 Treatment of error conditions

The behavior of each of the functions in **`<math.h>`** is defined for all representable values of its input arguments.

For all functions, a *domain error* occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[9] On a domain error, the function returns an implementation-defined value; whether the integer expression **`errno`** acquires the value **`EDOM`** is implementation-defined.

Similarly, a *range error* occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude. The result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result overflows and default rounding is in effect, or if the mathematical result is an exact infinity (for example **`log(0.0)`**), then the function returns the value of the macro **`HUGE_VAL`**, **`HUGE_VALF`**, or **`HUGE_VALL`** appropriate to the specified result type, with the same sign as the correct value of the function; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined. The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.[10] If the result underflows, the function returns a value whose magnitude is no greater than the smallest normalized positive number in the specified type and is otherwise implementation-defined; whether **`errno`** acquires the value **`ERANGE`** is implementation-defined.

---

[8] The type for determination doesn't matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

[9] In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

[10] The term underflow here is intended to encompass both *gradual underflow* as in IEC 559 and also *flush-to-zero* underflow.

[The setting of **errno** is no longer required because doing so is prohibitively expensive on many systems and because many implementations, including all IEC 559 ones, have more useful means of detecting domain and range errors.]

7.x.4 Trigonometric functions

No changes here (except for renumbering).

7.x.5 Hyperbolic functions

Merge in these subclauses:

7.x.5.1 The `acosh` function

Synopsis

```
#include <math.h>
double acosh(double x);
```

Description

The **acosh** function computes the (nonnegative) arc hyperbolic cosine of **x**. A domain error occurs for arguments less than 1. A range error occurs if **x** is too large.

Returns

The **acosh** function returns the arc hyperbolic cosine in the range [0, +∞].

7.x.5.2 The `asinh` function

Synopsis

```
#include <math.h>
double asinh(double x);
```

Description

The **asinh** function computes the arc hyperbolic sine of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

The **asinh** function returns the arc hyperbolic sine.

7.x.5.3 The `atanh` function

Synopsis

```
#include <math.h>
double atanh(double x);
```

Description

The **atanh** function computes the arc hyperbolic tangent of **x**. A domain error occurs for arguments not in the range [-1, +1].

Returns

The **atanh** function returns the arc hyperbolic tangent.

7.x.6 Exponential and logarithmic functions

Merge in these subclauses:

7.x.6.2 The exp2 function

Synopsis

```
#include <math.h>
double exp2(double x);
```

Description

The **exp2** function computes the base-2 exponential of **x**: 2^{x}. A range error occurs if the magnitude of **x** is too large.

Returns

The **exp2** function returns the base-2 exponential.

7.x.6.3 The expm1 function

Synopsis

```
#include <math.h>
double expm1(double x);
```

Description

The **expm1** function computes the base-e exponential of the argument, minus 1: $e^{x} - 1$. For small magnitude **x**, **expm1(x)** is expected to be more accurate than **exp(x) - 1**. A range error occurs if the **x** is too large.

Returns

The **expm1** function returns $e^{x} - 1$.

7.x.6.8 The log1p function

Synopsis

```
#include <math.h>
double log1p(double x);
```

Description

The **log1p** function computes the base-e logarithm of 1 plus the argument. For small magnitude **x**, **log1p(x)** is expected to be more accurate than **log(1 + x)**. A domain error occurs if the argument is less than -1. A range error may occur if the argument equals -1.

Returns

The **log1p** function returns the base-e logarithm of 1 plus the argument.

7.x.6.9 The log2 function

Synopsis

```
#include <math.h>
double log2(double x);
```

Description

The **log2** function computes the base-2 logarithm of **x**. A domain error occurs if the argument is less than zero. A range error may occur if the argument is zero.

Returns

The **log2** function returns the base-2 logarithm.

7.x.6.10 The logb function

Synopsis

```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Description

The **logb** function extracts the exponent of **x**, as a signed integral value in the format of **x**. If **x** is subnormal it is treated as though it were normalized; thus for positive finite **x**,

$$1 \leq \mathbf{x} * \mathbf{FLT_RADIX}^{-\mathbf{logb(x)}} < \mathbf{FLT_RADIX}$$

A range error may occur if the argument is zero.

> [The treatment of subnormal **x** follows the recommendation in IEEE standard 854, which differs from IEC 559 (IEEE standard 754) on this point. Even IEC 559 implementations are intended to follow this definition rather than the one recommended (not required) by IEC 559.
>
> Particularly on machines whose radix is not 2, **logb** can be expected to obtain the exponent more accurately and quickly than **frexp**.]

Returns

The **logb** function returns the signed exponent of its argument.

7.x.6.12 The scalb function

Synopsis

```
#include <math.h>
double scalb(double x, long int n);
```

Description

The **scalb** function computes **x * FLT_RADIX**$^{\mathbf{n}}$ efficiently, not normally by computing **FLT_RADIX**$^{\mathbf{n}}$ explicitly. A range error may occur.

Returns

The **scalb** function returns **x * FLT_RADIX**$^{\mathbf{n}}$.

[On machines whose radix is not 2, **scalb**, compared with **ldexp**, can be expected to have better accuracy, speed, and overflow and underflow behavior.

The second parameter has type **long int**, unlike the corresponding **int** parameter for **ldexp**, because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format allowed by Standard C.]

7.x.7 Power and absolute value functions

Rename the "Power functions" subclause as shown above. Move fabs into this subclause. In the first sentence of the description of pow, replace "is not an integral value" with "is finite and not an integral value". Merge in the following subclause:

7.x.7.2 The hypot function

Synopsis

```
#include <math.h>
double hypot(double x, double y);
```

Description

The **hypot** function computes the square root of the sum of the squares of **x** and **y**, without undue overflow or underflow. A range error may occur.

Returns

The **hypot** function returns the square root of the sum of the squares of **x** and **y**.

Add the following subclause:

7.x.8 Error and gamma functions

[See [23] regarding implementation.]

7.x.8.1 The `erf` function

Synopsis

```
#include <math.h>
double erf(double x);
```

Description

The **`erf`** function computes the error function of **`x`**.

Returns

The **`erf`** function returns the error function of **`x`**.

7.x.8.2 The `erfc` function

Synopsis

```
#include <math.h>
double erfc(double x);
```

Description

The **`erfc`** function computes the complementary error function of **`x`**.

Returns

The **`erfc`** function returns the complementary error function of **`x`**.

7.x.8.3 The `gamma` function

Synopsis

```
#include <math.h>
double gamma(double x);
```

Description

The **`gamma`** function computes the gamma function of **`x`**: $\Gamma(\mathbf{x})$. A domain error occurs if **`x`** is a negative integer or zero. A range error may occur.

Returns

The **`gamma`** function returns $\Gamma(\mathbf{x})$.

[In UNIX System V [10], both the **`gamma`** and **`lgamma`** functions compute $\log_e(|\Gamma(\mathbf{x})|)$.]

7.x.8.4 The `lgamma` function

Synopsis

```
#include <math.h>
double lgamma(double x);
```

Description

The **lgamma** function computes the logarithm of the absolute value of gamma of **x**: $\log_e(|\Gamma(\mathbf{x})|)$. A range error occurs if **x** is too large.

> [In UNIX System V [10], a call to **lgamma** sets an external variable **signgam** to the sign of **gamma(x)**, which is -1 if
>
> ```
> x < 0 && remainder(floor(x), 2) != 0
> ```
>
> Note that this specification does not remove the external identifier **signgam** from the user's name space. An implementation that supports, as an extension, **lgamma**'s setting of **signgam** must still protect the external identifier **signgam** if defined by the user.]

Returns

The **lgamma** function returns $\log_e(|\Gamma(\mathbf{x})|)$.

Split up the subclause "Nearest integer, absolute value, and remainder functions" into two subclauses, "Nearest integer functions" which will include ceil and floor and "Remainder functions" which will include fmod. Merge in the subclauses below.

7.x.9 Nearest integer functions

7.x.9.3 The `nearbyint` function

Synopsis

```
#include <math.h>
double nearbyint(double x);
```

Description

The **nearbyint** function differs from the **rint** function (7.x.9.4) only in that the **nearbyint** function does not raise the inexact exception. (See X.10.6.3-4.)

> [For implementations that do not support the inexact exception, **nearbyint** and **rint** are equivalent.]

Returns

The **nearbyint** function returns the rounded integral value.

7.x.9.4 The `rint` function

Synopsis

```
#include <math.h>
double rint(double x);
```

Description

The **`rint`** function rounds its argument to an integral value in floating-point format, using the current rounding direction.

Returns

The **`rint`** function returns the rounded integral value.

7.x.9.5 The `rinttol` function

Synopsis

```
#include <math.h>
long int rinttol(long double x);
```

Description

The **`rinttol`** function rounds its argument to the nearest integral value, rounding according to the current rounding direction. If the rounded value is outside the range of **`long int`**, the numeric result is unspecified.

Returns

The **`rinttol`** function returns the rounded **`long int`** value, using the current rounding direction.

7.x.9.6 The `round` function

Synopsis

```
#include <math.h>
double round(double x);
```

Description

The **`round`** function rounds its argument to the nearest integral value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

[This function rounds like the Fortran **`anint`** function.]

Returns

The **`round`** function returns the rounded integral value.

7.x.9.7 The `roundtol` function

Synopsis

```
#include <math.h>
long int roundtol(long double x);
```

Description

The **`roundtol`** function rounds its argument to the nearest integral value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of **`long int`**, the numeric result is unspecified.

[This function rounds like the Fortran **`nint`** function and the Pascal **`round`** function.]

Returns

The **`roundtol`** function returns the rounded **`long int`** value.

7.x.9.8 The `trunc` function

Synopsis

