

Proposal for C2Y WG N3943

Title: Annex F Special Cases - improve consistency, structure, readability, +++
Author: CFP Study Group
Date: October 15, 2026
Proposal Category: Editorial + minor technical
Reference: N3854

Introduction

The content of Annex F and Annex G exhibit an inconsistent use of language and sentence structure. This is most obvious in the mathematical domains for which some special case applies. Some domains with the same mathematical meaning are often expressed with different wording or different punctuations, and some are

- a. written using inequalities and equalities, some with less formal adjectives, and some are mixed;
- b. introduced by **for** and others by **if**, even when there is no good reason for such a variation.

Such inconsistencies can lead a reader to mistakenly think there are subtle differences in meaning between such phrases. To address this, a single syntax has been distilled from the dominant domain description styles seen in C2Y's Annex F's special cases, i.e. their structure, terminology, and punctuation. It uses mathematical inequalities or equalities for domain qualification across the board. This syntax spells out the function, the parameter(s) associated with the special case, the result for this case, any exceptions that might be raised, and any domain that might qualify when this special case applies. All specifications now adhere to this grammar:

```
<function-name> ( <argument> [ , <argument> ] )  
<consequence> [ and <consequence> ]  
[ <qualification> ]
```

where *<consequence>* is one or more of

- i. **returns** *value*
- ii. **raises the** "*<floating-point-exception-name>*" **floating-point exception**
- iii. **leaves the object pointed to by** *<name>* **containing value**

and *<qualification>* is one of

- i. **for domain** [[**and** | **or**] *domain*]
- ii. **when domain and the result is** *<classification>* ...
- iii. [**even**] **if condition** | *domain* [, **or condition** | *domain*]

Any domain qualification for a special case that was a mix of an adjective and an algebraic expression is now just an algebraic expression, e.g., what was "finite *x*" is now replaced by $-\infty < x < +\infty$. The same domain in separate special cases within (or between related) functions is now always described in the same manner.

As for the details within special cases themselves, there were some minor adjustments. Where two special cases had overlapping domains, the redundant overlap was eliminated. Any special case whose domain was expressed in the negative, i.e. what values of the parameters to the functions were excluded, has largely been rewritten in the positive, noting those values of *x* which actually are included in the domain. In a handful of special cases, the case itself was split into multiple domains for clarity. Lastly, a domain such as

all *x* (including NaN)

and its multiple variations has been replaced in ten places by the more explicit form:

$-\infty \leq x \leq +\infty$ or *x* a quiet NaN

with *x* replaced by either *y* or **cx* when appropriate. This domain definition style also avoids language that says what a function parameter is **not**, rather than what it actually is.

Where functions explicitly note that they raise no floating-point exceptions, this is normally the penultimate

mention within a sub-clause. Any mention of rounding behaviour is normally the final paragraph (or sentence) in a sub-clause.

This proposal replaces every instance of "*current rounding direction*" with *rounding direction mode*. The former term goes back to C99 when there was only the dynamic rounding direction mode, and no constant rounding mode.

Inconsistencies were also addressed to avoid confusion. Within the sub-clauses of F.10 of Annex F:

- a. All but two of the families of functions seen in those sub-clauses never cross reference their definitions in Section 7. The cross references in those two outlier cases have been deleted in the name of conformity.
- b. When parameters to a function are referenced within a sub-clause, they **always** appear appended to the function name within that sub-clause. (Some are omitted in C2Y).
- c. A parameter name is occasionally qualified by the word "argument". This is not only redundant but also wrong, and so the word "argument" has been deleted.
- d. The introduction of F.10.1 General says that the functions of F.3 are specified in ISO/IEC 60559. Both F.10.12 and F.10.13, functions in F.3, unnecessarily repeat this. These redundant statements have gone.

Elsewhere, expressions of the same concept in multiple ways has been rationalised. As an example, the term *result* is a replacement for *return value* and *returned value* unless stated otherwise. In two cases, footnotes were moved into the specification itself as it was easier to read. The language around overflow and underflow within special cases has been tightened. The current wording unnecessarily excludes exact underflow cases; the new wording broadens the statement to include those cases. Clunky or unclear wording was reworked, fonts were made consistent, and typos, grammar and punctuation were fixed.

Three technical issues are addressed:

- (a) F.10.4.18 introduces a specification that was previously missing;
- (b) F.10.5.2 has omitted the ISO/IEC 60559 special case of **compoundn**(0, *n*); and
- (c) F.10.4.19 infers that both a signaling NaN and a quiet NaN input to **scalbn**() needs special handling.

