Floating-Point Arithmetic—C9X Edits

WG14/N511 X3J11/95-112 (Draft 12/21/95)

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

This is a proposal for changes to existing parts of the C9X draft document, in order to incorporate FPCE. The edits refer to C9X Draft 3. (Subsequent C9X drafts are generally satisfactory for understanding the proposed edits.)

Feature: "Recommended practice" designation

3 Definitions and conventions

Insert the following definition:

> **3.xx recommended practice:** Sections so entitled contain specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations.

Feature: Expression evaluation methods

5.1.2.3 Program execution

Remove the last sentence of example 3.

5.2.4.2.2 Characteristics of floating types `<float.h>`

Replace the third paragraph with:

> Of the values in the `<float.h>` header, **`FLT_EVAL_METHOD`** and **`FLT_RADIX`** shall be constant expressions suitable for use in **`#if`** preprocessing directives; all other values need not be constant expressions. All except **`FLT_EVAL_METHOD`**, **`FLT_RADIX`**, and **`FLT_ROUNDS`** have separate names for all three floating-point types. The floating-point model representation is provided for all values except **`FLT_EVAL_METHOD`** and **`FLT_ROUNDS`**.

After the definition of `FLT_ROUNDS` add:

> The values of operations subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the value of **`FLT_EVAL_METHOD`**:

-1 indeterminable
0 evaluate all operations and constants just to the range and precision of the type
1 evaluate operations and constants of type `float` and `double` to the range and precision of the `double` type, evaluate `long double` operations and constants to the range and precision of the `long double` type
2 evaluate all operations and constants to the range and precision of the `long double` type

All other values for `FLT_EVAL_METHOD` characterize implementation-defined behavior.

[Well defined expression evaluation is essential for predictable arithmetic. 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.

Standard C generally defines the semantic type of a floating-point 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.

Representation of constants in a format commensurate with expression evaluation, not a traditionally uniform practice, better meets certain expectations than would representation strictly according to semantic type—for example, `0.1f == 1.0f/10.0f`, even if `float` operations are evaluated to `double` or `long double`. Viewed as translation-time operations that convert decimal strings to internal floating representations, literal floating constants naturally follow the method of expression evaluation.

Early drafts for this specification 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. The evaluation of `FLT_EVAL_METHOD` is intended to correctly reflect the expression evaluation method currently in effect.

In the following description of floating-point architectures, the terms *extended*, *double*, and *single* 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

NUMBERING CONTINUED WITH P. 94

74

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.

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. `FLT_EVAL_METHOD` equal 1 is a natural choice for double-based architectures.

Implementations for single/double and single/double/extended architectures may find `FLT_EVAL_METHOD` equal 0 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 have `FLT_EVAL_METHOD` equal 1, 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.

`FLT_EVAL_METHOD` equal 2 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 5.1.2.3).

"Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report defined *widest-need* expression evaluation methods: 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 an implementation-defined minimum evaluation format (`float`, `double`, or `long double`). 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. The definition of widest-need is based on a similar scheme for Fortran presented in [9]. It does not affect integer expression evaluation.

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.

This specification does not define widest-need methods because of the scarcity of prior art. Of course, an implementation could provide widest-need, as prescribed in the Technical Report, as an optional expression evaluation method.

"Floating-Point C Extensions" in the X3J11 Numerical C Extensions Technical Report specifies three pragmas granting 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. 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. The **`fp_wide_function_returns`** and **`fp_wide_function_parameters`** pragmas may affect function definitions or calls but not prototypes.

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. 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.

This specification does not include **`fp_wide`** pragmas, nor macro equivalents, because their use and meaning would vary so much from one implementation to the next. Of course and implementation could provide them as an extension.]

Feature: Dynamic FLT_ROUNDS

5.2.4.2.2 Characteristics of floating types `<float.h>`

In the fourth paragraph, add the following footnote to "rounding mode":

> Evaluation of `FLT_ROUNDS` correctly reflects any execution-time change of rounding mode through the function `fesetround` in `<fenv.h>`.

Feature: Hexadecimal floating constants

3. Definitions and conventions

Add the following definition:

> **3.x correctly rounded result:** A representation in the result format that is nearest in value, subject to the effective rounding mode, to what the result would be given unlimited range and precision.