```
#include <math.h>
double trunc(double x);
```

Description

The **`trunc`** function rounds its argument to the integral value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

The **`trunc`** function returns the truncated integral value.

7.x.10 Remainder functions

7.x.10.2 The `remainder` function

Synopsis

```
#include <math.h>
double remainder(double x, double y);
```

Description

The **`remainder`** function computes the remainder **`x`** REM **`y`** required by IEC 559.[11]

[11] "When $y \neq 0$, the remainder $r = x \; REM \; y$ is defined regardless of the rounding mode by the mathematical relation $r = x - y * n$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

Returns

The **remainder** function returns **x** REM **y**.

7.x.10.3 The remquo function

Synopsis

```
#include <math.h>
double remquo(double x, double y, int *quo);
```

Description

The **remquo** function computes the same remainder as the **remainder** function. In the object pointed to by **quo** it stores a value whose sign is the sign of **x/y** and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of **x/y**, where **n** is an implementation-defined integer at least 3.

Returns

The **remquo** function returns **x** REM **y**.

> [The **remquo** function is intended for implementing argument reductions, which can exploit a few low-order bits of the quotient. Note that **x** may be so large in magnitude relative to **y** that an exact representation of the quotient is not practical.]

Add the clause:

7.x.11 Manipulation functions

7.x.11.1 The copysign function

Synopsis

```
#include <math.h>
double copysign(double x, double y);
```

Description

The **copysign** function produces a value with the magnitude of **x** and the sign of **y**. It produces a NaN (with the sign of **y**) if **x** is a NaN. On implementations that represent a *signed* zero but do not treat negative zero consistently in arithmetic operations, the **copysign** function regards the sign of zero as positive.

> [The requirement that **copysign** regard a negative sign of zero as positive if the arithmetic treats negative zero like positive zero is justified in order to preserve more identities. For example, to preserve the identity, *the square root of the product is the product of the square roots*, the algorithm in [22] for the complex square root depends on consistency of **copysign** with the rest of the arithmetic: if -0 behaves like +0 then the square root of the product would yield
>
> $\sqrt{3 * (-1 - i0)}$ $= \sqrt{-3 + i0}$ $\rightarrow 0 - i\sqrt{3}$

but if **copysign** were to treat the sign of -0 as negative then the product of the square roots would yield

$$\sqrt{3} \quad * \sqrt{-1 - i0} \quad \text{--> } \sqrt{3} \quad * (0 - i) = 0 - i\sqrt{3}$$

]

Returns

The **copysign** function returns a value with the magnitude of **x** and the sign of **y**.

7.x.11.2 The nan function

Synopsis

```
#include <math.h>
double nan(const char *tagp);
```

Description

If the implementation supports quiet NaNs for the **double** type, then the call **nan("***n-char-sequence***")** is equivalent to **strtod("NAN(***n-char-sequence***)", (char**) NULL)**; the call **nan("")** is equivalent to **strtod("NAN()", (char**) NULL)**. If **tagp** does not point to an *n-char-sequence* string then the result NaN's content is unspecified. If the implementation does not support quiet NaNs for the **double** type, a call to the **nan** function is unspecified.

Returns

The **nan** function returns a quiet NaN, if available, with content indicated through **tagp**.

7.x.11.3 The nextafter functions

Synopsis

```
#include <math.h>
double nextafter(double x, double y);
```

Description

The **nextafter** function determines the next representable value, in the type of the function, after **x** in the direction of **y**, where **x** and **y** are first converted to the type of the function.[12] The **nextafter** function returns **y** if **x == y**.

Returns

The function **nextafter** returns the next representable value in the specified format after **x** in the direction of **y**.

[It's sometimes desirable to find the next representation after a value in the direction of a previously computed value—maybe smaller, maybe larger. The **nextafter** functions have

[12] The argument values are converted to the type of the function, even by a macro implementation of the function.

a second floating argument so that the program will not have to include floating-point tests for determining the direction in such situations. And, on some machines these tests may fail due to overflow, underflow, or roundoff.

For the case **x == y**, IEC 559 recommends that **x** be returned. This specification differs in order that **nextafter(-0.0, +0.0)** return **+0.0** and **nextafter(+0.0, -0.0)** return **-0.0**.]

7.x.12 Maximum, minimum, and positive difference functions

[The functions in this clause correspond to the standard Fortran functions **dim**, **max**, and **min**.

Their names have **f** prefixes to allow for integer versions—following the example of **fabs** and **abs**.]

7.x.12.1 The fdim function

Synopsis

```
#include <math.h>
double fdim(double x, double y);
```

Description

The **fdim** function determines the *positive difference* between its arguments:

x - y	if **x > y**
+0	if **x** ≤ **y**

A range error may occur.

Returns

The **fdim** function returns the positive difference between **x** and **y**.

7.x.12.2 The fmax function

Synopsis

```
#include <math.h>
double fmax(double x, double y);
```

Description

The **fmax** function determines the maximum numeric value of its arguments.[13]

Returns

The **fmax** function returns the maximum numeric value of its arguments.

[13] NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then **fmax** chooses the numeric value. See X.10.9.2.

7.x.12.3 The `fmin` function

Synopsis

```
#include <math.h>
double fmin(double x, double y);
```

Description

The **`fmin`** function determines the minimum numeric value of its arguments.[14]

Returns

The **`fmin`** function returns the minimum numeric value of its arguments.

7.x.13 Comparison macros

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships—*less*, *greater*, and *equal*—is true. Relational operators may raise the invalid exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.[15] The following subclauses provide macros that are *quiet* (non exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the invalid exception.

7.x.13.1 The `isgreater` macro

Synopsis

```
#include <math.h>
int isgreater(floating-expr x, floating-expr y);
```

Description

The **`isgreater`** macro determines whether its first argument is greater than its second argument. The value of **`isgreater(x,y)`** is always equal to **`(x) > (y)`**; however, unlike **`(x) > (y)`**, **`isgreater(x,y)`** does not raise the invalid exception when **`x`** and **`y`** are unordered.

Returns

The **`isgreater`** macro returns the value of **`(x) > (y)`**.

7.x.13.2 The `isgreaterequal` macro

Synopsis

```
#include <math.h>
int isgreaterequal(floating-expr x, floating-expr y);
```

[14] **`fmin`** is analogous to **`fmax`** in its treatment of NaNs.

[15] IEC 559 requires that the built-in relational operators raise the invalid exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs.

Description

The **isgreaterequal** macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal(x,y)** is always equal to **(x) >= (y)**; however, unlike **(x) >= (y)**, **isgreaterequal(x,y)** does not raise the invalid exception when **x** and **y** are unordered.

Returns

The **isgreaterequal** macro returns the value of **(x) >= (y)**.

7.x.13.3 The `isless` macro

Synopsis

```
#include <math.h>
int isless(floating-expr x, floating-expr y);
```

Description

The **isless** macro determines whether its first argument is less than its second argument. The value of **isless(x,y)** is always equal to **(x) < (y)**; however, unlike **(x) < (y)**, **isless(x,y)** does not raise the invalid exception when **x** and **y** are unordered.

Returns

The **isless** macro returns the value of **(x) < (y)**.

7.x.13.4 The `islessequal` macro

Synopsis

```
#include <math.h>
int islessequal(floating-expr x, floating-expr y);
```

Description

The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal(x,y)** is always equal to **(x) <= (y)**; however, unlike **(x) <= (y)**, **islessequal(x,y)** does not raise the invalid exception when **x** and **y** are unordered.

Returns

The **islessequal** macro returns the value of **(x) <= (y)**.

7.x.13.5 The `islessgreater` macro

Synopsis

```
#include <math.h>
int islessgreater(floating-expr x, floating-expr y);
```

Description

The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater(x,y)** macro is similar to **(x) < (y) || (x) > (y)**; however, **islessgreater(x,y)** does not raise the invalid exception when **x** and **y** are unordered (nor does it evaluate **x** and **y** twice).

Returns

The **islessgreater** macro returns the value of **(x) < (y) || (x) > (y)**.

7.x.13.6 The **isunordered** macro

Synopsis

```
#include <math.h>
int isunordered(floating-expr x, floating-expr y);
```

Description

The **isunordered** macro determines whether its arguments are unordered.

Returns

The **isunordered** macro returns 1 if its arguments are unordered and 0 otherwise.

[For implementations with NaNs, the translator should recognize the comparison macro in order to provide efficient implementation. Typical hardware offers efficient quiet comparisons. The semantically correct implementation

```
int _Isless(long double x, long double y)
{
    return ! (isnan(x) || isnan(y) || x >= y);
}
```

is unsuitable for efficiency reasons.

Programs written for (or ported to) systems with NaNs will be expected to handle invalid and NaN input in reasonable ways. The comparison functions support such programs. The arithmetic operators (**+**, *****, …) propagate NaNs quietly; the comparison functions facilitate directing NaNs through branches quietly. Thus NaNs can flow through many computations without the need for inefficient or unduly obfuscating code, and without raising inappropriate exceptions.

For implementations without NaNs, the comparison macros can be defined trivially:

```
#define isgreater(x,y) ((x)>(y))
...
#define islessgreater(x,y) ((x)!=(y))
#define isunordered(x,y) 0
```

Several previous versions of this specification proposed extending the relational operators:

Symbol	Relation
`<`	less
`>`	greater
`<=`	less or equal
`>=`	greater or equal
`==`	equal
`!=`	unordered, less, or greater
`!<>=`	unordered
`<>`	less or greater
`<>=`	less, equal, or greater
`!<=`	unordered or greater
`!<`	unordered, greater, or equal
`!>=`	unordered or less
`!>`	unordered, less, or equal
`!<>`	unordered or equal

The additional operators were to be analogous to, and have the same precedence as, the Standard C relational operators. The `!` symbol was to indicate awareness of NaNs, so operators including the `!` symbol would not raise the invalid exception for unordered operands. Where the operands have types and values suitable for relational operators, the semantics detailed in 6.3.8 were to apply. The *operator* syntax in 6.1.5 was to be augmented to include the additional operators. This approach would have had the advantages of brevity and clearer promise of efficiency, at no greater implementation cost for systems that support NaNs (the large majority of systems do support NaNs). However, it was rejected because of reluctance to extend the language definition for functionality which could be provided with a library interface, and because continuing contentiousness might discourage implementation. Also, in some cases, the functions provide a more straightforward articulation, e.g. `isless(x,y)` instead of `! (x !< y).`

IEC 559 enumerates 26 functionally distinct comparison predicates, from combinations of the four comparison results and whether invalid is raised. The following table shows how the previous and current specifications cover all important cases:

greater	less	equal	raises unordered	previous exception	current old proposal	current specification
		√			`x == y`	`x == y`
√	√		√		`x != y`	`x != y`
√				√	`x > y`	`x > y`
√		√		√	`x >= y`	`x >= y`
	√			√	`x < y`	`x < y`
	√	√		√	`x <= y`	`x <= y`
			√		`x !<>= y`	`isunordered(x,y)`
√	√			√	`x <> y`	`N/A`
√	√	√		√	`x <>= y`	`N/A`
√			√		`x !<= y`	`! islessequal(x,y)`
√		√	√		`x !< y`	`! isless(x,y)`
	√		√		`x !>= y`	`! isgreaterequal(x,y)`
	√	√	√		`x !> y`	`! isgreater(x,y)`
		√	√		`x !<> y`	`! islessgreater(x,y)`
	√	√	√	√	`! (x > y)`	`! (x > y)`
	√		√	√	`! (x >= y)`	`! (x >= y)`
√		√	√	√	`! (x < y)`	`! (x < y)`
√			√	√	`! (x <= y)`	`! (x <= y)`
√	√	√			`! (x !<>= y)`	`! isunordered(x,y)`
		√	√	√	`! (x <> y)`	`N/A`
			√	√	`! (x <>= y)`	`N/A`
	√	√			`! (x !<= y)`	`islessequal(x,y)`
	√				`! (x !< y)`	`isless(x,y)`
√		√			`! (x !>= y)`	`isgreaterequal(x,y)`
√					`! (x !> y)`	`isgreater(x,y)`
√	√				`! (x !<> y)`	`islessgreater(x,y)`

The previous proposal would have naturally covered the four N/A cases not covered by the current proposal. (The current proposal covers them, except for the invalid exception.) However, covering these cases per se is unimportant, because the facility would provide no additional capability except more ways to write NaN-unaware code.

In the interest of efficiency, note that each quiet combination of less, greater, equal, and unordered can be tested with a single comparison function or equality or relational operator.

The proposal for **!** operators supplanted an earlier proposal that would have augmented the set of relations by using the **?** symbol to denote unordered, for example **a ?>= b** instead of **a !< b**. Use of the **?** relationals would have had the advantage that the unordered case would have been dealt with explicitly. However, the **!** relationals seemed a more natural language extension, particularly from the point of view of programmers for (non-IEC 559) implementations not detecting unordered. Also, using **??** as proposed for the unordered operator would have conflicted with trigraphs.

Other macro approaches, such as

```
isrelation(x, FP_UNORDERED | FP_LESS | FP_EQUAL, y)
```

seemed more cumbersome.

Without any language or library support **isgreater(a,b)** might be implemented by the programmer as

```
! (a != a || b != b || a <= b)
```

However, even more awkward code would be required if **a** or **b** had side effects. The programmer would have to remember to put the NaN tests first, and trust the compiler not to replace **a != a || b != b** by *false*. Also, special optimization would be necessary to generate efficient code. Use of **isnan** helps only a little.]

Floating-Point Environment <fenv.h>

This is a proposal for adding the FPCE <fenv.h> header to C9X.

7.x Floating-point environment <fenv.h>

The header **<fenv.h>** declares two types and several macros and functions to provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[16] A *floating-point status flag* is a system variable whose value is set as a side effect of the arithmetic to provide auxiliary information. A *floating-point control mode* is a system variable whose value may be set by the user to affect the subsequent behavior of the arithmetic.

> [The floating-point environment as defined here includes only execution-time modes, not the myriad possible translation-time options that may affect a program's results. Examples of such translation-time options include: chopped or rounded multiplication on CRAY Y-MP systems, D or G format for VAX, and fast or correctly-rounded divide on the Intel 860.]

Certain programming conventions support the intended model of use for the floating-point environment[17]:

— A function call must not alter its caller's modes, clear its caller's flags, nor depend on the state of its caller's flags unless the function is so documented.

— A function call is assumed to require default modes, unless its documentation promises otherwise or unless the function is known not to use floating-point.

— A function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

> [Library implementors are encouraged to document their use, or non-use, of floating-point and their raising of floating-point exceptions.
>
> Names in this header have an **FE_** or **fe** prefix and employ certain abbreviations. The prefix calls attention to environmental access functions, which require an enabling **FENV_ACCESS** macro. The abbreviations **env** and **except** are used in Standard C and UNIX System V, respectively.
>
> The interface described here is not intended to support floating-point trap handlers.]

The type

fenv_t

---

[16] This header is designed to support the exception status flags and directed-rounding control modes required by IEC 559, and other similar floating-point state information. Also it is designed to facilitate code portability among all systems. Its implementation is trivial for systems that do not provide access to floating-point state.

[17] With these conventions, a programmer can safely assume default modes (or be unaware of them). Responsibilities, and (ordinarily modest) overhead, associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

represents the entire floating-point environment.

The type

```
fexcept_t
```

represents the floating-point exception flags collectively, including any status the implementation associates with the flags.

Each macro

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
```

is defined if and only if the implementation supports the exception by means of the functions in 7.x.1. The defined macros expand to integral constant expressions whose values are distinct powers of 2.

> [Macros for unsupported exceptions are left undefined in order to assure their use results in a translation error. A program might explicitly define such macros, to allow translation of code (perhaps never executed) containing the macros. The program could define the unsupported exception macros to be 0, for example
>
> ```
> #ifndef FE_INEXACT
> #define FE_INEXACT 0
> #endif
> ```
>
> so that a bitwise OR of macros has a reasonable effect.]

The macro

```
FE_ALL_EXCEPT
```

is simply the bitwise OR of all exception macros defined by the implementation.

Each macro

```
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the **`fegetround`** and **`fesetround`** functions. The defined macros expand to integral constant expressions whose values are distinct nonnegative values.[18]

[18] Even though the rounding direction macros may expand to constants corresponding to the values of **`FLT_ROUNDS`**, they are not required to do so.

The macro

```
FE_DFL_ENV
```

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

Additional macro definitions, beginning with **FE_** and having type *pointer to const-qualified* **fenv_t**, may also be specified by the implementation.

7.x.1 The FENV_ACCESS macros

Synopsis

```
#include <fenv.h>
FENV_ACCESS_ON;
FENV_ACCESS_OFF;
FENV_ACCESS_DEFAULT;
```

Description

The **FENV_ACCESS** macros provide a means to inform the implementation when a program might access the floating-point environment to test flags or run under non-default modes.[19] Each macro can occur either outside external declarations or preceding all declarations and statements inside a compound statement. When outside external declarations, the macro takes effect from its occurrence until another **FENV_ACCESS** macro is encountered, or until the end of the translation unit. When inside a compound statement, the macro takes effect from its occurrence until another **FENV_ACCESS** macro is encountered (within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the macros is restored to its condition just before the compound statement. The effect of one of these macros in any other context is undefined. If part of a program tests flags or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** macros *off*, then the behavior of that program is undefined. The default state (*on* or *off*) for the macros is implementation-defined.

> [Previous versions of this specifications used pragmas instead of macros for this mechanism. Macros were preferred because of general limitations with pragmas and because of the wish not to require standard pragmas.
>
> The ability to apply the macros to compound statements was added to avoid unnecessary restrictions on optimization and to support locality in source code.]

[19] The purpose of the **FENV_ACCESS** macros is to allow certain optimizations, for example *global common subexpression elimination*, *code motion*, and *constant folding*, that could subvert flag tests and mode changes. In general, if the state of **FENV_ACCESS** is *off* then the translator can assume that default modes are in effect and the flags are not tested.

Example