For (b), the missing case from ISO/IEC 60559 is introduced as an extra item in F.10.5.2. For (c), the correct (default) behavior for a signaling NaN and a quiet NaN input is already noted in F.2.2, both types of NaN can be, and now are, safely excluded from the domain seen in the second item of F.10.4.19.

Due to possible confusion between a parameter called *value* and the return value of a function, i.e. its result, the use of a parameter called *value* has been eliminated. These prototypes now conform more closely to argument naming practices used elsewhere throughout C2Y.

In the revised sample code for **modf**, it has been assumed that N3842 will be accepted by WG14 for inclusion in C2Y so that

```
#include <fenv.h>
```

no longer needs to be included prior to the use of

```
#pragma STDC . . . .
```

Note to Editor The trailing **s** in the term "rounding direction modes" is emboldened purely to highlight the changed pluralization of that word and make it more readable to the editor.

Anything struck-through in red is for deletion. Strike-through works at the middle of a character on the line. Hence, a footnote for deletion looks like it is underlined in red. It is meant for deletion. Only the revised sample code for **frexp** and **modf** appears for readability reasons with changes written in **blue**. The actual changes made are listed explicitly as including the old text in over-struck red is hard to read.

10.2.4#1 The `atan2` functions

[Note inserted plus sign in 7th item]

- `atan2`($-y, x$) and `-atan2`(y, x) are equivalent. [Not a new requirement - see F.10.1#3]
- ~~`atan2`($\pm 0, -0$) returns $\pm\pi$~~ ⁴⁰⁹⁾ [Drop footnote reference and incorporate directly, so ...]
`atan2`($\pm 0, -0$) returns $\pm\pi$ and raises no "invalid" floating-point exception.
- `atan2`($\pm 0, +0$) returns ± 0 and raises no "invalid" floating-point exception. [It is an exact zero]
- `atan2`($\pm 0, x$) returns $\pm\pi$ for $x < 0$.
- `atan2`($\pm 0, x$) returns ± 0 for $x > 0$.
- `atan2`($y, \pm 0$) returns $-\frac{\pi}{2}$ for $y < 0$ and raises no "divide-by-zero" floating-point exception.
- `atan2`($y, \pm 0$) returns $+\frac{\pi}{2}$ for $y > 0$ and raises no "divide-by-zero" floating-point exception.
- `atan2`($\pm y, -\infty$) returns $\pm\pi$ for ~~finite $y > 0$~~ $0 < y < +\infty$.
- `atan2`($\pm y, +\infty$) returns ± 0 for ~~finite $y > 0$~~ $0 < y < +\infty$.
- `atan2`($\pm\infty, x$) returns $\pm\frac{\pi}{2}$ for ~~finite x~~ $-\infty < x < +\infty$.
-

[Drop footnote - incorporated within item text]

⁴⁰⁹⁾ ~~`atan2`($0, 0$) does not raise the "invalid" floating-point exception; nor does `atan2`($y, 0$) raise the divide-by-zero floating-point exception.~~

10.2.11#1 The `atan2pi` functions

- `atan2pi`($-y, x$) and `-atan2pi`(y, x) are equivalent. [Not a new requirement - see F.10.1#3]
- ~~`atan2pi`($\pm 0, -0$) returns ± 1~~ ⁴¹⁰⁾ [Drop footnote reference and incorporate directly, so ...]
`atan2pi`($\pm 0, -0$) returns ± 1 and raises no "invalid" floating-point exception.
- `atan2pi`($\pm 0, +0$) returns ± 0 and raises no "invalid" floating-point exception. [It is an exact zero]
- `atan2pi`($\pm 0, x$) returns ± 1 for $x < 0$.
- `atan2pi`($\pm 0, x$) returns ± 0 for $x > 0$.
- `atan2pi`($y, \pm 0$) returns $-\frac{1}{2}$ for $y < 0$ and raises no "divide-by-zero" floating-point exception.
- `atan2pi`($y, \pm 0$) returns $+\frac{1}{2}$ for $y > 0$ and raises no "divide-by-zero" floating-point exception.
- `atan2pi`($\pm y, -\infty$) returns ± 1 for ~~finite $y > 0$~~ $0 < y < +\infty$.
- `atan2pi`($\pm y, +\infty$) returns ± 0 for ~~finite $y > 0$~~ $0 < y < +\infty$.
- `atan2pi`($\pm\infty, x$) returns $\pm\frac{1}{2}$ for ~~finite x~~ $-\infty < x < +\infty$.
-