6.1.3.1 Floating constants

Replace the syntax with the following syntax:

Syntax

floating-constant:
  decimal-floating-constant
  hexadecimal-floating-constant

decimal-floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

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*$_{opt}$

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

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

sign: one of
    `+ -`

digit-sequence:
    digit
    digit-sequence digit

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

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

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

hexadecimal-digit: one of

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
```

floating-suffix: one of
    `f l F L`

In the third sentence of the Description, replace "`e` or `E`*" with "*`e`, `E`, `p`, or `P`*".*

Replace the second clause of the last sentence of the Description with:

for decimal floating constants, either the period or the exponent part shall be present.

Replace the first paragraph of the Semantics with:

The significand part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significand part is to be scaled. For hexadecimal floating constants the exponent indicates the power of 2 by which the significand part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when `FLT_RADIX` is not a power of 2, if the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants, if `FLT_RADIX` is a power of 2 and the scaled value is in the range of representable values (for its type), then the result of a hexadecimal floating constant is correctly rounded.

Recommend practice

The implementation emits a non-fatal diagnostic if a hexadecimal constant cannot be represented exactly in its evaluation format.

6.1.3.2 Integer constants

Delete the hexadecimal-digit syntax (now in 6.1.3.1).

6.1.8 Preprocessing numbers

Before the last line of the Syntax insert:

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

[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).

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. The character sequence `0x1.FFFFFEp128f` might appear to be an IEEE single-format NaN, which is characterized by an unbiased exponent of 128 and a nonzero significand; however, this character sequence overflows to an infinity in the single format.

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 7.x.

Note that constants of `long double` type are not fully portable, even among IEEE implementations. See rationale in Annex X.2.

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

How about using hexadecimal constants in the examples in 5.2.4.2.2, instead of, or in addition to, the decimal ones?

7.10.1.4 The `strtod` function

Replace the first sentence of the second paragraph of the Description with:

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part as defined in 6.1.3.1;
- a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point character, then an optional binary-exponent part, where either the decimal-point character or the binary-exponent part is present;

but no floating suffix.

In the first sentence of the third paragraph of the Description, replace "if neither an exponent part nor a decimal-point character appears" *with* "if neither an exponent part, a binary-exponent part, nor a decimal-point character appears".

After the fourth paragraph of the Description insert the paragraph:

If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, then the value resulting from the conversion is correctly rounded.

After the Description insert:

Recommended practice

If the subject sequence has the hexadecimal form and `FLT_RADIX` is not a power of 2, then the result is 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 a correct sign for the current rounding direction.

7.16.4.1.1 The `wcstod` function

Replace the first sentence of the second paragraph of the Description with:

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part as defined for the corresponding single-byte characters in subclause 6.1.3.1;
- a `0x` or `0X`, then a nonempty sequence of hexadecimal digits optionally containing a decimal-point wide character, then an optional binary-exponent part, where either the decimal-point character or the binary-exponent part is present;

but no floating suffix.

In the first sentence of the third paragraph of the Description, replace "if neither an exponent part nor a decimal-point wide character appears" *with* "if neither an exponent part, a binary-exponent part, nor a decimal-point wide character appears".

After the fourth paragraph of the Description insert the paragraph:

If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, then the value resulting from the conversion is correctly rounded.

After the Description insert:

Recommended practice

If the subject sequence has the hexadecimal form and `FLT_RADIX` is not a power of 2, then the result is 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 a correct sign for the current rounding direction.

7.9.6.1 The `fprintf` function

In the third bullet of the second paragraph of the Description, replace "`e, E,`*" with "*`a, A, e, E,`*".*

In the fourth bullet of the second paragraph of the Description, replace "`e, E,`*" with "*`a, A, e, E,`*".*

