Floating-Point and Complex Arithmetic Enhancements

WG14/N657 X3J11/97-020 (Draft 1997-02-07)

Jim Thomas
jthomas@best.com

This documents describes potential C9X enhancements related to the floating-point and complex arithmetic specifications. Change bars mark changes from document N638 in the pre-Kona mailing. Section numbers refer to C9X Draft 8. Each enhancement could be accepted or rejected independently.

1. Enhancement: Allow as an implementation-defined option the `feraiseexcept` function in <fenv.h> to raise the inexact exception whenever it raises overflow or underflow.

Rationale: On some architectures the only practical way to raise an exception is to execute an instruction that has the exception as a side-effect. Any IEC 559 operation that raises either overflow or underflow raises inexact too. This should have little effect on use (except for tests).

Edits: *Add to the Description in 7.x.2.3 of the <fenv.h> section the sentence:* "Whether the `feraiseexcept` function additionally raises the inexact exception whenever it raises the overflow or underflow exception is implementation-defined."

2. Enhancement: Provide an inquiry macro, `isinf(x)`, to test whether `x` is an infinity.

Rationale: It's frequently needed (e.g. in the sample code for complex multiply and divide). For some uses, performance will be important and obvious implementations, e.g. `fabsl(x)==INFINITY`, may be far from optimal. It's possible for the user to get it wrong, e.g. `!isfinite(x)` would be wrong. There are already inquiries, `isnan`, `isfinite`, and `isnormal` to model after. There's shipping prior art. (`isinfinite` might have been better style).

Edits: *Add the following subclause in 7.6.2 of the <math.h> section:*

7.x.2.y The `isinf` macro

Synopsis

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

Description

The `isinf` macro determines whether its argument value is an infinity (positive or negative). 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 `isinf` macro returns a nonzero value if and only if its argument value is an infinity.

3. Enhancement: Provide a function for *floating multiply-add*: `fma(x, y, z)` computes `x * y + z` as one ternary operation (i.e. with just one rounding). IEC 559 implementations must deliver the result with just one IEC 559-style rounding.

Rationale: In many cases, clever use of floating multiply-add leads to much improved code. However, its unexpected use by the compiler can undermine carefully written code. The `FP_CONTRACT` directive in the current floating-point spec can be used to disallow use of floating multiply-add. What's still needed is a way to guarantee its use just in particular situations. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

Edits: *Insert the following subclause in 7.6 in the <math.h> section:*

7.6.y Floating multiply-add

7.6.y.1 The `fma` function

Synopsis

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

Description

The `fma` function computes the sum `z` plus the product `x` times `y`, rounded as one ternary operation: it computes the sum `z` plus the product `x` times `y` (as if) to infinite precision and rounds once to the result format, according to the rounding mode characterized by the value of `FLT_ROUNDS`.

Returns

The `fma` function returns the sum `z` plus the product `x` times `y`, rounded as one ternary operation.

[A longer alternative name might be `fmuladd`.]

Insert the following subclause in F.9 of the IEC 559 annex:

F.9.y Floating multiply-add

F.9.y.1 The `fma` function

`fma(x, y, z)` computes the sum `z` plus the product `x` times `y`, correctly rounded once.

- `fma(x, y, z)` returns one of its NaN arguments and raises no exception if `x` or `y` is a NaN.
- `fma(x, y, z)` returns a NaN and optionally raises the invalid exception if one of `x` and `y` is infinite, the other is zero, and `z` is a NaN.
- `fma(x, y, z)` returns a NaN and raises the invalid exception if one of `x` and `y` is infinite, the other is zero, and `z` is not a NaN.
- `fma(x, y, z)` returns a NaN and raises the invalid exception if `x` times `y` is an exact infinity and `z` is also an infinity but with the opposite sign.

Include fma in the list of <math.h> (but not <complex.h>) functions that have a type-generic macro.

In 7.6, paragraph [1], sentence [2], change "one or two" *to* "one or more".

4. Enhancement: Change the corresponding real type for the imaginary unit from the narrowest floating type for expression evaluation to the `float` type.

Rationale: This change assures reasonable presumable properties such as

```
sizeof(I * y) == sizeof(y)
sizeof(z) == sizeof(creal(z) + I * cimag(z))
```

It enhances consistency in that use of `I` for algorithmic or notational convenience will not result in widening types. Without the change, codes that depend on the semantic type of imaginary or complex expressions would potentially encounter problems when ported between implementations using different expression evaluation methods. Although this might not be a big problem, there appears to be no benefit in retaining the current spec.