[Drop footnote - incorporated within item text]

⁴¹⁰⁾ ~~`atan2pi`($0, 0$) does not raise the "invalid" floating-point exception; nor does `atan2pi`($y, 0$) raise the divide-by-zero floating-point exception.~~

F.10.2.12#1 The **cospi** functions

- ~~**cospi**($n + \frac{1}{2}$) returns +0, for integers n .~~
cospi($n + \frac{1}{2}$) returns +0 for integers n . [*The sole change is the deleted comma*]

F.10.2.13#1 The **sinpi** functions

- ~~**sinpi**($\pm n$) returns ± 0 , for positive integers n .~~
sinpi($\pm n$) returns ± 0 for integers $n > 0$.

F.10.2.14#1 The **tanpi** functions

[*The last two changes have a deleted comma in addition to their reordering*]

- **tanpi**(± 0) returns ± 0
- ~~**tanpi**(n) returns +0, for positive even and negative odd integers n .~~
tanpi(n) returns +0 for even integers $n > 0$ and odd integers $n < 0$.
- ~~**tanpi**(n) returns -0, for positive odd and negative even integers n .~~
tanpi(n) returns -0 for odd integers $n > 0$ and even integers $n < 0$.
- ~~**tanpi**($n + \frac{1}{2}$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception, for even integers n .~~
tanpi($n + \frac{1}{2}$) returns $-\infty$ and raises the "divide-by-zero" floating-point exception for odd integers n .
- ~~**tanpi**($n + \frac{1}{2}$) returns $-\infty$ and raises the "divide-by-zero" floating-point exception, for odd integers n .~~
tanpi($n + \frac{1}{2}$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for even integers n .

F.10.4.7#1 The **frexp** functions

- **frexp**(NaN, p) returns a NaN and stores an unspecified value in the object pointed to by p .
(~~and returns a NaN~~).

F.10.4.7#2 The **frexp** functions

[*Merge #2 and #3 as these two sentences appear in the same paragraph elsewhere in Annex F*]

frexp(x , p) raises no floating-point exceptions if $-\infty \leq x \leq +\infty$ or x is a quiet NaN. ~~value is not a signaling NaN.~~

~~The returned value is~~ Both the return value of **frexp**(x , p) and the value stored in the object pointed to by p are independent of the ~~current~~ rounding direction modes.

F.10.4.7#3 The **frexp** functions - RENUMBERED - was #4

EXAMPLE: For implementations that define `__STDC_IEC_60559_BFP__`, **frexp** behaves as though implemented by ~~On a binary system, the body of the **frexp** function may be~~

```
#include <math.h>
#pragma STDC FENV_ACCESS ON
double frexp(double x, int *p)
{
    *p = (x == 0 || !isfinite(x)) ? 0 : (int)(1 + logb(x));
    return scalbn(x, -(*p));
}
```

[*Note the value $\rightarrow x$ change + the #include and prototype lines seen above are new*]

F.10.4.8#1 The **ilogb** functions

~~When the correct result is representable in the range of the return type, the returned value is exact and is independent of the current rounding direction mode.~~

ilogb(*x*) returns an exact result independent of the rounding direction modes when the correct result is representable within the range of the return type.

F.10.4.8#2 The **ilogb** functions

~~If the correct result is outside the range of the return type, the numeric result is unspecified and the "invalid" floating-point exception is raised.~~

ilogb(*x*) returns an unspecified result and raises the "invalid" floating-point exception when the correct result is outside the range of the return type.

F.10.4.8#3 The **ilogb** functions

~~**ilogb**(*x*), for *x* zero, infinite, or NaN raises the "invalid" floating-point exception and returns the value specified in 7.12.7.8.~~

ilogb(*x*) returns the value specified in 7.12.6.8 and raises the "invalid" floating-point exception when $x = 0$ or $x = \pm\infty$ or *x* is either a quiet NaN or a signaling NaN.

F.10.4.9#1 The **ldexp** functions

~~On a binary system, **ldexp**(*x*, *p*) is equivalent to **scalbn**(*x*, *p*).~~

F.10.4.17#2 The **logb** functions

The ~~returned value~~ result is exact and is independent of the ~~current~~ rounding direction modes.

