A Rationale for Floating-Point C Extensions

Jim Thomas
Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233

Jerome T. Coonen
C-Labs
4035 Orme Street
Palo Alto, CA 94306

Abstract

This article describes a specification for C extensions designed to address two long-standing needs: (1) more predictable floating-point arithmetic and (2) a standard language binding for floating-point features, notably of IEEE standard arithmetic. The proposal is practical, as pioneer implementations attest, and its impact on C is modest. It reflects substantial experience from prior art and extensive review by experts from the C and numerical communities. The ANSI C committee (X3J11) has accepted the specification for inclusion in its forthcoming Technical Report on Numerical C Extensions.

Introduction

This article presents an overview and rationale for *Floating-Point C Extensions* [1] (FPCE), a specification of extensions to the C language to support floating-point computation. The rationale covers issues of specification, implementation, and use, because while FPCE's benefits are mainly to programmers and users, its costs, arguably modest, are to those defining and implementing the language. First, this introduction presents a sample of FPCE's features, describes FPCE's objectives, and explains some of the effort behind its development. The next two sections, "Predictability" and "Language Bindings," discuss how FPCE's language specification addresses the objectives. "Libraries" gives an overview of FPCE's libraries, which supersede the historical `<math.h>`, and "C Integration" itemizes FPCE's impact on C.

Readers acquainted with the IEEE floating-point arithmetic standards [2, 3] will find this discussion familiar territory. Readers should know C, though not necessarily to the level of detail of the standards documents [4, 5]. The draft of FPCE is available from the ANSI C Language Committee for anonymous FTP [1].

An Elementary Function

The best way to grasp the proposed extensions to C is to see them in action. Here is a sample implementation of the FPCE library function `atanh`, the inverse hyperbolic tangent, for a `double` argument. This is a robust implementation for IEEE standard environments, rich in FPCE features (set in bold face) and typical of a library function that would deal with the full range of input in a manner consistent with IEEE arithmetic.

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

#pragma fenv_access on
double sample_atanh( double x ) { /* no name conflict */
    double_t  tmp = fabs(x);
    double  result;
    fenv_t  saved_env;

    fegetenv(&saved_env);
    fesetenv(FE_DFL_ENV); /* default rounding, etc. */

    if (isless(tmp, 0x1.0p-27)) { /* threshold 2^-27 */
        if (tmp > 0.0)
            feraiseexcept(FE_INEXACT);
    } else
        tmp = 0.5 * log1p(2.0 * tmp / (1.0 - tmp)) );
    result = copysign(tmp, x);