In the # flag specification in the fourth paragraph of the Description, replace "`e, E,`*" with "*`a, A, e, E,`*".*

In the 0 flag specification in the fourth paragraph of the Description, replace "`e, E,`*" with "*`a, A, e, E,`*".*

After the g, G conversion specifier and its meaning in the fifth paragraph, insert:

> `a, A` The `double` argument is converted 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 and `FLT_RADIX` is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and `FLT_RADIX` is not a power of 2, then the precision is sufficient to distinguish values of type `double`, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point character is nonzero for normalized floating-point numbers and is otherwise unspecified; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `a` conversion specifier will produce a number with `x` and `p` and the `A` conversion specifier will produce a number with `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. If the value is zero, the exponent is zero.

In the second sentence of the preceding inserted text, footnote the word "distinguish" with:

> 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.

In the third sentence of the preceding inserted text, footnote the word "implementation" with:

> Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble boundaries.

At the end of the Description append the paragraphs:

For **a** and **A** conversions, if **FLT_RADIX** is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

If **FLT_RADIX** is not a power of 2, the result is one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have a correct sign for the current rounding direction.

[To illustrate alignment to nibble (4-bit) boundaries, 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 7.1.1. *Radix-point character* would be a better term.

Use of the **A** format specifier constitutes a minor extension to ISO/IEC 9899 : 1990 (E) which does not reserve it.]

7.16.2.1 The fwprintf function

*In the third bullet of the third paragraph of the Description, replace "***e, E,***" with "***a, A, e, E,***".*

*In the fourth bullet of the third paragraph of the Description, replace "***e, E,***" with "***a, A, e, E,***".*

*In the # flag specification in the fifth paragraph of the Description, replace "***e, E,***" with "***a, A, e, E,***".*

*In the 0 flag specification in the fifth paragraph of the Description, replace "***e, E,***" with "***a, A, e, E,***".*

After the g, G conversion specifier and its meaning in the sixth paragraph, insert:

a, A The **double** argument is converted in the style *[-]0xh.hhhhp±d*. The number of hexadecimal digits *h* after the decimal-point wide character is equal to the precision; if the precision is missing and **FLT_RADIX** is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and `FLT_RADIX` is not a power of 2, then the precision is sufficient to distinguish values of type double, except that trailing zeros may be omitted. The hexadecimal digit to the left of the decimal-point wide 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 wide character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `a` conversion specifier will produce a number with `x` and `p` and the `A` conversion specifier will produce a number with `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.

In the second sentence of the preceding inserted text, footnote the word "distinguish" with:

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 wide character.

In the third sentence of the preceding inserted text, footnote the word "implementation" with:

Binary implementations can choose the hexadecimal digit to the left of the decimal-point wide character so that subsequent digits align to nibble (4-bit) boundaries.

At the end of the Description append the paragraphs:

For `a` and `A` conversions, if `FLT_RADIX` is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

If `FLT_RADIX` is not a power of 2, the result is one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error have a correct sign for the current rounding direction.

7.9.5.8 The `fscanf` function

In the third bullet of the second paragraph of the Description, replace "`e`, `f`, and `g`" *with* "`a`, `e`, `f`, and `g`".

In the Description paragraph listing the conversion specifiers, replace "`e, f, g`" *with* "`a, e, f, g`".

In the fourth to last paragraph in the Description, replace "specifiers `E`," *with* "specifiers `A`, `E`,". *In the same paragraph, replace* "respectively, `e`," *with* "respectively, `a`, `e`,".

7.16.2.2 The `fwscanf` function

In the third bullet of the second paragraph of the Description, replace "`e`, `f`, and `g`" *with* "`a`, `e`, `f`, and `g`".

In the Description paragraph listing the conversion specifiers, replace "`e, f, g`" *with* "`a, e, f, g`".

In the third to last paragraph in the Description, replace "specifiers `E`," *with* "specifiers `A, E`,". *In the same paragraph, replace* "respectively, `e`," *with* "respectively, `a`, `e`,".

Feature: Output of wide exponents

7.9.6.1 The `fprintf` function

In the e item in the list of conversion specifiers and their meanings, in the next to last sentence, replace "at least two digits" *with* "at least two digits, and only as many more digits as necessary to represent the exponent".