F.10.4.18#1 The **modf** functions

- **modf**(*x*, *p*) ~~**modf**($\pm x$, *iptr*)~~ returns a result with the same sign as *x*. [~~remove $\pm$ sign and *iptr* $\rightarrow$ *p*~~]
- **modf**($\pm\infty$, ~~*iptr* *p*~~) returns ± 0 and stores $\pm\infty$ in the object pointed to by ~~*iptr* *p*~~.
- **modf**(NaN, ~~*iptr* *p*~~) (~~returns a NaN and~~) stores an unspecified value in the object pointed to by ~~*iptr* *p*~~. (~~and returns a NaN~~).

F.10.4.18#2 The **modf** functions

~~The returned values are exact and are independent of the current rounding direction mode.~~

modf(*x*, *p*) raises no floating-point exceptions if $-\infty \leq x \leq +\infty$ or *x* is a quiet NaN. Both the return value of **modf**(*x*, *p*) and the value stored in the object pointed to by *p* are independent of the rounding direction modes.

F.10.4.18#3 The **modf** functions

EXAMPLE: For implementations that define `__STDC_IEC_60559_BFP__`, **modf** behaves as though implemented by

```
#include <math.h>
#pragma STDC FENV_ACCESS ON
double modf(double x, double *p)
{
    int save_round = fegetround();
    fesetround(FE_TOWARDZERO);
    *p = nearbyint(x);
    fesetround(save_round);
    return copysign(isinf(x) ? 0.0 : x - (*p), x);
}
```

[Note the value $\rightarrow x$ and *iptr* $\rightarrow p$ change + the return is now a single line and `<fenv.h>` is dropped]

F.10.4.19#1 The **scalbn** and **scalbln** functions

- **scalbn**($x, 0$) returns x for $-\infty \leq x \leq +\infty$. [*Correction - See Introduction*]

F.10.4.19#2 The **scalbn** and **scalbln** functions

If ~~the calculation does not overflow or underflow~~, no "overflow" or "underflow" floating-point exception is raised, the ~~returned value~~ result is exact and is independent of the ~~current~~ rounding direction modes.

F.10.5.2#1 The **compoundn** functions

- **compoundn**($x, 0$) returns 1 for $x \geq -1$ or x a quiet NaN.
- **compoundn**(x, n) returns a NaN and raises the "invalid" floating-point exception for $x < 1$
-
- **compoundn**($-1, n$) returns +0 for $n > 0$
- **compoundn**($0, n$) returns 1 for all n . [*Insertion - See Introduction*]
- ~~**compoundn**($+\infty, n$) returns $+\infty$ for $n > 0$.~~
compoundn($+\infty, n$) returns +0 for $n < 0$. [*reversed with following for consistency*]
- ~~**compoundn**($+\infty, n$) returns +0 for $n < 0$.~~
compoundn($+\infty, n$) returns $+\infty$ for $n > 0$. [*reversed with preceding for consistency*]

F.10.5.3#2 The **fabs** functions

[*correct the font of the function parameter to italic +++ other stuff*]

fabs(x) raises no floating-point exceptions, even if x is a signaling NaN. The ~~returned value~~ result is independent of the ~~current~~ rounding direction modes.

F.10.5.4#1 The **hypot** functions

-
- **hypot**($x, \pm 0$) returns ~~the absolute value of x , if x is not a NaN~~ $|x|$ for $-\infty \leq x \leq +\infty$.
- **hypot**($x, \pm\infty$) returns $+\infty$ ~~, even if y is a NaN~~ for $-\infty \leq x \leq +\infty$ or x a quiet NaN. [*drop comma*]
- **hypot**(x, NaN) returns a NaN ~~, if x is not $\pm\infty$~~ for $-\infty < x < +\infty$ or x a quiet NaN. [*drop comma*]