```
#include <fenv.h>
void f(double x)
{
   FENV_ACCESS_ON;
   void g(double);
   void h(double);
   /*...*/
   g(x + 1);
   h(x + 1);
   /*...*/
}
```

If the function `g` might depend on status flags set as a side effect of the first `x + 1`, or if the second `x + 1` might depend on control modes set as a side effect of the call to function `g`, then the program must contain an appropriately placed invocation of `FENV_ACCESS_ON`. [20]

[The performance of code under the effect of an enabling `FENV_ACCESS` macro may well be important; in fact, an algorithm might access the floating-point environment specifically for performance. The implementation should optimize as aggressively as the `FENV_ACCESS` macros allow. (See §X.9.)

An implementation could simply honor the floating-point environment in all cases and ignore all `FENV_ACCESS` macros.

Dynamic modes are potentially problematic because:

- the programmer may have to defend against undesirable mode settings—which imposes intellectual, time, and space overhead; and
- the translator may not know which mode settings will be in effect or which functions change them at execution time—which inhibits optimization.

This specification attempts to address these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of *static modes*, with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. IEC 559 requires dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, Standard C has no facility, other than macros and pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling `FENV_ACCESS` macro would need such special treatment. To further limit the problem, the implementation might employ additional translation options specifically to indicate where non-default modes would be admissible.]

[20] The side effects impose a temporal ordering that requires two evaluations of `x + 1`. On the other hand, without the `FENV_ACCESS_ON` macro, and assuming the default state is *off*, just one evaluation of `x + 1` would suffice.

7.x.2 Exceptions

The following functions provide access to the exception flags.[21] The **int** input argument for the functions represents a subset of floating-point exceptions, and can be constructed by bitwise ORs of the exception macros, for example **FE_OVERFLOW | FE_INEXACT**. For other argument values the behavior of these functions is undefined.

> [In previous drafts of this specification, several of the exception functions returned an **int** indicating whether the **excepts** argument represented supported exceptions. This facility was deemed unnecessary because **excepts & ~FE_ALL_EXCEPT** can be used to test invalidity of the **excepts** argument.]

7.x.2.1 The `feclearexcept` function

Synopsis

```
#include <fenv.h>
void feclearexcept(int excepts);
```

Description

The **feclearexcept** function clears the supported exceptions represented by its argument.

7.x.2.2 The `fegetexceptflag` function

Synopsis

```
#include <fenv.h>
void fegetexceptflag(fexcept_t *flagp, int excepts);
```

Description

The **fegetexceptflag** function stores an implementation-defined representation of the exception flags indicated by the argument **excepts** in the object pointed to by the argument **flagp**.

> [Previous versions of this specification used the names **fegetexcept** and **fesetexcept** instead of **fegetexceptflag** and **fesetexceptflag**. The names were changed to distinguish these functions—which deal with the entire content of the flags, take an **fexcept_t*** argument, and are used as a pair—from the other exception functions.]

7.x.2.3 The `feraiseexcept` function

Synopsis

```
#include <fenv.h>
void feraiseexcept(int excepts);
```

---

[21] The functions **fetestexcept**, **feraiseexcept**, and **feclearexcept** support the basic abstraction of flags that are either set or clear. An implementation may endow exception flags with more information—for example, the address of the code which first raised the exception; the functions **fegetexceptflag** and **fesetexceptflag** deal with the full content of flags.

Description

The **feraiseexcept** function raises the supported exceptions represented by its argument.[22] The order in which these exceptions are raised is unspecified, except as stated in X.7.6.

> [The function is not restricted to accepting only IEC 559 valid coincident expressions for atomic operations, therefore the function can be used to raise exceptions accrued over several operations.]

7.x.2.4 The **fesetexceptflag** function

Synopsis

```
#include <fenv.h>
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

Description

The **fesetexceptflag** function sets the complete status for those exception flags indicated by the argument **excepts**, according to the representation in the object pointed to by **flagp**. The value of ***flagp** must have been set by a previous call to **fegetexceptflag**; if not, the effect on the indicated exception flags is undefined. This function does not *raise* exceptions, but only sets the state of the flags.

7.x.2.5 The **fetestexcept** function

Synopsis

```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

The **fetestexcept** function determines which of a specified subset of the exception flags are currently set. The **excepts** argument specifies the exception flags to be queried.[23]

> [The argument is a *mask* because querying all flags may be more expensive on some architectures.]

Returns

The **fetestexcept** function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in **excepts**.

Example

Call **f** if invalid is set, **g** if overflow is set:

---

[22] The effect is intended to be similar to that of exceptions raised by arithmetic operations. Hence, enabled traps for exceptions raised by this function are taken. The specification in X.7.6 is in the same spirit.

[23] This mechanism allows testing several exceptions with just one function call.

```
#include <fenv.h>
/*...*/
{
   FENV_ACCESS_ON;
   int set_excepts;
   /* maybe raise exceptions */
   set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
   if (set_excepts & FE_INVALID) f();
   if (set_excepts & FE_OVERFLOW) g();
   /*...*/
}
```

7.x.3 Rounding

The **fegetround** and **fesetround** functions provide control of rounding direction modes.

7.x.3.1 The `fegetround` function

Synopsis

```
#include <fenv.h>
int fegetround(void);
```

Description

The **fegetround** function gets the current rounding direction.

Returns

The **fegetround** function returns the value of the rounding direction macro representing the current rounding direction.

7.x.3.2 The `fesetround` function

Synopsis

```
#include <fenv.h>
int fesetround(int round);
```

Description

The **fesetround** function establishes the rounding direction represented by its argument **round**. If the argument does not match a rounding direction macro, the rounding direction is not changed.

Returns

The **fesetround** function returns a nonzero value if and only if the argument matches a rounding direction macro (that is, if and only if the requested rounding direction can be established).

Example

Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>
/*...*/
{
   FENV_ACCESS_ON;
   int save_round;
   int setround_ok;
   save_round = fegetround();
   setround_ok = fesetround(FE_UPWARD);
   assert(setround_ok);
   /*...*/
   fesetround(save_round);
   /*...*/
}
```

7.x.4 Environment

The functions in this section manage the floating-point environment—status flags and control modes—as one entity.

7.x.4.1 The `fegetenv` function

Synopsis

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

Description

The **`fegetenv`** function stores the current floating-point environment in the object pointed to by **`envp`**.

7.x.4.2 The `feholdexcept` function

Synopsis

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

The **`feholdexcept`** function saves the current environment in the object pointed to by **`envp`**, clears the exception flags, and installs a *non-stop* (continue on exceptions) mode, if available, for all exceptions.[24]

[24] IEC 559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the **`feholdexcept`** function can be used in conjunction with the **`feupdateenv`** function to write routines that hide spurious exceptions from their callers.

Returns

The **feholdexcept** function returns nonzero if and only if non-stop exception handling was successfully installed.

[More appropriate for the user model prescribed in 7.x, **feholdexcept** supersedes **feprocentry** which was equivalent to

```
fegetenv(envp);
fesetenv(FE_DFL_ENV);
```

]

7.x.4.3 The fesetenv function

Synopsis

```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

Description

The **fesetenv** function establishes the floating-point environment represented by the object pointed to by **envp**. The argument **envp** must point to an object set by a call to **fegetenv** or **feholdexcept**, or equal the macro **FE_DFL_ENV** or an implementation-defined environment macro. Note that **fesetenv** merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

7.x.4.4 The feupdateenv function

Synopsis

```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

Description

The **feupdateenv** function saves the current exceptions in its automatic storage, installs the environment represented through **envp**, and then raises the saved exceptions. The argument **envp** must point to an object set by a call to **feholdexcept** or **fegetenv**, or equal the macro **FE_DFL_ENV** or an implementation-defined environment macro.

[**feupdateenv** was called **feprocexit** in earlier drafts of this specification.]

Example

Hide spurious underflow exceptions:

```
#include <fenv.h>
double f(double x)
{
   FENV_ACCESS_ON;
   double result;
   fenv_t save_env;
   feholdexcept(&save_env);
   /* compute result */
   if (/* test spurious underflow */) feclearexcept(FE_UNDERFLOW);
   feupdateenv(&save_env);
   return result;
}
```

Annex X: IEC 559 Floating-Point Arithmetic

This is a proposal for a C9X annex to include the FPCE specification for IEC 559 implementations.

Annex X
(normative)
IEC 559 floating-point arithmetic

X.1 Introduction

This annex specifies C language support for the IEC 559 floating-point standard. The *IEC 559 floating-point standard* is specifically *Binary floating-point arithmetic for microprocessor systems* (IEC 559:1989), also known 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 559* generally refers to the floating-point standard, as in IEC 559 operation, IEC 559 format, etc. An implementation that defines **__STDC_IEC_559__** conforms to the specification in this annex.

X.2 Types

The C floating types match the IEC 559 formats as follows:

— The **float** type matches the IEC 559 single format.

— The **double** type matches the IEC 559 double format.

— The **long double** type matches an IEC 559 extended format[25], else a non-IEC 559 extended format, else the IEC 559 double format.

Any non-IEC 559 extended format used for the **long double** type has more precision than IEC 559 double and at least the range of IEC 559 double.[26]

Recommended practice

The **long double** type matches an IEC 559 extended format.

> [The primary objective of this specification is to facilitate writing portable code that exploits the IEC 559, including its standardized single and double data formats. Bringing the C data types and the IEC 559 formats into line advances this objective.
>
> Minimal conformance to the IEC 559 does not require a format wider than single. Standard C (5.2.4.2) requires its **double** type to be wider than IEC 559 single, and wider than the minimum IEC 559 *single-extended* format. (IEC 559 single-extended is an

[25] *Extended* is IEC 559's *double-extended* data format. Extended refers to both the common 80-bit and *quadruple* 128-bit IEC 559 formats.

[26] A non-IEC 559 **long double** type must provide infinities and NaNs, as its values must include all **double** values.

optional format intended only for those implementations that don't support double; it has at least 32 bits of precision.) Both Standard C and IEC 559 would be satisfied if **float** were IEC 559 single and **double** were an IEC 559 single-extended format with at least 35 bits of precision. However, this specification goes slightly further by requiring **double** to be IEC 559 double rather than just a wide IEC 559 single-extended.

Because of Standard C's bias toward **double**, extended-based architectures might appear to be better served by associating the C **double** type with IEC 559 extended. However, such an approach would not allow standard C types for both IEC 559 double and single and would go against current industry naming, in addition to undermining this specification's portability goal. The types **float_t** and **double_t** in **<math.h>**, are intended to allow effective use of architectures with more efficient, wider formats.