    feupdateenv(&saved_env);
    return result;
}
```

The two `#include` directives give access to the FPCE headers defining mathematical features for computation and environmental features for treatment of exceptional circumstances. The `#pragma` instructs the compiler that this code manipulates the floating-point environment, precluding any optimization, such as certain code movement, that would affect the setting of exception flags in the "natural" sequence of execution.

The temporary variable `tmp` has the type `double_t`, a `typedef` defined in `<fp.h>` providing a portable request for the implementation's most efficient type at least as wide as `double`. That is, the type `double_t` refers to `double` except on machines whose `long double` type is more efficient than `double`.

The computation is bracketed by calls to `fegetenv` and `fesetenv` to save and set the floating-point environment, and a call to `feupdateenv` to restore the environment. The first calls guarantee that the computation runs non-stop with the default rounding direction. The latter call restores the prior state and ensures that any exceptions that arise during the computation are raised in the caller's floating-point environment.

The function atanh is an odd function, that is, $\operatorname{atanh}(-z) = -\operatorname{atanh}(z)$, so for convenience the computation proceeds with `tmp = fabs(x)`, the absolute value of the argument. The argument's sign is copied back onto the result when it is coerced to `double`.

The approximation breaks down into two cases. First, if the argument is so small in magnitude that its square would round off completely when added to 1, then $\operatorname{atanh}(z) = z$ to within a rounding error. The threshold for this test, `0x1.0p-27`, exhibits the hexadecimal floating constant syntax—one of FPCE's few extensions to the language. Because the double format has 53 significant bits, 2^{-27} is a reasonable choice for the threshold (even if `double_t` is wider than `double`). Although a glance at a scientific calculator would give `7.45e-9` as a decimal approximation to 2^{-27}, the expression `0x1.0p-27` is a more natural representation. It can be guessed from the context that the fragment `0x1.0` is a fixed-point string of hexadecimal digits with a radix point separating its integer and fraction parts. The fragment `p-27` signifies the scale factor 2^{-27}.

The threshold test uses the macro function `isless(a, b)`, which performs the test `a < b`, to filter small operands. Unlike the built-in C relational, `isless` does not raise an invalid operation exception when either operand is a NaN, a non-numeric floating-point value specified by the IEEE standards. Once the argument is known to be small, it remains to distinguish the special case 0, because $\operatorname{atanh}(0) = 0$ exactly. At this stage the argument is known not to be a NaN (which compares unordered with, rather than less than, equal to, or greater than, any quantity), so the ordinary C relational operator suffices. The call `feraiseexcept(FE_INEXACT)` indicates the rounding error.

In the second case, for magnitudes above the threshold but no greater than 1, the formula $\operatorname{atanh}(z) = \frac{1}{2}\ln\left[\frac{1+z}{1-z}\right]$ applies [6]. A more numerically stable form of the latter expression is $\frac{1}{2}\ln\left[1+\frac{2z}{1-z}\right]$, which appears in the example. Note the use of `log1p`, which computes the natural logarithm of its argument as if added to 1 but without performing the addition—and without rounding off significant bits of $\frac{2z}{1-z}$. This branch of the code incorporates several special cases as well. If the magnitude of the argument is 1, then the formula produces the floating-point value ∞ and raises the divide by zero exception (which is consistent with the underlying arithmetic). If the magnitude is greater than 1, the formula produces a NaN and raises the invalid operation exception. And if the value is a NaN, that NaN simply passes through the computation without exception.

The final assignment, to the `double` variable `result`, restores the algebraic sign and coerces the value to `double` format, rounding and checking for overflow. (It is an interesting exercise to show that teeny values, though inexact and subnormal, are correctly rounded to full `double` precision, so that `atanh` need never raise underflow.)

This example conveys the spirit and many of the features of FPCE. Further examples given later in the discussion compare this programming style with what has been possible before this proposal.

Objectives

The objectives of FPCE for floating-point arithmetic within C are

- language bindings for certain features, especially those of the IEEE standards
- predictable behavior

Two more objectives, actually constraints, are

- practicality for implementation and use
- upward compatibility with Standard C and with C++

FPCE defines two levels of conformance. The basic level, for all general-purpose C implementations, is designed to promote more predictable floating-point arithmetic. The higher level, for implementations on IEEE standard systems, provides programmers convenient access to the features of their underlying arithmetic.

Background

In the late 1970s, when the microprocessor industry was moving toward hardware support of floating-point arithmetic, an IEEE standards committee began its deliberations on a standard that might avert the proliferation of designs that plagued minicomputers and mainframes. By the time the IEEE adopted its binary floating-point standard in 1985 [1], several conforming implementations were already available. Now, virtually all new implementations conform to IEEE 754 or its radix-free successor IEEE 854 [2]—at least in format, if not to the last detail. Although the standards describe the high-level environment in which numerical programmers work, their vague language falls far short of specific language bindings. So, despite their success in influencing hardware designers, the standards have not had their full impact at the programmers' level.

During the mid-1980s, the ANSI C committee attempted to eliminate conflicts with IEEE arithmetic, but did not specify language bindings. In the meantime, various companies have defined their own IEEE language extensions and libraries; not surprisingly, lack of portability has impeded programming for these interfaces. Finally, sixteen years after the first concerted standardization effort, FPCE offers a standard language binding for IEEE arithmetic.

Unpredictability of floating-point arithmetic is an even older problem, plaguing both programmers and users. For example, very large or small values may misbehave in what ought to be benign operations. A simple expression like `(a*b)+(s*t)` might yield a different value when rewritten as `(s*t)+(a*b)` because of the ill-conceived design or use of extra-precision registers. Or `a*x` might compare unequal to `a*y` even though `x==y`. Although the IEEE standards attempt to address these anomalies by careful specification of results of basic floating-point operations, the standards failed to achieve the expected level of predictability because of the myriad approaches to implementing and optimizing the language constructs.

In 1989 the Numerical C Extensions Group embarked on enhancements to C in support of numerical programming. NCEG launched the FPCE effort primarily to specify support for IEEE arithmetic. The goal was to define language bindings for IEEE constructs and to increase predictability without sacrificing efficiency. But from the beginning substantial portions of the specification suited non-IEEE implementations, too. Soon FPCE's charter expanded to encompass all Standard C systems.

Pursuing not only predictability but also practicality for the diversity of architectures proved to be FPCE's biggest challenge. The specification effort was variously drawn toward extremes of the theoretically ideal (where implementation is impractical or its outcome inefficient) and arithmetic anarchy (with its lack of predictability). The challenge was to see FPCE implemented across the variety of prevailing architectures at low cost in performance and complexity. FPCE's solution was to allow implementation options where dictated by architectural diversity, but to keep each option well defined and the total number of options to a minimum.

From the beginning, FPCE's designers sought compatibility with the philosophy and style of Standard C. FPCE extends the C language only where justified and then mimics similar existing constructs; its impact on Standard C is surprisingly small. Although specifically developed for C, FPCE was designed with C++ adaptability in mind. FPCE, with minor changes, has already been implemented in C++.

Credentials

At its initial meeting, NCEG identified support for IEEE floating-point arithmetic as one of its focus areas, and organized a subgroup to produce a specification for inclusion in a technical report. In December 1993, just before being incorporated into the ANSI C Language Committee (X3J11), NCEG (then X3J11.1, a subcommittee of X3J11) approved FPCE for forwarding to X3J11 for the technical report. In June 1994, X3J11 accepted FPCE (document numbers WG14/N350, X3J11/94-035) for its Technical Report on Numerical C Extensions. X3J11 will consider the technical report for its upcoming revision work on the C standard.

FPCE has benefited from the substantial C and IEEE floating-point expertise of the NCEG and X3J11 members and numerous other interested parties. Active contributors and reviewers have included key members of the C standard committee, creators of the IEEE floating-point standards, individuals who previously had developed proprietary language extensions for IEEE arithmetic, and others in the process of implementing FPCE features. Proprietary language extensions for IEEE support have provided prior art for many features.

FPCE has enjoyed wide exposure and review. NCEG and X3J11 mailings have included thirteen drafts of FPCE, reviewed by members of NCEG, the parent committee X3J11, and others. NCEG distributed FPCE to selected professional groups for preliminary review and distributed a later draft more widely for public comment. Now that both developmental and commercial systems have implemented FPCE's specifications, programmers have more than just an abstract specification to evaluate.

Predictability

Even with thoroughly predictable underlying arithmetic—IEEE or not—several factors can thwart the overall predictability of numerical results of compound calculations:

- choice of formats for expression evaluation
- optimizations
- characteristics of the `long double` type
- accuracy of library functions

FPCE primarily addresses the more troublesome first two factors. It recommends documentation for dealing with the last two, where the state of the art is currently evolving.

Expression Evaluation

Knowledge of how expressions are evaluated is essential for predictability. In recognition of the diversity of floating-point architectures, Standard C gives implementations license to evaluate expressions to a format wider than their semantic type. To reconcile predictability with implementation flexibility, FPCE narrows the possibilities to five well-defined expression evaluation methods, of which an implementation is required to implement (at least) one.

To facilitate discussion of expression evaluation, it's handy to have the concept of an implementation's *minimum evaluation format*—the narrowest format that can result from an arithmetic operation. (Technically, the relevant operations are those subject to C's *usual arithmetic conversions* governing the promotion of narrower types; informally, operations such as `+`, `-`, `*`, and `/` supported in hardware are of primary concern.) Early C systems, notably the DEC PDP-11, evaluated all expressions to `double`; CRAY and CYBER systems (on which both `float` and `double` are 64 bits wide) and the more modern PowerPC evaluate to `double` in hardware. They dictate a minimum evaluation format of `double` for their C environments. Some systems supplying a `long double` type in the form of an IEEE *extended* type naturally evaluate all expressions to `long double`; the Intel 8086 and Motorola 68000 families are typical of these systems, which dictate a minimum evaluation format of `long double`. Finally, some architectures treat all formats orthogonally, providing separate `float`, `double`, and possibly `long double` operations; MIPS, SPARC, and HP PA-RISC are typical of these systems, which support a minimum evaluation format of `float`. This macro indicates a system's minimum evaluation format:

```
_MIN_EVAL_FORMAT /* 0-float, 1-double, 2-long double */
```

Most systems evaluate an operation to the widest of the minimum evaluation format and the operands' formats. For systems whose minimum evaluation format is `float` or `double`, FPCE permits an alternative called *widest-need* evaluation, in which the evaluation format of an operation depends on its context. Intuitively, widest-need evaluation specifies that, for a given operation, the evaluation format is the widest of the operand formats in a certain enclosing expression (and is at least as wide as the minimum evaluation format). This approach makes it more convenient for the programmer to adjust tradeoffs between precision and performance, with less need for adding casts and constant suffixes. A macro identifies the use of widest-need evaluation:

```
_WIDEST_NEED_EVAL /* 1 if widest-need, 0 if not */
```

The two macros `_MIN_EVAL_FORMAT` (MEF) and `_WIDEST_NEED_EVAL` (WNE) determine FPCE's five methods of expression evaluation:

	Method	MEF	WNE
1.	evaluate to the wider of the operand formats (often called *evaluate to type*)	0	0
2.	evaluate to the widest of `double` and the operand formats	1	0
3.	evaluate to `long double`	2	0/1
4.	widest-need, with minimum evaluation format of `float`	0	1
5.	widest-need, with minimum evaluation format of `double`	1	1

FPCE specifies that floating constants be no narrower than the minimum evaluation format; in the case of widest-need evaluation, it specifies their width according to the context.

A Common Calculation—Inner Product

Here is an example illustrating possibilities for expression evaluation. The function `sdot` is adapted from the widely used LINPACK [7] and LAPACK [8] BLAS (basic linear algebra subroutines) in FORTRAN.

```
float sdot(int n, const float *sx,
           const float *sy) {
    int i;
    double_t stmp = 0.0;

    if (n > 0) { /* let n = 5*Q + R, Q, R >= 0 */
        for (i = 0; i < n % 5; i++) /* first R cases */
            stmp += *sx++ * *sy++;
        for (i = 0; i < n / 5; i++) { /* next 5*Q */
            stmp += *sx++ * *sy++;
            stmp += *sx++ * *sy++;
            stmp += *sx++ * *sy++;
            stmp += *sx++ * *sy++;
            stmp += *sx++ * *sy++;
        }
    }
    return stmp;
}
```

This function computes the dot product of two `float` vectors, that is, the sum of the products of their corresponding elements. To lessen the relative cost of the loop control when the vectors are long, the authors of the BLAS unrolled the main loop, computing five products per iteration. To improve the accuracy of the result beyond the `float`-only LINPACK form (written to avoid any performance penalty for mixed-format arithmetic), this routine accumulates the sum in a wider variable `stmp`, of type `double_t` for efficiency. The key issue is the translation of the (expanded) statement

```
stmp = stmp + (*sx++ * *sy++);
```

Evaluating the product in `double_t` format generally yields a more accurate result, and it might be guessed by the reader that such is what the author intended by using a `double_t` accumulator. But this is not guaranteed. Here is a breakdown of the three possibilities, depending on the system's expression evaluation method:

- the product is evaluated to `float` and the sum is evaluated to `double` when evaluating to type (method 1)
- all evaluation is to `double` when the minimum evaluation format is `double` or when using widest-need (methods 2, 4, and 5)
- all evaluation is to `long double` when the minimum evaluation format is `long double` (method 3)

Method 1, evaluate to type, requires a cast to force the desired promotion:

```
stmp += (double_t) *sx++ * *sy++;
```

The moral is that to be portable across all valid forms of expression evaluation, such mixed-format code requires casting to promote innermost expressions to a type wider than their semantic type.

Repeatable Results

For reasons of practicality, FPCE does not pursue the tempting goal of repeatable results across all IEEE implementations. The diversity of IEEE architectures assures that some implementations of such a specification would be unacceptably inefficient. However, FPCE's evaluate-to-type method of evaluation could be supported as an option in all implementations, and would be a natural part of a specification for repeatable results. Other aspects on FPCE's specification for predictable results, for example its treatment of optimization, do facilitate repeatable results.

Optimizations

Even with knowledge of the underlying arithmetic and of the implementation's method of expression evaluation, predictability may be elusive because of optimizations. FPCE attempts to achieve predictability, while still encouraging important optimizations.

An FPCE appendix identifies invalid "optimizations" which change result values. Some of these transformations change numerical results on any floating-point architecture; a typical example is rearrangement of parentheses, which Standard C already disallows. Others are invalid solely because of their behavior on infinities, NaNs, and signed zeros. These invalid optimizations are believed not to offer great performance benefits.

Another kind of optimization is the contraction of multiple C operations into one, typically to exploit special hardware instructions. For example, the PowerPC processor [9] offers a *fused* multiply-add, which can evaluate expressions of the form `x + y*z` with just one rounding. Such contractions can be significantly more efficient, and Standard C permits them. However, contractions deliver numerical results other than the source code would indicate, and hence undermine predictability, even as they improve accuracy. For example, use of the fused multiply-add may cause `a*b + c*d` and `c*d + a*b` to have different values, though each is more accurate than the non-fused calculation. FPCE allows contractions but specifies the `fp_contract` pragma as a means to avoid them.

It's worth noting here that FPCE uses pragmas as the mechanism at hand for effecting translation-time options. An alternative—to use corresponding macros recognized by the compiler—was rejected for a variety of minor reasons. Either approach would provide the desired floating-point facility without the need for language extensions providing translation options. All default states for FPCE pragmas conform to Standard C.

Implicit Widening

Implementations on architectures based on efficient wide registers have special problems in both exploiting the architecture's potential and also conforming to Standard C. The cost of meeting the C requirement to narrow function parameter and return values to their declared type has led some implementations not to conform on this point. Consider the following function and its invocation:

```
float harmonic_mean(float x, float y) {...}

float a, b, c;
c = harmonic_mean( (a + b), exp(a - b) );
```

A system evaluating all expressions to `long double`, while adhering strictly to Standard C, would evaluate the expression `(a + b)` to `long double`, (gratuitously) narrow it to `float` for the call to `harmonic_mean`, and then expand it back to `long double` within that function. On the other hand, Standard C makes special allowances for library functions like `exp`; the conforming system could evaluate the expression `(a - b)` to `long double`, pass it unchanged to `exp` and receive a `long double` result in return—which it would then (gratuitously) narrow to `float` for the call to `harmonic_mean`.

Predictability is not really the issue, because passing wide values could be done in a manner consistent with wide expression evaluation, which such an implementation would employ. FPCE aligns with Standard C's requirements, but embraces more than just built-in functions, specifying pragmas that allow optional wide representation of floating-point parameters and return values, and, in the same spirit, local variables.

```
#pragma fp_wide_function_returns    /* on/off */
#pragma fp_wide_function_parameters /* on/off */
#pragma fp_wide_variables           /* on/off */
```

The first and second pragmas permit function arguments and results to be kept in their evaluation format, if that is wider than the declared semantic type. The last pragma allows the widening of local variables to the system's most efficient (wider) type. Note that the pragmas only permit widening, and do not require it. The implementation's function call mechanism determines whether the wide function pragmas can be safely effective, and, if so, whether the user should place the pragmas at the function call site or definition site.

Explicit Widening

Also in the interest of efficiency on wide-register architectures, an FPCE library provides the typedefs `float_t` and `double_t`, which are the implementation's most efficient types at least as wide as `float` and `double`, respectively. They facilitate writing portable code that runs efficiently on the various floating-point architectures. Most algorithms work at least as well if their variables are given a wider format. These types appear throughout the examples in this paper.

Binary-Decimal Conversion

Even perfect binary-decimal conversion can surprise the uninitiated user. Consider the simple example:

```
float x = 1.0/3.0; /* one-third, rounded */
printf("%12.9f", x);
```

This displays the rather dubious decimal value `0.333333343`, looking more like an I/O bug than what it is—the closest 9-significant-digit number to the value of one-third rounded to 24 significant bits. But this is typical of the vagaries of radix conversion, in this case an artifact of the relative densities of 9-digit decimals and `float` values (the former being thirty times as dense as the latter near 1/3).

If accurate binary-decimal conversion is subtle, imperfect conversion is a serious impediment to predictability. For simplicity and efficiency, the IEEE floating-point standards demand slightly less of binary-decimal conversion than the correct rounding they require of basic operations, though the requirements are more than sufficient to eliminate ambiguity when performing round-trip binary-decimal-binary conversions. In response to the development of improved algorithms, FPCE recommends correctly-rounded binary-decimal conversion for all implementations and requires it for IEEE ones.

Hexadecimal Constants

To facilitate exact determination of floating-point values, FPCE specifies hexadecimal floating constants. They provide an expressive notation which, for all binary (or hexadecimal) implementations with sufficiently wide formats, is totally portable and yields exact values. FPCE specifies I/O support for hexadecimal representations, which entails an extension to the syntax accepted by `strtod` and related functions. The `atanh` example earlier used a hexadecimal floating constant for a threshold test. Here is code showing several ways to obtain the next representable `float` value larger than 1 on IEEE systems:

```
float x1, x2, x3, x4, x5;
x1 = 0x0.800001p1;           /* hex floating constant */
x2 = nextafterf(1.0, 2.0);   /* next from 1 to 2 */
x3 = 1.00000012f;            /* 2^-23 approx= 1.2e-7 */
x4 = 1.0 + pow(2.0, -23.0); /* evaluate increment */
x5 = nextrep(1.0f);          /* defined later */
```

The first three options are clearly superior, with `nextafter` being the most portable across non-IEEE formats. Hexadecimal floating constants are powerful for programmers familiar with the underlying representation of floating-point values, and are more useful than `nextafter` for creating arbitrary values such as

```
float zebra = 0x0.AAAAAAp1; /* sum 2^-2k, k=0...11 */
```

Decimal constants, though robust given their rigid specification, can require that the reader have a calculator handy. The latter two options are unnecessarily complex. Here is an implementation of `nextrep` that, given a finite value, returns its nearest representable neighbor toward $+\infty$. It is (arbitrarily) defined to be unexceptional, so it simply overwrites any exception flags that arise. (The algorithm would raise inexact for all but tiny input, would never raise underflow, and would raise overflow only for the input value `FLT_MAX`.) This algorithm depends on its argument being of type `float`. It is typical of the special circumstances in which thoughtless use of the implicit widening allowed by FPCE pragmas foils a carefully tuned algorithm.

```
#pragma fenv_access on
#pragma fp_wide_function_parameters off /* default */
float nextrep(float x) { /* 32-bit IEEE value */
    fenv_t saved_env;

    (void) feholdexcept(&saved_env);
    fesetround(FE_UPWARD); /* + will round up */
    x += 0x1.0p-149; /* add tiniest subnormal */
    fesetenv(&saved_env); /* restore rounding, flags */
    return x;
}
```

Language Bindings

FPCE's specification for IEEE implementations supports the major features of the IEEE arithmetic:

1. standard floating-point data formats
2. correctly rounded results for basic operations
3. consistent, nonstop treatment of exceptional cases

Features 1 and 2 are cornerstones for predictability and portability. Feature 3, which includes flags for detecting exceptions, promotes more robust algorithms that handle special cases without resorting to pervasive and expensive preflighting.

Although motivated by the need for access to features of IEEE arithmetic, parts of the specification discussed here apply more generally. In some cases the specification is broad enough to cover non-IEEE systems with similar features, and in others the specification explicitly supports portability between IEEE and non-IEEE environments.

IEEE Arithmetic Features

The following list of features and associated examples indicates FPCE's direct support for IEEE floating-point arithmetic. The examples are culled from functions and fragments throughout the paper:

- matching of IEEE formats to C types, with IEEE single and double formats specified for C's `float` and `double` types, and IEEE's optional double extended format (if supported) recommended for C's `long double` type

  ```
  float x;
  x += 0x1.0p-149; /* add tiniest subnormal */
  ```

- matching of IEEE operations (+, -, *, /, square root, remainder, conversions) to C in-line operations and library functions

  ```
  stmp += *sx++ * *sy++;
  ```

- representation, I/O support, and inquiry for special values—signed zeros (± 0), signed infinities ($\pm\infty$), NaNs (not-a-number symbols indicating invalid or missing values)—mostly handled through new library macros and functions, but also through extensions to functions like `strtod`, `fprintf`.

```
if (isnan(*sx)) break;
```

- library macros to handle comparisons involving NaNs

```
if (isless(tmp, 0x1.0p-27)) {...}
```

- library functions to control the rounding direction (to-nearest, upward, downward, toward-zero)

```
fesetround(FE_UPWARD);
```

- library functions for accessing IEEE exception flags (invalid, overflow, underflow, divide-by-zero, and inexact)

```
feraiseexcept(FE_OVERFLOW | FE_INEXACT);
```

- library functions recommended in appendixes to the IEEE floating-point standards

```
result = copysign(tmp, x); /* tmp with sign of x */
```

- mechanisms to assure IEEE semantics—in particular that the side effects of arithmetic operations occur as the source code indicates

```
#pragma fenv_access on
```

- clarification of restrictions on optimizations, for predictability and IEEE semantics