Edits: *In 7.7 Complex arithmetic, replace* "... whose corresponding real type is the narrowest real floating type used for expression evaluation[1]" *with* "... whose corresponding real type is `float`", *removing the footnote.*

5. Enhancement: Make `imaginary` a keyword token in the normative standard. Like `complex`, it would become a keyword just in translation units that include **<complex.h>**. The imaginary types per se would remain non-normative.

Rationale: This allows implementing the complex arithmetic (informative) annex without breaking conformance with the normative standard.

Edits: *In 6.1.1 [2], add* `imaginary` *to the list of keyword tokens.*

Replace 6.1.1 [3] with:

Where reserved, the above tokens (entirely in lowercase) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise. The keyword tokens `complex` and `imaginary` are reserved in translation units where the header **<complex.h>** is included, but not otherwise; if one of the tokens `complex` and `imaginary` is used prior to the first inclusion of the header, the behavior is undefined. All other keyword tokens are reserved in all translation units.

---

[1] If `FLT_EVAL_METHOD` equals 0, 1, or 2, then `_Imaginary_I` has the corresponding real type `float`, `double`, or `long double`, respectively.

Remove G.2 from the complex arithmetic annex (and renumber subsequent subclauses).

At the beginning of the first paragraph in G.3 (before renumbering), insert the sentence: "Inclusion of the header `<complex.h>` enables designations for imaginary types."

6. Enhancement: Replace the current `scalb(double, long)` function with the pair of functions `scalbn(double, int)` and `scalb(double, double)`, and provide the `int` function `ilogb` in addition to the current `double` function `logb`.

Rationale: `scalb(double, double)` and `ilogb` are in XPG4 V2 and are widely implemented. Some applications are better served by a floating second argument to `scalb`[`n`], others by an integral one, and similarly regarding the return type for [`i`]`logb`. The transition problems for early C9X implementors and users is mitigated by the fact that (almost all) calls to `scalb(double, long)` would still work (after recompilation) when silently replaced by calls to `scalb(double, double)`.

Edits: *Change the name of the current scalb function to:* `scalbn`. *Replace the synopsis for the old scalb function with:*

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

Throughout, replace references to the old ***`scalb`*** *function with ones to* ***`scalbn`****.*

Insert the following subclause in 7.6.6 of the <math.h> section:

7.6.6.y The `scalb` function

Synopsis

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

Description

The `scalb` function computes `x * FLT_RADIX`$^{\mathtt{y}}$, for integral values `y`. If `y` is nonintegral, the result is implementation-defined. A range error may occur.

Returns

The `scalb` function returns the value of `x * FLT_RADIX`$^{\mathtt{y}}$, for integral values `y`.

Insert the following subclause in F.9.3:

F.9.3.y The `scalb` function

- `scalb(-x, y)` and `-scalb(x, y)` are equivalent.
- `scalb(x, y)` returns a NaN if `y` is nonintegral.
- `scalb(x,` ±0`)` returns `x`.
- `scalb(`+0`,` +∞`)` returns a NaN and raises the invalid exception.

- `scalb(+0, y)` returns +0, for integral `y` < +∞.
- `scalb(+∞, y)` returns +∞, for integral `y` > -∞.
- `scalb(x, -∞)` returns +0, for finite positive-signed `x`.
- `scalb(x, +∞)` returns +∞, for positive `x`.
- `scalb(+∞,-∞)` returns a NaN.
- `scalb(x, y)` returns a NaN, if `x` or `y` is a NaN.

Include scalb in the list of <math.h> (but not <complex.h>) functions that have a type-generic macro.

Insert the following subclause in 7.6.6:

7.6.6.y The `ilogb` function

Synopsis

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

Description

The `ilogb` function extracts the exponent of `x` as a signed `int` value. It is equivalent to `(int)logb(x)`, for finite nonzero `x`; it computes the value `INT_MIN` if `x` is zero; and it computes the value `INT_MAX` if `x` is infinite or NaN.

