WG14/N291
X3J11/93-037

Floating-Point C Extensions

Numerical C Extensions Group / X3J11.1
FP/IEEE Subcommittee
Technical Report
Public Comment Draft (X3J11.1/93-001)
January 20, 1993

Foreword

NCEG, at its initial meeting in May 1989, identified support for IEEE floating-point arithmetic as one of its focus areas and organized a subgroup to produce a technical report. NCEG, now the ANSI working group X3J11.1, has had the benefit of both C language expertise and also IEEE floating-point expertise. It has included individuals with substantial experience with language extensions (albeit proprietary) for IEEE floating-point. And, following the C language standardization, NCEG has had a stable, well defined host for its extensions. Thus NCEG has had a unique opportunity to solve this long-standing problem.

When IEEE binary floating-point standard 754 became an official standard in July 1985, 26 months before the radix-independent standard 854, several IEEE implementations were already shipping. Now virtually all new floating-point implementations conform to the IEEE standards—at least in format, if not to the last detail. Although these standards have been enormously successful in influencing hardware implementation, many IEEE features remain impractical or unavailable for use by programmers. The IEEE standards do not include language bindings—part of the cost of delivering the basic standard in a timely fashion. The ANSI C committee attempted to eliminate conflicts with IEEE arithmetic, but did not specify IEEE support. In the meantime, particular companies have defined their own IEEE language extensions and libraries [4, 8, 10]; not surprisingly, lack of portability has impeded programming for these interfaces.

The initial version of this document, in August 1989, was organized foremost as a specification for IEEE implementations, with notes for other implementations. However, from the beginning, substantial portions of the specification were not specific to IEEE floating-point. For broader utility the document was reorganized as a general floating-point specification with additional specification for IEEE implementations.

NCEG mailings have included ten drafts of this document, which have been reviewed to varying degrees by NCEG's FP/IEEE subgroup, NCEG as a whole, and numerous other interested parties. Proprietary extensions for IEEE support have provided prior art for many features. Substantial portions of the specification have been implemented in both developmental and commercial compilers and libraries.

The previous draft of "Floating-Point C Extensions" (X3J11.1/92-040) was distributed to various professional organizations, including X3J11, WG14, X3J16/WG21, X3T2, and X/Open, in an effort to obtain broader, but still limited, review. At its December 1992 meeting, NCEG, approved unanimously a number of minor changes (detailed in X3J11.1/92-070 and X3J11.1/92-082), mostly in response to suggestions from the limited review, and approved distributing for public comment this draft (X3J11.1/93-001) which incorporates those changes.

People who have made especially substantial contributions to this document include, in alphabetical order: Jerome Coonen, Bill Gibbons, David Hough, Rex Jaeschke, W. Kahan, Clayton Lewis, Stuart McDonald, Colin McMaster, Rick Meyers, David Prosser, and Fred Tydeman.

Others who have provided invaluable contributions, reviews, or administrative help include, in alphabetical order: Joel Boney, Norris Boyd, Larry Breed, Walter Bright, W. J. Cody, Elizabeth Crockett, Karen Deach, James Demmel, Fred Dunlap, Yinsun Feng, James Frankel, Scott Fraser, David Gay, Eric Grosse, Ron Guilmette, Bill Homer, Kenton Hanson, Paul Hilfinger, Martha Jaffe, Bob Jervis, David Keaton, Earl Killian, David Knaak, John Kwan, Roger Lawrence, Tom MacDonald, Randy Meyers, Jon Okada, Tom Pennello, Tim Peters, Sanjay Poonen, Ali Sazegari, Roger Schlafly, Steve Sommars, Richard Stallman, Linda Stanberry, Gordon Sterling, Bill Torkelson, and Terrence Yee.

This document benefited from the author's previous experience with Apple Computer, Inc.'s numerics and languages groups, developing the Standard Apple Numerics Environment (SANE) and its various language bindings.

Jim Thomas
January 20, 1993

Floating-Point C Extensions

X3J11.1/93-001

Jim Thomas
Taligent, Inc.
10725 N. DeAnza Blvd.
Cupertino, CA 95014-2000
jim_thomas@taligent.com

1. INTRODUCTION

1.1 Purpose

This document specifies:

1. a set of Standard C extensions, suitable for most implementations, and designed to facilitate a wide range of numerical programming;
2. a further set of extensions and definitions, suitable for implementations that support IEEE floating-point standards 754 and 854, and designed to provide full access to the features of those standards—allowing access also to similar features in non-IEEE implementations.

This document is intended for incorporation into a Technical Report of the Numerical C Extensions Group (NCEG/X3J11.1).

1.2 Scope

The specifications of this document, while extending Standard C, still lie within the scope of that standard ([17] §4.2; [1] §1.2).

This document does not:

1. describe Standard C, except where it relates to extensions or subtleties of implementation, but instead refers implicitly to the Standard C documents [17], [1];
2. describe IEEE floating-point per se, but instead refers implicitly to the 754 and 854 standards documents;
3. address other NCEG areas, such as complex arithmetic, which are expected to accommodate IEEE standard arithmetic.

1.3 References

1. American National Standard for Information Systems—Programming Language C (X3.159-1989).

2. *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985).

3. *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).

4. *Apple Numerics Manual*, Second Edition, Addison-Wesley (1988).

5. "Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic", by W. J. Cody and Jerome T. Coonen (to appear, *Transactions on Mathematical Software*).

6. "Contributions to a Proposed Standard for Binary Floating-Point Arithmetic", by Jerome Coonen, PhD thesis, University of California, Berkeley (1984).

7. "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", by W. Kahan, Proceedings of the joint IMA/SIAM conference on The State of the Art in Numerical Analysis, 14-18 April 1986, Clarendon Press (1987).

8. *Numerical Computation Guide*, Sun Microsystems, Inc. (March 16, 1990).

9. "Enhanced Arithmetic for Fortran", by R. P. Corbett, *ACM Signum Newsletter*, Vol. 18, No. 1 (January 1983) and *ACM Sigplan Newsletter*, Vol. 17, No. 12 (December 1982).

10. *UNIX System V/386 Release 4 Programmer's Reference Manual*, Prentice Hall (1990).

11. "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic", by W. J. Cody et al, *IEEE Micro*, Vol. 4, No. 4 (August 1984).

12. "Correctly Rounded Binary-Decimal and Decimal-Binary Conversion", by David M. Gay, Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories (November 30, 1990)—NCEG 91-019.

13. *Numerics Programming Guide*, Zortech C++ Compiler 3.0, Zortech Limited (May 1991). A pioneer implementation of substantial portions of this specification.

14. "Overloading Floating-Point Functions in C", Bill Gibbons (November 1991)—NCEG 91-047.

15. "Generic Functions", David Hough (November 1991)—NCEG 91-046.

16. *American National Standard Programming Language FORTRAN* (ANSI X3.9-1978).

17. *International Standard Programming Languages—C*, (ISO/IEC 9899:1990 (E)).

018

18. *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley (1990).

19. "Contracted Multiply-Adds", W. Kahan (September 20, 1991)—NCEG 91-042.

20. "An Argument of Apple's Support of the Extended Floating-Point Type", Jim Thomas and Jerome Coonen (September 14, 1989)—NCEG 89-037.

21. *C* Language Reference Manual*, James L. Frankel, Draft Technical Report, Thinking Machines Corporation (May 16, 1991)—NCEG 91-023.

22. "Augmenting a Programming Language with Complex Arithmetic", W. Kahan and J. W. Thomas (November 15, 1991)—NCEG 91-039.

23. "Algorithm XXX: Specfun—A Portable Package of Special Functions and Test Drivers", W. J. Cody (work in progress).

24. "Floating-Point C Extensions", Jim Thomas (August 10, 1991)—NCEG 91-028.

1.4 Organization of Document

The major subsections are:

1. this introduction;
2. characteristics of the translation and execution environments;
3. language syntax, constraints, and semantics;
4. library facilities;

A. summary of language syntax extensions;

B. optimization notes;

C. summary of library facilities;

D. implementation limits;

E. documentation guide for operators and functions;

F. specification of library facilities for IEEE implementations.

Sections that can be regarded as extending or modifying a particular section of the Standard C documents are marked with the section numbers from the ISO C [17] and ANSI C [1] documents, for example

3.1.2 Floating constants (ISO §6.1.3.1; ANSI §3.1.3.1)

Rationale, in smaller font size, accompanies sections that are controversial or where further explanation seems needed. Subsections entitled "For IEEE Implementations" contain specification that applies specifically to implementations that support features of the IEEE floating-point standards.

As in the IEEE floating-point standards, use of the word *shall* signifies that which is obligatory for conformance; use of the word *should* signifies that which is strongly recommended, though impractical for certain implementations.

1.5 Definition of Terms

- Double—the IEEE floating-point standards' double data format. Written in program-text style, `double` refers to the Standard C data type. Where noted, double refers locally to both IEEE and also non-IEEE double precision formats.

- Double-based architecture—a floating-point architecture designed to produce primarily double format results.

- Evaluation format—the format used to represent the result of an expression. It may be wider than the semantic type of the expression.

- Exception flag—a flag signifying that a floating-point exception has occurred. IEEE implementations support overflow, underflow, invalid, divide-by-zero, and inexact exception flags.

- Expression evaluation method—a method for determining the evaluation formats for expressions.

- Extended—the IEEE floating-point standards' *double-extended* data format. Extended refers to both the common 80-bit and so-called *quadruple* 128-bit IEEE formats. Where noted, *extended* refers locally to both IEEE and also non-IEEE formats that are wider than double.

- Extended-based architecture—a floating-point architecture designed to produce primarily extended format results.

- Flag—see *exception flag*.

- Floating-point environment—collectively all floating-point status flags and control modes.

- Floating-point standards—specifically *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) and *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE Std 854-1987).

- IEEE implementation—an implementation that conforms to one or both of the IEEE floating-point standards. See *floating-point standards*.

- Implementation-defined behavior—behavior that depends on the characteristics of the implementation and that each implementation shall document. (As in Standard C.)

- Mode—short for floating-point control mode.

- NaN—in a floating-point format, an encoding that signifies *Not-a-Number.* In this document the term NaN generally denotes quiet NaNs.

- Normal number—a nonzero, finite number that is not subnormal. (As in IEEE standard 854.)

- Quiet NaN—a NaN that propagates through almost every arithmetic operation without raising an exception.

- Rounding direction mode—a dynamic mode that controls the style of rounding used by floating-point operations. For IEEE implementations the rounding directions are upward, downward, toward zero, and to nearest.

- Rounding precision mode—a dynamic or static mode that controls the precision to which floating-point operations round their results.

- Signaling NaN—a NaN that generally raises an exception when occurring as an arithmetic operand. This specification does not cover signaling NaNs.

- Single—the IEEE floating-point standards' single data format. Where noted, *single* refers locally to both IEEE and also non-IEEE single precision formats.

- Single/double architecture—a floating-point architecture designed to produce with like efficiency either single or double format results, with any extended arithmetic being substantially slower.

- Single/double/extended architecture—a floating-point architecture designed to produce with like efficiency either single, double, or extended results.

- Standard C—the C language standard, *International Standard ISO/IEC 9899* and *American National Standard Programming Language C* .

- Subnormal number—a nonzero floating-point number whose exponent is the format's minimum and whose leading significand digit is zero. (As in IEEE standard 854.)

- Undefined behavior—behavior, for an erroneous program or erroneous data, for which this specification imposes no requirement. (As in Standard C.)

- Unspecified behavior—behavior, for a correct program and correct data, for which this specification imposes no requirement. (As in Standard C.)

- Warning—a translation-time message alerting the user when reasonable or common expectations for a formally correct program may not be met.

1.6 Compliance

In order to comply as an implementation supporting IEEE standard floating-point, an implementation must conform to one, or both, of the IEEE standards 754 and 854—which ones is implementation-defined.

2. ENVIRONMENT

Any floating-point status flags and control modes supported by the implementation are here referred to collectively as the *floating-point environment.* Programs that test flags or run under non-default modes must do so under the effect of an enabling `fenv_access` pragma.

`#pragma fenv_access` *on-off-switch*

on-off-switch: one of:
`on` **`off`** **`default`**

can occur outside external declarations and takes effect from its occurrence until another `fenv_access` pragma is encountered, or until the end of the translation unit. The effect of this pragma appearing inside an external declaration is undefined. The behavior of a program that tests flags or runs under non-default mode settings but that is not under the effect of an enabling `fenv_access` pragma is undefined. The default state (`on` or `off`) for the pragma is implementation-defined. (§4.4 specifies facilities for accessing the floating-point environment.)

The floating-point environment as defined here includes only execution-time modes, not the myriad of possible translation-time options that may affect a program's results. Examples of such translation-time options include: chopped or rounded multiplication on CRAY Y-MP systems, D or G format for VAX, and fast or correctly-rounded divide on the Intel 860. Each option's implementation-defined or deviant properties, relative to this specification, should be well documented.

The purpose of the `fenv_access` pragma is to allow certain optimizations, namely *global common subexpression elimination, code motion,* and *constant folding,* that could subvert the testing of flags and changing of modes. For example, in

```
{
        double x;
        void f(double);
        void g(double);
        ...
        f(x + 1);
        g(x + 1);
        ...
}
```

the function `f` might depend on status flags set as a side effect of the first `x + 1`. The second `x + 1` might depend on control modes set as a side effect of the function call `f`. The imposed temporal ordering would require two evaluations of `x + 1`. This specification says that if in fact the program tests flags or changes modes through the call to `f`, it must so declare with an enabling `fenv_access` pragma. In general, if the state of `fenv_access` is `off` then the translator can assume that default modes are in effect.

The user model is supported by certain programming conventions:

1. A function call must not alter its caller's modes, clear its caller's flags, nor depend on the state of its caller's flags unless the function is so documented. (To do otherwise would be extremely dangerous, irrespective of this specification.)

2. A function call is assumed to require default modes, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

3. A function call is assumed to have the potential of raising floating-point exceptions, unless its documentation promises otherwise, or unless the function is known not to use floating-point.

With these conventions a programmer can assume default modes, or ignore them altogether, safely and without code or intellectual overhead. Responsibilities, and ordinarily modest overhead, associated with accessing the floating-point environment fall entirely on the programmer or program that does so.