7.16.2.1 The `fwprintf` function

In the e item in the list of conversion specifiers and their meanings, in the next to last sentence, replace "at least two digits" *with* "at least two digits, and only as many more digits as necessary to represent the exponent".

Feature: Support for infinities, NaNs, and -0

6.1.2.5 Types

To the sixth paragraph append :

> Values of floating types might include infinities and NaNs, as well as floating-point numbers. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising an exception. A *signaling NaN* generally raises an exception when occurring as an arithmetic operand.

7.10.1.4 The `strtod` function

In the first sentence of the second paragraph of the Description, as amended by the change above for hexadecimal floating constants, after the last bullet, insert:

- one of `INF` or `INFINITY`, ignoring case
- one of `NAN` or `NAN(`*n-char-sequence*$_{opt}$ `)`, ignoring case in the `NAN` part, where

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

In the first sentence of the third paragraph of the Description replace "expected form" *with* "expected form for a floating-point number".

After the first sentence of the third paragraph of the Description insert:

A character sequence `INF` or `INFINITY` is interpreted as an infinity, if representable in the `double` type, else like a floating constant that is too large for the range of `double`. A character sequence `NAN` or `NAN(`*n-char-sequence*$_{opt}$`)` is interpreted as a quiet NaN, if supported in the `double` type, else like a subject sequence part that does not have the expected form; the meaning of the n-char sequences is implementation-defined.

In the preceding insertion, footnote "implementation-defined" with:

An implementation may use the *n-char-sequence* to determine extra information to be represented in the NaN's significand.

In the next to last sentence of the third paragraph of the Description, footnote "negated" with:

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

[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 standard does 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 were provided through straightforward extensions, rather than through a new locale. Note also that Standard C locale categories do not affect the representations of infinities and NaNs.

A proposal that `strtod` be allowed to return a NaN for invalid numeric input, as recommended by IEEE standard 854, was withdrawn because of the incompatibility with the C standard ISO/IEC 9899 : 1990 (E), which demands that `strtod` return 0 for invalid numeric input.]

7.16.4.1.1 The `wcstod` function

In the first sentence of the second paragraph of the Description, as amended by the change above for hexadecimal floating constants, after the last bullet, insert:

- one of `INF`, `INFINITY`, or any wide string equivalent except for case

- one of **`NAN`**, or **`NAN(`***n-wchar-sequence*$_{opt}$ **`)`**, or any wide string equivalent except for case in the **`NAN`** part, where

> *n-wchar-sequence:*
> *digit*
> *nondigit*
> *n-wchar-sequence digit*
> *n-wchar-sequence nondigit*

In the first sentence of the third paragraph of the Description replace "expected form" *with* "expected form for a floating-point number".

After the first sentence of the third paragraph of the Description insert:

> A wide character sequence **`INF`** or **`INFINITY`** is interpreted as an infinity, if representable in the **`double`** type, else like a floating constant that is too large for the range of **`double`**. A wide character sequence **`NAN`** or **`NAN(`***n-wchar-sequence*$_{opt}$**`)`** is interpreted as a quiet NaN, if supported in the **`double`** type, else like a subject sequence part that does not have the expected form; the meaning of the n-wchar sequences is implementation-defined.

In the preceding insertion, footnote "implementation-defined" with:

> An implementation may use the *n-wchar-sequence* to determine extra information to be represented in the NaN's significand.

In the next to last sentence of the third paragraph of the Description, footnote "negated" with:

> The **`wcstod`** function honors the sign of zero if the arithmetic supports signed zeros.

7.9.6.1 The `fprintf` function

In the third bullet of the second paragraph of the Description, replace "**`E`**, and **`f`**" *with* "**`E`**, **`f`**, and **`F`**".

In the fourth bullet of the second paragraph of the Description, replace "**`E`**, **`f`**," *with* "**`E`**, **`f`**, **`F`**,".

Footnote the + flag specification in the fourth paragraph of the Description with:

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

In the # flag specification in the fourth paragraph of the Description, replace "**`E`**, **`f`**," *with* "**`E`**, **`f`**, **`F`**,".

In the 0 flag specification in the fourth paragraph of the Description, replace "**`E`**, **`f`**," *with* "**`E`**, **`f`**, **`F`**,".

In the list of conversion specifiers and their meanings, change the "**`f`**" *bullet to* "**`f`**, **`F`**".

In the f item in the list of conversion specifiers and their meanings, replace "**`double`** argument" *with* "**`double`** argument representing a floating-point number".