F.10.5.5#1 The **pow** functions

- **pow**($\pm 0, y$) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for ~~y an odd integer < 0~~ . $-\infty < y < 0$ and y an odd integer.
- **pow**($\pm 0, y$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for ~~$y < 0$, finite, and not an odd integer~~. $-\infty < y < 0$ and y not an odd integer.
- **pow**($\pm 0, -\infty$) returns $+\infty$
- **pow**($\pm 0, y$) returns ± 0 for ~~y an odd integer > 0~~ $0 < y < +\infty$ and y an odd integer.
- **pow**($\pm 0, y$) returns +0 for ~~$y > 0$ and not odd integer~~ $y > 0$ and y not an odd integer.
- **pow**($-1, \pm\infty$) returns 1.
- **pow**($+1, y$) returns 1 for ~~any y , even a NaN~~ $-\infty \leq y \leq +\infty$ or y a quiet NaN.
- **pow**($x, \pm 0$) returns 1 for ~~any x , even a NaN~~ $-\infty \leq x \leq +\infty$ or x a quiet NaN.
- **pow**(x, y) returns a NaN and raises the "invalid" floating-point exception for ~~finite $x < 0$ and finite non-integer y~~ . $-\infty < x < 0$ and $-\infty < y < +\infty$ and y not an integer.
- **pow**($x, -\infty$) returns $+\infty$ for $|x| < 1$.

-
- **pow**($x, +\infty$) returns $+\infty$ for $|x| > 1$.
- **pow**($-\infty, y$) returns -0 for ~~y an odd integer < 0~~ $-\infty < y < 0$ and y an odd integer.
- **pow**($-\infty, y$) returns $+0$ for ~~$y < 0$ and not an odd integer~~ $y < 0$ and y not an odd integer.
- **pow**($-\infty, y$) returns $-\infty$ for ~~y an odd integer > 0~~ $0 < y < +\infty$ and y an odd integer.
- **pow**($-\infty, y$) returns $+\infty$ for ~~$y > 0$ and not an odd integer~~ $y > 0$ and y not an odd integer.
- **pow**($+\infty, y$) returns $+0$ for $y < 0$.
-

F.10.5.6#1 The **pown** functions

- **pown**($x, 0$) returns 1 for ~~all x not a signaling NaN~~ $-\infty \leq x \leq +\infty$ or x a quiet NaN.
- **pown**($\pm 0, n$) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for odd $n < 0$.
- **pown**($\pm 0, n$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for even $n < 0$.
- ~~**pown**($\pm 0, n$) returns $+0$ for even $n > 0$.~~
pown($\pm 0, n$) returns ± 0 for odd $n > 0$. [*reversed with following for consistency*]
- ~~**pown**($\pm 0, n$) returns ± 0 for odd $n > 0$.~~
pown($\pm 0, n$) returns $+0$ for even $n > 0$. [*reversed with preceding for consistency*]
- **pown**($\pm\infty, n$) is equivalent to **pown**($\pm 0, -n$) for ~~n not 0, except that the "divide-by-zero" floating-point exception is not raised.~~ $n < 0$.
- **pown**($\pm\infty, n$) returns $\pm\infty$ for odd $n > 0$.
- **pown**($\pm\infty, n$) returns $+\infty$ for even $n > 0$.

F.10.5.7#1 The **powr** functions

- **powr**($x, \pm 0$) returns 1 for ~~finite $x > 0$.~~ $0 < x < +\infty$.
- **powr**($\pm 0, y$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for ~~finite $y < 0$.~~ $-\infty < y < 0$.
- **powr**($\pm 0, -\infty$) returns $+\infty$.
- **powr**($\pm 0, y$) returns $+0$ for $y > 0$.
- **powr**($+1, y$) returns 1 for ~~finite y .~~ $-\infty < y < +\infty$.
- **powr**($+1, \pm\infty$) returns a NaN and raises the "invalid" floating-point exception.
- **powr**(x, y) returns a NaN and raises the "invalid" floating-point exception for $x < 0$ and ~~all y (including NaN)~~ either $-\infty \leq y \leq +\infty$ or y a quiet NaN.
-

F.10.5.8#1 The **rootn** functions

[The domains of the very last and 3rd last items no longer overlap]

- **rootn**($\pm 0, n$) returns $\pm\infty$ and raises the "divide-by-zero" floating-point exception for odd $n < 0$.
- **rootn**($\pm 0, n$) returns $+\infty$ and raises the "divide-by-zero" floating-point exception for even $n < 0$.
- ~~**rootn**($\pm 0, n$) returns $+0$ for even $n > 0$.~~
rootn($\pm 0, n$) returns ± 0 for odd $n > 0$.
- ~~**rootn**($\pm 0, n$) returns ± 0 for odd $n > 0$.~~
rootn($\pm 0, n$) returns $+0$ for even $n > 0$.
- **rootn**($+\infty, n$) returns $+\infty$ for $n > 0$.
-
- **rootn**($-\infty, 0$) returns a NaN and raises the "invalid" floating-point exception for even $n < 0$.
- **rootn**($x, 0$) returns a NaN and raises the "invalid" floating-point exception for ~~all x (including NaN)~~
 $-\infty \leq x \leq +\infty$ or x a quiet NaN.
- **rootn**(x, n) returns a NaN and raises the "invalid" floating-point exception for $x < 0$ and ~~n even~~ $n \neq 0$.