The **long double** type is not required to be IEC 559 extended because

1. some of the major IEC 559 architectures for C implementations do not support extended,

2. double precision is adequate for a broad assortment of numerical applications, and

3. extended is less standard than single or double in that only bounds for its range and precision are specified in IEC 559.

For certain implementations, non-IEC 559 extended arithmetic written in software to exploit special hardware features can provide useful extra precision but with significantly better performance than could IEC 559 extended in software. [???] explains advantages of extended precision.

This specification accommodates what are expected to be the most important IEC 559 architectures for general C implementations—see Rationale 5.2.4.2.2.

Specification for a variable-length extended type—one whose width could be changed by the user—was deemed premature. However, not unduly encumbering experimentation and future extensions, for example for variable length extended, is a goal of this specification.

Some narrow-format C implementations, namely ones for digital signal processing, provide only the IEC 559 single format, possibly augmented by single-extended, which may be narrower than IEC 559 double or Standard C **double**, and possibly further augmented by double in software. These non-conforming implementations might generally adopt this specification, though not matching its requirements for types.

One approach would be: match Standard C **float** with single; match Standard C **double** with single-extended, else single; and match Standard C **long double** with double, else single-extended, else single. Then most of this specification could be applied straightforwardly. Users should be clearly warned that the types may not meet expectations.

Another approach would be to refer to a single-extended format as **long float** and then not recognize any C types not truly supported. This would provide ample warning for programs requiring double. The translation part of porting programs could be accomplished easily with the help of type definitions. In the absence of a double type, most of this specification for **double** could be adopted for the **long float** type. Having distinct types for **long float** and **double**, previously synonyms, requires more imagination.]

X.2.1 Infinities, signed zeros, and NaNs

IEC 559 formats provide representations for signed infinities, signed zeros, and quiet and signaling NaNs. This specification adopts by reference the IEC 559-specified

behavior for infinities, signed zeros, and quiet NaNs. This specification does not define the behavior of signaling NaNs.[27] It generally uses the term *NaN* to denote quiet NaNs.

The **NAN** and **INFINITY** macros and the **nan** function in **<math.h>** provide designations for NaNs and infinities.

> [Signaling NaNs are not created by any standard operations or functions, but can be created as bit patterns. Operations that trigger a signaling NaN argument generally return a quiet NaN result provided no trap is taken; neither traps nor any other facility for signaling NaNs is required by IEC 559. True support for signaling NaNs implies restartable traps, such as the optional traps specified in IEC 559.
>
> This specification fully supports the primary utility of quiet NaNs—"to handle otherwise intractable situations, such as providing a default value for 0.0/0.0" [11] and thereby provide closure for the arithmetic.
>
> Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information, for example about the NaN's origin [4]. Signaling NaNs are good candidates for filling uninitialized storage; and their available parts could distinguish uninitialized floating objects. IEC 559 signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetics.
>
> However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. IEC 559 requires that NaNs propagate, but not all implementations faithfully propagate the entire contents. And even IEC 559 fails to specify the contents of NaNs through format conversion, which is pervasive in some C implementation mechanisms. For these reasons this specification does not specify the behavior of signaling NaNs nor the interpretation of NaN significands.
>
> An early draft of the Floating-Point C Extensions part of the X3J11 Numerical C Extensions Technical Report contains specification for signaling NaNs, which could serve as a guide for extensions in support of signaling NaNs.]

X.3 Operators and functions

C operators and functions provide IEC 559 required and recommended facilities as listed below. Except where noted, this specification adopts by reference their IEC 559-specified behavior.

— The **+**, **-**, *****, and **/** operators provide the IEC 559 add, subtract, multiply, and divide operations.

— The **sqrt** function in **<math.h>** provides the IEC 559 square root operation.

— The **rem** function in **<math.h>** provides the IEC 559 remainder operation. The **remquo** function in **<math.h>** provides the same operation but with additional information.

— The **rint** function in **<math.h>** provides the IEC 559 operation that rounds a floating-point number to an integral value (in the same precision). The C **nearbyint** function in **<math.h>** provides the similar nearbyinteger function recommended in the Appendix to IEEE standard 854.

[27] Since NaNs created by IEC 559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

— The conversions for floating types provide the IEC 559 conversions between floating-point precisions.

— The conversions from integral to floating types provide the IEC 559 conversions from integer to floating point.

— The conversions from floating to integral types provide IEC 559-like conversions but always round toward zero.

— The **rinttol** function in **<math.h>** provides the IEC 559 conversions, which honor the directed rounding mode, from floating point to the **long** integer format. The **rinttol** function can be used in conjunction with casts to provide IEC 559 conversions from floating to other integer formats.

— The translation time conversion of floating constants and the **strtod**, **fprintf**, and related library functions in **<stdlib.h>** and **<stdio.h>** provide IEC 559 binary-decimal conversions. The **strtold** function in **<stdlib.h>** provides the conv function recommended in the Appendix to IEEE standard 854.

— The relational and equality operators provide IEC 559 comparisons. IEC 559 identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison functions (**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 559.

— The **feclearexcept**, **feraiseexcept**, and **fetestexcept** functions in **<fenv.h>** provide the facility to test and alter the IEC 559 floating-point exception flags. The **fegetexceptflag** and **fesetexceptflag** functions in **<fenv.h>** provide the facility to save and restore all five status flags at one time. These functions are used in conjunction with the type **fexcept_t** and the 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 559 directed rounding modes represented by the rounding direction macros (**FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, **FE_TOWARDZERO**) also in **<fenv.h>**.

— The **fegetenv**, **feholdexcept**, **fesetenv**, and **feupdateenv** functions in **<fenv.h>** provide a facility to manage the floating-point environment, comprising the IEC 559 status flags and control modes.

— The **copysign** function in **<math.h>** provides the copysign function recommended in the Appendix to IEC 559.

— The unary minus (**-**) operator provides the minus (-) operation recommended in the Appendix to IEC 559.

— The **scalb** function in **<math.h>** provides the scalb function recommended in the Appendix to IEC 559.

— The **logb** function in **<math.h>** provides the logb function recommended in the Appendix to IEC 559, but following the newer specification in IEEE 854.

— The **nextafterf**, **nextafterd**, and **nextafterl** functions in **<math.h>** provide the nextafter function recommended in the Appendix to IEC 559 (but with a minor change to better handle signed zeros). The **nextafter** function in **<math.h>** provides the same function but in a manner reflecting the implementation's evaluation method.

— The **isfinite** macro in **<math.h>** provides the finite function recommended in the Appendix to IEC 559.

— The **isnan** macro in **<math.h>** provides the isnan function recommended in the Appendix to IEC 559.

— The **signbit** macro and the **fpclassify** macro, used in conjunction with the number classification macros (**FP_NAN**, **FP_INFINITE**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO**), in **<math.h>**provide the facility of the class function recommended in the Appendix to IEC 559 (except that **fpclassify** does not distinguish signaling from quiet NaNs).

X.4 Floating to integral conversion

If the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the invalid exception is raised and the resulting value is unspecified. Whether conversion of nonintegral floating values whose integral part is within the range of the integer type raises the inexact exception is unspecified.[28]

X.5 Binary-decimal conversion

Conversion from the widest supported IEC 559 format to decimal with **DECIMAL_DIG** digits and back is the identity function.[29]

Conversions involving IEC 559 formats follow the recommended practice in 6.1.3.1, 7.9.6.1, 7.10.1.4, 7.16.2.1, and 7.16.4.1.1. In particular, conversion between any supported IEC 559 format and decimal with **DECIMAL_DIG** or fewer significant digits is correctly rounded.

> [IEC 559 requires perfect rounding for a large though incomplete subset of decimal conversions. This specification goes beyond IEC 559 by requiring perfect rounding for all decimal conversions, involving **DECIMAL_DIG** or fewer decimal digits and a supported IEC 559 format, because practical methods are now available. Although not requiring correct rounding for arbitrarily wide decimal numbers, this specification is sufficient in the sense that it ensures that every internal numeric value in an IEC 559 format can be determined as a decimal constant.]

---

[28] IEEE 854, but not IEC 559 (IEEE 754), directly specifies that floating-to-integral conversions raise the inexact exception for nonintegral in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the inexact exception. See **rint**, **rinttol**, and **nearbyint** in **<math.h>**.

[29] If the minimum-width IEC 559 extended format (64 bits of precision) is supported, **DECIMAL_DIG** must be at least 21. If IEC 559 double (53 bits of precision) is the widest IEC 559 format supported, then **DECIMAL_DIG** must be at least 17. (By contrast, **LDBL_DIG** and **DBL_DIG** are 19 and 15, respectively, for these formats.)

X.6 Contracted expressions

A contracted expression treats infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by IEC 559.

Recommended practice

A contracted expression raises exceptions in a manner generally consistent with the basic arithmetic operations. A contracted expression delivers the same value as its uncontracted counterpart, else is correctly rounded (once).