No standard library function, except those in <fenv.h>, tests or clears its caller's flags or changes its caller's modes.

Libraries are encouraged to document their use, or non-use, of floating-point and their raising of floating-point exceptions.

The performance of code under the effect of an enabling fenv_access pragma may well be important; in fact, an algorithm may access the floating-point environment specifically for performance. The implementation should optimize as aggressively as the fenv_access pragma allows. (See §B.1.)

An implementation could simply honor the floating-point environment in all cases and ignore the pragma.

Dynamic vs static modes

Dynamic modes are potentially problematic because

1. the programmer may have to defend against undesirable mode settings—which imposes intellectual, as well as time and space, overhead.

2. the translator may not know which mode settings will be in effect or which functions change them at execution time—which inhibits optimization.

This proposal attempts to address these problems without changing the dynamic nature of the modes.

An alternate approach would have been to present a model of *static modes*, with explicit utterances to the translator about what mode settings would be in effect. This would have avoided any uncertainty due to the global nature of dynamic modes or the dependency on unenforced conventions. However, some essentially dynamic mechanism still would have been needed in order to allow functions to inherit (honor) their caller's modes. The IEEE standards require dynamic rounding direction modes. For the many architectures that maintain these modes in control registers, implementation of the static model would be more costly. Also, Standard C has no facility, other than pragmas, for supporting static modes.

An implementation on an architecture that provides only static control of modes, for example through opword encodings, still could support the dynamic model, by generating multiple code streams with tests of a private global variable containing the mode setting. Only modules under an enabling fenv_access pragma would need such special treatment. To further limit the problem, the implementation might employ additional pragmas specifically to indicate where non-default modes would be admissible.

For IEEE Implementations

The IEEE floating-point standards require that an implementation support certain status flags and control modes.

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 *rounding precision modes* cause 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. Some systems need not implement rounding precision modes—see §4.4.2.

The traps recommended by the IEEE floating-point standards require modes for enabling and disabling. Trap-enable modes, which will be covered by a separate NCEG/X3J11.1 subgroup focusing on exception handling, lie outside the scope of this document.

2.1 Translation

An implementation should provide a warning for each translation-time floating-point exception, other than inexact. The implementation shall document all mode settings that affect the behavior of translation-time arithmetic.

For IEEE Implementations

During translation the IEEE standard default modes are in effect:

rounding direction:	to-nearest
rounding precision:	results not shortened
trapping:	all traps disabled

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.

2.2 Execution

2.2.1 Startup

At program startup any floating-point modes are initialized as for translation-time arithmetic. The implementation shall document all startup mode settings that affect the behavior of arithmetic.

This promotes consistency between translation- and execution-time computations.

For IEEE Implementations

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

exception flags:	all clear
rounding direction:	to-nearest
rounding precision:	results not shortened
trapping:	all traps disabled

2.2.2 Changing the environment

Operations defined in ISO §6.3 (ANSI §3.3) and functions and macros defined for standard libraries change flags and modes just as indicated by their specification (which may include conformance to standards); they do not change flags or modes (so as to be detectable by the user) in any other cases.

2.3 Environmental Limits

2.3.1 Characteristics of floating types (ISO §5.2.4.2; ANSI §2.2.4.2)

The value of the Standard C macro

```
FLT_ROUNDS
```

is dynamically determined to represent the effective rounding direction.

> Thus IEEE implementations, and any others that allow changing the rounding direction at execution time, must implement `FLT_ROUNDS` as an execution-time inquiry, not as a constant (when the state of `fenv_access` is *on*).

This specification augments `<float.h>` with two macros that characterize the evaluation method (§3.2.3.1) for evaluating floating-point expressions. The minimum evaluation format is characterized by the value of `_MIN_EVAL_FORMAT`:

```
0  float
1  double
2  long double
```

Whether widest-need evaluation is performed is characterized by `_WIDEST_NEED_EVAL`:

```
0  no
1  yes
```

> If the minimum evaluation format is `long double` then the value of `_WIDEST_NEED_EVAL` is irrelevant.

The values `_MIN_EVAL_FORMAT` and `_WIDEST_NEED_EVAL` need not be constants.

An implementation that provides alternate expression evaluation methods must assure that translation-time evaluation of _MIN_EVAL_FORMAT and _WIDEST_NEED_EVAL reflects the current one.

016

3. LANGUAGE

3.1 Lexical Elements (ISO §6.1; ANSI §3.1)

3.1.1 Types (ISO §6.1.2.5; ANSI §3.1.2.5)

The `long double` type should have strictly more precision than `double` which should have at least twice the number of digits of precision as `float`. If not, the implementation should emit a warning when processing a translation unit that uses distinct floating types with the same precision.

This facilitates porting code that, intentionally or not, depends on differences in type widths. Many results are exact or correctly rounded when computed with twice the number of digits of precision as the data. For example, the calculation

```
float d, x, y, z, w;
d = (double) x * y - (double) z * w;
```

yields a correctly rounded determinant if `double` has twice the precision of `float` and the individual operations are correctly rounded. (The casts to `double` are unnecessary if the minimum evaluation format is `double` or `long double`—see §3.2.3.1.)

For IEEE Implementations

The C floating types match the IEEE standard floating-point formats as follows:

C type	IEEE format
`float`	single
`double`	double
`long double`	extended, else a non-IEEE extended format, else double

Any non-IEEE extended format, used as the `long double` format for an IEEE implementation, has more precision than IEEE double and at least the range.

Minimal conformance to the IEEE floating-point standards does not require a format wider than single. The narrowest C `double` type allowed by Standard C (ISO §5.2.4.2; ANSI §2.2.4.2) 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 standards 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.

The primary objective of the IEEE part of this specification is to facilitate writing portable code that exploits the IEEE floating-point standards, including their standardized single and double data formats. Bringing the C data types and the IEEE standard formats into line advances this objective.

This specification accommodates what are expected to be the most important IEEE floating-point architectures for general C implementations—see §3.2.3.

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. Other features in this document, for example the type definitions `float_t` and `double_t` (defined in §4.3), 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 floating-point 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. [20] explains advantages of extended precision.

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.

Narrow-format Implementations

Some C implementations, namely ones for digital signal processing, provide only the IEEE floating-point standards' 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.

3.1.1.1 NaNs

Floating types may support *NaN* (Not-a-Number) values, which do not represent numbers. A NaN that generally raises an exception when encountered as an operand of arithmetic operations is called a *signaling NaN*, and the operation is said to *trigger* the signaling NaN; this document does not define the behavior of signaling NaNs. A NaN that behaves predictably and does not raise exceptions in arithmetic operations is called a *quiet NaN*; this document uses the term *NaN* to denote quiet NaNs.

The IEEE floating-point standards specify quiet and signaling NaNs, but these terms can be applied for some non-IEEE implementations as well—for example, the VAX *reserved operand* and the CDC and CRAY *indefinite* qualify as signaling NaNs. In IEEE standard arithmetic, 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. True support for signaling NaNs implies restartable traps, such as the optional traps specified in the IEEE floating-point standards.

The primary utility of quiet NaNs—"to handle otherwise intractable situations, such as providing a default value for 0.0/0.0" [11]—can be supported through straightforward extensions to C. See §3.3.2, 4.2.1.2, 4.2.2.1-2, 4.3, 4.3.9.2.

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 floating-point standards require that NaNs propagate, but not all implementations faithfully propagate the entire contents. And even the IEEE standards fail to specify the contents of NaNs through format conversion, which is pervasive in some C implementation mechanisms. For these reasons this document does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.

[24], a previous version of this document, contains specification for signaling NaNs. It could serve as a guide for implementation extensions in support of signaling NaNs.

3.1.2 Floating constants (ISO §6.1.3.1; ANSI §3.1.3.1)

Floating constants are converted to their internal representation at translation time. Each constant is represented in a format (perhaps wider than required by the constant's type) determined by the effective expression evaluation method (§3.2.3.1). The resulting values from translation-time conversion of (valid) floating constants match those from execution-time conversion with default rounding modes by library functions, like `strtod`. See §4.2.1.2.

Note that since floating constants are converted to appropriate internal representations at translation time, default rounding direction and precision will be in effect and execution-time exceptions will not be raised, even under the effect of an enabling `fenv_access` pragma. Library functions, for example `strtod`, provide execution-time conversion of decimal strings.

3.1.2.1 Hexadecimal floating constants (ISO §6.1.3.1, 6.1.8; ANSI §3.1.3.1, 3.1.8)

The Standard C *floating-constant* syntax is augmented to include hexadecimal floating constants.

Syntax

floating-constant:
  ...
  hexadecimal-floating-constant

hexadecimal-floating-constant:
  `0x` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
  `0X` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
  `0x` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$
  `0X` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$

hexadecimal-fractional-constant:
  hexadecimal-digit-sequence$_{opt}$. *hexadecimal-digit-sequence*
  hexadecimal-digit-sequence .

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence hexadecimal-digit

binary-exponent-part:
  `p` *sign*$_{opt}$ *digit-sequence*
  `P` *sign*$_{opt}$ *digit-sequence*

The binary exponent gives a decimal integer representing a power of 2. If `FLT_RADIX` is a power of two, hexadecimal floating constants are correctly rounded. The implementation shall emit a warning if a hexadecimal constant cannot be represented exactly in its evaluation format (see §3.2.3.1). Further accuracy specification for the conversion of hexadecimal floating constants to internal format is implied by the specification for the `strtod` functions (§4.2.1.2).

In order to accommodate hexadecimal floating constants, the Standard C syntax for preprocessing numbers (ISO §6.1.8; ANSI §3.1.8) is augmented to include

pp-number:
  pp-number `p` *sign*
  pp-number `P` *sign*

Example

`<float.h>` for an IEEE 754 implementation might contain:

```
#define  FLT_MAX      0x1.FFFFFEp127f  /* or 0xF.FFFFFp124f */
#define  FLT_EPSILON  0x1p-23f
#define  FLT_MIN      0x1p-126f
```

Example

The constant `-0x1.0000000000001p0` represents the largest IEEE 754 double precision value less than -1.

Hexadecimal more clearly expresses the significance of floating constants.

The binary-exponent part is required to avoid ambiguity from an `f` suffix (being mistaken as a hexadecimal digit).

Constants of `long double` type are not generally portable, even among IEEE implementations.

Unlike integers, floating values cannot all be represented directly by hexadecimal constant syntax. A sign can be prefixed for negative numbers and -0. Infinities might be produced by hexadecimal constants that overflow. NaNs require some other mechanism. Note that `0x1.FFFFFEp128f`, which might appear to be an IEEE single-format NaN, in fact overflows to an infinity in the single format (and causes a translation-time warning).

Infinity and NaN constants, useful for static and aggregate initialization, should be considered for future extensions.

An alternate approach might have been to represent bit patterns. For example

```
#define    FLT_MAX     0x.7F7FFFFF
```

This would have allowed representation of NaNs and infinities. However, numerical values would have been more obscure owing to bias in the exponent and the implicit significand bit. NaN representations would not have been portable—even the determination of IEEE quiet NaN vs signaling NaN is implementation-defined. NaNs and infinities are provided through macros in §4.3.

The straightforward approach of denoting octal constants by a `o` prefix would have been inconsistent with allowing a leading `0` digit—a moot point as the need for octal floating constants was deemed insufficient.

The caret ^ was ruled out as a character to introduce the exponent because doing so would have used up a potential operator.

3.2 Conversions

3.2.1 Floating and integral (ISO §6.2.1.3; ANSI §3.2.1.3)

For conversion from floating to integer type, 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, if available, is raised and the value of the result is unspecified.

For IEEE Implementations

Whether conversion of nonintegral floating values whose integral part is within the range of the integer type raises the inexact exception is unspecified.

Conversion from floating to integral rounds toward zero, consistent with Standard C. IEEE rounding is provided by the library function `rinttol`.

IEEE standard 854, though not 754, directly specifies that floating-to-integral conversion 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 §F.6.

Conversion from integer to floating type is an IEEE standard floating-point operation, rounding according to the current rounding direction mode.

Note that rounding indeed will be required if an integer is too wide to represent exactly in the floating-point format.

3.2.2 Floating types (ISO §6.2.1.4; ANSI §3.2.1.4)

For IEEE Implementations

Conversions between floating types are specified by the IEEE floating-point standards.

3.2.3 Expression evaluation (ISO §6.2.1.5; ANSI §3.2.1.5)

This specification recognizes five distinct *expression evaluation methods.* A floating-point expression has both a semantic type and an *evaluation format.* Each evaluation format is the format of one of the floating types. The result of the expression is represented in the expression's evaluation format. An expression evaluation method determines the evaluation formats for expressions. Which expression evaluation methods are provided is implementation-defined.

For most floating operations Standard C defines the semantic type of the operation to be the widest type of its operands, but gives explicit license to represent the operation's result in a format wider than its type.

The use here of the term *method* is unrelated to its use in Smalltalk.

Although specifying just one method would have facilitated porting code, any one method would have been unacceptably inefficient on some important architectures. On the other hand, still other expression evaluation methods are conceivable, for example evaluating `float` operations to `float` format, and all others to `long double`. The expression evaluation methods described in this section comprise an intentionally small set with at least one method that is efficient for any of the existing or anticipated, commercially significant, floating-point architectures:

Floating-point architectures

In the following description of floating-point architectures, the terms *extended, double*, and *single* have a slightly broader meaning than in the rest of this document. They still refer to floating formats, but apply to both IEEE and non-IEEE systems. Generally, extended is wider than double which is wider than single.

Extended-based. The arithmetic engine is extended. Source operands can be single, double, or extended, though generally arithmetic with single and double types is less efficient, requiring extra conversions. Examples include Intel 80x87, Cyrix 3D87, Motorola 6888x, and AT&T WE 32x06. The Motorola 88110 and Intel 960 can be used as extended-based architectures, or alternatively as single/double/extended ones (see below).

Double-based. The arithmetic engine is double. Source operands can be single or double, though generally arithmetic with the single type is less efficient, requiring extra conversions. Extended may be available, but implemented in software. Examples include IBM RISC System/6000, PDP-11 in double mode, CRAY, and CYBER 180. On CRAY and CYBER, single and double may be the same format. The CYBER provides some hardware support for extended.

