C9X Floating-Point

WG14/N595 X3J11/96-059 (Draft 9/12/96)

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

This document describes the detailed changes to made to C9X Draft 6 (N559) in order to incorporate the floating-point proposal accepted at the Amsterdam meeting. It has been approved by the editorial review committee and is intended for the redactor. It is basically the proposal N546 (post Irvine mailing), with

1. overview and rationale removed (to reappear elsewhere)
2. C9X references updated to Draft 6
3. the footnote (presented to X3J11 at Amsterdam) to further highlight the conditional nature of the IEC 559 annex
4. long long versions of two functions
5. some corrections (from Fred Tydeman) regarding when domain and range errors occur or might occur
6. a few author's editorial changes
7. positive values of FLT_EVAL_METHOD reserved for future use
8. several clarifications and corrections from the editorial review committee (Fred Tydeman, Douglas Walls, Frank Farance, and John Kwan).

(Note that the page number is not printed on the blank pages inserted in order for major sections to begin on right-hand pages.)

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 last three sentences of [1] 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 paragraph [2] add:

The values of operations with floating operands and 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 negative values for `FLT_EVAL_METHOD` characterize implementation-defined behavior.

6.3.4 Cast operators

Footnote the last sentence of [4] with:

If the value of the expression is represented with greater precision or range than required by the type named by the cast (6.2.1.5), then the cast specifies a conversion even if the type of the expression is the same as the named type.

Feature: Dynamic FLT_ROUNDS

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

In [2], 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.

Recommended 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*

7.11.1.4 The `strtod` function

Replace the first sentence of [4] 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 [5], 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 paragraph [6] 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.

7.17.4.1.1 The `wcstod` function

Replace the first sentence of [4] 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 [5], 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 paragraph [6], 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.

7.10.6.1 The `fprintf` function

In the third bullet of [4], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the fourth bullet of [4], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the # flag specification in [6], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the 0 flag specification in [6], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

After the g, G conversion specifier and its meaning in [8], 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 (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.

7.17.2.1 The `fwprintf` function

In the third bullet of [5], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the fourth bullet of [5], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the # flag specification in [7], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

In the 0 flag specification in [7], replace "**`e`**, **`E`**," *with* "**`a`**, **`A`**, **`e`**, **`E`**,".

After the g, G conversion specifier and its meaning in [8], 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.

7.10.6.2 The `fscanf` function

In the third bullet of [4], replace "**e**, **f**, and **g**" *with* "**a**, **e**, **f**, and **g**".

In [13], replace "**e, f, g**" *with* "**a, e, f, g**".

In [17], replace "specifiers **E**," *with* "specifiers **A**, **E**,". *In the same paragraph, replace* "respectively, **e**," *with* "respectively, **a**, **e**,".

7.17.2.2 The `fwscanf` function

In [4], replace "**e**, **f**, and **g**" *with* "**a**, **e**, **f**, and **g**".

In [12], replace "**e, f, g**" *with* "**a, e, f, g**".

In [17], replace “specifiers **`E`**,” *with* “specifiers **`A`**, **`E`**,”. *In the same paragraph, replace* “respectively, **`e`**,” *with* “respectively, **`a`**, **`e`**,”.

Feature: Output of wide exponents

7.10.6.1 The `fprintf` function

In the "e, E" item in [8], 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”.

7.17.2.1 The `fwprintf` function

In the "e, E" item in [8], 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

5.2.4.2.2 Characteristics of floating types <float.h>

In [1], before the sentence beginning "Of the values in the <float.h> header ...", insert:

Floating types might include values that are not normalized floating-point numbers, for example subnormal numbers ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$), infinities, and NaNs. 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.

Footnote the last inserted sentence above with:

IEC 559 specifies quiet and signaling NaNs. For implementations that do no support IEC 559, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

7.11.1.4 The `strtod` function

In the first sentence of [4], 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 [5], replace “expected form” *with* “expected form for a floating-point number”.

After the first sentence of the [5], 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(`**$n\text{-}char\text{-}sequence_{opt}$**`)`** 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 [5], footnote “negated” with:

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

7.17.4.1.1 The `wcstod` function

In the first sentence of [4], 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(`**$n\text{-}wchar\text{-}sequence_{opt}$ **`)`**, 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 [5], replace “expected form” *with* “expected form for a floating-point number”.

After the first sentence of [5], 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(`**$n\text{-}wchar\text{-}sequence_{opt}$**`)`** 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 [5], footnote "negated" with:

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

7.10.6.1 The `fprintf` function

In the third bullet of [4], replace "**E**, and **f**" *with* "**E**, **f**, and **F**".

In the fourth bullet of [4], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

Footnote the + flag specification in [6] with:

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

In the # flag specification in [6], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the 0 flag specification in [6], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the list of conversion specifiers and their meanings in [8], change the "**f**" *bullet to* "**f, F**".

In the f item in the list of conversion specifiers and their meanings in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], replace "style **E**" *with* "style **F** or **E**".

