Annex X
(informative)
IEEE standard floating-point arithmetic

WG14/N469 X3J11/95-070 (Draft 8/15/95)

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233
`jim_thomas@taligent.com`

X.1 Introduction

This annex specifies C language support for the IEEE floating-point standard. The *IEEE floating-point standard* is specifically *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985), AKA *Binary floating-point arithmetic for microprocessor systems* (IEC 559:1989). *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987) generalizes ANSI/IEEE Std 754-1985 to remove dependencies on radix and word length. *IEEE* generally refers to the floating-point standard, as in IEEE operation, IEEE format, etc..

X.2 Types

The C floating types match the IEEE formats as follows:

— The C `float` type matches the IEEE single format.

— The C `double` type matches the IEEE double format.

— The C `long double` type matches the IEEE extended format[1], else a non-IEEE extended format, else the IEEE double format.

Any non-IEEE extended format used for the `long double` type has more precision than IEEE double and at least the range or IEEE double.

> [The primary objective of this specification is to facilitate writing portable code that exploits the IEEE standard, including its standardized single and double data formats. Bringing the C data types and the IEEE formats into line advances this objective.
>
> Minimal conformance to the IEEE standard does not require a format wider than single. The narrowest C `double` type allowed by Standard C (ISO 5.2.4.2) is wider than IEEE single, and wider than the minimum IEEE *single-extended* format. (IEEE single-extended is an optional format intended only for those implementations that don't support double; it has at least 32 bits of precision.) Both Standard C and the IEEE standard would be satisfied if `float` were IEEE single and `double` were an IEEE single-extended format with at least 35 bits of precision. However, this specification goes slightly further by requiring `double` to be IEEE double rather than just a wide IEEE single-extended.

[1] *Extended* is the IEEE standard's *double-extended* data format. Extended refers to both the common 80-bit and so-called *quadruple* 128-bit IEEE formats.

Because of Standard C's bias toward `double`, extended-based architectures might appear to be better served by associating the C `double` type with IEEE extended. However, such an approach would not allow standard C types for both IEEE double and single and would go against current industry naming, in addition to undermining this specification's portability goal. The type definitions `float_t` and `double_t` in `<fp.h>`, are intended to allow effective use of architectures with more efficient, wider formats.

The `long double` type is not required to be IEEE extended because

1. some of the major IEEE architectures for C implementations do not support extended,
2. double precision is adequate for a broad assortment of numerical applications, and
3. extended is less standard than single or double in that only bounds for its range and precision are specified in IEEE standard 754.

For implementations without extended in hardware, non-IEEE extended arithmetic written in software, exploiting double in hardware, provides some of the advantages of IEEE extended but with significantly better performance than true IEEE extended in software. [???] explains advantages of extended precision.

What to do about bibliography references?

This specification accommodates what are expected to be the most important IEEE architectures for general C implementations—see Rationale 5.2.4.2.2.

Specification for a variable-length extended type—one whose width could be changed by the user—was deemed premature. However, not unduly encumbering experimentation and future extensions, for example for variable length extended, is a goal of this specification.

Some narrow-format C implementations, namely ones for digital signal processing, provide only the IEEE single format, possibly augmented by single-extended, which may be narrower than IEEE double or Standard C `double`, and possibly further augmented by double in software. These non-conforming implementations might generally adopt this specification, though not matching its requirements for types.

One approach would be: match Standard C `float` with single; match Standard C `double` with single-extended, else single; and match Standard C `long double` with double, else single-extended, else single. Then most of this specification could be applied straightforwardly. Users should be clearly warned that the types may not meet expectations.

Another approach would be to refer to a single-extended format as `long float` and then not recognize any C types not truly supported. This would provide ample warning for programs requiring double. The translation part of porting programs could be accomplished easily with the help of type definitions. In the absence of a double type, most of this specification for `double` could be adopted for the `long float` type. Having distinct types for `long float` and `double`, previously synonyms, requires more imagination.]

X.2.1 Infinities, signed zeros, and NaNs

IEEE formats provide representations for signed infinities, signed zeros, and quiet and signaling NaNs. 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. This specification adopts by reference the IEEE-specified behavior for infinities, signed zeros, and quiet NaNs. This specification does not define the behavior of signaling NaNs.[2] It generally uses the term *NaN* to denote quiet NaNs.