F.10.5.10#1 The **sqrt** functions

- ~~**sqrt**($+\infty$) return $+\infty$.~~
sqrt(x) returns a NaN and raises the "invalid" floating-point exception for $x < 0$.
- ~~**sqrt**(x) returns a NaN and raises the "invalid" floating-point exception. for $x < 0$~~
sqrt($+\infty$) returns $+\infty$.

F.10.5.10#2 The **sqrt** functions

The ~~returned-value~~ **result** is dependent on the ~~current~~ **applicable** rounding direction mode.

F.10.6.3#1 The **lgamma** functions - 2nd item only

- floating-point exception for ~~x a negative integer or zero.~~ integers $x \leq 0$.

F.10.6.4#1 The **tgamma** functions - 2nd item only

- floating-point exception for ~~x a negative integer.~~ integers $x < 0$.

F.10.7.1#1 The **ceil** functions

The **ceil** functions round "toward positive infinity", choosing only from among integer values of the parameter type.

- **ceil**(± 0) returns ± 0 .
- **ceil**($\pm\infty$) returns $\pm\infty$.

F.10.7.1#2 The **ceil** functions

The ~~returned-value~~ **result** is exact and is independent of the ~~current~~ rounding direction modes.

F.10.7.2#1 The **floor** functions

The **floor** functions round "toward negative infinity", choosing only from among integer values of the parameter type.

- **floor**(± 0) returns ± 0 .
- **floor**($\pm\infty$) returns $\pm\infty$.

F.10.7.2#2 The **floor** functions

The ~~returned-value~~ **result** is exact and is independent of the ~~current~~ rounding direction modes.

F.10.7.3#1 The **nearbyint** functions

~~The **nearbyint** functions use ISO/IEC 60559 rounding according to the current rounding direction.~~
The **nearbyint** functions round according to the applicable rounding direction mode, choosing only from among integer values of the parameter type.

- **nearbyint** (± 0) returns ± 0 .
- **nearbyint** ($\pm \infty$) returns $\pm \infty$.

~~They do not raise the "inexact" floating-point exception if the result differs in value from the argument.~~

F.10.7.3#2 The **nearbyint** functions - NEW paragraph

Even if the result differs in value from the argument, the "inexact" floating-point exception is not raised.

F.10.7.4#1 The **rint** functions

~~The **rint** functions differ from the **nearbyint** functions only in that they do raise the "inexact" floating-point exception if the result differs in value from the argument.~~

The **rint** functions round according to the applicable rounding direction mode, choosing only from among integer values of the parameter type.

- **rint** (± 0) returns ± 0 .
- **rint** ($\pm \infty$) returns $\pm \infty$.

F.10.7.4#2 The **rint** functions - NEW paragraph

If the result differs in value from the argument, the "inexact" floating-point exception is raised.

F.10.7.5#1 The **lrint** and **llrint** functions

The **lrint** and **llrint** functions provide floating-to-integer conversion, ~~as prescribed by ISO/IEC 60559.~~
~~They round~~ rounding according to the ~~current rounding direction.~~ applicable rounding direction mode. If the rounded value is outside the range of the return type, the ~~numeric~~ result is unspecified and the "invalid" floating-point exception is raised. ~~When they raise no other floating-point exception and the result differs~~ If the rounded value is representable within the range of the return type, the result is that value cast to the return type and, if this result differs in value from the argument, ~~they raise~~ the "inexact" floating-point exception is raised.

F.10.7.6#1 The **round** functions

The **round** functions round "to nearest with ties away from zero", choosing only from among integer values of the parameter type.

- **round** (± 0) returns ± 0 .
- **round** ($\pm \infty$) returns $\pm \infty$.

F.10.7.6#2 The **round** functions

The ~~returned value~~ result is exact and is independent of the ~~current~~ rounding direction modes.