```
for (i = 0; i < N; i++) /* example appears here only */
    v[i] = -(v[i] - b) / 5.0;
```

FPCE points out that applying the distributive law to the difference and replacing the multiply with a divide to "optimize" the right hand side to `(b - v[i]) * 0.2` would generally produce a different result and so violates Standard C. In this case, rearranging the difference would affect the sign of a zero result and replacing the exact denominator with a rounded factor `0.2` could alter a nonzero result.

FPCE Design Decisions

The value of having a standard C binding for IEEE features is widely appreciated, and most of FPCE's detailed specification has proved non-controversial. In some cases the FPCE committee explored a variety of approaches with different benefits before settling on the current proposal. And some details appear rather arbitrary—for example, names. Although a vendor already providing IEEE extensions might prefer conformance with its particular implementation, this effort has considered the diverse, incompatible existing implementations as prior art, deciding issues on technical merit.

In order to set practical limits on the scope of the effort, and in recognition of a still evolving state of the art, the IEEE standards do not cover certain areas where IEEE principles might apply and consistency is desirable. Transcendental functions, like `exp`, `sin`, and `log`, and complex arithmetic are two important examples. FPCE's specification for IEEE implementations details how transcendental functions should treat exceptional cases in order to be consistent with IEEE arithmetic, but does not specify accuracy requirements (provably correct rounding being an essentially intractable goal). Complex arithmetic is a sufficiently complicated subject to warrant a proposal of its own [10].

The FPCE's general approach was to accept IEEE arithmetic as is. But in certain cases, FPCE omits specific support for IEEE features without precluding them:

- For exception handling, FPCE limits its scope to non-stop execution with access to exception flags. (At one time another NCEG subgroup addressed the knotty subject of numerical exception handling; more recent efforts have yet to become aligned with the ANSI C Language Committee.)

- FPCE supports quiet NaNs, the kind produced by invalid operations. FPCE abandoned support for signaling NaNs, which must be specially constructed for each implementation and whose utility depends on exception-handling facilities, as too pervasive, too hostile to optimization, and insufficiently portable.

- FPCE omits support for precision control, the mechanism for specifying that results be limited in precision and perhaps range though delivered in a wider format. Because the floating-point standards are vague about whether support should be dynamic or static, there are hardware implementations of both; consequently, any specification would have to cover both, ruining the prospect of a portable construct. (Implementations can adhere to IEEE requirements by providing an option for evaluate-to-type expression evaluation, or by mimicking the FPCE protocol for access to dynamic rounding directions.)

NaNs and Comparisons

Typically, NaNs arise in a computation as the result of missing or invalid data. They pass through arithmetic operations without exception, manifesting themselves when results are displayed. What makes NaNs such effective messengers of arithmetic mishaps past is their tenacious nature, their tendency to propagate through subsequent operations. However, consider the following code fragment from a hypothetical securities management application:

```
if (buy_factor < SURE_THING)
    /* proceed conservatively */
else
    buy_stock(0.9 * current_assets); /* go for broke */
```

What happens if `buy_factor` is a NaN? The IEEE standards specify that a NaN is *unordered* with all other values. That is, a NaN is never less than, equal to, or greater than any other value. So the relation above is `false` and the `else` clause is executed, perhaps to the user's ruin.

The IEEE arithmetic standards attempt to protect programmers and their customers from such pitfalls by specifying that when a NaN participates in a C relational operation (`<`, `<=`, `>=`, `>`) not only should the system deliver the result `false` but it should raise the invalid operation exception as well, as a hedge against the inability to propagate the NaN. The NaNs' deviation from the well-known *trichotomy law* of real number comparisons represents their greatest intrusion into a programming environment originally designed with a strict subset of the real numbers in mind. Programmers writing code robust in the face of all possible input must cope with NaNs in comparisons yet avoid spurious invalid operation exceptions. Clearly, the standard set of C relationals won't suffice.

Perhaps the simplest extension would be a Boolean macro of the form

```
isrelation( x, FP_UNORD | FP_LESS, y ) /* not FPCE */
```

where the logical OR of the predefined constants would indicate which combination of the four relations was being tested. This proposal died of its own inelegance. Two other alternatives for relational operators that would not raise the invalid exception extended the set of relationals using '`?`' and '`!`'. In one proposal, the relational `?<` would mean "unordered with or less than", while in the other the relational `!<` would mean "not less than," that is, "greater than, equal to, or unordered with." Aside from technical problems with the use of '`?`', both proposals would have added several new relational operators to the language. Though more elegant than macros, the added relationals were viewed by some as cluttering things up for the sake of NaNs.

FPCE specifies a set of macros mirroring the existing relational operators, but specified to raise no exception if either operand is a NaN. The test above would take this form to ensure that NaNs are treated suitably conservatively:

```
if (!isgreaterequal(buy_factor, SURE_THING)) {...}
```

These relational macros map to a single test in most underlying hardware. With a modicum of compiler support they can be very efficient.

Filtering NaNs—An Example

Here is another example from the LINPACK [7] BLAS. The function `isamax` returns the index of a vector's first element with largest magnitude. The vector, pointed to by `sx`, has length `n`. This first version, adapted from the original FORTRAN, raises the invalid operation exception if any element of the vector is a NaN. If the vector has no numerical elements, it returns `-1`.