The **NAN** and **INFINITY** macros and the **nan** function in **<fp.h>** provide designations for NaNs and infinities. Any meaning associated with arguments to the **nan** function is implementation defined. An implementation might interpret **nan** function arguments to determine extra information to be represented in the NaN's significand.

[Signaling NaNs are not created by any standard operations or functions, but can be created as bit patterns. Operations that trigger a signaling NaN argument generally return a quiet NaN result provided no trap is taken; neither traps nor any other facility for signaling NaNs is required by the IEEE standard. True support for signaling NaNs implies restartable traps, such as the optional traps specified in the IEEE standard.

This specification fully supports the primary utility of quiet NaNs—"to handle otherwise intractable situations, such as providing a default value for 0.0/0.0" [11] and thereby provide closure for the arithmetic.

Other applications of NaNs may prove useful. Available parts of NaNs have been used to encode auxiliary information, for example about the NaN's origin [4]. Signaling NaNs are good candidates for filling uninitialized storage; and their available parts could distinguish uninitialized floating objects. IEEE signaling NaNs and trap handlers potentially provide hooks for maintaining diagnostic information or for implementing special arithmetics.

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEEE standard requires that NaNs propagate, but not all implementations faithfully propagate the entire contents. And even the IEEE standard fails to specify the contents of NaNs through format conversion, which is pervasive in some C implementation mechanisms. For these reasons this specification does not specify the behavior of signaling NaNs nor the interpretation of NaN significands.

An early draft of the Floating-Point C Extensions part of the X3J11 Numerical C Extensions Technical Report contains specification for signaling NaNs, which could serve as a guide for extensions in support of signaling NaNs.]

X.3 Operators and functions

C operators and functions provide IEEE required and recommended facilities as listed below. Except where noted, this specification adopts by reference their IEEE-specified behavior.

— The C **+**, **-**, *****, and **/** operators provide the IEEE add, subtract, multiply, and divide operations.

— The C **sqrt** function in **<fp.h>** provides the IEEE square root operation.

— The C **rem** function in **<fp.h>** provides the IEEE remainder operation. The **remquo** function in **<fp.h>** provides the same operation but with additional information.

— The C **rint** function in **<fp.h>** provides the IEEE operation that rounds a floating-point number to an integral value (in the same precision). The C **nearbyint** function in **<fp.h>** provides the similar nearbyinteger function recommended in the Appendix to IEEE standard 854.

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

— The C conversions for floating types provide the IEEE conversions between floating-point precisions.

— The C conversions from integral to floating types provide the IEEE conversions from integer to floating point.

— The C conversions from floating to integral types provide IEEE-like conversions but always round toward zero.

— The C **rinttol** function in **<fp.h>** provides the IEEE 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 IEEE conversions from floating to other integer formats.

— The C translation time conversion of floating constants and the **strtod**, **fprintf**, and related C library functions in **<stdlib.h>** and **<stdio.h>** provide IEEE binary-decimal conversions. The **strtold** function in **<stdlib.h>** provides the conv function recommended in the Appendix to IEEE standard 854.

— The C relational and equality operators provide IEEE comparisons. The IEEE standard identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The relational macros (**isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, and **isunordered**) in **<fp.h>** supplement the C language operators to address this need. The **islessgreater** and **isunordered** macros provide a quiet version of the <> predicate and the unordered predicate recommended in the Appendix to the IEEE standard.

— The **feclearexcept**, **feraiseexcept**, and **fetestexcept** functions in **<fenv.h>** provide the facility to test and alter the IEEE floating-point exception flags. The **fegetexcept** and **fesetexcept** 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 **fexcept_t** typedef 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 IEEE 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, composing the IEEE status flags and control modes.

— The **copysign** function in **<fp.h>** provides the copysign function recommended in the Appendix to the IEEE standard.

— The C unary minus (-) operator provides the minus (-) operation recommended in the Appendix to the IEEE standard.

— The **scalb** function in **<fp.h>** provides the scalb function recommended in the Appendix to the IEEE standard.