Single/double. These provide orthogonal operations for single and double arithmetic. Single is typically faster than double. Extended may be available, but implemented in software. Examples include MIPS, SPARC, HP PA-RISC, Motorola 88100, Intel 860, and systems assembled with Weitek or BIT processors. The MIPS, SPARC, and HP PA-RISC architectures specify extended, though it is not yet in hardware.

Single/double/extended. These provide orthogonal operations for single, double, and extended arithmetic. Single is faster than double, which is faster than extended. Examples include Motorola 88110, Intel 960, IBM S/370, and VAX.

3.2.3.1 Methods

The five expression evaluation methods are characterized by two properties: (1) the minimum-width format for expression evaluation, and (2) whether determination of the evaluation format is affected by the context of the operation according to the rules for *widest-need* evaluation. (Widest-need is irrelevant if the minimum width format is `long double`.)

Minimum evaluation format. The minimum evaluation format may be `float`, `double`, or `long double`. The evaluation format for operations subject to the usual arithmetic conversions and for floating constants is at least this wide, even if the semantic type is less.

> The early C implementations provided just the `float` and `double` floating-point types and evaluated all floating expressions to `double`. Intentional or not, some C programs have relied on extra precision for their computation with `float` operands. A minimum evaluation format of `double` is a natural choice for double-based architectures.
>
> Implementations for single/double and single/double/extended architectures may find a minimum evaluation format of `float` compellingly more efficient, despite potential problems of conformity to expectations based on C's tradition of wide evaluation.
>
> Even on a single/double or single/double/extended architecture, an implementation might provide double as a minimum evaluation format, for compatibility reasons. Common statements of the form

```
    f1 = f2 op f3;     /* where f1, f2, f3 are of type float  */
```

> can be done optimally by many such implementations, including all IEEE ones, where rounding the result to `double` and then to `float` is equivalent to rounding to `float` directly.
>
> A minimum evaluation format of `long double` is common on extended-based architectures. Programs that run under one of the other expression evaluation methods generally run at least as well when all expressions are evaluated to `long double`. Most program failures due to extra precision arise from its inconsistent use (see §B.5).
>
> Representation of constants in a format commensurate with expression evaluation, not a traditional practice, better meets certain expectations than would representation strictly according to semantic type—for example, `0.1f == 1.0f/10.0f`. Viewed as translation-time operations that convert decimal strings to internal floating representations, literal floating constants naturally follow the method of expression evaluation.

Widest-need evaluation. Without widest-need the evaluation format for an operation or constant is simply the wider of its semantic type and the minimum evaluation format. With widest-need, the evaluation format for an operation is the widest of the semantic types appearing in a certain enclosing expression and at least as wide as the minimum evaluation format. More precisely, the evaluation format for an operation subject to the usual arithmetic conversions, or for an assignment (including the assignment of function arguments to parameters, but not cast conversions), is propagated to its operands (or arguments): if an operand is a variable or an operation not subject to the usual arithmetic conversions it is converted to the evaluation format; if the operand is an operation subject to the usual arithmetic conversions, or a floating constant, the evaluation format is imposed recursively.

Examples

With the declarations

```
float f;
double d;
long double ld;
double dd(double);
double fd(float);
```

widest-need with a minimum evaluation format of `float` implies:

Outer-most expression	Operation	Evaluation format
ld + (f * f)	*	long double
	+	long double
ld + fd(f * f)	*	float
	+	long double
ld + dd(f * f)	*	double
	+	long double
ld + (d = f * f)	*	double
	+	long double
f + (d = ld * ld)	*	long double
	+	double
ld + (double)(f * f)	*	float
	+	long double
ld + (d -= f * f)	*	double
	-	double
	+	long double
ld + (f++ * --d)	++	float
	--	double
	*	long double
	+	long double
ld == f * f	*	long double
f / (d + d, f * f)	+	double
	*	float
	/	float
f + (f - f ? d / d : ld)	-	float
	/	long double
	+	long double
ld + 0.1f	0.1f	long double
	+	long double

`f + 0.1`	`0.1`	`double`
	`+`	`double`

The definition of widest-need is based on a *widest need* evaluation for Fortran presented in [9]. It does not affect integer expression evaluation, which is covered by Standard C.

As computer speed and memory size increase, so will the number of problems attempted and the size of data sets. Thus, the likelihood that a program will suffer serious roundoff error for some actual data will increase. Wider precision, not for the entire computation but just for expressions containing certain variables, often will fix the problem, without unduly affecting performance, and without requiring costly error analysis. With widest-need, an expression is automatically evaluated to the format of its widest component. To achieve the same effect without widest-need expression evaluation, the programmer must add or delete casts and constant suffixes throughout the program.

Widest-need expression evaluation is a particularly attractive compromise for architectures whose wider formats are significantly slower. It offers the accuracy of wide evaluation where likely needed and also the speed of narrow evaluation where clearly intended. Note that casts can be used to inhibit widest-need widening, even within a wide expression.

Previous drafts defined a `#pragma evaluate` which allowed switching expression evaluation methods between external declarations. This facility was believed to be without sufficient utility and somewhat error prone. Implementations that support multiple expression evaluation methods can supply translation options.

3.2.3.2 Contraction operators

A floating-point engine may provide an atomic operation for multiple binary operators. Such an atomic operation is referred to here as a *contraction* operator and the multiple binary operators as *contracted*, because rounding errors or side effects from their computation may be omitted. Contracted operators should incur no greater-magnitude rounding error than if they were not contracted. Whether contraction operators are employed is implementation-defined.

An implementation that is able to multiply two `double` operands and produce a `float` result in just one machine instruction might contract the multiplication and assignment in:

```
float f;
double d1, d2;
...
f = d1 * d2;
```

Other examples of potential contraction operators include:

compound assignments	`+=`, `-=`, etc.;
ternary add	`x + y + z;`
multiply-add	`x * y + z.`

Contractions can lead to subtle anomalies even while increasing accuracy. The value of expressions like `a * b + c * d` will depend on how the translator uses a contracted multiply-add. Knowing that the implementation contracts multiply-adds, the programmer should be able to control results (and reap the benefits of contraction) through simple coding measures, for example parenthesizing `(a * b) + c * d`. However, the Intel 860's multiply-add is slightly more problematic. Since it keeps a wide but partial product, `a * b + z` may differ from `c * d + z` even though the exact mathematical products `a * b` and `c * d` are equal; the result depends not just on the mathematical result and the format, as ordinarily expected for error analysis, but also on the particular values of the operands.

The extra accuracy of the IBM RISC System/6000's *fused* multiply-add, which produces a result with just one rounding, can be exploited for simpler and faster codes. See [19] for details.

The fp_contract pragma can be used to allow or disallow contraction operators.

```
#pragma  fp_contract  on-off-switch
```

This pragma can occur outside external declarations, and allows or disallows contraction operators from its occurrence until another fp_contract pragma is encountered, or until the end of the translation unit. The default state (on or off) for the pragma is implementation-defined.

For IEEE Implementations

Contraction operators honor infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by the IEEE floating-point standards. Contraction operators should raise exceptions in a manner generally consistent with the basic arithmetic operations covered by the IEEE floating-point standards.

Contraction operators should deliver the same value as their uncontracted counterpart, else be correctly rounded—that is, deliver the infinitely precise result of their contracted operations, rounded (once) to the destination format according to the effective rounding method.

3.2.4 Wide representation

Three pragmas give license for the implementation to represent function return values, function parameters, and variables in a format wider than their declared type.

```
#pragma   fp_wide_function_returns   on-off-switch
#pragma   fp_wide_function_parameters   on-off-switch
#pragma   fp_wide_variables   on-off-switch
```

#pragma fp_wide_function_returns on—allows floating return values to be represented in a wider format than (and not narrowed to) the declared type of the function.

#pragma fp_wide_function_parameters on—allows floating parameters to be in a wider format than (and not narrowed to) the declared type of the formal parameter.

#pragma fp_wide_variables on—allows values of automatic scalar floating variables to be in a wider format than their declared type.

If the *on-off-switch* is off then widening is disallowed. The default state for these pragmas is off.

Standard C compatibility requires that the default state for the pragmas be off.

These pragmas can occur outside external declarations, and allow (if on) or disallow (if off) widening from their occurrence until another pragma instructing otherwise is encountered, or until the end of the translation unit. The effect of these pragmas appearing inside an external declaration is undefined. Widening is consistent throughout the effect of an enabling pragma: either all instances of the returns, parameters, or variables are widened or none are; the representation format for a parameter or variable does not vary. However, which, if any, of these pragmas actually cause widening is implementation-defined.

When the implementation detects an address or `sizeof` operator of a widened parameter or variable it emits a translation-time warning; execution-time behavior is then undefined.

This mechanism facilitates generating efficient code for extended-based or double-based architectures. §3.2.3 and §4.3 provide rationale. The typedefs `float_t` and `double_t` allow finer application than do the pragmas, but require more extensive changes to existing code.

The function writer who decides that narrowing arguments and returns to their semantic type is less desirable than efficiency can write the function under the effect of enabling `fp_wide_function_parameters` and `fp_wide_function_returns` pragmas. Similarly the programmer who uses the function can apply these pragmas to the call site. In either case widening (not narrowing) may or may not occur. These pragmas do not demand widening, nor even recommend it, but merely declare that widening would be acceptable. It is up to the implementation to determine whether widening would be both safe and also more efficient. For example, implementations that pass different type (floating) parameters in different formats can not widen parameters of external functions safely.

Even among implementations which respond to the pragmas, the location in the source code where the pragmas must be placed to be effective may vary. For example, an implementation might perform requisite narrowing of parameters at the call site, in which case the call would have to be under the effect of an enabling `fp_wide_function_parameters` pragma; or, it might narrow within the function, in which case the function definition would have to be under the effect of the pragma.

The pragmas do not affect function prototypes. Doing so might have benefited implementations that pass different floating type arguments and returns in different ways. However, maintaining consistency between the prototype and implementation seemed particularly error prone. A language extension to assure the consistency seemed unjustified. The `float_t` and `double_t` type definitions are available for such prototypes.

Casts are not affected by any of these pragmas, nor by wide expression evaluation, so can be used portably to force narrowing.

Another approach would have been to introduce `register` as a function qualifier that would have allowed widening of both the parameters and the return value. This would have been inconsistent with the `register` storage class specifier, which is unrelated to widening, and would not have helped with variables.

3.3 Expressions (ISO §6.3; ANSI §3.3)

3.3.1 The arithmetic operators (ISO §6.3.3.1, 6.3.3.3, 6.3.4-6.3.6, 6.3.16; ANSI §3.3.3.1, 3.3.3.3, 3.3.4-3.3.6, 3.3.16)

The implementation should follow the guide referenced in Appendix E to document its arithmetic operators.

For IEEE Implementations

These operations fall under the specification of the IEEE floating-point standards:

```
 ++      --      +      -      *      /
 =      casts    +=     -=     *=     /=
```

3.3.2 Relational and equality operators (ISO §6.1.5, 6.3.8, 6.3.9; ANSI §3.1.5, 3.3.8, 3.3.9)

The Standard C relational and equality operators support the usual mathematical relationships between numeric values. These operators are augmented to support relationships involving nonnumeric values, NaNs. For any ordered pair of numeric values exactly one of the relationships—*less*, *greater*, and *equal*—is true. For a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true.

Symbol	Relation
`<`	less
`>`	greater
`<=`	less or equal
`>=`	greater or equal
`==`	equal
`!=`	unordered, less, or greater
`!<>=`	unordered
`<>`	less or greater
`<>=`	less, equal, or greater
`!<=`	unordered or greater
`!<`	unordered, greater, or equal
`!>=`	unordered or less
`!>`	unordered, less, or equal
`!<>`	unordered or equal

The additional operators are analogous to, and have the same precedence as, the Standard C relational operators. Where the operands have types and values suitable for relational operators, the semantics detailed in ISO §6.3.8 (ANSI §3.3.8) apply.

The Standard C *operator* syntax (ISO §6.1.5; ANSI §3.1.5) is augmented to include the additional operators.

Programs written for (or ported to) systems with NaNs will be expected to handle invalid and NaN input in reasonable ways. The additional relational operators support such programs. As operators they, together with the arithmetic operators (+, *, ...), provide a consistent approach to handling NaNs: the arithmetic operators simply propagate them; the relational operators facilitate directing them through branches. Thus NaNs can flow through many computations without the need for obfuscating or inefficient code.

The alternative of function syntax, such as

```
isrelation(x, FP_UNORDERED | FP_LESS | FP_EQUAL, y)
```

instead of `x !> y`, would not have required language extensions, and could have been as efficient as built-in operators, though only if the functions were built into the translator. However, special function calls for comparisons would have been arguably more awkward and inconsistent with the overall approach to NaNs.

The set of relational operators is obtained by allowing the <> combinations and extending the notion of negated operators, previously only !=, to the relational operators, !<, etc.

The expressions `(a !`*op*` b)` and `!(a `*op*` b)` always have the same logical value, that is `(a !`*op*` b) == !(a `*op*` b)`, where *op* is `<`, `>`, `<=`, `>=`, `<>`, or `<>=`. But if the types of `a` and `b` admit NaNs, then `!(a `*op*` b)` and `(a !`*op*` b)` may differ in their exception behavior. Implementations without NaNs, and non-IEEE implementations with NaNs but which practically cannot support the specification below for IEEE implementations, can regard the new relationals simply as notational alternatives.

In its definitions for the logical AND, logical OR, and conditional operators and the `if` statement, Standard C assumes that any expression compares equal to 0 if and only if it does not compare unequal to 0. This assumption is still valid: `a == 0` is true if and only if `a != 0` is false, even if `a` is a NaN.

The implementation should follow the guide referenced in Appendix E to document its relational and equality operators.

For IEEE Implementations

For just those relational operators that include `<` or `>` but not `!`, the invalid exception is raised if the operands compare unordered.

For example,

```
(a < b) == !(a !< b)
```

in all cases, but `(a < b)` raises invalid when `a` and `b` compare unordered whereas `!(a !< b)` does not.