At the end of the f item in the list of conversion specifiers and their meanings, append the paragraph:

> A `double` argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity`—which style is implementation-defined. A `double` argument representing a NaN 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. The `F` conversion specifier will produce an `INF`, `INFINITY`, or `NAN` instead of an `inf`, `infinity`, or `nan`.

Footnote the preceding inserted paragraph with:

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

In the e item in the list of conversion specifiers and their meanings, replace "`double` argument" *with* "`double` argument representing a floating-point number".

At the end of the e item in the list of conversion specifiers and their meanings, append the sentence:

> A `double` argument representing an infinity or a NaN is converted in the style of an `f` or `F` conversion specifier.

In the newly inserted a item in the list of conversion specifiers and their meanings, replace "`double` argument" *with* "`double` argument representing a floating-point number".

In the newly inserted a item in the list of conversion specifiers and their meanings, append the sentence:

> A `double` argument representing an infinity or a NaN is converted in the style of an `f` or `F` conversion specifier.

In the g item in the list of conversion specifiers and their meanings, replace "style `E`" *with* "style `E` or `F`".

> [See the rationale above for `strtod`.
>
> Use of the `F` format specifier constitutes a minor extension to ISO/IEC 9899 : 1990 (E) which does not reserve it.]

7.16.2.1 The `fwprintf` function

In the third bullet of the third paragraph of the Description, replace "`E`, and `f`" *with* "`E`, `f`, and `F`".

In the fourth bullet of the third paragraph of the Description, replace "`E`, `f`," *with* "`E`, `f`, `F`,".

Footnote the + flag specification in the fifth paragraph of the Description with:

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

In the # flag specification in the fifth paragraph of the Description, replace "`E, f,`" *with* "`E, f, F,`".

In the 0 flag specification in the fifth paragraph of the Description, replace "`E, f,`" *with* "`E, f, F,`".

In the list of conversion specifiers and their meanings, change the "`f`" *bullet to* "`f, F`".

In the f item in the list of conversion specifiers and their meanings, replace "`double` argument" *with* "`double` argument representing a floating-point number".

At the end of the f item in the list of conversion specifiers and their meanings, append the paragraph:

> A `double` argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity`—which style is implementation-defined. A `double` argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-wchar-sequence*`)`—which style, and the meaning of any *n-wchar-sequence*, is implementation-defined. The `F` conversion specifier will produce an `INF`, `INFINITY`, or `NAN` instead of an `inf`, `infinity`, or `nan`.

Footnote the preceding inserted paragraph with:

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

In the e item in the list of conversion specifiers and their meanings, replace "`double` argument" *with* "`double` argument representing a floating-point number".

At the end of the e item in the list of conversion specifiers and their meanings, append the sentence:

> A `double` argument representing an infinity or a NaN is converted in the style of an `f` or `F` conversion specifier.

In the newly inserted a item in the list of conversion specifiers and their meanings, replace "`double` argument" *with* "`double` argument representing a floating-point number".

In the newly inserted a item in the list of conversion specifiers and their meanings, append the sentence:

> A `double` argument representing an infinity or a NaN is converted in the style of an `f` or `F` conversion specifier.

In the g item in the list of conversion specifiers and their meanings, replace "style `E`" *with* "style `E` or `F`".

Feature: Strings -> float, long double

7.10.1 String conversion functions

Add two new subclauses and renumber others accordingly (note that HUGE_VALF and HUGE_VALL are defined in <math.h>):

7.10.1.4 The `strtof` function

Synopsis

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

Description

The `strtof` function is similar to the `strtod` function, except the returned value has type `float` and plus or minus `HUGE_VALF` is returned for values outside the range.

7.10.1.7 The `strtold` function

Synopsis

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

Description

The `strtold` function is similar to the `strtod` function, except the returned value has type `long double` and plus or minus `HUGE_VALL` is returned for values outside the range.

The names strtold and wcstold match strtol and wcstol in the first six characters. Also the <fenv.h> functions fesetenv and fesetexcept match in the first six characters. An issue?

7.16.4.1 Wide-string numeric conversion functions

Add two new subclauses and renumber others accordingly (note that HUGE_VALF and HUGE_VALL are defined in <math.h>):

7.16.4.1.x The `wcstof` function

Synopsis

