Floating-Point and Complex Arithmetic Enhancements

WG14/N638 X3J11/97-001 (Draft 1997-01-03)

Jim Thomas
jthomas@best.com

This documents describes potential C9X enhancements related to the floating-point and complex arithmetic specifications. It is a revision of the document N619 introduced as a Monday paper and discussed in Toronto. 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.x.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.x in the <math.h> section:*

7.x.y Floating multiply-add

7.x.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 X.9 of the IEC 559 annex:

X.9.y Floating multiply-add

X.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.

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.x 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.

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

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

At the beginning of the first paragraph in Y.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, long)** 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 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**. *Throughout, replace references to the old* ***scalb*** *function with ones to* ***scalbn***.

Insert the following subclause in 7.x.6 of the <math.h> section:

7.x.6.y The scalb function

Synopsis

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

Description

The **scalb** function computes **x * FLT_RADIX**$^{\mathbf{y}}$, 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**$^{\mathbf{y}}$, for integral values **y**.

Insert the following subclause in X.9.3:

X.9.y.1 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, +INFINITY)** returns a NaN and raises the invalid exception.
- **scalb(+0, y)** returns +0, for integral **y** < **+INFINITY**.
- **scalb(+INFINITY, y)** returns **+INFINITY**, for integral **y** > **-INFINITY**.
- **scalb(x, -INFINITY)** returns +0, for finite positive-signed **x**.
- **scalb(x, +INFINITY)** returns **+INFINITY**, for positive **x**.
- **scalb(+INFINITY, -INFINITY)** 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.x.6:

7.x.6.y The `ilogb` function

Synopsis

```
#include <math.h>
int ilogb(long 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 or NaN; and it computes the value **`INT_MAX`** if **`x`** is infinite.

[The special cases follow the X/Open specification, though returning the same value for NaN arguments as for infinite ones would be more natural from an implementation point of view.

For functions with a floating argument and an integer return type, C9X has taken the approach of declaring the parameter to be **`long double`**. The rationale is to avoid unnecessary multiple versions of the function. The implementation need not actually promote a **`float`** or **`double`** argument to **`long double`**, so any potential inefficiency can be avoided. For **`ilogb`** this leads to a minor inconsistency with X/Open which declares the parameter to be **`double`**. This difference should have no affect on real X/Open programs.]

Returns

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

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 X.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. Enhancement: Provide the **`cis`** function, defined by cis(x) = cos(x) + i*sin(x), to **`<complex.h>`**.

Rationale: The obvious implementation is correct, which suggests leaving it for the user to implement. However, most implementations do or can provide a way of obtaining both the cosine and sine (which are frequently needed in tandem) more efficiently than with two function calls. Having a standard library function provides a portable way for programs to benefit from this effort.

Edits: *Add to 7.x.2 in <complex.h> the subclause:*

7.x.2.y The cis function

Synopsis

```
#include <complex.h>
double complex cis(double x);
```

Description

The **cis** function computes cos(**x**) + i*sin(**x**).

Returns

The **cis** function returns the value of cos(**x**) + i*sin(**x**).

Add cis to the list of type-generic macros for <complex.h> functions.

[Longer alternative names might be **cosisin** or, since the expression is equivalent to e^{ix}, **cexpi**.

A different approach would be to provide in **<math.h>** a function for obtaining both the sine and cosine, e.g.

```
double sincos(double x, double* cosptr);
```

or

```
void sincos(double x, double* sinptr, double* cosptr);
```

Prior art might suggest other possibilities. This approach would have the advantage of not bringing in **<complex.h>** (some clients of this facility would not be modeling with complex numbers), but the call itself might be less efficient.]

9. Enhancement: Provide the math functions: cube root (**cbrt**), Bessel functions of the first kind (**j0**, **j1**, **jn**), Bessel functions of the second kind (**y0**, **y1**, **yn**), and reentrant log gamma (**lgamma_r**).

[Longer alternative names might be

```
bslj0  bslj1  bsljn  bsly0  bsly1  bslyn
```

or

```
j0bsl  j1bsl  jnbsl  y0bsl  y1bsl  ynbsl
```

]

Rationale: These are specified in XPG4 V2 and are widely implemented.

Edits: *Add specification for the functions to the* **<math.h>** *spec and to the IEC 559 annex, and add the names to the list of type-generic macros.*

10. Enhancement: Provide foundational financial/growth functions: (1+r)^n (**`compound`**) and (1 - (1 + r)^(-n)) / r (**`annuity`**).

Rationale: These functions are key to implementation of several other financial functions. Straight forward implementation loses needed accuracy for some arguments. Shipping prior art has been around for years.

Edits: *Add specification for the functions to the* **`<math.h>`** *spec and to the IEC 559 annex, and add the names to the list of type-generic macros.*

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 7.14.4, 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.