[This specification differs from both X/Open and Sun's fdlibm (for which X/Open granted a permanent waiver):

```
X/Open  ilogb(0) =  INT_MIN, ilogb(NaN) = INT_MIN, ilogb(+/-Inf) = INT_MAX
fdlibm  ilogb(0) = -INT_MAX, ilogb(NaN) = INT_MAX, ilogb(+/-Inf) = INT_MAX
C9X     ilogb(0) =  INT_MIN, ilogb(NaN) = INT_MAX, ilogb(+/-Inf) = INT_MAX
```

The value `INT_MAX` for both NaN and infinite arguments is more convenient for both implementation and use; the implementation can simply test for a maximum exponent; the user can test for `INT_MAX` to determine the argument is not a finite number. `INT_MIN` is more natural than the expression `-INT_MAX`, for testing for a zero argument.]

Returns

The `ilogb` function returns the exponent of `x` as a signed `int` value.

If enhancement [6] but not [12] is accepted, in 7.6, paragraph [1], insert after the second sentence:

For the functions `ilogb`, there are functions `ilogbf` and `ilogbl` which are corresponding functions with `float` and `long double` arguments.

7. Enhancement: Identify `x / x` --> `1.0` as an invalid optimizations, in the IEC 559 annex.

Rationale: Some compilers do this optimization, which is invalid for IEC 559 implementations if `x` can be zero, infinite, or NaN.

Edits: *In F.8.2, add the item*

`x / x` --> `1.0`	The expressions `x / x` and `1.0` are not equivalent if `x` can be zero, infinite, of NaN.

8. The proposal to provide the `cis` function, defined by cis(x) = cos(x) + i*sin(x), to `<complex.h>` is being withdrawn, based on insufficient need: `cis(x)` should be equivalent to `cexp(I*x)`. Rationale might encourage implementors to optimize for this use.

9. Enhancement: Provide the `cbrt` (cube root) function.

Rationale: It is specified in XPG4 V2 and is widely implemented. For some applications, a true cube root function, which returns negative results for negative arguments, is more appropriate than `pow(x, 1.0/3.0)`, which returns NaN for `x` < 0.

The proposal for Bessel functions, also specified by XPG4 V2, is being withdrawn, because they're believed to be insufficiently useful.

Edits: *In 7.6.7, add the subclause:*

7.6.7.x The `cbrt` function

Synopsis

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

Description

The `cbrt` function computes the real cube root of `x`.

Returns

The `cbrt` function returns the value of the cube root.

Insert the following subclause in F.9.4:

F.9.4.y The `cbrt` function

- `cbrt(`+∞`)` returns +∞.
- `cbrt(`-∞`)` returns -∞.
- `cbrt(`+0`)` returns +0.
- `cbrt(`-0`)` returns -0.

Include cbrt in the list of <math.h> (but not <complex.h>) functions that have a type-generic macro.

Include cbrt in the list of function names in item [11] below.

10. The proposal to provide the foundational financial/growth functions (`compound` and `annuity`) is being withdrawn. They don't seem to amount to a sufficiently large contribution to this important application area (and I don't have the time to put together a more comprehensive solution).

11. Enhancement: Reserve the `c`-prefixed names of math functions that could have complex counterparts but don't in C9X.

Rationale: This enhancement reserves the names for future library directions, and provides guidance for extensions. A straw vote in Toronto, taken during the discussion of type-generic math functions, strongly favored reserving such names.

Edits: *In 7.18.4, append the paragraph:*

The function names

```
exp2
expm1
log10
log1p
log2
erf
erfc
gamma
lgamma
```

and the same names suffixed with `f` or `l`, all prefixed with `c`, are reserved for corresponding functions with complex arguments and return values.

12. Enhancement: Change the prototypes

```
long rinttol(long double);
long long rinttoll(long double);
long roundtol(long double);
long long roundtoll(long double);
```

to

```
long lrint(double);
long long llrint(double);
long lround(double);
long long llround(double);
```

The existence of `float` and `long double` versions for the new functions is implicit.

Rationale: There are two changes here. First, the parameter type changes to `double`, matching the X/Open function `ilogb`, which like these functions returns an integer-type result; this makes the interface style more consistent. Second, the names change to make way for `f` and `l` suffixed versions of the functions, which become needed because of the first change: else `rinttoll` could be either the `double` version of the `long long` function or the `long double` version of the `long` function. Although this enhancement is independent of enhancement [6] above, its rationale would be somewhat stronger if enhancement [6] were accepted.

For functions with a floating argument and an integer return type, the current C9X specification took the approach of declaring the parameter to be **`long double`**. The rationale was to avoid unnecessary multiple versions of the function in the interface. The implementation need not actually promote a **`float`** or **`double`** argument to **`long double`**, so any potential inefficiency could be avoided.