7.17.2.1 The fwprintf function

In the third bullet of [5], replace "**E**, and **f**" *with* "**E**, **f**, and **F**".

In the fourth bullet of [5], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

Footnote the + flag specification in [7] with:

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

In the # flag specification in [7], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the 0 flag specification in [7], replace "**E**, **f**," *with* "**E**, **f**, **F**,".

In the list of conversion specifiers and their meanings in [8], change the "**f**" *bullet to* "**f, F**".

In the f item in the list of conversion specifiers and their meanings in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], 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 in [8], replace "style `E`" *with* "style `F` or `E`".

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: (note the subclause number will be incremented by a subsequent change)

The specification for the library functions recommends more accurate conversion than required for floating constants. See `strtod` (7.11.1.5).

7.11.1.4 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.

7.17.4.1.1 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.

7.10.6.1 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.

7.17.2.1 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: FP environment management

5.1.2.3 Program execution

In the first sentence of [2], 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: Strings -> float, long double

7.11.1 String conversion functions

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

7.11.1.4 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.

7.11.1.7 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.

7.17.4.1 Wide-string numeric conversion functions

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

7.17.4.1.1 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.

7.17.4.1.4 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: Contraction control

6.3 Expressions

Append after [7] 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.

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 rules for addition or multiplication, nor the distributive rule, 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 rules for real numbers 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

4 Compliance

Footnote the first sentence of [1] with:

This implies that a strictly conforming program can use features in a conditionally normative annex provided the use is conditioned by a **#ifdef** directive with the conformance macro for the annex, as in

```
#ifdef __STDC_IEC_559__  /* FE_UPWARD defined */
    ...
    fesetround(FE_UPWARD);
    ...
#endif
```

6.8.8 Predefined macro names

After paragraph [1], 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>`

In [1], replace the sentence beginning "Of the values in the <float.h> header ..." 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.

Note to redactor: The change immediately above needs to be applied after the earlier change that introduced FLT_EVAL_METHOD.

Mathematics (Enhanced) <math.h>

Replace 7.6 with the following:

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 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.

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]

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.

---

[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.

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

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 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.

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]

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

---

[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
$10^{n-1} > \texttt{FLT\_RADIX}^{m}$	otherwise

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.

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.

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.

Example

The **fpclassify** macro might be implemented in terms of ordinary functions 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. A floating 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 a floating 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`** when a range error occurs 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.

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.

Returns

The **`acosh`** function returns the arc hyperbolic cosine in the range $[0, +\infty]$.

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`**.

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: $e^{x} - 1$.[11] 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);
```

---

[11] For small magnitude `x`, `expm1(x)` is expected to be more accurate than `exp(x) - 1`.

Description

The `log1p` function computes the base-e logarithm of 1 plus the argument.[12] 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);
```

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`,

$$1 \le \mathtt{x * FLT\_RADIX}^{\mathtt{-logb(x)}} < \mathtt{FLT\_RADIX}$$

A range error may occur if the argument is zero.

Returns

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

---

[12] For small magnitude `x`, `log1p(x)` is expected to be more accurate than `log(1 + x)`.

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`^{n} efficiently, not normally by computing `FLT_RADIX`^{n} explicitly. A range error may occur.

Returns

The `scalb` function returns `x * FLT_RADIX`^{n}.

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

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**. A range error occurs if **x** is too large.

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 if the magnitude of **x** is too large.

Returns

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

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 or if **x** is a negative integer or zero.

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.)

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. A range error may occur if the magnitude of **`x`** is too large.

Returns

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

7.x.9.6 The `rinttoll` function

Synopsis

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

Description

The `rinttoll` function is similar to the `rintol` function, except that the returned value has type `long long`.

7.x.9.7 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.

Returns

The `round` function returns the rounded integral value.

7.x.9.8 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. A range error may occur if the magnitude of `x` is too large.

Returns

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

7.x.9.9 The `roundtoll` function

Synopsis

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

Description

The **`roundtoll`** function is similar to the **`roundtol`** function, except that the returned value has type **`long long`**.

7.x.9.10 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.[13]

Returns

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

---

[13] "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.

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 $2^{\mathtt{n}}$ 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`.

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.

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` function

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.[14] 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`.

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

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:

$$\begin{array}{ll} x - y & \text{if } x > y \\ +0 & \text{if } x \le y \end{array}$$

A range error may occur.

Returns

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

---

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

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.[15]

Returns

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

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.[16]

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.[17] 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);
```

---

[15] 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.

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

[17] 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; the result in these cases is false.

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);
```

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.

Floating-Point Environment <fenv.h>

In 7, insert this subclause:

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.[18] 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.

Certain programming conventions support the intended model of use for the floating-point environment[19]:

— 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.

The type

```
fenv_t
```

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.

---

[18] 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.

[19] 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.

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.2. The defined macros expand to integral constant expressions whose values are distinct powers of 2.