The somewhat awkward articulation `!(a !< b)` for a non-exceptional *less than* is a consequence of the IEEE floating-point standards' priority on supporting existing code that does not account for NaNs. When `x` or `y` is a NaN, any value for `x < y`, whether the actual result 0 (false) or its alternative 1 (true), is likely to be misleading; so the IEEE standards require that such tests raise the invalid exception. Tests of equality give generally useful results when operands are NaNs, hence do not raise the invalid exception.

The IEEE floating-point standards enumerate 26 functionally distinct comparison predicates, from combinations of the four comparison results and whether invalid is raised. Use of logical negation extends the 14 relational and equality operators listed above to the full 26. (There are 26 not 28 because `==` and `!=` never raise invalid).

This proposal supplants a previous one which would have augmented the set of relations by using the `?` symbol to denote unordered, for example `a ?>= b` instead of `a !< b`. Use of the `?` relationals would have had the advantage that the unordered case would have been dealt with explicitly. However, the `!` relationals are a more natural language extension, particularly from the point of view of programmers for (non-IEEE) implementations not detecting unordered. Also, using `??` as proposed for the unordered operator would have conflicted with Standard C trigraphs.

Without any language or library support `a !> b` might be implemented by

```
a != a || b != b || a <= b.
```

However, even more awkward code would be required if `a` or `b` had side effects. The programmer would have to remember to put the NaN tests first, and trust the compiler not to replace `a != a || b != b` by *false*. Also, special optimization would be necessary to generate efficient code. Use of functions like `isnan` or `isunordered`, which are recommended in the appendixes to the IEEE floating-point standards, is similarly problematic.

3.4 Constant Expressions (ISO §6.4; ANSI §3.4)

A constant expression, 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.

This is merely a clarification of Standard C. See §B.4 regarding optimization.

Example

```
#pragma 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;              /* may raise an exception */
    double z = 0.0/0.0;             /* may raise 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.

Previous versions of this section allowed translation-time constant arithmetic, but empowered the unary + operator, when applied to an operand, to inhibit translation-time evaluation of constant expressions. The programmer can achieve the efficiency of translation-time evaluation by using static initialization, for example

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

or to decrease the likelihood of storage allocation

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

For IEEE Implementations

This specification acknowledges the semantics of IEEE arithmetic for the *as if* principle. In particular, execution-time operations must be affected by modes and must raise exceptions as required by the IEEE floating-point standards.

Note that 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.

3.5 Initialization (ISO §5.1.2, 6.5.7; ANSI §2.1.2, 3.5.7)

All computation for automatic scalar initialization is done (as if) at execution time. 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.

Standard C does not specify when aggregate and union initialization is done. Otherwise, this is merely a clarification of Standard C. Note that, under the effect of an enabling fenv_access pragma, any exception resulting from execution-time initialization must be raised at execution time. See §B.4 regarding optimization.

Example

```
#pragma 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 exceptions on some implementations, including all IEEE ones. The automatic initializations of x and y entail execution-time assignment, but, in some expression evaluation methods, not to a narrower format, in which case no exceptions need be raised. The automatic initialization of z entails execution-time assignment, 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.

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.

The specification of §3.4-5 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.

For IEEE Implementations

This specification acknowledges the semantics of IEEE arithmetic for the *as if* principle. In particular, execution-time operations must be affected by modes and must raise exceptions as required by the IEEE floating-point standards.

3.6 Predefined Macro Names (ISO §6.8.8; ANSI §3.8.8)

__FPCE__ The decimal constant 1, indicating conformance to this specification.

__FPCE_IEEE__ The decimal constant 1, indicating conformance to the parts of this specification for IEEE implementations.

4. Library

4.1 Introduction (ISO §7.1.2, 7.1.6; ANSI §4.1.2, 4.1.6)

Section 4.2 covers numerical extensions to the Standard C libraries. Section 4.3 documents a new header <fp.h> which provides facilities for general floating-point programming; <fp.h> supersedes <math.h>. Section 4.4 documents a new header <fenv.h> which provides access to the floating-point environment. The new headers, <fp.h> and <fenv.h>, follow the Standard C specification for standard headers and reserved identifiers (ISO §7.1.2, 7.1.2.1; ANSI §4.1.2, 4.1.2.1).

In some cases names are not unique in the first six characters, unlike in Standard C.

4.2 Standard C Library Extensions

4.2.1 General utilities <stdlib.h>

4.2.1.1 The atof function (ISO §7.10.1.1; ANSI §4.10.1.1)

```
#include <stdlib.h>
double atof(const char *nptr);
```

The atof function meets the specifications below for strtod.

This document does not require float and long double versions of atof, but instead encourages the use of strtof and strtold which have a more generally useful interface.

4.2.1.2 The strtof, strtod, and strtold functions (ISO §7.10.1.4; ANSI §4.10.1.4)

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

The strtof and strtold functions are float and long double versions of strtod.

Translation-time conversion of valid floating constants exactly matches execution-time conversion provided the default rounding modes are in effect. See §3.1.2.

The expected form of the subject sequence accepted by strtod is broadened to include an optional plus or minus sign, then a 0x or 0X, and then followed by one of:

1. a nonempty sequence of hexadecimal digits optionally containing a "decimal-point" character, then a binary-exponent part, but no floating suffix;

2. a nonempty sequence of hexadecimal digits containing a "decimal-point" character, but no floating suffix.

For implementations whose `FLT_RADIX` is a power of 2, a hexadecimal floating source value is correctly rounded to the appropriate internal format. For implementations whose `FLT_RADIX` is not a power of 2, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error have the correct sign for the current rounding direction.

Also, the syntax accepted by `strtod` includes:

sign_{opt} `INF`
sign_{opt} `INFINITY`
sign_{opt} `NAN`
sign_{opt} `NAN(`*n-char-sequence_{opt}* `)`

where

n-char-sequence:
  digit
  nondigit
  n-char-sequence digit
  n-char-sequence nondigit

and any strings equivalent except for case in the `INF`, `INFINITY`, or `NAN` part.

The character sequences `INF` and `INFINITY` produce an infinity, if representable, else are treated as numeric input that overflows.

Character sequences of the form `NAN` or `NAN(`*n-char-sequence_{opt}*`)` are converted to quiet NaNs, if supported, else are treated as invalid input. An implementation may use the *n-char-sequence* to determine extra information to be represented in the NaN's significand; which *n-char-sequence*'s are meaningful is implementation-defined.

So much is implementation-defined because so little is portable. Attaching meaning to NaN significands is problematic, even for one implementation, even an IEEE one. For example, the IEEE floating-point standards do not specify the effect of format conversions on NaN significands—conversions, perhaps generated by the compiler, may alter NaN significands in obscure ways.

Requiring a sign for NaN or infinity input was considered as a way of minimizing the chance of mistakenly accepting nonnumeric input. The need for this was deemed insufficient, partly on the basis of prior art.

For simplicity, the infinity and NaN representations are provided through straightforward extensions, rather than through a new locale (ISO §7.4; ANSI §4.4). Note also that Standard C locale categories do not affect the representations of infinities and NaNs.

The `strtod` function should honor the sign of zero if the arithmetic supports signed zeros.

For IEEE Implementations

Conversion from a numeric decimal string with DECIMAL_DIG (defined in §4.3) or fewer significant digits to the widest supported IEEE format is correctly rounded according to the effective rounding direction. For conversion of a numeric decimal string D, with more than DECIMAL_DIG digits, consider the two bounding, adjacent decimal strings L and U, both having DECIMAL_DIG significant digits, such that the values of L, D, and U satisfy L ≤ D ≤ U. The result of conversion is one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D has the correct sign for the current rounding direction.

> *Correct rounding* refers to the style of rounding that the IEEE floating-point standards prescribe for their arithmetic operations: the delivered value is the (destination) format's value that is closest, in the sense of the effective rounding direction, to the infinitely precise result.
>
> DECIMAL_DIG is sufficiently large that L and U will usually round to the same internal floating value, but if not will round to adjacent values.
>
> The IEEE floating-point standards require perfect rounding for a large though incomplete subset of decimal conversions. This specification goes beyond the IEEE floating-point standards by requiring perfect rounding for all decimal conversions, involving DECIMAL_DIG or fewer decimal digits and a supported IEEE format, because practical methods are becoming publicly available (see [12]). Although not requiring 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.

The `strtod` function honors the sign of zero. It treats the overflow, underflow, and inexact exceptions in accordance with the IEEE floating-point standards. And, it raises the invalid exception on invalid numeric input.

> This document's previous specification that `strtod` return a NaN for invalid numeric input, as recommended by IEEE standard 854, was withdrawn because of the incompatibility with Standard C, which demands that `strtod` return 0 for invalid numeric input.

4.2.2 Input / output <stdio.h>

4.2.2.1 The fprintf function (ISO §7.9.6.1; ANSI §4.9.6.1)

The `a` and `A` conversion specifiers provide hexadecimal floating representations. The conversion specifier `a` means that the argument is converted to a hexadecimal floating representation in the style `[-]0xh.hhhhp±d`. The number of hexadecimal digits `h` after the decimal-point character is equal to the precision; if the precision is missing then for implementations whose FLT_RADIX is a power of 2 the precision is sufficient for an exact representation of the source value, and for other implementations the precision should be sufficient to distinguish (see rationale below) values of the source type, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point character is nonzero for normal values and is otherwise chosen by the implementation; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The `A` conversion specifier will produce a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2.

Binary implementations should choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble boundaries. For example, the next value greater than one in the common IEEE 754 80-bit extended format should be

```
0x8.000000000000001p-3
```

The next value less than one in IEEE 754 double should be

```
0x1.fffffffffffffp-1
```

Note that if the precision is missing, trailing zeros may be omitted. For example, the value positive zero might be

```
0x0.p+0
```

The more suggestive conversion specifiers for hexadecimal formatting, namely `x` and `h`, were unavailable. Since `h` was taken `H` was ruled out in favor of a lower/upper case option. Possibilities other than `a` included: `b j k m q r t v w y z`. The optional `h` to indicate hexadecimal floating, as in `%he`, was deemed a less natural fit with the established scheme for specifiers and options.

The *decimal-point character* is defined in Standard C (ISO §7.1.1; ANSI §4.1.1). *Radix-point character* would have been a better term.

The precision *p* is sufficient to *distinguish* values of the source type if

$$16^{p-1} > b^n$$

where *b* is `FLT_RADIX` and *n* is the number of base-*b* digits in the significand of the source type. A smaller *p* might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character. (See [6], Ch 7.)

For implementations whose `FLT_RADIX` is a power of 2, the source value is correctly rounded to a hexadecimal floating number with the given precision. For implementations whose `FLT_RADIX` is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have the correct sign for the current rounding direction.

The `F` conversion specifier has the same effect as `f` except for infinity and NaN formatting (specified below).

The optional precision, optional `L`, and `0` and `#` flag characters all have the same effect for `a`, `A`, and `F` conversions as for `e`, `E`, `f`, `g`, and `G`.

With `e`-style formatting, the exponent field contains at least two digits, and only as many more digits as necessary to represent the exponent.

For all floating conversions, an infinite value is converted in one of the styles `[-]inf` or `[-]infinity`—which style is implementation-defined.

For all floating conversions, a NaN value is converted in one of the styles `[-]nan` or `[-]nan(`*n-char-sequence*`)`—which style, and the meaning of any *n-char-sequence*, is implementation-defined.

See §4.2.1.2 Rationale.

When applied to infinite and NaN values, the `-`, `+`, and *space* flag characters have their usual meaning; the `#` and `0` flag characters have no effect.

Use of an upper case format specifier, `A`, `E`, `F`, or `G`, results in `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`.

> Use of the `A` and `F` format specifiers constitutes a minor extension to Standard C which does not reserve them.

The results of all floating conversions of a negative zero should include a minus sign.

For IEEE Implementations

If the number of significant decimal digits is at most `DECIMAL_DIG`, then the result from converting a value from an IEEE format is correctly rounded for the current rounding direction. If the number of significant decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, then the result is an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having `DECIMAL_DIG` significant digits; the value of the result decimal string D satisfies $L \le D \le U$, with the extra stipulation that the error have the correct sign for the current rounding direction.

> See §4.2.1.2 Rationale. For binary-to-decimal conversion, the *infinitely precise result* is just the source value, and the destination format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

The results of all floating conversions of a negative zero shall include a minus sign.

4.2.2.2 The fscanf function (ISO §7.9.6.2; ANSI §4.9.6.2)

For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, all valid syntax—including hexadecimal-floating-constant, infinity, NaN, and signed zero—is treated in the same manner as `strtod`.

4.3 Floating-Point Extensions <fp.h>

The header `<fp.h>` declares several types, macros and functions to support general floating-point programming. It provides essentially all of Standard C `<math.h>`, as well as facilities required or recommended by the IEEE floating-point standards, and a few other facilities that have proved broadly useful. [17] and [1] contain basic descriptions, not repeated in this document, of the Standard C functions. Appendix F contains specification that is particularly suited for IEEE implementations. The implementation should follow the guide referenced in Appendix E to document its `<fp.h>` functions.

The typedefs

```
    float_t
    double_t
```

are defined to be the implementation's most efficient floating types at least as wide as `float` and `double`, respectively. (Every value of type `float_t` is also a value of type `double_t`.)

> Facility to use wider types is needed for writing portable efficient code. Currently Standard C gives no way of asking for the most efficient floating type with at least a given width. Thus different prototypes are more efficient on different floating-point architectures.
>
> | architecture (see §3.2.3) | most efficient prototype |
> |---|---|
> | extended-based | `long double f(long double)` |
> | double-based | `double f(double)` |
> | single/double | `float f(float)` |
> | single/double/extended | `float f(float)` |
>
> Differences may involve whether values can be kept in registers, hence are substantial. Implementations for the various floating-point architectures might use these type definitions:
>
> | architecture | `float_t` | `double_t` |
> |---|---|---|
> | extended-based | `long double` | `long double` |
> | double-based | `double` | `double` |
> | single/double | `float` | `double` |
> | single/double/extended | `float` | `double` |
>
> An alternate approach of modifying the semantics of the `register` storage-class specifier, when applied to a floating type, to mean that the associated value may be wider than the type, was rejected as inconsistent with existing use of `register` in Standard C.