— The `logb` function in `<fp.h>` provides the logb function recommended in the Appendix to the IEEE standard, but following the newer specification in IEEE Standard 854.

— The `nextafterf`, `nextafterd`, and `nextafterl` functions in `<fp.h>` provide the nextafter function recommended in the Appendix to the IEEE standard (but with a minor change to better handle signed zeros). The `nextafter` function in `<fp.h>` provides the same function but in a manner reflecting the implementation's evaluation method.

— The `isfinite` macro in `<fp.h>` provides the finite function recommended in the Appendix to the IEEE standard.

— The `isnan` macro in `<fp.h>` provides the isnan function recommended in the Appendix to the IEEE standard.

— 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 `<fp.h>`provide the facility of the class function recommended in the Appendix to the IEEE standard (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 integer 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 integer type raises the inexact exception is unspecified.[3]

X.5 Binary-decimal conversion

Conversion from the widest supported IEEE format to decimal with `DECIMAL_DIG` digits and back is the identity function.[4]

The implementation follows the accuracy recommendations in 7.9.5.7 and 7.10.1.4. In particular, conversion between any supported IEEE format and decimal with `DECIMAL_DIG` or fewer significant digits is correctly rounded.

> [The IEEE standard requires perfect rounding for a large though incomplete subset of decimal conversions. This specification goes beyond the IEEE standard by requiring perfect rounding for all decimal conversions, involving `DECIMAL_DIG` or fewer decimal digits and a supported IEEE format, because practical methods are now available (see ???). Although not requiring

[3] IEEE standard 854, though not 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 <fp.h>**.

[4] If the minimum-width IEEE 754 extended format (64 bits of precision) is supported, `DECIMAL_DIG` must be at least 21. If IEEE 754 double (53 bits of precision) is the widest IEEE 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.)

correct rounding for arbitrarily wide decimal numbers, this specification is sufficient in the sense that it ensures that every internal numeric value in an IEEE format can be determined as a decimal constant.]

X.6 Contracted expressions

Contracted expressions treat infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by the IEEE standard. Contracted expressions should raise exceptions in a manner generally consistent with the basic arithmetic operations.

A contracted expressions should deliver the same value as its uncontracted counterpart, else be correctly rounded (once).

X.7 Environment

The floating-point environment defined in `<fenv.h>` includes the IEEE exception status flags and directed-rounding control modes. It includes also IEEE dynamic rounding precision and trap enablement modes, if the implementation supports them.[5]

[The *exception flags*—invalid, overflow, underflow, divide-by-zero, and inexact—can be queried at execution time to determine whether a floating-point exception has occurred since the beginning of execution or since its flag was explicitly cleared. (The flags are *sticky*.)

The *rounding direction modes*—to-nearest, toward-zero, upward (toward +∞), and downward (toward -∞)—can be altered at execution time to control the rounding direction for floating-point operations.

The IEEE standard prescribes *rounding precision modes* as a means for a system whose results are always double or extended to shorten the precision of its results, in order to mimic systems that deliver results to single or double precision. An implementation of C can meet this goal in any of the following ways:

1. By supporting the method of evaluating expressions to their semantic type.
2. By providing translation options to shorten results by rounding to IEEE single or double precision.
3. By providing functions to set and get dynamic rounding-precision modes which shorten results by rounding to IEEE single or double precision. Functions `fesetprec` and `fegetprec` and macros `FE_FLTPREC`, `FE_DBLPREC`, and `FE_LDBLPREC`, analogous to the functions and macros for the rounding direction modes, would serve the purpose.

This specification does not include a portable interface for precision control because the IEEE standard is ambivalent on whether it intends for precision control to be dynamic (like the rounding direction modes) or static. Indeed, some floating-point architectures provide control modes, suitable for a dynamic mechanism, and others rely on instructions to deliver single- and double-format results, suitable only for a static mechanism.

The traps recommended by the IEEE standard require modes for enabling and disabling.]

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

X.7.1 Environment management

The IEEE standard 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.[6]

X.7.2 Translation

During translation the IEEE 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.

The implementation should provide a warning for each translation-time floating-point exception, other than inexact.

Is there a need for the term *warning* for the rest of the C standard?