```
int isamax (int n, const double *sx) {
    int i, ixmax = -1; /* if n <= 0, return -1 */
    double_t sxmax = -1.0;

    if (n > 0)
        for (i = 0; i < n; i++, sx++)
            if (fabs(*sx) > sxmax) {
                ixmax = i;
                sxmax = fabs(*sx);
            }
    return ixmax;
}
```

The next variation of the test in the inner loop uses `isgreater` to suppress the invalid exception. This is similar in style to library function `fmax` in that it unexceptionally finds the largest number:

```
if (isgreater(fabs(*sx), sxmax)) {
    ixmax = i;
    sxmax = fabs(*sx);
}
```

The prior versions have interpreted NaNs as missing or otherwise uninteresting data. This rendition of the inner loop returns the index of the first occurrence of the biggest magnitude, or the index of the first NaN:

```
for (i = 0; i < n; i++, sx++)
    if (!islessequal(fabs(*sx), sxmax) {
        ixmax = i;
        if (isnan(*sx)) /* exit if NaN */
            break;
        sxmax = fabs(*sx);
    }
```

This logic is more complicated because of the need to exit prematurely on the first occurrence of a NaN and because of the use of `!islessequal` to effect the test "is greater than or unordered with."

IEEE Semantics

The mechanisms behind certain key features of IEEE floating-point imply special semantics for floating-point operations. The rounding direction controls are dynamic modes that affect the results of operations. And operations have the side effect of raising global flags when given exceptional operands. Honoring these semantics implies certain restrictions on significant optimizations. For example, constants can't be folded if their execution-time evaluation might depend on execution-time changes in the rounding direction. Code can't be eliminated, or perhaps even moved, if it might raise flags that could be tested later. FPCE addresses this dilemma by establishing an implicit assumption that code does not depend on these special semantics (the ordinary case), and by specifying the `fenv_access` pragma as a convenient means to declare otherwise.

FPCE clarifies which operations are to be done at translation time and which at execution time. For IEEE implementations, it requires that IEEE special semantics be treated as part of the program behavior to be protected by the *as-if* rule: that program execution, including side effects, proceed as if ordered by the sequence of statements. This specification is essential for the predictability of numeric results. In order not to constrain the optimizer unduly, the as-if rule pertains to the special IEEE semantics only when the programmer employs the `fenv_access` pragma.

Without these FPCE specifications, authors of portable code must resort to subterfuge, and do so with no guarantee of success. Consider this fragment, taken verbatim from a portable implementation [11] of `nextafterd`, defined in FPCE. This code appears after NaN cases have been filtered out and a numerical result `z` has been computed from source argument `argx` and target argument `argy`. The code raises the underflow exception if `z` is zero or subnormal and raises overflow if `z` is infinite:

```
if ((highpart(z) & ~MDMaxDenorm) == DZero) /* teeny? */
    sideF = z * z + argx * argx; /* underflow */
else if (!finite(z)) /* infinite? */
    sideF = argx * argx; /* overflow */
if ((highpart(sideF) & SignMask) == SignMask)
    z = Zero; /* never executed */
```

The macro function `highpart` gives access to the high 32 bits of a `double` value as an unsigned integer (a technique largely obviated by FPCE). The first test is a bit-wise check for an exponent field of zero—the mark of a subnormal number or zero. The consequent expression is the sum of two products, one of which is guaranteed to underflow; its purpose is purely to stimulate the side effect. The second test uses the finite function provided in the portable library (and in FPCE). Its consequent expression is guaranteed to overflow, once again just for its side effect. The third test—effectively a test of the sign bit of a number known to be the sum of squares—always fails, but (probably) suffices to keep compilers from optimizing the computation of sideF away. This is a very subtle piece of code, based on intimate knowledge of the underlying arithmetic and the behavior of compilers. With FPCE the sequence becomes much more transparent:

```
#pragma fenv_access on
if (fabs(z) < DBL_MIN) /* teeny or 0 */
    feraiseexcept(FE_UNDERFLOW | FE_INEXACT);
else if (fabs(z) == INFINITY)
    feraiseexcept(FE_OVERFLOW | FE_INEXACT);
```

Libraries

FPCE introduces two new headers, <fp.h> and <fenv.h>. The floating-point header <fp.h> includes

- the macro and functions of <math.h>—HUGE_VAL, acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, ceil, fabs, floor, fmod
- facilities required or recommended by the IEEE standards—INFINITY, NAN, nan, rint, rinttol, nearbyint, remainder, copysign, scalb, logb, nextafter, isfinite, isnormal, isnan, fpclassify, signbit, isgreater, isgreaterequal, isless, islessequal, islessgreater, isunordered
- widely useful facilities from other libraries—acosh, asinh, atanh, exp2, expm1, log1p, log2, hypot, erf, erfc, gamma, lgamma, round, roundtol, trunc, remquo, fdim, fmax, fmin

The mathematical header <fp.h> supersedes <math.h>. For IEEE implementations and others with similar features, FPCE specifies how all <fp.h> functions should treat special arguments (singularities, invalid input, infinities, NaNs, signed zeros). Details include values to be returned and exceptions to be raised.

The floating-point environment header <fenv.h> provides access to the execution-time control modes and status flags. For IEEE implementations, these are (at least) the rounding direction mode (to-nearest, upward, downward, toward-zero) and the exception flags (invalid, overflow, divide-by-zero, underflow, inexact). For implementations not supporting access to modes or flags, implementation of <fenv.h> is vacuous.

Overloading

The handiest new feature of <fp.h> as a whole is function overloading—the automatic selection of an implementation of a function to match the expression evaluation formats of its arguments. For example, a function like sin takes arguments of any numerical type. With function overloading, the implementation invokes an internal float, double, or long double version of sin according to its argument.

The specification is consistent with what can be achieved in C++ with a set of overloaded prototypes and enhances compatibility between C and C++. However, FPCE does not dictate the manner of implementation, and, in particular, does not require a general-purpose overloading mechanism outside of `<fp.h>`.

Overloading obsoletes most explicit suffixing. Without it, programmers must choose a suitable version—`sin`, `sinf`, or `sinl` in the case of the sine function. The advantages of overloading are twofold.

First, FPCE's overloading bestows on `<fp.h>` functions the convenience of C's built-in operators. The operands and expression evaluation method determine the type and evaluation format of a '+' operator, which is far simpler (if notationally less perfectly precise) than explicitly identifying the type of each addition. Also, allowing the implementation's evaluation method to determine the evaluation format means that the one natural articulation of the expression can be optimally efficient on the various floating-point architectures. Consider the simple example:

```
double_t z, a, b, t;
z = sin(a*t + b);
```

Although `double_t` gives access to the implementation's most efficient wide type, without overloading the programmer would have no portable way to request the suitable implementation of the sine function. A `double` version of `sin` might force unwanted conversions on a system evaluating to `long double`, while the choice of a `long double` version would cause unwanted promotions on a system with less efficient `long double`.

A second advantage of overloading is the stemming of name proliferation, especially with the proposed extensions for complex arithmetic. For example, the sine functions would require `sin`, `sinf`, `sinl`, `csin`, `csinf`, and `csinl`, whereas with overloading a single `sin` suffices.

It is important to distinguish between the two portable FPCE mechanisms: function overloading and implicit widening of function parameters, variables, and results. Overloading matches an implementation of a library function to its arguments. On the other hand, widening exploits the most efficient types in the underlying implementation; although its primary purpose is to speed up a given calculation, in almost all cases the use of wider types produces results at least as accurate as would have been obtained using the declared types. Recall the use of `harmonic_mean` in an earlier example:

```
#pragma fp_wide_function_returns on
#pragma fp_wide_function_parameters on
#pragma fp_wide_variables on

float harmonic_mean(float x, float y) {...}
float f() {
    float a, b, c;
    ...
    c = harmonic_mean( (a + b), exp(a - b) );
    ...
}
```

Widening allows the implementation to use its most efficient format for the three `float` variables. The sum and difference of `a` and `b` are evaluated in that format.

Overloading ensures that the exponential is computed in that format. Finally, widening allows harmonic_mean to be called with arguments in the implementation's most efficient format, even though its arguments are specified to be of type float.

Note the difference between overloading and widening here. Overloading provides that a suitably accurate approximation to the exponential function e^x is called, corresponding to the format of its argument. This flexibility is already permitted by Standard C, though FPCE overloading goes a step further by ensuring that the one (portable) name exp in the library serves all floating-point types. FPCE widening permits the passing of wider parameters to the sole (non-overloaded) user function harmonic_mean.

C Integration

Considering its scope, FPCE's impact on Standard C is surprisingly small:

- Much of its specification, including the matching of formats to types and the evaluation methods, amounts to limiting the range of implementation options already allowed by Standard C.
- Hexadecimal floating constants require extension of C's syntax for floating constants and (to a slight extent) preprocessor numbers.
- The translator must recognize the pragmas fenv_access, fp_contract, fp_wide_function_returns, fp_wide_function_parameters, and fp_wide_variables, though response will require no change in translation for some implementations.
- <float.h> gets two new macros characterizing the expression evaluation method.
- <stdlib.h> gets new float and long double versions of strtod. The syntax accepted by strtod functions is augmented to handle hexadecimal floating numbers, infinities, and NaNs.
- In <stdio.h>, the input accepted by fscanf is implicitly augmented by the above mentioned changes to strtod. fprintf is required to write infinities and NaNs in a manner compatible with fscanf. The conversion specifier a is adopted for hexadecimal representation, and A and F (which Standard C does not reserve for future extensions) are claimed as specifiers to force capitalization.

Summary

The ANSI C committee's Technical Report on Numerical C Extension will include the specification, "Floating-Point C Extensions" [5], which offers (1) more predictable, yet still efficient, arithmetic, even for legacy implementations, and (2) program-level access to the features already built into IEEE hardware, the predominant industry standard. This paper has argued that the language definition and implementation costs are low relative to the benefits for programmers and users. The ANSI C committee will be considering this specification in its effort to revise the C Standard. Acceptance of FPCE will compound the benefits for those working on multiple platforms.

References

[1] Jim Thomas, *Floating-Point C Extensions,* (WG14/N350, X3J11/94-035). July 7, 1994. Available for anonymous FTP on `ftp.dmk.com` in the directory `DMK/x3j11/floating-point`.

[2] *IEEE Standard for Binary Floating-Point Arithmetic,* (ANSI/IEEE Std 754-1985). Also, *Binary floating-point arithmetic for microprocessor systems,* (IEC 559:1989).

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

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

[5] *American National Standard for Information Systems—Programming Language C,* (X3.159-1989).

[6] Milton Abramowitz and Irene A. Stegun, 1964, *Handbook of Mathematical Functions,* Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York).

[7] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, 1979, *LINPACK Users' Guide,* SIAM, Philadelphia, PA.

[8] E. Anderson, Z. Bai, *et al.*, 1992, *LAPACK Users' Guide,* SIAM, Philadelphia, PA.

[9] *PowerPC 601: RISC Microprocessor User's Manual,* Motorola, Inc. and IBM, Inc., 1993.

[10] Jim Thomas, *Complex C Extensions,* (WG14/N351, X3J11/94-036). July 7, 1994.

[11] W. J. Cody and Jerome T. Coonen, "Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic," *ACM Transactions on Mathematical Software,* Vol. 19, No. 4, pp. 443-451 (December 1993).