The macro

```
HUGE_VAL
```

is as defined in ANSI C §4.5.

The macro

```
INFINITY
```

expands to a floating expression representing an implementation-defined positive or unsigned infinity, if available, else to a positive floating constant that overflows at translation time.

The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs. It expands to a floating-point expression representing an implementation-defined quiet NaN.

> Ideally the `INFINITY` and `NAN` macros would be suitable for static and aggregate initialization, though currently such is not required by this specification.

No type is specified for the `INFINITY` and `NAN` macros because conversions between floating-point types convert infinities to infinities and NaNs to NaNs, without raising exceptions.

The macros

```
FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

are for number classification. They represent the mutually exclusive kinds of floating-point values. They expand to `int` constant expressions with distinct values and suitable for use in `#if` preprocessing directives.

Some prior art uses a finer classification: `FP_POS_INFINITE`, `FP_NEG_INFINITE`, etc. The current consensus is that those specified, in conjunction with the `signbit` macro, are generally preferable.

Results for the inquiry macros specified in the remainder of this section are undefined if the argument is not of floating type.

This allows efficient implementation.

The macro

`fpclassify(`*`floating-expr`*`)`

evaluates to the value of the number classification macro appropriate to the value of its argument. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

It is important to know the type that classification is based on. For example, a value might be zero when converted to `float`, subnormal when converted to `double`, and normal when converted to `long double`.

`fpclassify` might be implemented as

```
#define fpclassify(x) ((sizeof(x) == sizeof(float)) ? __fpclassifyf(x) \
    : (sizeof(x) == sizeof(double)) ?__fpclassifyd(x) \
    : __fpclassifyl(x))
```

The macro

`signbit(`*`floating-expr`*`)`

evaluates to an `int` expression that is nonzero if and only if the sign of the argument (infinities, zeros, and NaNs included) is negative.

The IEEE floating-point standards do not specify the effect of arithmetic operations on a NaN's sign bit. However, `signbit` of a NaN is valid in the sense that, even if `x` is a NaN, `signbit(x)` is invariant until `x` changes.

The macro

`isfinite(`*`floating-expr`*`)`

evaluates to an `int` expression that is nonzero if and only if its argument has a finite value (zero, subnormal, or normal). 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.

The macro

`isnormal(`*`floating-expr`*`)`

evaluates to an `int` expression that is nonzero if and only if its argument value is *normal*: neither zero, subnormal, infinite, nor NaN. 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.

The macro

`isnan(`*`floating-expr`*`)`

evaluates to an `int` expression that is nonzero if and only if its argument value is a NaN.

The macro

`DECIMAL_DIG`

expands to an `int` constant expression suitable for use in `#if` preprocessing directives. Its value is an implementation-defined number of decimal digits which is supported by conversion between decimal and internal floating-point representations. Conversion from (at least) `double` to decimal with `DECIMAL_DIG` digits and back should be the identity function. And, `DECIMAL_DIG` should give an appropriate number of digits to carry in canonical decimal representations.

> In order that correctly rounded conversion from an internal floating-point format with precision *m* to decimal with `DECIMAL_DIG` digits and back be the identity function, `DECIMAL_DIG` should be a positive integer *n* satisfying
>
> $$\begin{array}{ll} n \geq m, & \text{if } \mathtt{FLT\_RADIX} \text{ is } 10 \\ 10^{n-1} > \mathtt{FLT\_RADIX}^{m}, & \text{otherwise} \end{array}$$
>
> Refer to [6], Chapter 7, for details.
>
> `DECIMAL_DIG` is distinct from Standard C's `DBL_DIG`, which is defined in terms of conversion from decimal to `double` and back.
>
> `DECIMAL_DIG` was deemed more useful than `FP_CONV_DIG`, which previous versions of this document defined as the number of decimal digits for which the implementation guaranteed correctly rounded conversion.

For IEEE Implementations

Conversion from the widest supported IEEE-standard format to decimal with `DECIMAL_DIG` digits and back is the identity function. See §4.2.1.2 and 4.2.2.1 regarding conversion requirements.

Examples

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.

4.3.1 Overloaded functions

The specification of the header `<fp.h>` uses the notion of *overloaded* functions. The function designators are *overloaded* with multiple prototypes which become declared when `<fp.h>` is included. They are used much like ordinary functions, but they behave more like typical built-in arithmetic operators: their semantic type is affected by their arguments; their result is at least as wide as the minimum evaluation format; and, with widest-need expression evaluation, the evaluation format propagates from the call to its arguments.

For each overloaded function defined in this section, the implementation provides a set of functions—one for the minimum evaluation format and one for each wider floating format. Their prototypes are identical except for their floating-point return type and one or more overloading parameters whose type is identical to the return type. Prototypes have the form

```
floating-type function-designator ( argument-list ) ;
```

Subsequent sections give a form in this style for each overloaded function. The form determines the required prototypes.

Example

The square root function has the prototype form

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

So, if the minimum evaluation format were `float` then the implementation would have

```
float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

Example

The `remquo` function has the form

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

So, if the minimum evaluation format were `double` then the implementation would have

```
double remquo(double, double, int *);
long double remquo(long double, long double, int *);
```

A function call that is compatible with one of the prototypes for an overloaded function is compatible with all of them. For such a function call, occurring after inclusion of `<fp.h>` in the translation unit, the implementation determines the evaluation format for the call and chooses the matching prototype. First, any integer arguments for the overloading parameters are converted to `double`. Then the evaluation format for the call is determined to be the widest of

1. the arguments for the overloading parameters,

2. the minimum evaluation format,

and in the case of widest-need (§3.2.3.1)

3. any evaluation format propagated down to the call.

With widest-need expression evaluation, the evaluation format for the call, which matches the type of the invoked function, is propagated through the call to the arguments corresponding to the overloading parameters.

Taking the address of an overloaded function or passing it as an argument to a function results in undefined behavior.

Overloaded functions cooperate with expression evaluation methods much as arithmetic operators do. For example, with

```
double d; float f;
d = sqrt(f + f);
```

the type of `sqrt` and evaluation format of `+` actually used are:

min-eval-format	widest-need	`sqrt()`	`f + f`
`float`	no	`float`	`float`
`double`	no	`double`	`double`
`long double`	either	`long double`	`long double`
`float`	yes	`double`	`double`
`double`	yes	`double`	`double`

Modeled after C++, the rudimentary overloading required by this specification can reasonably be expected to be compatible with future overloading extensions to C. General purpose overloading for C is beyond the scope of this document.

This specification does not prescribe any particular implementation mechanism. It could be implemented simply with built-in macros. A function such as

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

could be implemented with

```
#undef sqrt
#define sqrt(x) __BUILTIN_OVERLOAD_sqrt(x)
```

Note that implementations whose minimum evaluation format is `long double` require no overloading mechanism since only one prototype, the `long double` one, is needed for each function.

This specification borrows heavily from [14] and [15]. [15] is functionally similar, but is presented in the style of Fortran generic intrinsic functions. Like this specification, [14] is consistent with C++ overloading. Adding a suitable subset of C++ overloading to C, as prescribed by [14], would support the overloading required here, and as well provide a useful user-extensible overloading mechanism. (With the current C++ overloading rules, additional prototypes would be required to handle integer arguments. [14] suggests how C++ overloading might be extended to accommodate widest-need.)

The C* language, an extended C, includes a function overloading scheme that supports its parallel types. The C* definition in [21] differs from this specification and current C++ in that its overloading resolution depends on the order of parameters.

The great majority of existing C programs are expected to run correctly straightaway when `<fp.h>` is included instead of `<math.h>`. Overloaded functions differ from corresponding Standard C library functions essentially only in the semantic types of return values.

Use of Standard C's reserved `f` and `l` suffixed identifiers could be supported with macros such as

```
#define sinf(x) sin((float)(x))
```

(Taking the address of such a function or passing it as an argument would be disallowed.)

The ability to overload on integer as well as floating types would have been useful for some functions, for example `copysign`. Overloading with different numbers of arguments would have allowed reusing names, for example `remainder` for `remquo`. However, these facilities would have complicated the specification somewhat and their natural consistent use, such as for a floating `abs` or a two argument `atan`, would have introduced further inconsistencies with Standard C for insufficient benefit.

This specification in no way limits the implementation's options for efficiency, including inlining library functions (ISO §7.1.6; ANSI §4.1.6).

4.3.2 Trigonometric functions (ISO §7.5.2; ANSI §4.5.2)

The header `<fp.h>` declares overloaded versions of all the Standard C trigonometric functions.

4.3.2.1 The acos function (ISO §7.5.2.1; ANSI §4.5.2.1)

Synopsis

```
#include <fp.h>
floating-type acos(floating-type x);
```

4.3.2.2 The asin function (ISO §7.5.2.2; ANSI §4.5.2.2)

Synopsis

```
#include <fp.h>
floating-type asin(floating-type x);
```

4.3.2.3 The atan function (ISO §7.5.2.3; ANSI §4.5.2.3)

Synopsis

```
#include <fp.h>
floating-type atan(floating-type x);
```

4.3.2.4 The atan2 function (ISO §7.5.2.4; ANSI §4.5.2.4)

Synopsis

```
#include <fp.h>
floating-type atan2(floating-type y, floating-type x);
```

4.3.2.5 The cos function (ISO §7.5.2.5; ANSI §4.5.2.5)

Synopsis

```
#include <fp.h>
floating-type cos(floating-type x);
```

4.3.2.6 The sin function (ISO §7.5.2.6; ANSI §4.5.2.6)

Synopsis

```
#include <fp.h>
floating-type sin(floating-type x);
```

4.3.2.7 The tan function (ISO §7.5.2.7; ANSI §4.5.2.7)

Synopsis

```
#include <fp.h>
floating-type tan(floating-type x);
```

4.3.3 Hyperbolic functions (ISO §7.5.3; ANSI §4.5.3)

The header `<fp.h>` declares overloaded versions of the Standard C hyperbolic functions and their *arc* counterparts.

4.3.3.1 The acosh function

Synopsis

```
#include <fp.h>
floating-type acosh(floating-type x);
```

Description

The `acosh` function computes the (nonnegative) arc hyperbolic cosine of `x`.

Returns

The `acosh` function returns the arc hyperbolic cosine.

4.3.3.2 The asinh function

Synopsis

```
#include <fp.h>
floating-type asinh(floating-type x);
```

Description

The `asinh` function computes the arc hyperbolic sine of `x`.

Returns

The `asinh` function returns the arc hyperbolic sine.

4.3.3.3 The atanh function

Synopsis

```
#include <fp.h>
floating-type atanh(floating-type x);
```

Description

The `atanh` function computes the arc hyperbolic tangent of `x`.

Returns

The `atanh` function returns the arc hyperbolic tangent .

4.3.3.4 The cosh function (ISO §7.5.3.1; ANSI §4.5.3.1)

Synopsis

```
#include <fp.h>
floating-type cosh(floating-type x);
```

4.3.3.5 The sinh function (ISO §7.5.3.2; ANSI §4.5.3.2)

Synopsis

```
#include <fp.h>
floating-type sinh(floating-type x);
```

4.3.3.6 The tanh function (ISO §7.5.3.3; ANSI §4.5.3.3)

Synopsis

```
#include <fp.h>
floating-type tanh(floating-type x);
```

4.3.4 Exponential and logarithmic functions (ISO §7.5.4; ANSI §4.5.4)

The header <fp.h> declares overloaded versions of the Standard C exponential and logarithmic functions—except for modf which is declared but not overloaded—and several related functions.

4.3.4.1 The exp function (ISO §7.5.4.1; ANSI §4.5.4.1)

Synopsis

```
#include <fp.h>
floating-type exp(floating-type x);
```

4.3.4.2 The exp2 function

Synopsis

```
#include <fp.h>
floating-type exp2(floating-type x);
```

Description

The exp2 function computes the base-2 exponential of x: 2^x.

Returns

The exp2 function returns the base-2 exponential.

4.3.4.3 The expm1 function

Synopsis

```
#include <fp.h>
floating-type expm1(floating-type x);
```

Description

The expm1 function computes the base-e exponential of the argument, minus 1: $e^x - 1$. For small magnitude x, expm1(x) is expected to be more accurate than exp(x) - 1.

Returns

The expm1 function returns $e^x - 1$.

4.3.4.4 The frexp function (ISO §7.5.4.2; ANSI §4.5.4.2)

Synopsis

```
#include <fp.h>
floating-type frexp(floating-type value, int *exp);
```

4.3.4.5 The ldexp function (ISO §7.5.4.3; ANSI §4.5.4.3)

Synopsis

```
#include <fp.h>
floating-type ldexp(floating-type x, int exp);
```

4.3.4.6 The log function (ISO §7.5.4.4; ANSI §4.5.4.4)

Synopsis

```
#include <fp.h>
floating-type log(floating-type x);
```

4.3.4.7 The log10 function (ISO §7.5.4.5; ANSI §4.5.4.5)

Synopsis

```
#include <fp.h>
floating-type log10(floating-type x);
```

4.3.4.8 The log1p function

Synopsis

```
#include <fp.h>
floating-type log1p(floating-type x);
```

Description

The `log1p` function computes the base-e logarithm of 1 plus the argument. For small magnitude `x`, `log1p(x)` is expected to be more accurate than `log(1 + x)`.

Returns

The `log1p` function returns the base-e logarithm of 1 plus the argument.

4.3.4.9 The log2 function

Synopsis

```
#include <fp.h>
floating-type log2(floating-type x);
```

Description

The `log2` function computes the base-2 logarithm of `x`.

Returns

The `log2` function returns the base-2 logarithm.

4.3.4.10 The logb function

Synopsis

```
#include <fp.h>
floating-type logb(floating-type x);
```

Description

The logb function extracts the exponent of x, as a signed integral value in the format of x. If x is subnormal it is treated as though it were normalized; thus for positive finite x,

$$1 \le x * \mathtt{FLT\_RADIX}^{-\mathrm{logb}(x)} < \mathtt{FLT\_RADIX}$$

The treatment of subnormal x follows the recommendation in IEEE standard 854, which differs from IEEE standard 754 on this point. Even 754 implementations should follow this definition rather than the one recommended (not required) by 754.