[An implementation is not required to provide a facility for altering the modes for translation-time arithmetic, or for making exception flags from the translation available to the executing program. The language and library provide facilities to cause floating-point operations to be done at execution time, when they can be subjected to varying dynamic modes and their exceptions detected. The need does not seem sufficient to require similar facilities for translation.]

As floating constants are converted to appropriate internal representations at translation time, the default rounding modes are in effect and execution-time exceptions are not raised (even when the state of `fenv_access` in *on*).[7]

X.7.3 Execution

At program startup the floating-point environment is initialized as prescribed by the IEEE standard:

— 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.

[6] 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.9.

[7] Library functions, for example `strtod`, provide execution-time conversion of numeric strings.

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 the IEEE standard (provided the state for **fenv_access** is *on*).[8]

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.

> [A previous approach allowed translation-time constant arithmetic, but empowered the unary + operator, when applied to an operand, to inhibit translation-time evaluation of constant expressions.]

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 the IEEE standard (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;                /* may raise 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

[8] Results of inexact expressions like `1.0/3.0` can be affected by rounding modes set at execution time, and expressions such as `0.0/0.0` and `1.0/0.0` can be used reliably to generate execution-time exceptions. Or, the efficiency of translation-time evaluation can be achieved by using static initialization, such as

```
const static double one_third = 1.0/3.0;
```

exceptions. The automatic initializations of `x` and `y` entail execution-time conversion, but, in some expression evaluation methods, not to a narrower format, in which case no exception is raised.[9] 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.

> [This specification does not suit C++, whose static and aggregate initializers need not be constant. Specifying all floating-point constant arithmetic and initialization to be (as if) at execution time would be consistent with C++ and, given the `fenv_access` mechanism, still would allow the bulk of constant arithmetic to be done, in actuality, at translation time.]

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 the IEEE standard). 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 IEEE valid coincident exceptions for atomic operations—namely overflow and inexact, or underflow and inexact—then overflow or underflow is raised before inexact.

> [IEEE operations order exceptions this way.]

X.8 Predefined macro name

An implementation that follows this specification defines the macro name `__IEEE_FP__` which expands to the decimal constant `1`.

X.9 Optimization

This section identifies code transformations that might subvert IEEE-specified behavior, and others that do not.

X.9.1 "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

[9] 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.

```
#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 ≥ `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.9.2 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 IEEE 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 IEEE machines, among others).[10]

`x - y` ↔ `x + (-y)` The expressions `x - y`, `x + (-y)`, and `(-y) + x` are equivalent (on IEEE 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).[11]

`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

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

[11] The IEEE standard 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*))

`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.9.3 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 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.9.4 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*, then a further check is required to assure that changing the rounding direction to downward does not alter the sign of the result.[12] Implementations that support dynamic rounding precision modes should assure further that the result of the operation raises no exception when converted to the semantic type of the operation.

X.10 <fp.h>

This section contains specification of `<fp.h>` facilities that is particularly suited for IEEE 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; their evaluations raise no exceptions.[13]

Special cases for functions in `<fp.h>` are covered directly or indirectly by the IEEE standard. X.3 identifies the functions that the IEEE standard specifies directly. This specification recommends that the other functions in `<fp.h>` honor infinities, NaNs, signed zeros, subnormals, and the exception flags in a manner consistent with the basic arithmetic operations covered by the IEEE standard.

> Does ISO 7.5.1 need amending, as exceptions will be visible by means of <fenv.h> functions?

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.[14] This specification for transcendental functions allows raising the underflow (and inexact) exception when a result is tiny and *probably* inexact.

The inexact exception is raised whenever the rounded result is not identical to the mathematical result. Except as noted, this specification for transcendental functions allows raising the inexact exception when a result is *probably* inexact.

[For some functions, for example `pow`, determining exactness in all cases may be too costly.]

As implied by X.7.6, the `<fp.h>` functions do not raise spurious exceptions (detectable by the user).[15]

[12] 0 - 0 yields -0 instead of +0 just when the rounding direction is downward.

[13] `HUGE_VAL` could not be implemented as