F.10.7.7#1 The **lround** and **llround** functions

The **lround** and **llround** functions provide floating-to-integer conversion, ~~differ from the **lrint** and **llrint** functions with the default rounding direction just in that the **lround** and **llround** functions round halfway cases away from zero and are not required to raise the "inexact" floating-point exception for non-integer arguments that round to within the range of the return type.~~ rounding "to nearest with ties away from zero". If the rounded value is outside the range of the return type, the result is unspecified and the "invalid" floating-point exception is raised. If the rounded value is representable within the range of the return type, the result is that value cast to the return type, and no floating-point exception is raised.

F.10.7.8#1 The **roundeven** functions

The **roundeven** functions round "to nearest with ties to even", choosing only from among integer values of the parameter type.

- **roundeven**(± 0) returns ± 0 .
- **roundeven**($\pm\infty$) returns $\pm\infty$.

F.10.7.8#2 The **roundeven** functions

The **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.7.9#1 The **trunc** functions

The **trunc** functions round "toward zero", choosing only from among integer values of the parameter type.

- **trunc**(± 0) returns ± 0 .
- **trunc**($\pm\infty$) returns $\pm\infty$.

F.10.7.9#2 The **trunc** functions

The **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.7.10#1 The **fromfp** and **ufromfp** functions

The **fromfp**(x , rnd , $width$) and **ufromfp**(x , rnd , $width$) functions return a NaN and raise the "invalid" floating-point exception ~~and return a NaN if the argument $width$ is zero or if the floating-point argument x is infinite or NaN or if $width = 0$ or $x = \pm\infty$ or x is a quiet NaN or x rounds to an integer integral value that is outside the range determined by $width$. the argument $width$ (see 7.12.10.10).~~

F.10.7.11#1 The **fromfpx** and **ufromfpx** functions

The **fromfpx**(x , rnd , $width$) and **ufromfpx**(x , rnd , $width$) functions return a NaN and raise the "invalid" floating-point exception ~~and return a NaN if the argument $width$ is zero or if the floating-point argument x is infinite or NaN or if $width = 0$ or $x = \pm\infty$ or x is a quiet NaN or x rounds to an integer integral value that is outside the range determined by $width$. the argument $width$ (see 7.12.10.11).~~

F.10.7.11#2 The **fromfpx** and **ufromfpx** functions

..... differs in value from ~~the floating-point argument x~~ . [Note that the font of x is italic]

F.10.8.1#1 The **fmod** functions

- **fmod**(± 0 , y) return ± 0 for ~~y not zero~~, $y < 0$ or $y > 0$.
- **fmod**(x , y) returns a NaN and raises the "invalid" floating-point exception for ~~x infinite or y zero (and neither is a NaN)~~ $x = \pm\infty$ or $y = 0$ (and neither is a quiet NaN).
- **fmod**(x , $\pm\infty$) return x for ~~finite x~~ $-\infty < x < +\infty$.

F.10.8.1#2 The **fmod** functions

When subnormal results are supported, the **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.8.2#1 The **remainder** functions

- **remainder**(± 0 , y) return ± 0 for ~~y not zero~~, $y < 0$ or $y > 0$.
- **remainder**(x , y) returns a NaN and raises the "invalid" floating-point exception for ~~x infinite or y zero (and neither is a NaN)~~ $x = \pm\infty$ or $y = 0$ (and neither is a quiet NaN).
- **remainder**(x , $\pm\infty$) return x for ~~finite x~~ $-\infty < x < +\infty$.

F.10.8.2#2 The **remainder** functions

When subnormal results are supported, the **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.8.3#2 The **remquo** functions

When subnormal results are supported, the **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.9.1#1 The **copysign** functions

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

F.10.9.2#2 The **nan** functions

~~The returned-value is exact and is~~ The **nan** functions raise no floating-point exceptions. Their results are independent of the **current** rounding direction modes.

F.10.9.3#1 The **nextafter** functions

- **nextafter**(x , y) raises the "overflow" and "inexact" floating-point exceptions ~~for x finite and the function value infinite when $-\infty < x < +\infty$ and the result is infinite.~~
- **nextafter**(x , y) raises the "underflow" and "inexact" floating-point exceptions ~~for the function value subnormal or zero and $x \neq y$ when $x \neq y$ and the result is subnormal or zero.~~

F.10.9.3#2 The **nextafter** functions

Even though ~~underflow or overflow can occur,~~ the "overflow" or "underflow" floating-point exception can be raised, the **returned-value result** is independent of the **current** rounding direction modes.

F.10.9.4#1 The **nexttoward** functions

These functions have the same ~~No additional~~ special cases and requirements ~~beyond those on~~ as **nextafter**.