Particularly on machines whose radix is not 2, logb can be expected to obtain the exponent more accurately and quickly than frexp.

Returns

The logb function returns the signed exponent of its argument.

4.3.4.11 The modf functions (ISO §7.5.4.7; ANSI §4.5.4.6)

Synopsis

```
#include <fp.h>
float modff(float value, float *iptr);
double modf(double value, double *iptr);
long double modfl(long double value, long double *iptr);
```

4.3.4.12 The scalb function

Synopsis

```
#include <fp.h>
floating-type scalb(floating-type x, long int n);
```

Description

The scalb function computes $x * \mathtt{FLT\_RADIX}^{n}$ efficiently, not normally by computing $\mathtt{FLT\_RADIX}^{n}$ explicitly.

Returns

The scalb function returns $x * \mathtt{FLT\_RADIX}^{n}$.

On machines whose radix is not 2, scalb, compared with ldexp, can be expected to have better accuracy, speed, and overflow and underflow behavior.

4.3.5 Power and absolute value functions (ISO §7.5.5, 7.5.6; ANSI §4.5.5, 4.5.6)

The header <fp.h> declares overloaded versions of the Standard C power and absolute value functions and a hypotenuse function.

4.3.5.1 The fabs function (ISO §7.5.6.2; ANSI §4.5.6.2)

Synopsis

```
#include <fp.h>
floating-type fabs(floating-type x);
```

4.3.5.2 The hypot function

Synopsis

```
#include <fp.h>
floating-type hypot(floating-type x, floating-type y);
```

Description

The hypot function computes the square root of the sum of the squares of x and y, without undue overflow or underflow.

Returns

The hypot function returns the square root of the sum of the squares of x and y.

4.3.5.3 The pow function (ISO §7.5.5.1; ANSI §4.5.5.1)

Synopsis

```
#include <fp.h>
floating-type pow(floating-type x, floating-type y);
```

4.3.5.4 The sqrt function (ISO §7.5.5.2; ANSI §4.5.5.2)

Synopsis

```
#include <fp.h>
floating-type sqrt(floating-type x);
```

4.3.6 Error and gamma functions

See [23] regarding implementation.

4.3.6.1 The erf function

Synopsis

```
#include <fp.h>
floating-type erf(floating-type x);
```

Description

The `erf` function computes the error function of x.

Returns

The `erf` function returns the error function of x.

4.3.6.2 The erfc function

Synopsis

```
#include <fp.h>
floating-type erfc(floating-type x);
```

Description

The `erfc` function computes the complementary error function of x.

Returns

The `erfc` function returns the complementary error function of x.

4.3.6.3 The gamma function

Synopsis

```
#include <fp.h>
floating-type gamma(floating-type x);
```

Description

The `gamma` function computes the gamma function of x: $\Gamma(x)$.

Returns

The `gamma` function returns $\Gamma(x)$.

In UNIX System V [10], both the `gamma` and `lgamma` functions compute $\log_e(|\Gamma(x)|)$.

4.3.6.4 The lgamma function

Synopsis

```
#include <fp.h>
floating-type lgamma(floating-type x);
```

Description

The `lgamma` function computes the logarithm of the absolute value of gamma of x: $\log_e(|\Gamma(x)|)$.

In UNIX System V [10], a call to `lgamma` sets an external variable `signgam` to the sign of `gamma(x)`, which is -1 if

```
    x < 0 && remainder(floor(x), 2) != 0
```

Note that this specification does not remove the external identifier `signgam` from the user's name space. An implementation that supports, as an extension, `lgamma`'s setting of `signgam` must still protect the external identifier `signgam` if defined by the user.

Returns

The `lgamma` function returns $\log_e(|\Gamma(x)|)$.

4.3.7 Nearest integer functions (ISO §7.5.6; ANSI §4.5.6)

The header `<fp.h>` declares overloaded versions of the Standard C nearest integer functions, nearest integer functions specified by the IEEE standards, and functions similar to common Fortran nearest integer functions.

4.3.7.1 The ceil function (ISO §7.5.6.1; ANSI §4.5.6.1)

Synopsis

```
#include <fp.h>
floating-type ceil(floating-type x);
```

4.3.7.2 The floor function (ISO §7.5.5.1; ANSI §4.5.5.1)

Synopsis

```
#include <fp.h>
floating-type floor(floating-type x);
```

4.3.7.3 The nearbyint function

Synopsis

```
#include <fp.h>
floating-type nearbyint(floating-type x);
```

Description

The `nearbyint` function differs from the `rint` function (§4.3.7.4) only in that the `nearbyint` function does not raise the inexact exception. (See §F.6.3-4.)

For implementations that do not support the inexact exception, `nearbyint` and `rint` are equivalent.

Returns

The `nearbyint` function returns the rounded integral value.

4.3.7.4 The rint function

Synopsis

```
#include <fp.h>
floating-type rint(floating-type x);
```

Description

The `rint` function rounds its argument to an integral value in floating-point format, using the current rounding direction.

Returns

The `rint` function returns the rounded integral value.

4.3.7.5 The rinttol function

Synopsis

```
#include <fp.h>
long int rinttol(long double x);
```

Description

The `rinttol` function rounds its argument to the nearest `long int`, rounding according to the current rounding direction. If the rounded value is outside the range of `long int`, the numeric result is unspecified.

Returns

The `rinttol` function returns the rounded `long int` value, using the current rounding direction.

4.3.7.6 The round function

Synopsis

```
#include <fp.h>
floating-type round(floating-type x);
```

Description

The `round` function rounds its argument to the nearest integral value in floating-point format, using *add half to the magnitude and chop* rounding a la the Fortran `anint` function, regardless of the current rounding direction.

Returns

The `round` function returns the rounded integral value.

4.3.7.7 The roundtol function

Synopsis

```
#include <fp.h>
long int roundtol(long double x);
```

Description

The roundtol function returns the rounded long int value, using *add half to the magnitude and chop* rounding a la the Fortran nint function and the Pascal round function, regardless of the current rounding direction. If the rounded value is outside the range of long int, the numeric result is unspecified.

Returns

The roundtol function returns the rounded long int value.

4.3.7.8 The trunc function

Synopsis

```
#include <fp.h>
floating-type trunc(floating-type x);
```

Description

The trunc function rounds its argument to the integral value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

The trunc function returns the truncated integral value.

4.3.8 Remainder functions (ISO §7.5.6; ANSI §4.5.6)

The header <fp.h> declares an overloaded Standard C fmod function and two versions of the IEEE floating-point standards' remainder function.

4.3.8.1 The fmod function (ISO §7.5.6.4; ANSI §4.5.6.4)

Synopsis

```
#include <fp.h>
floating-type fmod(floating-type x, floating-type y);
```

4.3.8.2 The remainder function

Synopsis

```
#include <fp.h>
floating-type remainder(floating-type x,
                                           floating-type y);
```

032

Description

The `remainder` function computes the remainder `x` REM `y` required by the IEEE floating-point standards.

> "When $y \neq 0$, the remainder $r = x\ REM\ y$ is defined regardless of the rounding mode by the mathematical relation $r = x - y * n$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." [3] §5.1.

(This definition can be implemented for non-IEEE as well as IEEE machines.)

Returns

The `remainder` function returns `x` REM `y`.

4.3.8.3 The remquo function

Synopsis

```
#include <fp.h>
floating-type remquo(floating-type x, floating-type y,
                                        int *quo);
```

Description

The `remquo` function computes the same remainder as the `remainder` function. In the object pointed to by `quo` it stores a value whose sign is the sign of `x/y` and whose magnitude is congruent mod 2^n to the magnitude of the integral quotient of `x/y`, where `n` is an implementation-defined integer at least 3.

Returns

The `remquo` function returns `x REM y`.

> The `remquo` function is intended for implementing argument reductions, which can exploit a few low-order bits of the quotient. Note that `x` may be so large in magnitude relative to `y` that an exact representation of the quotient is not practical.

4.3.9 Manipulation functions

These extension functions manipulate representations of floating-point numbers.

4.3.9.1 The copysign function

Synopsis

```
#include <fp.h>
floating-type copysign(floating-type x, floating-type y);
```

Description

The `copysign` function produces a value with the magnitude of `x` and the sign of `y`. It produces a NaN (with the sign of `y`) if `x` is a NaN. On implementations that represent a *signed* zero but do not treat negative zero consistently in arithmetic operations, the `copysign` function regards the sign of zero as positive.

The requirement that `copysign` regard a negative sign of zero as positive if the arithmetic treats negative zero like positive zero is justified in order to preserve more identities. For example, to preserve the identity, the square root of the product is the product of the square roots, the algorithm in [22] for the complex square root depends on consistency of `copysign` with the rest of the arithmetic: if -0 behaves like +0 then the square root of the product would yield

$$\sqrt{3 * (-1 - 0i)} = \sqrt{-3 + 0i} \rightarrow 0 + \sqrt{3}i$$

but if `copysign` were to treat the sign of -0 as negative then the product of the square roots would yield

$$\sqrt{3} * \sqrt{-1 - 0i} \rightarrow \sqrt{3} * (0 - i) = 0 - \sqrt{3}i$$

Returns

The `copysign` function returns a value with the magnitude of `x` and the sign of `y`.

4.3.9.2 The nan functions

Synopsis

```
#include <fp.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

If the implementation supports quiet NaNs in the type of the function, then the call `nan("`*`n-char-sequence`*`")` is equivalent to `strtod("NAN(`*`n-char-sequence`*`)", (char**) NULL)`; the call `nan("")` is equivalent to `strtod("NAN()", (char**) NULL)`. Similarly `nanf` and `nanl` are defined in terms of `strtof` and `strtold`. If `tagp` does not point to an *n-char-sequence* string then the result NaN's content is unspecified. A call to a `nan` function of a type for which the implementation does not support quiet NaNs is unspecified.

Returns

The `nan` functions return a quiet NaN, if available, with content indicated through `tagp`.

4.3.9.3 The nextafter functions

Synopsis

```
#include <fp.h>
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);
```

Description

The `nextafter` functions determine the next representable value, in the type of the function, after `x` in the direction of `y`. The `nextafter` functions return `y` if `x == y`.

Returns

The `nextafter` functions return the next representable value after `x` in the direction of `y`.

> It's sometimes desirable to find the next representation after a value in the direction of a previously computed value—maybe smaller, maybe larger. The `nextafter` functions have a second floating argument so that the program will not have to include floating-point tests for determining the direction in such situations. And, on some machines these tests may fail due to overflow, underflow, or roundoff.
>
> The overloaded `nextafter` function depends substantially on the expression evaluation method—which is appropriate for certain uses but not for others. The explicitly typed functions can be employed to obtain next values in a particular format. For example,
>
> ```
> nextafterf(x, y)
> ```
>
> will return the next `float` value after `(float) x` in the direction of `(float) y` regardless of the evaluation method.
>
> The second parameter of the overloaded `nextafter` function has type `long double` primarily to keep the overloading scheme simple. Promotion of the second argument to `long double` is harmless but unnecessary.
>
> For the case `x == y`, the IEEE floating-point standards recommend that `x` be returned. This specification differs in order that `nextafter(-0.0, +0.0)` return `+0.0` and `nextafter(+0.0, -0.0)` return `-0.0`.

4.3.10 Maximum, minimum, and positive difference functions

These extension functions correspond to standard Fortran functions, `dim`, `max`, and `min` [16].

> Their names have `f` prefixes to allow for integer versions—following the example of `fabs` and `abs`.

4.3.10.1 The fdim function

Synopsis

```
#include <fp.h>
floating-type fdim(floating-type x, floating-type y);
```

Description

The `fdim` function determines the *positive difference* between its arguments:

```
x - y ,  if x > y
 +0   ,  if x ≤ y
```

Returns

The fdim function returns the positive difference between x and y.

4.3.10.2 The fmax function

Synopsis

```
#include <fp.h>
floating-type fmax(floating-type x, floating-type y);
```

Description

The fmax function determines the maximum numeric value of its arguments.

NaN arguments should be treated as missing data. If one argument is a NaN and the other numeric, then fmax should chose the numeric value. (See §F.9.2.)

Returns

The fmax function returns the maximum numeric value of its arguments.

4.3.10.3 The fmin function

Synopsis

```
#include <fp.h>
floating-type fmin(floating-type x, floating-type y);
```

Description

The fmin function determines the minimum numeric value of its arguments.

NaN arguments should be treated as missing data. If one argument is a NaN and the other numeric, then fmin should chose the numeric value. (See §F.9.3.)

Returns

The fmin function returns the minimum numeric value of its arguments.

4.4 Floating-Point Environment <fenv.h>

The header <fenv.h> declares two types and several macros and functions to provide access to the floating-point environment.

Names of macros and functions in this section consistently include an FE_ or fe prefix and employ certain abbreviations. The prefix calls attention to environmental access functions, which require an enabling fenv_access pragma (see §2). The abbreviations env and except are used in Standard C and UNIX System V, respectively.

The interface described here is not intended to support trap handlers, which are outside the scope of this document.

The typedef

```
fenv_t
```

is a type for representing the entire floating-point environment.

The typedef

```
fexcept_t
```

is a type for representing the floating-point exception flags collectively, including any status the implementation associates with the flags.

See rationale in §4.4.1.

Each macro

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
```

is defined if and only if the implementation supports the exception by means of the functions in §4.4.1. The defined macros expand to `int` constant expressions whose values are distinct powers of 2, all suitable for use in `#if` preprocessing directives.

The macro

```
FE_ALL_EXCEPT
```

is simply the bitwise OR of all exception macros defined by the implementation.