```
#define HUGE_VAL (1.0/0.0)
```

whose use raises the divide-by-zero exception.

[14] The IEEE standard allows different definitions of underflow. All resulting in the same values, the options differ only in that the thresholds when the exception is raised may differ by a rounding error.

[15] For example, the implementation must hide an underflow generated by an intermediate computation of a non-tiny result.

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

> [Although correct rounding for transcendentals is desirable, costs may be prohibitive at this time.]

Generally, one-parameter functions of a NaN argument return that same NaN and raise no exception.

The specification in the following subsections appends to the definitions in `<fp.h>`.

X.10.1 Trigonometric functions (ISO 7.5.2)

X.10.1.1 The acos function (ISO 7.5.2.1)

```
floating-type acos(floating-type x);
```

- `acos(1)` returns `+0`.
- `acos(x)` returns a NaN and raises the invalid exception for `|x| > 1`.

X.10.1.2 The asin function (ISO 7.5.2.2)

```
floating-type asin(floating-type x);
```

- `asin(±0)` returns `±0`.
- `asin(x)` returns a NaN and raises the invalid exception for `|x| > 1`.

X.10.1.3 The atan function (ISO 7.5.2.3)

```
floating-type atan(floating-type x);
```

- `atan(±0)` returns `±0`.
- `atan(±INFINITY)` returns `±π/2`.

X.10.1.4 The atan2 function (ISO 7.5.2.4)

```
floating-type atan2(floating-type y, floating-type x);
```

- 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`.[16]
- `atan2(±0, x)` returns `±π`, for `x < 0`.
- `atan2(±0, -0)` returns `±π`.
- `atan2(y, ±0)` returns `π/2` for `y > 0`.
- `atan2(y, ±0)` returns `-π/2` for `y < 0`.
- `atan2(±y, INFINITY)` returns `±0`, for finite `y > 0`.
- `atan2(±INFINITY, x)` returns `±π/2`, for finite `x`.
- `atan2(±y, -INFINITY)` returns `±π`, for finite `y > 0`.
- `atan2(±INFINITY, INFINITY)` returns `±π/4`.
- `atan2(±INFINITY, -INFINITY)` returns `±3π/4`.

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

[The more contentious cases are `y` and `x` both infinite or both zeros. See [7] for a justification of the choices above.]

X.10.1.5 The cos function (ISO 7.5.2.5;)

```
floating-type cos(floating-type x);
```

- `cos(±INFINITY)` returns a NaN and raises the invalid exception.

X.10.1.6 The sin function (ISO 7.5.2.6)

```
floating-type sin(floating-type x);
```

- `sin(±0)` returns ±0.
- `sin(±INFINITY)` returns a NaN and raises the invalid exception.

X.10.1.7 The tan function (ISO 7.5.2.7)

```
floating-type tan(floating-type x);
```

- `tan(±0)` returns ±0.
- `tan(±INFINITY)` returns a NaN and raises the invalid exception.

X.10.2 Hyperbolic functions (ISO 7.5.3)

X.10.2.1 The acosh function

```
floating-type acosh(floating-type x);
```

- `acosh(1)` returns +0.
- `acosh(+INFINITY)` returns `+INFINITY`.
- `acosh(x)` returns a NaN and raises the invalid exception if `x < 1`.

X.10.2.2 The asinh function

```
floating-type asinh(floating-type x);
```

- `asinh(±0)` returns ±0.
- `asinh(±INFINITY)` returns `±INFINITY`.

X.10.2.3 The atanh function

```
floating-type atanh(floating-type x);
```

- `atanh(±0)` returns ±0.
- `atanh(±1)` returns `±INFINITY`.
- `atanh(x)` returns a NaN and raises the invalid exception if `|x| > 1`.

X.10.2.4 The cosh function (ISO 7.5.3.1)

```
floating-type cosh(floating-type x);
```

- `cosh(±INFINITY)` returns `+INFINITY`.

X.10.2.5 The sinh function (ISO 7.5.3.2)

```
floating-type sinh(floating-type x);
```

- `sinh(±0)` returns ±0.
- `sinh(±INFINITY)` returns ±INFINITY.

X.10.2.6 The tanh function (ISO 7.5.3.3)