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.[20]

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;
```

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

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.[21] 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.

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**.[22]

7.x.2 Exceptions

The following functions provide access to the exception flags.[23] 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.

---

[21] 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.

[22] 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.

[23] 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.

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`**.

7.x.2.3 The `feraiseexcept` function

Synopsis

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

Description

The **`feraiseexcept`** function raises the supported exceptions represented by its argument.[24] The order in which these exceptions are raised is unspecified, except as stated in X.7.6.

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

---

[24] 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.

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.[25]

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, then **g** if overflow is set:

```
#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);
```

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

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.[26]

Returns

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

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.

---

[26] 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.

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.

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

Insert the following annex:

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. Where a binding between the C language and IEC 559 is indicated, the IEC 559-specified behavior is adopted by reference, unless stated otherwise.

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[27], 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.[28]

Recommended practice

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

X.2.1 Infinities, signed zeros, and NaNs

This specification does not define the behavior of signaling NaNs.[29] 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 IEC 559 NaNs and infinities.

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

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

X.3 Operators and functions

C operators and functions provide IEC 559 required and recommended facilities as listed below.

— 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.

— 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

---

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

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 **nextafter** functions in **<math.h>** provides the nextafter function recommended in the Appendix to IEC 559 (but with a minor change to better handle signed zeros).

— The **isfinite** macro in **<math.h>** provides the finite function recommended in the Appendix to IEC 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 integral 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 integral type raises the inexact exception is unspecified.[30]

[30] 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>**.

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.[31]

Conversions involving IEC 559 formats follow all pertinent recommended practice. In particular, conversion between any supported IEC 559 format and decimal with `DECIMAL_DIG` or fewer significant digits is correctly rounded.

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.[32]

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.[33]

[31] 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.)

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

[33] 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.

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.[34]

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.

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*).[35]

---

[34] 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.

[35] 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.

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.[36] The automatic initialization of **z** entails 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.

---

[36] 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;
```

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

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.

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;
   /*...*/
}
```

`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).[37]

`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).[38]

`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).

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

---

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

[38] 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*))

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,[39] 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.

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.

[39] 0 - 0 yields -0 instead of +0 just when the rounding direction is downward.

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 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.[40]

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.

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-defined.[41] Otherwise, as implied by X.7.6, the `<math.h>` functions do not raise spurious exceptions (detectable by the user).

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

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. This will be resolved with the redactor.

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 $|x| > 1$.

[40] IEC 559 allows different definitions of underflow. They all result in the same values, but differ on when the exception is raised.
[41] It is intended that undeserved underflow and inexact exceptions are raised only if determining inexactness would be too costly.

X.9.1.3 The atan function

- **`atan(±0)`** returns ±0.
- **`atan(±INFINITY)`** returns $\pm\pi/2$.

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.[42]
- **`atan2(±0, x)`** returns $\pm\pi$, for **`x`** < 0.
- **`atan2(±0, -0)`** returns $\pm\pi$.
- **`atan2(y, ±0)`** returns $\pi/2$ for **`y`** > 0.
- **`atan2(y, ±0)`** returns $-\pi/2$ for **`y`** < 0.
- **`atan2(±y, INFINITY)`** returns ±0, for finite **`y`** > 0.
- **`atan2(±INFINITY, x)`** returns $\pm\pi/2$, for finite **`x`**.
- **`atan2(±y, -INFINITY)`** returns $\pm\pi$, for finite **`y`** > 0.
- **`atan2(±INFINITY, INFINITY)`** returns $\pm\pi/4$.
- **`atan2(±INFINITY, -INFINITY)`** returns $\pm 3\pi/4$.

X.9.1.5 The cos function

- **`cos(±0)`** returns 1.
- **`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`**.

---

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

X.9.2.3 The atanh function

- **`atanh(±0)`** returns ±0.
- **`atanh(±1)`** returns ±**`INFINITY`** and raises the divide-by-zero exception.
- **`atanh(x)`** returns a NaN and raises the invalid exception if $|\mathbf{x}| > 1$.

X.9.2.4 The cosh function

- **`cosh(±0)`** returns 1.
- **`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(±0)`** returns 1.
- **`exp(+INFINITY)`** returns **`+INFINITY`**.
- **`exp(-INFINITY)`** returns +0.

X.9.3.2 The exp2 function

- **`exp2(±0)`** returns 1.
- **`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

- **`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`**.
- **`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)
```

X.9.3.6 The log function

- **`log(±0)`** returns **`-INFINITY`** and raises the divide-by-zero exception.
- **`log(1)`** returns +0.
- **`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(1)`** returns +0.
- **`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.
- **scalb(x, 0)** returns **x**.

X.9.4 Power and absolute value functions

X.9.4.1 The fabs function

- **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 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.

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

- **`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.

- **rint(**±**0)** returns ±0 (for all rounding directions).
- **rint(**±**INFINITY)** returns ±**INFINITY** (for all rounding directions).

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 function

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

X.9.8.3 The nextafter function

- 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 or zero 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; }`[43]

X.9.9.3 The fmin function

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

[43] 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.