X.7 Environment

The floating-point environment defined in **`<fenv.h>`** includes the IEC 559 exception status flags and directed-rounding control modes. It includes also IEC 559 dynamic rounding precision and trap enablement modes, if the implementation supports them.[30]

> [The *exception flags*—invalid, overflow, underflow, divide-by-zero, and inexact—can be queried at execution time to determine whether a floating-point exception has occurred since the beginning of execution or since its flag was explicitly cleared. (The flags are *sticky*.)
>
> The *rounding direction modes*—to-nearest, toward-zero, upward (toward +∞), and downward (toward -∞)—can be altered at execution time to control the rounding direction for floating-point operations.
>
> IEC 559 prescribes *rounding precision modes* as a means for a system whose results are always double or extended to shorten the precision of its results, in order to mimic systems that deliver results to single or double precision. An implementation of C can meet this goal in any of the following ways:
>
> 1. By supporting the method of evaluating expressions to their semantic type.
> 2. By providing translation options to shorten results by rounding to IEC 559 single or double precision.
> 3. By providing functions to set and get dynamic rounding-precision modes which shorten results by rounding to IEC 559 single or double precision. Functions **`fesetprec`** and **`fegetprec`** and macros **`FE_FLTPREC`**, **`FE_DBLPREC`**, and **`FE_LDBLPREC`**, analogous to the functions and macros for the rounding direction modes, would serve the purpose.
>
> This specification does not include a portable interface for precision control because IEC 559 is ambivalent on whether it intends for precision control to be dynamic (like the rounding direction modes) or static. Indeed, some floating-point architectures provide control modes, suitable for a dynamic mechanism, and others rely on instructions to deliver single- and double-format results, suitable only for a static mechanism.
>
> The traps recommended by IEC 559 require modes for enabling and disabling.]

[30] This specification does not require dynamic rounding precision nor trap enablement modes.

X.7.1 Environment management

IEC 559 requires that floating-point operations implicitly raise 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** macros (defined in **<fenv.h>**) is *on*, these changes to the floating-point state are treated as side effects which respect sequence points.[31]

X.7.2 Translation

During translation the IEC 559 default modes are in effect:

— The rounding direction mode is rounding to nearest.

— The rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all exceptions.

Recommended practice

The implementation provides a non-fatal diagnostic for each translation-time floating-point exception, other than inexact.[32]

> [An implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program. The language and library provide facilities to cause floating-point operations to be done at execution time, when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.]

X.7.3 Execution

At program startup the floating-point environment is initialized as prescribed by IEC 559:

— All exception status flags are clear.

— The rounding direction mode is rounding to nearest.

— The dynamic rounding precision mode (if supported) is set so that results are not shortened.

— Trapping or stopping (if supported) is disabled on all exceptions.

---

[31] If the state for the **FENV_ACCESS** macros is *off*, the implementation is free to assume the modes will be the default ones and the flags will not be tested, which allows certain optimizations—see X.8.

[32] 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 exceptions (even when the state of **FENV_ACCESS** in *on*). Library functions, for example **strtod**, provide execution-time conversion of numeric strings.

X.7.4 Constant expressions

An arithmetic constant expression of floating type, other than one in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type, is evaluated (as if) during execution. As execution-time evaluation, it is affected by any operative modes and raises exceptions as required by IEC 559 (provided the state for **`FENV_ACCESS`** is *on*).[33]

[33] Where the state for **`FENV_ACCESS`** is *on*, results of inexact expressions like `1.0/3.0` are affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` generate execution-time exceptions. The programmer can achieve the efficiency of translation-time evaluation through static initialization, such as

```
const static double one_third = 1.0/3.0;
```

Example

```
#include <fenv.h>
FENV_ACCESS_ON;
void f(void)
{
      float w[] = { 0.0/0.0 };      /* does not raise an exception */
      static float x = 0.0/0.0;     /* does not raise an exception */
      float y = 0.0/0.0;            /* raises an exception */
      double z = 0.0/0.0;           /* raises an exception */
      /*...*/
}
```

For the aggregate and static initializations, the division is done at translation time, raising no (execution-time) exceptions. On the other hand, for the two automatic scalar initializations the invalid division occurs at execution time.

> [A previous approach allowed translation-time constant arithmetic, but empowered the unary **+** operator, when applied to an operand, to inhibit translation-time evaluation of constant expressions.]

X.7.5 Initialization

All computation for automatic scalar initialization is done (as if) at execution time. As execution-time evaluation, it is affected by any operative modes and raises exceptions as required by IEC 559 (provided the state for **fenv_access** is *on*). All computation for initialization of objects that have static storage duration or that have aggregate or union type is done (as if) at translation time.

Example

```
#include <fenv.h>
FENV_ACCESS_ON;
void f(void)
{
      float u[] = { 1.1e75 };        /* does not raise exceptions */
      static float v = 1.1e75;       /* does not raise exceptions */
      float w = 1.1e75;              /* raises exceptions */
      double x = 1.1e75;             /* may raise exceptions */
      float y = 1.1e75f;             /* may raise exceptions */
      long double z = 1.1e75;        /* does not raise exceptions */
      /*...*/
}
```

The aggregate and static initializations of **u** and **v** raise no (execution-time) exceptions because their computation is done at translation time. The automatic initialization of **w** requires an execution-time conversion to **float** of the wider value **1.1e75**, which raises exceptions. The automatic initializations of **x** and **y** entail execution-time conversion; however, in some expression evaluation methods, the conversions is not to a narrower format, in which case no exception is raised.[34] The automatic initialization of **z** entails

[34] Use of **float_t** and **double_t** variables increases the likelihood of translation-time computation. For example, the automatic initialization

```
double_t x = 1.1e75;
```

execution-time conversion, but not to a narrower format, so no exception is raised. Note that the conversions of the floating constants **`1.1e75`** and **`1.1e75f`** to their internal representations occur at translation time in all cases.

[Note that automatic aggregate or union initialization is not equivalent to assignment, which always has execution-time semantics.

This specification does not suit C++, whose static and aggregate initializers need not be constant. Specifying translation-time semantics for initialization of non-local statics with constant-expression initializers, and execution-time semantics for all other floating-point arithmetic, would be consistent with C++ and, given the **`FENV_ACCESS`** mechanism, still would allow the bulk of constant arithmetic to be done, in actuality, at translation time.]

X.7.6 Changing the environment

Operations defined in 6.3 and functions and macros defined for the standard libraries change flags and modes just as indicated by their specification (including conformance to IEC 559). They do not change flags or modes (so as to be detectable by the user) in any other cases.

If the argument to the **`feraiseexcept`** function in **`<fenv.h>`** represents IEC 559 valid coincident exceptions for atomic operations—namely overflow and inexact, or underflow and inexact—then overflow or underflow is raised before inexact.

[IEC 559 operations order exceptions this way.]

X.8 Optimization

This section identifies code transformations that might subvert IEC 559-specified behavior, and others that do not.

X.8.1 "Global" transformations

Floating-point arithmetic operations and external function calls may entail side effects which optimization must honor, at least where the state for **`FENV_ACCESS`** is *on*. The flags and modes in the floating-point environment may be regarded as global variables; floating-point operations (**`+`**, **`*`**, etc.) implicitly read the modes and write the flags.

Concern about side effects may inhibit code motion and removal of seemingly useless code. For example, in

```
#include <fenv.h>
FENV_ACCESS_ON;
void f(double x)
{
   /*...*/
   for (i = 0; i < n; i++) x + 1;
   /*...*/
}
```

---

could be done at translation time, regardless of the expression evaluation method.

`x + 1` might raise exceptions, so cannot be removed. And since the loop body might not execute (maybe $0 \geq$ `n`), `x + 1` cannot be moved out of the loop. (Of course these optimizations are valid if the implementation can rule out the nettlesome cases.)

This specification does not require support for trap handlers that maintain information about the order or count of exceptions. Therefore, between function calls exceptions need not be precise: the actual order and number of occurrences of exceptions (> 1) may vary from what the source code expresses. Thus the preceding loop could be treated as

```
if (0 < n) x + 1;
```

X.8.2 Expression transformations

`x / 2` <--> `x * 0.5` Although similar transformations involving inexact constants generally do not yield numerically equivalent expressions, if the constants are exact then such transformations can be made on IEC 559 machines and others that round perfectly.

`1 * x`, `x / 1` --> `x` The expression `1 * x` and `x`, and `x / 1` and `x` are equivalent (on IEC 559 machines, among others).[35]

`x - y` <--> `x + (-y)` The expressions `x - y`, `x + (-y)`, and `(-y) + x` are equivalent (on IEC 559 machines, among others).

`x - y` <--> `-(y - x)` The expressions `x - y` and `-(y - x)` are not equivalent because 1 - 1 is +0 but -(1 - 1) is -0 (in the default rounding direction).[36]

`x - x` --> `0.0` The expressions `x - x` and `0.0` are not equivalent if `x` is a NaN or infinite.

`0 * x` --> `0.0` The expressions `0 * x` and `0.0` are not equivalent if `x` is a NaN or infinite.

`x + 0` --> `x` The expressions `x + 0` and `x` are not equivalent if `x` is -0, because (-0) + (+0) yields +0 (in the default rounding direction), not -0.

`x - 0` --> `x` (+0) - (+0) yields -0 when rounding is downward (toward -∞), but +0 otherwise, and (-0) - (+0) always yields -0; so, if the state for `FENV_ACCESS` is *off*, promising default rounding, then the implementation can replace `x - 0` by `x`, even if `x` might be zero.

`-x` <--> `0 - x` The expressions `-x` and `0 - x` are not equivalent if `x` is +0, because -(+0) yields -0, but 0 - (+0) yields +0 (unless rounding is downward).

---

[35] Strict support for signaling NaNs—not required by this specification—would invalidate these and other transformations that remove arithmetic operators.

[36] IEC 559 prescribes a signed zero to preserve mathematical identities across certain discontinuities. Examples include

1 / (1 / (±∞)) is ±∞
complex_conjugate(sqrt(*z*)) is sqrt(complex_conjugate(*z*))

X.8.3 Relational operators

`x != x` --> *false* The statement **`x != x`** is *true* if **`x`** is a NaN.

`x == x` --> *true* The statement **`x == x`** is *false* if **`x`** is a NaN.