```
#include <wchar.h>
float wcstof(const wchar_t *nptr, wchar_t **endptr);
```

Description

The `wcstof` function is similar to the `wcstod` function, except the returned value has type `float` and plus or minus `HUGE_VALF` is returned for values outside the range.

7.16.4.1.x The `wcstold` function

Synopsis

```
#include <wchar.h>
long double wcstold(const wchar_t *nptr, wchar_t **endptr);
```

Description

The `wcstold` function is similar to the `wcstod` function, except the returned value has type `long double` and plus or minus `HUGE_VALL` is returned for values outside the range.

Feature: FP environment management

5.1.2.3 Program execution

In the first sentence of the second paragraph, footnote "side effects" with:

The ANSI/IEEE floating-point standard 754 (IEC 559) requires certain status flags and control modes, with user access. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state will need to regard changes to it as side effects—see Annex X for details. The floating-point environment library `<fenv.h>` provides a programming facility for indicating when these side effects matter, freeing the implementation in other cases.

Feature: Correctly rounded binary-decimal conversion

6.1.3.1 Floating constants

Insert after semantics (as revised above) the paragraph:

Recommended practice

The translation-time conversion of floating constants matches the execution-time conversion of character strings by library functions, such as `strtod`, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.

Footnote the preceding paragraph with:

The specification for the library functions recommends more accurate conversion than required for floating constants. See 7.10.1.4.

7.10.1.4 The `strtod` function

After the Description insert the paragraph:

Recommended practice

If the subject sequence has the decimal form and at most `DECIMAL_DIG` (defined in `<math.h>`) significant digits, then the value resulting from the conversion is correctly rounded. If the subject sequence *D* has the decimal form and more than `DECIMAL_DIG` significant 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 $L \leq D \leq U$. 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 a correct sign for the current rounding direction.

Footnote the preceding inserted paragraph with:

`DECIMAL_DIG`, defined in `<math.h>`, is recommended to be sufficiently large that *L* and *U* will usually round to the same internal floating value, but if not will round to adjacent values.

7.16.4.1.1 The `wcstod` function

After the Description insert the paragraph:

Recommended practice

If the subject sequence has the decimal form and at most `DECIMAL_DIG` (defined in `<math.h>`) significant digits, then the value resulting from the conversion is correctly rounded. If the subject sequence *D* has the decimal form and more than `DECIMAL_DIG` significant 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 $L \leq D \leq U$. 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 a correct sign for the current rounding direction.

Footnote the preceding inserted paragraph with:

`DECIMAL_DIG`, defined in `<math.h>`, is recommended to be sufficiently large that *L* and *U* will usually round to the same internal floating value, but if not will round to adjacent values.

7.9.6.1 The `fprintf` function

At the end of the Description insert the paragraph:

Recommended practice

For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most `DECIMAL_DIG`, then the result is correctly rounded. 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 $L \leq D \leq U$, with the extra stipulation that the error have a correct sign for the current rounding direction.

In the first sentence of the preceding paragraph, footnote "correctly rounded" with:

For binary-to-decimal conversion, the result 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.

7.16.2.1 The `fwprintf` function

At the end of the Description insert the paragraph:

Recommended practice

For `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most `DECIMAL_DIG`, then the result is correctly rounded. 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 $L \leq D \leq U$, with the extra stipulation that the error have a correct sign for the current rounding direction.

In the first sentence of the preceding paragraph, footnote "correctly rounded" with:

For binary-to-decimal conversion, the result 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.

Feature: Contraction control

6.3 Expressions

Append to the subclause the paragraph :

A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method. The `fp_contract` macros in `<math.h>` provide a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation defined.

Footnote the first sentence in the preceding new paragraph with:

A contracted expression might also omit side effects such as the raising of floating-point exception flags.

Footnote the preceding new paragraph with:

This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use must be well-defined and clearly documented.

[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 contractions include:

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

A contraction might omit a rounding error that would have fortuitously improved accuracy, so that the more accurate evaluation of a subexpression might result in a less accurate evaluation of the containing expression.

Contractions can lead to subtle anomalies even while increasing overall 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 and HP PA8000's *fused* multiply-add, which produces a result with just one rounding, can be exploited for simpler and faster codes. See [19] for details.]

Feature: Optimization guidance

5.1.2.3 Program execution

Add an example after current example 3:

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: for the fragment

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

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

Add an example after current example 4:

Rearrangement for floating-point expressions is restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative laws for addition or multiplication, nor the distributive law, because of roundoff error, even in the absence of overflow and underflow. Likewise, the implementation cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical laws are not valid. See Annex X.8.

```
double x, y, z;
/*...*/
x = (x * y) * z;       /* not equivalent to x *= y * z; */
z = (x - y) + y;       /* not equivalent to z = x; */
z = x + x * y;         /* not equivalent to z = x * (1.0 + y); */
y = x / 5.0;           /* not equivalent to y = x * 0.2; */
```

Feature: Optional IEEE support

6.8.8 Predefined macro names

After the first paragraph insert the paragraph:

The following macro name is defined if and only if the implementation conforms to Appendix X:

`__IEEE_FP__` The decimal constant 1.

94