```
floating-type tanh(floating-type x);
```

- `tanh(±0)` returns ±0.
- `tanh(±INFINITY)` returns ±1.

X.10.3 Exponential and logarithmic functions (ISO 7.5.4)

X.10.3.1 The exp function (ISO 7.5.4.1;)

```
floating-type exp(floating-type x);
```

- `exp(+INFINITY)` returns +INFINITY.
- `exp(-INFINITY)` returns +0.

X.10.3.2 The exp2 function

```
floating-type exp2(floating-type x);
```

- `exp2(+INFINITY)` returns +INFINITY.
- `exp2(-INFINITY)` returns +0.

X.10.3.3 The expm1 function

```
floating-type expm1(floating-type x);
```

- `expm1(±0)` returns ±0.
- `expm1(+INFINITY)` returns +INFINITY.
- `expm1(-INFINITY)` returns -1.

X.10.3.4 The frexp function (ISO 7.5.4.2)

```
floating-type frexp(floating-type value, int *exp);
```

- `frexp(±0, exp)` returns ±0, and returns 0 in `*exp`.
- `frexp(±INFINITY, exp)` returns ±INFINITY, and returns an unspecified value in `*exp`.
- `frexp` of a NaN argument is that same NaN, and returns an unspecified value in `*exp`.
- Otherwise, `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.10.3.5 The ldexp function (ISO 7.5.4.3;)

```
floating-type ldexp(floating-type x, int exp);
```

On a binary system, `ldexp` is equivalent to

```
scalb(x, exp)
```

[Note that `ldexp` may not provide the full functionality of `scalb` for extended values, because the power required to scale from the smallest (subnormal) to the largest extended value exceeds the minimum `INT_MAX` allowed by Standard C.]

X.10.3.6 The log function (ISO 7.5.4.4)

```
floating-type log(floating-type x);
```

- `log(±0)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log(x)` returns a NaN and raises the invalid exception if `x < 0`.
- `log(+INFINITY)` returns `+INFINITY`.

X.10.3.7 The log10 function (ISO 7.5.4.5)

```
floating-type log10(floating-type x);
```

- `log10(±0)` returns `-INFINITY` and raises the divide-by-zero exception.
- `log10(x)` returns a NaN and raises the invalid exception if `x < 0`.
- `log10(+INFINITY)` returns `+INFINITY`.

X.10.3.8 The log1p function

```
floating-type log1p(floating-type x);
```

- `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.10.3.9 The log2 function

```
floating-type log2(floating-type x);
```

- `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.10.3.10 The logb function

```
floating-type logb(floating-type x);
```

- `logb(±INFINITY)` returns `+INFINITY`.
- `logb(±0)` returns `-INFINITY` and raises the divide-by-zero exception.

X.10.3.11 The modf functions (ISO 7.5.4.6)

```
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

- `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 <fp.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.10.3.12 The scalb function

```
floating-type scalb(floating-type x, long int n);
```

- `scalb(x, n)` returns `x` if `x` is infinite, zero, or a NaN.
- `scalb` handles overflow and underflow like the basic IEEE standard arithmetic operations.

X.10.4 Power and absolute value functions (ISO 7.5.5, 7.5.6)

X.10.4.1 The fabs function (ISO 7.5.6.2)

```
floating-type fabs(floating-type x);
```

- `fabs(±0)` returns `+0`.
- `fabs(±INFINITY)` returns `+INFINITY`.

X.10.4.2 The hypot function

```
floating-type hypot(floating-type x, floating-type y);
```

- `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.10.4.3 The pow function (ISO 7.5.5.1)

```
floating-type pow(floating-type x, floating-type y);
```

- 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, finite, 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.

[See [7].

pow(NaN, 0). [7] provides extensive justification for the value 1. An opposing point of view is that any return value of a function of a NaN argument should be a NaN, even if the function is independent of that argument—as, in the IEEE standard, NAN <= +INFINITY is false and raises the invalid exception.]

X.10.4.4 The sqrt functions (ISO 7.5.5.2)

```
floating-type sqrt(floating-type x);
```

sqrt is fully specified as a basic arithmetic operation in the IEEE standard.

X.10.5 Error and gamma functions

X.10.5.1 The erf function

```
floating-type erf(floating-type x);
```

- erf(±0) returns ±0.
- erf(±INFINITY) returns ±1.

