Floating-Point and Complex Arithmetic Enhancements

WG14/N619 X3J11/96-83 (Draft 10/15/96)

Jim Thomas
jthomas@best.com

This documents describes potential enhancements for the floating-point and complex arithmetic specifications. 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 *fused multiply-add*: **`fma(x,y,z)`** computes **`x * y + z`** (as if) as one ternary operation. IEC 559 implementations must deliver the result with just one IEC 559-style rounding.

Rationale: In many cases, clever use of fused multiply-add leads to much improved code. However, unexpected use of fused multiply-add 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 fused multiply-add. What's still needed is a way to guarantee its use just in particular situations. Many current machines provide hardware fused 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 Fused 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 `x * y + z`, rounded as one ternary operation.

Returns

The `fma` function returns the value of `x * y + z`, rounded as one ternary operation.

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

X.9.y Fused multiply-add

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

`fma(x,y,z)` computes `x * y + z`, correctly rounded (IEC 559 style) once. The function treats infinities, NaNs, signed zeros, and subnormals in a manner consistent with the basic arithmetic operations covered by IEC 559.

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 facilitates portable use of expressions involving `I` as arguments to `printf`, without regard to the expression evaluation method. For example,

```
double y;
...
printf("%e", I*y);
```

would be erroneous with the current spec if the implementation uses long double expression evaluation.

Edits: *In 7.x in the complex arithmetic spec, 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: Change `imaginary` from a keyword to a macro and reserve it in the manner of other library macros.

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

Edits: *Remove section Y.2.*

At the beginning of (old) Y.3 add the paragraph:

Inclusion of the header `<complex.h>` enables designations for imaginary types. The header defines the macro

```
imaginary
```

At the beginning of (old) Y.6 add:

The header `<complex.h>` defines the macro

```
imaginary
```

[The implementation might define `imaginary` to be a keyword in the implementation's namespace.]

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 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:

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

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`**$^{\mathtt{y}}$, 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 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`**, computes the value **`INT_MIN`** if **`x`** is zero, and computes the value **`INT_MAX`** if **`x`** is infinite or NaN.

[**`ilogb`**(NaN) = **`INT_MAX`** matches the fdlibm implementation, not XPG4 V2, which specifies **`INT_MIN`**; **`INT_MAX`** is more natural in some implementations.]

Returns

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

7. Enhancement: Add `x / x` --> `1.0` to the list of invalid optimizations, in the new example added as part of the floating-point spec.

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

Edits: *To the new example in 5.1.2.3, add the line*

```
y = x / x;     /* not equivalent to y = 1.0; */
```

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

Rationale: Although the obvious implementation is correct (which would suggest leaving it for the user to implement), having a library function might promote more efficient implementation.

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.

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`).

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.*

Other ideas:

Use an encoding of the standardization date for the value of **`__STD_IEC_559__`**. (Fred?)

Use new alternate form of pragmas for translation directives. (Fred?)

Remove use of trailing semicolons with the translation directive macros. (Douglas?)

Add the requirement that some form of error/exception reporting be implemented. (Fred?)

Provide the XPG4 V2 double-precision math constants.

Provide a **`significand`** function to compute scalb(x, (long) logb(x)).

Clarify that **`-z`** negates real and imaginary parts. (rationale?)