`x < y` --> **`isless(x,y)`** (and similarly for **`<=`**, **`>`**, **`>=`**) Though numerically equal, these expressions are not equivalent because of side effects when **`x`** or **`y`** is a NaN and the state for **`FENV_ACCESS`** is *on*. This transformation, which would be desirable if extra code were required to cause the invalid exception for unordered cases, could be performed provided the state for **`FENV_ACCESS`** is *off*.

The sense of relational operators must be maintained. This includes handling unordered cases as expressed by the source code.

Example

```
if (a < b) f(); else g();   /* calls g and raises invalid if a and
                                                    b are unordered */
```

is not equivalent to

```
if (a >= b) g(); else f();  /* calls f and raises invalid if a and
                                                    b are unordered */
```

nor to

```
if (isgreaterequal(a, b)) g(); else f();        /* calls f without
                          raising invalid if a and b are unordered */
```

nor, unless the state of **`FENV_ACCESS`** is *off*, to

```
if (isless(a, b)) f(); else g();       /* calls g without raising
                                    invalid if a and b are unordered */
```

but is equivalent to

```
if (!(a < b)) g(); else f();
```

X.8.4 Constant arithmetic

The implementation must honor exceptions raised by execution-time constant arithmetic wherever the state for **`FENV_ACCESS`** is *on*. (See X.7.4 and X.7.5.) An operation on constants that raises no exception can be folded during translation, except, if the state for **`FENV_ACCESS`** is *on*, a further check is required to assure that changing the rounding direction to downward does not alter the sign of the result,[37] and implementations that support dynamic rounding precision modes must assure further that the result of the operation raises no exception when converted to the semantic type of the operation.

[37] 0 - 0 yields -0 instead of +0 just when the rounding direction is downward.

X.9 <math.h>

This subclause contains specification of **<math.h>** facilities that is particularly suited for IEC 559 implementations.

The Standard C macro **HUGE_VAL** and its **float** and **long double** analogs, **HUGE_VALF** and **HUGE_VALL**, expand to expressions whose values are positive infinities.

> [As standard library macros, **HUGE_VAL**, **INFINITY**, and **NAN** raise no exceptions when evaluated. Hence, for example, **HUGE_VAL** could not be implemented as
>
> ```
> #define HUGE_VAL (1.0/0.0)
> ```
>
> whose use raises the divide-by-zero exception.]

Special cases for functions in **<math.h>** are covered directly or indirectly by IEC 559. X.3 identifies the functions that IEC 559 specifies directly. The other functions in **<math.h>** treat infinities, NaNs, signed zeros, subnormals, and (provided the state for **FENV_ACCESS** is *on*) the exception flags in a manner consistent with the basic arithmetic operations covered by IEC 559.

> [The implementation need not honor the exception flags if the state for **FENV_ACCESS** is *off*. For example, the translator could maintain a variable, say **_Fenv_access**, whose value would be nonzero just when the state for **FENV_ACCESS** is *on*. Then a header could define macros in the style
>
> ```
> #define foo(x) (_Fenv_access?_Foo(x):_FastFoo(x))
> ```
>
>]

The invalid and divide-by-zero exceptions are raised as specified in subsequent subclauses of this annex.

The overflow exception is raised whenever an infinity—or, because of rounding direction, a maximal-magnitude finite number—is returned in lieu of a value whose magnitude is too large.

The underflow exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy.[38]

Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and **lgamma** functions raise the inexact exception is implementation-defined. For other functions, the inexact exception is raised whenever the rounded result is not identical to the mathematical result.

> [The functions exempted from careful handling of the inexact flag are almost always inexact. Presumably there is little value in testing whether they are inexact.]

Whether the inexact exception may be raised when the rounded result actually does equal the mathematical result is implementation-defined. Whether the underflow (and inexact) exception may be raised when a result is tiny but not inexact is implementation-

[38] IEC 559 allows different definitions of underflow. They all result in the same values, but differ on when the exception is raised.

defined.[39] Otherwise, as implied by X.7.6, the **<math.h>** functions do not raise spurious exceptions (detectable by the user).

> [For some functions, for example **pow**, determining exactness in all cases might be too costly.]

Whether the functions honor the rounding direction mode is implementation-defined.

> [Although correct rounding for transcendentals is desirable, costs may be prohibitive at this time.]

Generally, one-parameter functions of a NaN argument return that same NaN and raise no exception.

The specification in the following subclauses appends to the definitions in **<math.h>**.

X.9.1 Trigonometric functions

Type-face used in the following subclauses is misleading in that Courier bold is used for non-program elements. It will be cleaned up.

X.9.1.1 The acos function

- **acos(1)** returns +0.
- **acos(x)** returns a NaN and raises the invalid exception for |**x**| > 1.

X.9.1.2 The asin function

- **asin(±0)** returns ±0.
- **asin(x)** returns a NaN and raises the invalid exception for |**x**| > 1.

X.9.1.3 The atan function

- **atan(±0)** returns ±0.
- **atan(±INFINITY)** returns ±π/2.

X.9.1.4 The atan2 function

- If one argument is a NaN then **atan2** returns that same NaN; if both arguments are NaNs then **atan2** returns one of its arguments.
- **atan2(±0, x)** returns ±0, for **x** > 0.
- **atan2(±0, +0)** returns ±0.[40]
- **atan2(±0, x)** returns ±π, for **x** < 0.
- **atan2(±0, -0)** returns ±π.
- **atan2(y, ±0)** returns π/2 for **y** > 0.
- **atan2(y, ±0)** returns -π/2 for **y** < 0.
- **atan2(±y, INFINITY)** returns ±0, for finite **y** > 0.
- **atan2(±INFINITY, x)** returns ±π/2, for finite **x**.
- **atan2(±y, -INFINITY)** returns ±π, for finite **y** > 0.

---

[39] It is intended that undeserved underflow and inexact exceptions are raised only if determining inexactness would be too costly.

[40] **atan2(0, 0)** does not raise the invalid exception, nor does **atan2(y, 0)** raise the divide-by-zero exception.

- **atan2(±INFINITY, INFINITY)** returns ±π/4.
- **atan2(±INFINITY, -INFINITY)** returns ±3π/4.

[The more contentious cases are **y** and **x** both infinite or both zeros. See [7] for a justification of the choices above.]

X.9.1.5 The cos function

- **cos(±INFINITY)** returns a NaN and raises the invalid exception.

X.9.1.6 The sin function

- **sin(±0)** returns ±0.
- **sin(±INFINITY)** returns a NaN and raises the invalid exception.

X.9.1.7 The tan function

- **tan(±0)** returns ±0.
- **tan(±INFINITY)** returns a NaN and raises the invalid exception.

X.9.2 Hyperbolic functions

X.9.2.1 The acosh function

- **acosh(1)** returns +0.
- **acosh(+INFINITY)** returns **+INFINITY**.
- **acosh(x)** returns a NaN and raises the invalid exception if **x** < 1.

X.9.2.2 The asinh function

- **asinh(±0)** returns ±0.
- **asinh(±INFINITY)** returns ±**INFINITY**.

X.9.2.3 The atanh function

- **atanh(±0)** returns ±0.
- **atanh(±1)** returns ±**INFINITY**.
- **atanh(x)** returns a NaN and raises the invalid exception if |**x**| > 1.

X.9.2.4 The cosh function

- **cosh(±INFINITY)** returns **+INFINITY**.

X.9.2.5 The sinh function

- **sinh(±0)** returns ±0.
- **sinh(±INFINITY)** returns ±**INFINITY**.

X.9.2.6 The tanh function

- **tanh(±0)** returns ±0.
- **tanh(±INFINITY)** returns ±1.

X.9.3 Exponential and logarithmic functions

X.9.3.1 The exp function

- **exp(+INFINITY)** returns **+INFINITY**.
- **exp(−INFINITY)** returns +0.

X.9.3.2 The exp2 function

- **exp2(+INFINITY)** returns **+INFINITY**.
- **exp2(−INFINITY)** returns +0.

X.9.3.3 The expm1 function

- **expm1(±0)** returns ±0.
- **expm1(+INFINITY)** returns **+INFINITY**.
- **expm1(−INFINITY)** returns -1.

X.9.3.4 The frexp function (ISO 7.5.4.2)

- **frexp(±0, exp)** returns ±0, and returns 0 in ***exp**.
- **frexp(±INFINITY, exp)** returns ±**INFINITY**, and returns an unspecified value in ***exp**.
- **frexp(x, exp)** returns **x** if **x** is a NaN, and stores an unspecified value in ***exp**.
- Otherwise, **frexp** raises no exception.

On a binary system, **frexp** is equivalent to the comma expression

```
( (*exp = (value == 0) ? 0 : (int)(1 + logb(value))),
   scalb(value, −(*exp)) )
```

X.9.3.5 The ldexp function

On a binary system, **ldexp(x, exp)** is equivalent to

```
scalb(x, exp)
```

[Note that **ldexp** may not provide the full functionality of **scalb** for extended values, because the power required to scale from the smallest (subnormal) to the largest extended value exceeds the minimum **INT_MAX** allowed by Standard C.]

X.9.3.6 The log function

- **log(±0)** returns **−INFINITY** and raises the divide-by-zero exception.
- **log(x)** returns a NaN and raises the invalid exception if **x** < 0.
- **log(+INFINITY)** returns **+INFINITY**.

X.9.3.7 The log10 function

- **log10(±0)** returns **−INFINITY** and raises the divide-by-zero exception.
- **log10(x)** returns a NaN and raises the invalid exception if **x** < 0.
- **log10(+INFINITY)** returns **+INFINITY**.