F.10.9.4#2 The **nexttoward** functions

~~Even though underflow or overflow can occur, the returned-value is independent of the current rounding direction mode.~~

F.10.9.5#2 The **nextup** functions

[correct the font of the function parameter to be italic +++ other stuff]

nextup($*x$) raises no floating-point exceptions if $-\infty \leq x \leq +\infty$ or x is a quiet NaN. ~~x is not a signaling NaN.~~ The **returned-value result** is independent of the **current** rounding direction modes.

F.10.9.6#2 The **nextdown** functions

[correct the font of the function parameter to be italic +++ other stuff]

nextdown($*x$) raises no floating-point exceptions if $-\infty \leq x \leq +\infty$ or x is a quiet NaN. ~~x is not a signaling NaN.~~ The **returned-value result** is independent of the **current** rounding direction modes.

F.10.9.7#1 The **canonicalize** functions

All floating-point values have a unique canonical representation in ISO/IEC 60559 formats (5.3.5.3.3).

1. **canonicalize**(cx , x) stores the canonical representation of $*x$ in the object pointed to by cx if $-\infty \leq *x \leq +\infty$ or $*x$ is a quiet NaN.
2. **canonicalize**(cx , x) stores a canonical representation of a quiet NaN in the object pointed to by cx and raises the "invalid" floating-point exception if $*x$ is a signaling NaN.

The **canonicalize** functions produce ⁴¹²⁾ the canonical version of the representation in the object pointed to by the argument x . If the input $*x$ is a signaling NaN, the "invalid" floating-point exception is raised and a (canonical) quiet NaN (which should be the canonical version of that signaling NaN made quiet) is produced. For quiet NaN, infinity, and finite inputs, the functions raise no floating-point exceptions.

⁴¹²⁾ ~~As if $*x * 1e0$ were computed.~~ *[Delete the footnote]*

F.10.9.7#2 The **canonicalize** functions

[*New paragraph*]

Recommended practice

The representation stored in the object pointed to by *cx* should be the "quieted" canonical representation of **x* if **x* is a signaling NaN. See ISO/IEC 60559 for details on how signaling NaNs are "quieted".

F.10.10.2#2 The **fmax** functions

The **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.10.3#2 The **fmin** functions

The **returned-value result** is exact and is independent of the **current** rounding direction modes.

F.10.12#1 Functions that round to narrower type

~~The functions that round their result to narrower type (7.12.15) are fully specified in ISO/IEC 60559.~~ The **returned-value result** is dependent on the **current applicable** rounding direction mode.

F.10.13.2#2 The **totalorder** functions

The **totalorder** functions determine whether the total ordering relationship ~~, defined by ISO/IEC 60559,~~ is true for the ordered pair of **x*, **y*. ~~These functions are~~ The total ordering relationship is fully specified ~~in~~ by ISO/IEC 60559. These functions ~~are independent of the current rounding direction mode and~~ raise no floating-point exceptions, even if **x* or **y* is a signaling NaN. ~~Their results are independent of the rounding direction modes.~~

[*Like other sub-clauses, mention floating-point exceptions before rounding direction modes*]

F.10.13.3#2 The **totalordermag** functions

The **totalordermag** functions determine whether the total ordering relationship ~~, defined by ISO/IEC 60559,~~ is true for the ordered pair of the magnitudes ~~of~~ **x*, **y*. ~~These functions are~~ The total ordering relationship is fully specified ~~in~~ by ISO/IEC 60559. These functions ~~are independent of the current rounding direction mode and~~ raise no floating-point exceptions, even if **x* or **y* is a signaling NaN. ~~Their results are independent of the rounding direction modes.~~

[*See comments in prior sub-clause*]

F.10.13.3#3 The **totalordermag** functions

[*Use the same wording as seen in the previous paragraph*]

The **totalordermag** functions return nonzero if and only if the total ordering relation is true for the ordered pair of the magnitudes ~~of~~ **x*, **y*. *|*x|*, *|*y|*.

F.10.15 Quantum functions

F.10.15#1 The **samequantum** functions

samequantum(*x*, *y*) raises no floating-point exceptions, even if *x* or *y* is a signaling NaN.

F.10.16 Comparison macros - RENUMBERED - was F.10.15

F.10.16.1 General - RENUMBERED - was F.10.15.1

F.10.16.2 The **iseqsig** macro - RENUMBERED - was F.10.15.2