Each macro

```
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the `fegetround` and `fesetround` functions. The defined macros expand to `int` constant expressions whose values are distinct nonnegative values, all suitable for use in `#if` preprocessing directives.

The rounding direction macros might expand to constants corresponding to the values of `FLT_ROUNDS`, the Standard C inquiry for the rounding direction of addition, but need not.

The macro

```
FE_DFL_ENV
```

represents the default floating-point environment—the one installed at program startup—and has type *pointer to* fenv_t. It can be used as an argument to <fenv.h> functions that manage the floating-point environment.

> Unsupported macros are not defined in order to assure that their use results in a translation error. A program might explicitly define such macros, to allow translation of code (perhaps never executed) containing the macros. An unsupported exception macro should be defined to be 0, for example
>
> ```
> #ifndef FE_INEXACT
> #define FE_INEXACT 0
> #endif
> ```
>
> so that a bitwise OR of macros has a reasonable effect.

4.4.1 Exceptions

The following functions provide access to the exception flags. The int input argument for the functions represents a subset of floating-point exceptions, and can be constructed by bitwise ORs of the exception macros, for example FE_OVERFLOW | FE_INEXACT. For other argument values the behavior of these functions is undefined.

> The IEEE standards require that exception flags represent, at least, whether the flag is set or clear; the functions fetestexcept, feraiseexcept, and feclearexcept support this basic abstraction. However, an implementation may endow exception flags with more information—for example, the address of the code which raised the exception; the functions fegetexcept and fesetexcept deal with the full content of flags.

> In previous drafts of this specification, several of the exception functions returned an int indicating whether the excepts argument represented supported exceptions. This facility was deemed unnecessary because excepts & ~FE_ALL_EXCEPT can be used to test invalidity of the excepts argument.

4.4.1.1 The feclearexcept function

Synopsis

```
#include <fenv.h>
void feclearexcept(int excepts);
```

Description

The feclearexcept function clears the supported exceptions represented by its argument. The argument excepts represents exceptions as a bitwise OR of exception macros.

4.4.1.2 The fegetexcept function

Synopsis

```
#include <fenv.h>
void fegetexcept(fexcept_t *flagp, int excepts);
```

Description

The fegetexcept function stores an implementation-defined representation of the exception flags indicated by the argument excepts through the pointer argument flagp.

4.4.1.3 The feraiseexcept function

Synopsis

```
#include <fenv.h>
void feraiseexcept(int excepts);
```

Description

The feraiseexcept function raises the supported exceptions represented by its argument. The argument excepts represents exceptions as a bitwise OR of exception macros. The order in which these exceptions are raised is unspecified.

It is intended that enabled traps for exceptions raised by the feraiseexcept function are taken.

For IEEE Implementations

If the argument excepts represents coincident exceptions that are valid for atomic operations—namely overflow and inexact, or underflow and inexact—then overflow or underflow is raised before inexact.

The function is not restricted to accept only valid coincident expressions for atomic operations, so that the function can be used to raise exceptions accrued over several operations.

4.4.1.4 The fesetexcept function

Synopsis

```
#include <fenv.h>
void fesetexcept(const fexcept_t *flagp, int excepts);
```

Description

The fesetexcept function sets the exception flags indicated by the argument excepts, according to the representation in the object pointed to by flagp. The value of *flagp must have been set by a previous call to fegetexcept; if not, the effect on the indicated exception flags is undefined. This function does not *raise* exceptions, but only sets the state of the flags.

4.4.1.5 The fetestexcept function

Synopsis

```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

The `fetestexcept` function determines which of a specified subset of the exception flags are currently set. The `excepts` argument specifies—as a bitwise OR of the exception macros—the exception flags to be queried.

Returns

The `fetestexcept` function returns the bitwise OR of the exception macros corresponding to the currently set exceptions included in `excepts`.

Example

Call `f` if invalid is set, `g` if overflow is set:

```
#pragma fenv_access on
int set_excepts;
set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (set_excepts & FE_INVALID) f();
if (set_excepts & FE_OVERFLOW) g();
```

This mechanism allows testing several exceptions with just one function call. The argument is a *mask* because querying all flags may be more expensive on some architectures.

4.4.2 Rounding

The `fegetround` and `fesetround` functions provide control of rounding direction modes.

The IEEE floating-point standards prescribe *rounding precision* modes (in addition to the rounding direction modes covered by the functions in this section) as a means for systems whose results are always double or extended to mimic systems that deliver results to narrower formats. An implementation of C can meet this goal in any of the following ways:

1. By supporting the `float` minimum evaluation format (see §3.2.3).

2. By providing pragmas or compile options to shorten results by rounding to IEEE single or double precision.

3. By providing functions to set and get, dynamically, rounding precision modes which shorten results by rounding to IEEE single or double precision. Recommended are functions `fesetprec` and `fegetprec` and macros `FE_FLTPREC`, `FE_DBLPREC`, and `FE_LDBLPREC`, analogous to the functions and macros for the rounding direction modes.

This specification does not include a portable interface for precision control because the IEEE floating-point standards are ambivalent on whether they intend 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.

4.4.2.1 The fegetround function

Synopsis

```
#include <fenv.h>
int fegetround(void);
```

Description

The fegetround function gets the current rounding direction.

Returns

The fegetround function returns the value of the rounding direction macro representing the current rounding direction.

4.4.2.2 The fesetround function

Synopsis

```
#include <fenv.h>
int fesetround(int round);
```

Description

The fesetround function establishes the rounding direction represented by its argument round. If the argument does not match a rounding direction macro, the rounding direction is not changed.

Returns

The fesetround function returns a nonzero value if and only if the argument matches a rounding direction macro (that is, if and only if the requested rounding direction can be established).

Example

Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#pragma fenv_access on
int save_round;
save_round = fegetround();
int setround_ok = fesetround(FE_UPWARD);
assert(setround_ok);
. . .
fesetround(save_round);
```

4.4.3 Environment

The functions in this section manage the floating-point environment—exception flags, dynamic rounding modes, and any dynamic modes for handling floating-point exceptions—as one entity.

4.4.3.1 The fegetenv function

Synopsis

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

Description

The `fegetenv` function stores the current floating-point environment in the object pointed to by `envp`.

4.4.3.2 The feholdexcept function

Synopsis

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

The `feholdexcept` function saves the current environment in the object pointed to by `envp`, clears the exception flags, and installs the non-stop mode, if available, for all exceptions. For implementations whose floating-point environment contains modes for exception handling, including a *non-stop* (continue on exceptions) mode, the `feholdexcept` function can be used in conjunction with the `feupdateenv` function to write routines that hide spurious exceptions from their callers.

Returns

The `feholdexcept` function returns nonzero if and only if non-stop exception handling was successfully installed.

`feholdexcept` should be effective on typical IEEE implementations which have the default non-stop mode and at least one other mode for trap handling or aborting. If the implementation provides only the non-stop mode then installing the non-stop mode is trivial.

More appropriate for the user model prescribed in §2, `feholdexcept` supersedes `feprocentry` which is equivalent to

```
fegetenv(envp);
fesetenv(FE_DFL_ENV);
```

4.4.3.3 The fesetenv function

Synopsis

```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

Description

The `fesetenv` function establishes the floating-point environment represented by the object pointed to by `envp`. The value of `*envp` must be set by a call to `fegetenv`, or by the macro FE_DFL_ENV or an implementation-defined macro of type fenv_t. Note that fesetenv merely installs the state of the exception flags represented through its argument, and does not raise these exceptions.

4.4.3.4 The feupdateenv function

Synopsis

```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

Description

The feupdateenv function saves the current exceptions in its automatic storage, installs the environment represented through envp, and then raises (actually re-raises) the saved exceptions. For implementations whose floating-point environment contains modes for exception handling, including a *non-stop* (continue on exceptions) mode, the feupdateenv function can be used in conjunction with the feholdexcept function to write routines that hide spurious exceptions from their callers.

feupdateenv was called feprocexit in earlier drafts of this specification.

Example

Hide spurious underflow exceptions:

```
#pragma fenv_access on
fenv_t save_env;
feholdexcept(&save_env);
...
if (... underflow is spurious ...) feclearexcept(FE_UNDERFLOW);
feupdateenv(&save_env);
```

APPENDIXES

A. Language Syntax Extensions (ISO §B; ANSI §A)

This appendix summarizes language syntax extensions to Standard C.

A.1 Constants (ISO §B.1.4; ANSI §A.1.4)

floating-constant:
. . .
hexadecimal-floating-constant

hexadecimal-floating-constant:
`0x` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-fractional-constant binary-exponent-part floating-suffix*$_{opt}$
`0x` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$
`0X` *hexadecimal-digit-sequence binary-exponent-part floating-suffix*$_{opt}$

hexadecimal-fractional-constant:
hexadecimal-digit-sequence$_{opt}$ `.` *hexadecimal-digit-sequence*
hexadecimal-digit-sequence `.`

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

binary-exponent-part:
`p` *sign*$_{opt}$ *digit-sequence*
`P` *sign*$_{opt}$ *digit-sequence*

A.2 Operators (ISO §B.1.6; ANSI §A.1.6)

operator: one of

```
. . .
!<>=    <>    <>=    !<=    !<    !>=    !>    !<>
```

A.3 Preprocessor Numbers (ISO §B.1.9; ANSI §A.1.9)

pp-number:
. . .
pp-number `p` *sign*
pp-number `P` *sign*

A.4 Relational Expressions (ISO §B.2.1; ANSI §A.2.1)

relational expression:

```
    ...
    relational-expression  !<>=  shift-expression
    relational-expression  <>    shift-expression
    relational-expression  <>=   shift-expression
    relational-expression  !<=   shift-expression
    relational-expression  !<    shift-expression
    relational-expression  !>=   shift-expression
    relational-expression  !>    shift-expression
    relational-expression  !<>   shift-expression
```

A.5 Pragmas

#pragma fenv_access *on-off-switch*

on-off-switch: one of:

```
    on      off      default
```

#pragma fp_contract *on-off-switch*

#pragma fp_wide_function_returns *on-off-switch*
#pragma fp_wide_function_parameters *on-off-switch*
#pragma fp_wide_variables *on-off-switch*

B. Optimization

Many common optimizations for integer arithmetic are not generally valid (nor as useful) for floating-point. This appendix is an implementation note explaining some of the constraints.

B.1 Expression evaluation

For IEEE Implementations

Floating-point arithmetic operations and external function calls may entail side effects which optimization must honor, at least within the scope of an enabling `fenv_access` pragma (§2). 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, removal of *useless* code, and certain transformations. For example, in

```
#pragma fenv_access on
double x;
...
for (i = 0; i < n; i++) x + 1;
```

x + 1 might overflow, 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.

Within a basic block exceptions need not be precise: Thus the preceding loop could be treated as

```
    if (0 < n) x + 1;
```

B.2 Expression transformations

The following transformations are not generally valid.

Removal of parentheses. This is invalid for any floating-point arithmetic.

$x - x \to 0$. The expressions x - x and 0 are not equivalent on machines with NaNs or infinities. Such machines include CRAY, VAX, and IEEE machines. (Of course, the two expression are not equivalent on any machine if x has side-effects.)

$x - y \leftrightarrow -(y - x)$. For IEEE machines, in the default rounding direction, 1 - 1 is +0 but -(1 - 1) is -0. The expressions x - y and -(y - x) are not equivalent on the CDC CYBER 205, because of abuse of 1's complement arithmetic. On the other hand, the expressions x - y, x + (-y), and (-y) + x are equivalent on IEEE machines, among others.

$x / 5.0 \leftrightarrow x * 0.2$. Such transformations involving 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.

$0 * x \to 0$. The expressions 0 * x and 0 are not equivalent if x is a NaN or infinite.

$1 * x, x / 1 \to x$. Neither 1 * x and x nor x / 1 and x are equivalent if x has almost overflowed on a CRAY or if x has almost underflowed on a CDC CYBER 170.

$x + 0 \to x$. For IEEE machines, if x is -0 then, in the default rounding direction, x + (+0) yields +0, not x; the implementation cannot replace x + 0 by x unless it can determine that x cannot be -0.

$x - 0 \to x$. For IEEE machines, (+0) - (+0) yields -0 when rounding is downward (toward -∞), but +0 otherwise, and (-0) - (+0) always yields -0; so, the implementation could replace x - 0 by x, even if x might be zero, if it could determine that rounding would not be downward.

$-x \leftrightarrow 0 - x$. For IEEE machines, -(+0) yields -0, but 0 - (+0) yields +0 (unless rounding is downward); the expressions are numerically equivalent for all other cases.

The IEEE floating-point standards prescribe a signed zero to preserve mathematical identities across certain discontinuities. Examples include

```
    1 / (1 / (±INFINITY)) is ±INFINITY
    complex_conjugate(sqrt(z)) is sqrt(complex_conjugate(z))
```

B.3 Relational operators

x != x → false. In IEEE implementations, the statement `x != x` is true if `x` is a NaN. In non-IEEE implementations with NaNs (reserved operands, indefinites) `x != x` may trap.

x == x → true. In IEEE implementations, the statement `x == x` is false if `x` is a NaN. In non-IEEE implementations with NaNs (reserved operands, indefinites) `x == x` may trap.

x !op y ↔ !(x op y). Though equal these expressions are not equivalent on IEEE implementations if *op* includes `<` or `>`, `x` or `y` might be a NaN, and the state of `fenv_access` is *on.*

x op y → !(x !op y). 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 of `fenv_access` is *off.*

The sense of relational operators must be maintained. For IEEE implementations, this includes handling unordered cases as expressed by the source code. (Handling relationals correctly might not require generating worse 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 (a !>= b) f(); else g();    /* calls f without raising invalid
                                         if a and b are unordered */
```

nor, unless the state of `fenv_access` is *off,* to

```
    if (a !< b) g(); else f();     /* calls g without raising invalid
                                         if a and b are unordered */
```

but is equivalent to

```
    if (!(a < b)) g(); else f();
```

B.4 Constant arithmetic

Under the effect of an enabling `fenv_access` pragma, the implementation must honor exceptions raised because of execution-time constant arithmetic. See §3.4, 3.5. Even under the effect of an enabling `fenv_access` pragma, an operation on constants that raises no exception can be folded during translation; implementations that support dynamic rounding precision modes (§4.4.2) should assure further that the result of the operation raises no exception when converted to the semantic type of the operation.