X.9.3.8 The log1p function

- **log1p(±0)** returns ±0.
- **log1p(-1)** returns **-INFINITY** and raises the divide-by-zero exception.
- **log1p(x)** returns a NaN and raises the invalid exception if **x** < -1.
- **log1p(+INFINITY)** returns **+INFINITY**.

X.9.3.9 The log2 function

- **log2(±0)** returns **-INFINITY** and raises the divide-by-zero exception.
- **log2(x)** returns a NaN and raises the invalid exception if **x** < 0.
- **log2(+INFINITY)** returns **+INFINITY**.

X.9.3.10 The logb function

- **logb(±INFINITY)** returns **+INFINITY**.
- **logb(±0)** returns **-INFINITY** and raises the divide-by-zero exception.

X.9.3.11 The modf functions

- **modf(value, iptr)** returns a result with the same sign as the argument **value**.
- **modf(±INFINITY, iptr)** returns ±0 and stores ±**INFINITY** through **iptr**.
- **modf** of a NaN argument returns that same NaN and also stores it through **iptr**.

modf behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
FENV_ACCESS_ON;
double modf(double value, double *iptr)
{
   int save_round = fegetround();
   fesetround(FE_TOWARDZERO);
   *iptr = nearbyint(value);
   fesetround(save_round);
   return copysign(
   (fabs(value) == INFINITY) ? 0.0 : value - (*iptr),
       value);
}
```

X.9.3.12 The scalb function

- **scalb(x, n)** returns **x** if **x** is infinite, zero, or a NaN.

X.9.4 Power and absolute value functions

X.9.4.1 The fabs function (ISO 7.5.6.2)

- **fabs(±0)** returns +0.
- **fabs(±INFINITY)** returns **+INFINITY**.

X.9.4.2 The hypot function

- **hypot(x, y)**, **hypot(y, x)**, and **hypot(x, -y)** are equivalent.
- **hypot(x, y)** returns **+INFINITY** if **x** is infinite.
- If both arguments are NaNs then **hypot** returns one of its arguments; otherwise, if **x** is a NaN and **y** in not infinite then **hypot** returns that same NaN.
- **hypot(x, ±0)** is equivalent to **fabs(x)**.

[**hypot(INFINITY, NAN)** returns **+INFINITY**, under the justification that **hypot(INFINITY, y)** is **+INFINITY** for any numeric value **y**.]

X.9.4.3 The pow function

- **pow(x, ±0)** returns 1 for any **x**.
- **pow(x, +INFINITY)** returns **+INFINITY** for |**x**| > 1.
- **pow(x, +INFINITY)** returns +0 for |**x**| < 1.
- **pow(x, -INFINITY)** returns +0 for |**x**| > 1.
- **pow(x, -INFINITY)** returns **+INFINITY** for |**x**| < 1.
- **pow(+INFINITY, y)** returns **+INFINITY** for **y** > 0.
- **pow(+INFINITY, y)** returns +0 for **y** < 0.
- **pow(-INFINITY, y)** returns **-INFINITY** for **y** an odd integer > 0.
- **pow(-INFINITY, y)** returns **+INFINITY** for **y** > 0 and not an odd integer.
- **pow(-INFINITY, y)** returns -0 for **y** an odd integer < 0.
- **pow(-INFINITY, y)** returns +0 for **y** < 0 and not an odd integer.
- **pow(x, y)** returns one of its NaN arguments if **y** is a NaN, or if **x** is a NaN and **y** is nonzero.
- **pow(±1, ±INFINITY)** returns a NaN and raises the invalid exception.
- **pow(x, y)** returns a NaN and raises the invalid exception for finite **x** < 0 and finite nonintegral **y**.
- **pow(±0, y)** returns ±**INFINITY** and raises the divide-by-zero exception for **y** an odd integer < 0.
- **pow(±0, y)** returns **+INFINITY** and raises the divide-by-zero exception for **y** < 0, finite, and not an odd integer.
- **pow(±0, y)** returns ±0 for **y** an odd integer > 0.
- **pow(±0, y)** returns +0 for **y** > 0 and not an odd integer.

[See [7].

pow(NaN, 0). [7] provides extensive justification for the value 1. An opposing point of view is that any return value of a function of a NaN argument should be a NaN, even if the function is independent of that argument—as, in IEC 559, **NAN <= +INFINITY** is false and raises the invalid exception.]

X.9.4.4 The sqrt function

sqrt is fully specified as a basic arithmetic operation in IEC 559.

X.9.5 Error and gamma functions

X.9.5.1 The erf function

- **erf(±0)** returns ±0.
- **erf(±INFINITY)** returns ±1.

X.9.5.2 The erfc function

- **erfc(+INFINITY)** returns +0.
- **erfc(-INFINITY)** returns 2.

X.9.5.3 The gamma function

- **gamma(+INFINITY)** returns **+INFINITY**.
- **gamma(x)** returns a NaN and raises the invalid exception if **x** is a negative integer or zero.
- **gamma(-INFINITY)** returns a NaN and raises the invalid exception.

X.9.5.4 The lgamma function

- **lgamma(+INFINITY)** returns **+INFINITY**.
- **lgamma(x)** returns **+INFINITY** and raises the divide-by-zero exception if **x** is a negative integer or zero.
- **lgamma(-INFINITY)** returns a NaN and raises the invalid exception.

X.9.6 Nearest integer functions

X.9.6.1 The ceil function (ISO 7.5.6.1)

- **ceil(x)** returns **x** if **x** is ±**INFINITY** or ±0.

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

```
#include <math.h>
#include <fenv.h>
FENV_ACCESS_ON;
double ceil(double x)
{
   double result;
   int save_round = fegetround();
   fesetround(FE_UPWARD);
   result = rint(x); /* or nearbyint instead of rint */
   fesetround(save_round);
   return result;
}
```

X.9.6.2 The floor function

- **floor(x)** returns **x** if **x** is ±**INFINITY** or ±0.

See the sample implementation for **ceil** in X.9.6.1.

X.9.6.3 The nearbyint function

The **nearbyint** function differs from the **rint** function only in that the **nearbyint** function does not raise the inexact flag.

X.9.6.4 The rint function

The **rint** function uses IEC 559 rounding according to the current rounding direction. It raises the inexact exception if its argument differs in value from its result.

X.9.6.5 The rinttol function

The **rinttol** function provides floating-to-integer conversion as prescribed by IEC 559. It rounds according to the current rounding direction. If the rounded value is outside the range of **long int**, the numeric result is unspecified and the invalid exception is raised. When it raises no other exception and its argument differs from its result, **rinttol** raises the inexact exception.

X.9.6.6 The round function

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

```
#include <math.h>
#include <fenv.h>
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** function may but is not required to raise the inexact exception for nonintegral numeric arguments, as this implementation does.

X.9.6.7 The roundtol function

The **roundtol** function differs from **rinttol** with the default rounding direction just in that **roundtol** (1) rounds halfway cases away from zero and (2) may but need not raise the inexact exception for nonintegral arguments that round to within the range of **long int**.

X.9.6.8 The trunc function

The **trunc** function uses IEC 559 rounding toward zero (regardless of the current rounding direction).

X.9.7 Remainder functions

X.9.7.1 The fmod function

- If one argument is a NaN then **fmod** returns that same NaN; if both arguments are NaNs then **fmod** returns one of its arguments.
- **fmod(±0, y)** returns ±0 if **y** is not zero.
- **fmod(x, y)** returns a NaN and raises the invalid exception if **x** is infinite or **y** is zero.
- **fmod(x, ±INFINITY)** returns **x** if **x** is not infinite.

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

```
#include <math.h>
#include <fenv.h>
FENV_ACCESS_ON;
double fmod(double x, double y)
{
   double result;
   result = remainder(fabs(x), (y = fabs(y)));
   if (signbit(result)) result += y;
   return copysign(result, x);
}
```

X.9.7.2 The remainder function

The **remainder** function is fully specified as a basic arithmetic operation in IEC 559.

X.9.7.3 The remquo function

The **remquo** function follows the specification for **remainder**. It has no further specification special to IEC 559 implementations.

X.9.8 Manipulation functions

X.9.8.1 The copysign function

copysign is specified in the Appendix to IEC 559.

X.9.8.2 The nan functions

All IEC 559 implementations support quiet NaNs, in all floating formats.

X.9.8.3 The nextafter functions

- If one argument is a NaN then **nextafter** returns that same NaN; if both arguments are NaNs then **nextafter** returns one of its arguments.
- **nextafter(x, y)** raises the overflow and inexact exceptions if **x** is finite and the function value is infinite.
- **nextafter(x, y)** raises the underflow and inexact exceptions if the function value is subnormal and **x != y**.

X.9.9 Maximum, minimum, and positive difference functions

X.9.9.1 The fdim function

- If one argument is a NaN then **`fdim`** returns that same NaN; if both arguments are NaNs then **`fdim`** returns one of its arguments.

X.9.9.2 The fmax function

- If just one argument is a NaN then **`fmax`** returns the other argument; if both arguments are NaNs then **`fmax`** returns one of its arguments.

  The **`fmax`** function might be implemented as

  ```
  { return (isgreaterequal(x, y) || isnan(y)) ? x : y; }
  ```
  [41]

  [Some applications might be better served by a *max* function that would return a NaN if one of its arguments were a NaN:

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

  Note that two branches are still required, for symmetry in NaN cases.]

X.9.9.3 The fmin function

`fmin` is analogous to **`fmax`**. See X.9.9.2.

[41] Ideally, **`fmax`** would be sensitive to the sign of zero, for example **`fmax(-0.0, +0.0)`** would return +0; however, implementation in software might be impractical.