However, with the current interface a programmer is left to worry about the risk of incurring a costly promotion to **`long double`**. Also, the proposed enhancement seems more consistent with the rest of the interface, where all the other functions come in three sizes; a programmer might initially be surprised not to find **`float`** and **`double`** versions.

Edits: *In 7.6.9.6, 7.6.9.7, 7.6.9.9, 7.6.9.10, F.9.6.5, and F.9.6.7, change the subclause titles, synopses prototypes, and all references to reflect the function name and parameter type changes indicated above.*

Globally find and replace the names as indicated above.

In 7.6, paragraph [1], insert after the second sentence:

Some synopses specify a function which takes a **`double`** argument and returns an integer-type 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.

13. Enhancement: Fix the complex division sample to cover a case that was overlooked.

Rationale: The current version does not handle the infinity/zero case correctly. In particular, the first calculation of **`x`** and **`y`** yields NaN+iNaN, then execution goes into the **`if ((isinf(a) ... )`** block, but comes out again as NaN+iNaN, instead of an infinity.

Edits: Replace example [2] in G.5.1 with:

```
#include <math.h>
#include <complex.h>
/* isinf is as in example 1 above */

/* Divide z / w ... */
double complex _Cdivd(double complex z, double complex w)
{
      FP_CONTRACT_OFF
      double a, b, c, d, logbw, denom, x, y;
      long llogbw = 0;
      a = creal(z); b = cimag(z);
      c = creal(w); d = cimag(w);
      logbw = logb(fmax(fabs(c), fabs(d)));
      if (isfinite(logbw)) {
         llogbw = (long)logbw;
         c = scalb(c, -llogbw);
         d = scalb(d, -llogbw);
      }
      denom = c * c + d * d;
      x = scalb((a * c + b * d) / denom, -llogbw);
      y = scalb((b * c - a * d) / denom, -llogbw);
```

```
        /*
         * Recover infinities and zeros that computed as NaN+iNaN;
         * the only cases are non-zero/zero, infinite/finite, and
         * finite/infinite, ...
         */
        if (isnan(x) && isnan(y)) {
           if ((denom == 0.0) && (!isnan(a) || !isnan(b))) {
              x = copysign(INFINITY, c) * a;
              y = copysign(INFINITY, c) * b;
           }
           else if ((isinf(a) || isinf(b)) && isfinite(c) && isfinite(d)) {
              a = copysign(isinf(a) ? 1.0 : 0.0, a);
              b = copysign(isinf(b) ? 1.0 : 0.0, b);
              x = INFINITY * (a * c + b * d);
              y = INFINITY * (b * c - a * d);
           }
           else if (isinf(logbw) && isfinite(a) && isfinite(b)) {
              c = copysign(isinf(c) ? 1.0 : 0.0, c);
              d = copysign(isinf(d) ? 1.0 : 0.0, d);
              x = 0.0 * (a * c + b * d);
              y = 0.0 * (b * c - a * d);
           }
        }
        return x + I * y;
}
```

14. Enhancement: Change the special cases for `cexp` to be more consistent with the other complex functions.

Rationale: Some of these changes give the implementation latitude with respect to the sign of infinity or zero where the sign isn't meaningfully determined by the argument, which is the approach for the other complex functions. The other changes reflect the fact that `cexp` maps the real axis to the real axis; these also follow the approach for other functions (e.g. `csinh`).

Edits: *In G.5.10, replace these five bullets:*

- `cexp`(-∞+i∞) returns +0+i0.
- `cexp`(+∞+i∞) returns +∞+iNaN and raises the invalid exception.

...

- `cexp` (-∞+iNaN) returns +0±i0 (where the sign of the imaginary part of the result is unspecified).
- `cexp` (+∞+iNaN) returns +∞+iNaN.
- `cexp` (NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all numbers y.

with these six bullets:

- `cexp`(-∞+i∞) returns ±0±i0 (where the signs of the real and imaginary parts of the result are unspecified).
- `cexp`(+∞+i∞) returns ±∞+iNaN and raises the invalid exception (where the signs of the real and imaginary parts of the result are unspecified).

...

- **cexp** (-∞+iNaN) returns ±0±i0 (where the signs of the real and imaginary parts of the result are unspecified).
- **cexp** (+∞+iNaN) returns ±∞+iNaN (where the sign of the real part of the result is unspecified).
- **cexp** (NaN+i0) returns NaN+i0.
- **cexp** (NaN+iy) returns NaN+iNaN and optionally raises the invalid exception, for all nonzero numbers y.