B.5 Wide representation

Implementations employing wide registers must take care to honor appropriate semantics. Values must be independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register must not alter the value. Also, an explicit *store and load* must round to the precision of the storage type. In particular, casts and assignments must perform their specified conversion (ISO §6.2.1.5; ANSI §3.2.1.5): for

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float)expression;
```

the values assigned to `d1` and `d2` must have been converted to `float`.

C. Library Summary

This appendix summarizes library facilities specified, all or in part, in this document.

C.1 General utilities <stdlib.h> (ISO §D.11; ANSI §C.11)

```
double atof(const char *nptr);
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

C.2 Input/output <stdio.h> (ISO §D.10; ANSI §C.10)

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

C.3 Floating-point <fp.h>

```
HUGE_VAL
INFINITY
NAN
FP_NAN
FP_INFINITE
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
fpclassify(floating-expr)
signbit(floating-expr)
isfinite(floating-expr)
isnormal(floating-expr)
isnan(floating-expr)
DECIMAL_DIG
```

```
float_t
double_t
floating-type acos(floating-type x);
floating-type asin(floating-type x);
floating-type atan(floating-type x);
floating-type atan2(floating-type y, floating-type x);
floating-type cos(floating-type x);
floating-type sin(floating-type x);
floating-type tan(floating-type x);
floating-type acosh(floating-type x);
floating-type asinh(floating-type x);
floating-type atanh(floating-type x);
floating-type cosh(floating-type x);
floating-type sinh(floating-type x);
floating-type tanh(floating-type x);
floating-type exp(floating-type x);
floating-type exp2(floating-type x);
floating-type expm1(floating-type x);
floating-type frexp(floating-type value, int *exp);
floating-type ldexp(floating-type x, int exp);
floating-type log(floating-type x);
floating-type log10(floating-type x);
floating-type log1p(floating-type x);
floating-type log2(floating-type x);
floating-type logb(floating-type x);
float modff(float value, float *iptr);
double modf(double value, double *iptr);
long double modfl(long double value, long double *iptr);
floating-type scalb(floating-type x, long int n);
floating-type fabs(floating-type x);
floating-type hypot(floating-type x, floating-type y);
floating-type pow(floating-type x, floating-type y);
floating-type sqrt(floating-type x);
floating-type erf(floating-type x);
floating-type erfc(floating-type x);
floating-type gamma(floating-type x);
floating-type lgamma(floating-type x);
floating-type ceil(floating-type x);
floating-type floor(floating-type x);
floating-type nearbyint(floating-type x);
floating-type rint(floating-type x);
long int rinttol(long double x);
floating-type round(floating-type x);
long int roundtol(long double x);
floating-type trunc(floating-type x);
floating-type fmod(floating-type x, floating-type y);
floating-type remainder(floating-type x, floating-type y);
floating-type remquo(floating-type x, floating-type y, int *quo);
floating-type copysign(floating-type x, floating-type y);
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
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);
```

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

C.4 Floating-point environment <fenv.h>

```
FE_INEXACT
FE_DIVBYZERO
FE_UNDERFLOW
FE_OVERFLOW
FE_INVALID
FE_ALL_EXCEPT
FE_TONEAREST
FE_UPWARD
FE_DOWNWARD
FE_TOWARDZERO
FE_DFL_ENV
fenv_t
fexcept_t
void feclearexcept(int excepts);
void fegetexcept(fexcept_t *flagp, int excepts);
void feraiseexcept(int excepts);
void fesetexcept(const fexcept_t *flagp, int excepts);
int fetestexcept(int excepts);
int fegetround(void);
int fesetround(int round);
void fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
void fesetenv(const fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

D. Implementation Limits

§2.3.1 includes supplementary specification for the Standard C component

```
#define FLT_ROUNDS
```

and specification for additional components

```
#define _MIN_EVAL_FORMAT
#define _WIDEST_NEED_EVAL
```

E. Operator / Function Documentation Guide

Adapted from "Documentation of Special Properties of Programs that Compute Elementary Transcendental Functions in the Context of IEEE Standards 754/854 for Floating-Point Arithmetic, with advice for other systems too" (work in progress), by W. Kahan (July 1991).

This appendix previews a forthcoming guide to the documentation of floating-point operators and functions, including most of those in the C language and libraries. For each operator and function, the guide will discuss in detail the prescribed documentation.

"... Read by a system's user, this guide explores many of the issues that could undermine the portability of numerical software if not addressed. Read by a system's implementor, this guide explains what information ought to be supplied and offers a format for its presentation. An implementation's quality can be gauged by the absence of excuses for some of the sad compromises mentioned here. On the other hand, an implementor who resorted to one of those compromises without documenting it might expect users to regard that compromise more as a bug than a feature and to demand its correction.

"Real functions will be dealt with before complex. At first, only the IEEE's default rounding mode to-nearest will be presumed, and only the default responses to exceptions...

"For systems that do not conform to IEEE 754/854, the number ∞, if unavailable, may be approximated by the largest finite number. If no usable -0 is available, use just 0 instead. And if no NaN-like object is available, replace *NaN* by whatever is the system's usual response to an invalid operation, probably a trap or abortion of execution. Similarly, serious exceptions like overflow and divide-by-zero, if they cannot raise an appropriate flag testable by the program, may have to trap or abort. In any event, serious disruption of a program should be accompanied by a user visible message more informative than, say, a hexadecimal dump. Subnormal numbers, if unusable, must be replaced by zeros with underflow signaled; of course, this signal may be unavailable or practically inaccessible, and hence usually ignored.

"For each function f(x) considered, implementors should supply the following tableau of information: "

Invalid Operands

Invalid operands are those x for which f(x) becomes a new NaN. Example: $x < 0$ for the square root function.

Exact Unexceptional Values

Document these for transcendental functions. Example: sin(±0) returns ±0. f(NaN) returning a NaN need not be included.

Exact Exceptional Values

f(x) returns ±∞ and raises the divide-by-zero exception. Example: log(0).

Inexact Exceptional Values

Overflow	f(x) returns ±∞ and raises the overflow exception.
Underflow	f(x) is subnormal or 0 and raises the underflow exception.
Divide-by-zero	—signaled when f(x) returns ±∞ though x is only approximately a pole, e.g. $\tan(\pi / 2)$.

Symmetries

Like f(-x) == f(x) if f is an even function.

Monotonicity

"(...more important than accuracy!)"

Accuracy

"Accuracy should be described, if not *correctly rounded.* Accuracy should normally be specified in terms of worst-case error in ulps (units in the last place). (Average error or standard deviation is an abuse of statistics, since worst case error is the killer.) Correctly rounded functions are accurate to within at worst half an ulp; most transcendental functions are very difficult to compute much more accurately than to within less than one ulp. Unfortunately, cases exist for which a kind of backward error-bound may be the best that can be described; this means that the computed value of f(x) may actually be within an ulp or two of f(X) where X is within an ulp or two of the given argument x. Such a bound is acceptable in situations where x is uncertain by a few ulps and uncorrelated with anything else that will be combined with f(x); otherwise such a bound has disagreeable consequences when x comes close to singularities of f, so it is a compromise that should be avoided whenever practical."

Other Noteworthy Properties

"In case *correctly rounded* is an impracticable accuracy goal, implementations may yet preserve important properties that would be preserved if every computed value were correctly rounded. Some, like monotonicity and symmetry, have already been mentioned; others are inequalities and identities that involve related functions, for instance `sqrt(x*x) == fabs(x)` absent over/underflow."

F. <fp.h> for IEEE Implementations (ISO §7.5; ANSI §4.5)

This appendix contains recommended specification of library facilities that is particularly suited for IEEE implementations.

The Standard C macro `HUGE_VAL` expands to an expression whose value is a positive infinity; its evaluation raises no exceptions.

`HUGE_VAL` could not be implemented as

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

whose use may raise the divide-by-zero exception.

Special cases for functions in `<fp.h>` are covered directly or indirectly by the IEEE floating-point standards. The functions `sqrt`, `remainder`, `rint`, and `rinttol` have direct counterparts in these standards. Appendixes to these standards specify the functions `copysign`, `logb`, `nextafter`, `scalb`, and `nearbyint`. The function `trunc` is equivalent to `rint` (or, at the implementor's option, to `nearbyint`) with rounding-toward-zero. The `ldexp` function is equivalent to the binary version of the `scalb` function specified in an appendix to the IEEE binary floating-point standard 754. The `fmod` function is readily derived from the standards' `remainder` function (see §F.7.1). The other functions are not covered explicitly by the IEEE floating-point standards; however, except as noted, this specification requires that they honor infinities, NaNs, signed zeros, subnormals, and the exception flags in a manner consistent with the basic arithmetic operations covered by the IEEE floating-point standards.

> Detectable only through invocation of non-Standard C inquiries or exception handling, the exceptions required here are not in conflict with ISO §7.5.1 (ANSI §4.5.1).

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. (See §2.)

The underflow exception is raised whenever a result is tiny (essentially subnormal or zero) and suffers loss of accuracy. This NCEG specification for transcendental functions allows raising the underflow (and inexact) exception when a result is tiny and *probably* inexact.

> The IEEE floating-point standards offer the implementation multiple 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.

> According to the IEEE floating-point standards, an enabled underflow trap is taken if the result would be tiny, regardless of loss of accuracy.

The inexact exception is raised whenever the rounded result is not identical to the mathematical result. Except as noted, this NCEG 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 §2.2.2, the `<fp.h>` functions do not raise spurious exceptions (detectable by the user). 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 and any rounding precision mode is implementation-defined.

Generally, one-parameter functions of a NaN argument return that same NaN and raise no exception. (See §3.1.1.1.)

For the Standard C `<math.h>` functions, the specification in the subsections of §F.1 appends to the Standard C definitions. For the NCEG extensions, it appends to the definitions in §4.3 of this document.

F.1 Trigonometric functions (ISO §7.5.2; ANSI §4.5.2)

F.1.1 The acos function (ISO §7.5.2.1; ANSI §4.5.2.1)

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

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

F.1.2 The asin function (ISO §7.5.2.2; ANSI §4.5.2.2)

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

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

F.1.3 The atan function (ISO §7.5.2.3; ANSI §4.5.2.3)

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

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

F.1.4 The atan2 function (ISO §7.5.2.4; ANSI §4.5.2.4)

```
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.
- `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.

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

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

F.1.5 The cos function (ISO §7.5.2.5; ANSI §4.5.2.5)

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

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

F.1.6 The sin function (ISO §7.5.2.6; ANSI §4.5.2.6)

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

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

F.1.7 The tan function (ISO §7.5.2.7; ANSI §4.5.2.7)

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

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

F.2 Hyperbolic functions (ISO §7.5.3; ANSI §4.5.3)

F.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`.

F.2.2 The asinh function

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

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

F.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`.

F.2.4 The cosh function (ISO §7.5.3.1; ANSI §4.5.3.1)

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

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

F.2.5 The sinh function (ISO §7.5.3.2; ANSI §4.5.3.2)

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

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

F.2.6 The tanh function (ISO §7.5.3.3; ANSI §4.5.3.3)

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

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

F.3 Exponential and logarithmic functions (ISO §7.5.4; ANSI §4.5.4)

F.3.1 The exp function (ISO §7.5.4.1; ANSI §4.5.4.1)

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

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

F.3.2 The exp2 function

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

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

F.3.3 The expm1 function

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

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

F.3.4 The frexp function (ISO §7.5.4.2; ANSI §4.5.4.2)

```
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)) )
```

F.3.5 The ldexp function (ISO §7.5.4.3; ANSI §4.5.4.3)

```
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.

F.3.6 The log function (ISO §7.5.4.4; ANSI §4.5.4.4)

```
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`.

F.3.7 The log10 function (ISO §7.5.4.5; ANSI §4.5.4.5)

```
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`.

F.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`.

F.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`.

F.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.

F.3.11 The modf functions (ISO §7.5.4.6; ANSI §4.5.4.6)

```
float modff(float value, float *iptr);
double modf(double value, double *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 <fenv.h>
#include <fp.h>
#pragma 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);
}
```

F.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.

F.4 Power and absolute value functions (ISO §7.5.5, 7.5.6; ANSI §4.5.5, 4.5.6)

F.4.1 The fabs function (ISO §7.5.6.2; ANSI §4.5.6.2)

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

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

F.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).

Note that hypot(INFINITY, NAN) returns +INFINITY, under the justification that hypot(INFINITY, y) is +INFINITY for any numeric value y.

F.4.3 The pow function (ISO §7.5.5.1; ANSI §4.5.5.1)

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 floating-point standards, `NAN <= +INFINITY` is false and raises the invalid exception.

F.4.4 The sqrt functions (ISO §7.5.5.2; ANSI §4.5.5.2)

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

`sqrt` is fully specified as a basic arithmetic operation in the IEEE floating-point standards.

F.5 Error and gamma functions

F.5.1 The erf function

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

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

F.5.2 The erfc function

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

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

F.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.

F.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.

F.6 Nearest integer functions (ISO §7.5.6; ANSI §4.5.6)

F.6.1 The ceil function (ISO §7.5.6.1; ANSI §4.5.6.1)

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 <fenv.h>
#include <fp.h>
#pragma 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;
}
```

F.6.2 The floor function (ISO §7.5.6.3; ANSI §4.5.6.3)

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

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

See the sample implementation for ceil in §F.6.1.

F.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.

F.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.

F.6.5 The rinttol function

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

The `rinttol` function provides floating-to-integer conversion as prescribed by the IEEE standards [2], [3] §5.4. 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.

F.6.6 The round function

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

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

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

F.6.7 The roundtol function

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

roundtol differs from rinttol 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.

F.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).

F.7 Remainder functions (ISO §7.5.6; ANSI §4.5.6)

F.7.1 The fmod function (ISO §7.5.6.4; ANSI §4.5.6.4)

```
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>
double fmod(double x, double y)
{
    double result;
    result = remainder(fabs(x), (y = fabs(y)));
    if (signbit(result)) result += y;
    return copysign(result, x);
}
```

F.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 floating-point standards.

F.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.

F.8 Manipulation functions

F.8.1 The copysign function

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

copysign is specified in appendices to the IEEE floating-point standards. It has no specification special to IEEE implementations.

F.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.

F.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.

F.9 Maximum, minimum, and positive difference functions

F.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.

F.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

```
    isnan(y) ? x : (y !<= x ? y : x)
```

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.

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

```
isnan(y) ? y : (y !> x ? x : y)
```

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

F.9.3 The fmin functions

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

`fmin` is analogous to `fmax`. See §F.9.2.