X.10.5.2 The erfc function

```
floating-type erfc(floating-type x);
```

- `erfc(+INFINITY)` returns +0.
- `erfc(-INFINITY)` returns 2.

X.10.5.3 The gamma function

`floating-type` `gamma(`*`floating-type`* `x);`

- `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.10.5.4 The lgamma function

`floating-type` `lgamma(`*`floating-type`* `x);`

- `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.10.6 Nearest integer functions (ISO 7.5.6)

X.10.6.1 The ceil function (ISO 7.5.6.1)

`floating-type` `ceil(`*`floating-type`* `x);`

- `ceil(x)` returns `x` if `x` is ±`INFINITY` or ±0.

The `double` version of `ceil` behaves as though implemented by

```
#include <fp.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.10.6.2 The floor function (ISO 7.5.6.3)

`floating-type` `floor(`*`floating-type`* `x);`

- `floor(x)` returns `x` if `x` is ±`INFINITY` or ±0.

See the sample implementation for `ceil` in X.10.6.1.

X.10.6.3 The nearbyint function

```
floating-type nearbyint(floating-type x);
```

The `nearbyint` function differs from the `rint` function only in that the `nearbyint` function does not raise the inexact flag.

X.10.6.4 The rint function

```
floating-type rint(floating-type x);
```

The `rint` function use IEEE standard rounding according to the current rounding direction. It raises the inexact exception if its argument differs in value from its result.

X.10.6.5 The rinttol function

```
long int rinttol(long double x);
```

The `rinttol` function provides floating-to-integer conversion as prescribed by the IEEE standard. 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.10.6.6 The round function

```
floating-type round(floating-type x);
```

The `double` version of `round` behaves as though implemented by

```
#include <fp.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;
}
```

`round` may but is not required to raise the inexact exception for nonintegral numeric arguments, as this implementation does.

X.10.6.7 The roundtol function

```
long int roundtol(long double x);
```

`roundtol` 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.10.6.8 The trunc function

```
floating-type trunc(floating-type x);
```

The `trunc` function uses IEEE standard rounding toward zero (regardless of the current rounding direction).

X.10.7 Remainder functions (ISO 7.5.6)

X.10.7.1 The fmod function (ISO 7.5.6.4)

```
floating-type fmod(floating-type x, floating-type y);
```

- 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 <fp.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.10.7.2 The remainder function

```
floating-type remainder(floating-type x, floating-type y);
```

`remainder` is fully specified as a basic arithmetic operation in the IEEE standard.

X.10.7.3 The remquo function

```
floating-type remquo(floating-type x, floating-type y, int *quo);
```

`remquo` follows the specification for `remainder`. It has no further specification special to IEEE implementations.

X.10.8 Manipulation functions

X.10.8.1 The copysign function

```
floating-type copysign(floating-type x, floating-type y);
```

copysign is specified in the Appendix to the IEEE standard.

X.10.8.2 The nan functions

```
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

All IEEE implementations support quiet NaNs, hence declare all the nan functions.

X.10.8.3 The nextafter functions

```
floating-type nextafter(floating-type x, long double y);
float nextafterf(float x, float y);
double nextafterd(double x, double y);
long double nextafterl(long double x, long double y);
```

- 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 and x != y.

X.10.9 Maximum, minimum, and positive difference functions

X.10.9.1 The fdim function

```
floating-type fdim(floating-type x, floating-type y);
```

- 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.10.9.2 The fmax function

```
floating-type fmax(floating-type x, floating-type y);
```

- 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.

fmax might be implemented as

```
    { return (isgreaterequal(x, y) || isnan(y)) ? x : y; }[17]
```

Ideally, fmax would be sensitive to the sign of zero, for example fmax(-0.0, +0.0) should return +0; however, implementation in software may be impractical.

[17] Some applications might be better served by a *max* function that would return a NaN if one of its arguments were a NaN:

```
{ return (isgreaterequal(x, y) || isnan(x)) ? x : y; }
```

Note that two branches still are required, for symmetry in NaN cases.

X.10.9.3 The fmin function

```
floating-type fmin(floating-type x, floating-type y);
```

`fmin` is analogous to `fmax`. See X.10.9.2.