WG14/N338
X3J11/94-022

A Rationale for Floating-Point C Extensions

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

This article is scheduled to appear in The C Users Journal.

Abstract

This article describes a specification for C extensions designed to address two long-standing needs of programmers: (1) more predictable floating-point arithmetic for all implementations, and (2) a standard language binding for features of IEEE arithmetic. This proposal is eminently practical on any general purpose hardware, as pioneer implementations attest. Its impact on C is modest. It reflects substantial experience from prior art and has benefited from copious review by both C and floating-point experts.

The specification, already a subgroup-approved technical report, now comes under consideration in the Standard C revision effort.

Introduction

This article presents an overview and rationale for a specification for Floating-Point C Extensions, here referred to as FPCE. The rationale covers issues of specification, implementation, and use, because while FPCE's benefits are mainly to programmers and users, its costs, however modest, are to those defining and implementing the language. First, this introduction describes FPCE's objectives and the effort behind its development. The next two sections, Predictability and IEEE Support, discuss how FPCE's language specification addresses the objectives. The section, Libraries, gives an overview of the FPCE libraries. The section, C Integration, itemizes FPCE's impact on C, and precedes the Summary.

Objectives

The objectives are:

- predictability of floating-point arithmetic
- practicality across the diversity of hardware architectures
- support for IEEE standard arithmetic
- compatibility with Standard C, adaptability for C++

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

327

Background

FPCE's seminal objective was to *specify support for IEEE arithmetic.* When IEEE arithmetic became an official standard in 1985, several IEEE implementations were already available. Now virtually all new floating-point implementations conform to the IEEE standards 754/854—at least in format, if not to the last detail. Although these standards have been enormously successful in influencing hardware implementation, many features remain impractical or unavailable for use by programmers—the IEEE standards do not include language bindings. The ANSI C committee attempted to eliminate conflicts with IEEE arithmetic, but did not specify its support. In the meantime, particular companies have defined their own IEEE language extensions and libraries; not surprisingly, lack of portability has impeded programming for these interfaces. The lack a of standard language binding for IEEE arithmetic is a decade-long problem.

Unpredictability of floating-point arithmetic is an even older problem, plaguing both programmers and users. Horror stories abound. Although the IEEE specifications determine results of basic floating-point operations, the promised level of predictability has been lost because of the myriad of approaches to implementing (and optimizing) the language constructs that combine the basic operations in order to compute useful results. For utilization of its direct support for IEEE arithmetic, FPCE had to address the complementary objective, to *promote predictable floating-point arithmetic.*

Although FPCE was initially chartered for IEEE implementations, from the beginning substantial portions of the specification suited non-IEEE implementations too. The central problem of predicting numerical results of calculations, based on knowledge of the underlying arithmetic and the language, pertains on any implementation. Standard C already makes some progress on this front, for example in requiring that the implementation honor parentheses. Also, non-IEEE arithmetics have features similar to IEEE ones. Today FPCE is for all C implementations intended for general purpose floating-point programming.

Pursuing not only predictability but also *practicality* for the diversity of architectures proved to be FPCE's biggest challenge. The specification effort was pulled precariously toward extremes of impractical idealism and practical anarchy. The intent was for a specification that actually would be implemented and used, not one that was theoretically perfect but unsuited for real implementation on real hardware, or if implemented, undesirable for performance reasons—users should not need to resort routinely to a translation option that forfeits predictability. On the other hand, a specification too loose, though certainly practical to implement, clearly would not address the predictability objective. FPCE's general approach is to allow implementation options where dictated by architectural diversity, but to keep each option well defined and the number of options to a minimum.

Compatibility with the philosophy and style of Standard C has been an objective throughout. FPCE extends the C language only where justified and then mimics similar existing constructs. Detailed below, FPCE's impact on Standard C is rather small.

Although written for C, FPCE was designed with *C++ adaptability* in mind. C++ implementations of FPCE, with very minor changes, exist.

328

Credentials

The Numerical C Extensions Group, NCEG, at its initial meeting in May 1989, identified support for IEEE floating-point arithmetic as one of its focus areas and organized a subgroup to produce a technical report. In 1993, just before being incorporated into the ANSI C Language Committee (X3J11), NCEG (then X3J11.1) approved "Floating-Point C Extensions" for forwarding to X3J11 as this technical report. FPCE (document numbers WG14/N319, X3J11/94-003) arrives just in time for consideration by X3J11 for its revision work on the C standard.

FPCE has benefited from the substantial C and IEEE floating-point expertise of the NCEG 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.

NCEG mailings have included twelve drafts of FPCE, reviewed to varying degrees by NCEG's FP/IEEE subgroup, NCEG as a whole, and others. NCEG distributed FPCE to selected professional groups for preliminary review and distributed it again for public comment. Proprietary extensions for IEEE support have provided prior art for many features. Both developmental and commercial language systems have implemented the specification.

Predictability

Even with thoroughly predictable underlying arithmetic (IEEE or other) several factors can thwart the predictability of numerical results of compound calculations:

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

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

Knowledge of how expressions are evaluated is essential for predictability. In recognition of the diversity of floating-point architectures, the C standard gives implementations license to evaluate expressions to a format wider than their semantic type. To reconcile these important but potentially adverse needs, FPCE narrows the possibilities to a few well-defined expression evaluation methods, of which each implementation is required to implement (at least) one. The options regularize methods in common use:

- evaluate each expression to its semantic type (evaluate-to-type)
- evaluate `float` and `double` expressions to `double`, `long double` expressions to `long double`
- evaluate all expressions to `long double`

Constants of a narrower type are widened to the minimum evaluation format.

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.

An implementation whose minimum evaluation format is `float` or `double` is given the additional option of widest-need expression evaluation, which is a well defined scheme for incorporating certain context into the determination of the evaluation format of an expression. This approach makes it more convenient for the programmer to adjust tradeoffs between precision and performance. Widening the precision of a variable causes calculations with that variable to be widened, without the need for adding or deleting many casts and constant suffixes.

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 the C standard 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 category of optimizations is characterized by the contraction of multiple C operations into one, typically to exploit special hardware instructions, for example the PowerPC's fused multiply-add. Such contractions can be significantly more efficient, and the C standard seems to permit them. However, contractions deliver numerical results other than the source code would indicate, 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, both more accurate than the non-fused calculation. FPCE allows contractions but specifies the `fp_contract` pragma as a means to disallow them.

FPCE uses pragmas as the mechanism at hand for effecting translation-time options. A proposed alternative was to use macros, that would be used in the same places and for the same purposes as the corresponding pragmas. Either approach would provide the desired floating-point facility. C extensions for translation options, without the limitations of pragmas or macros, were not required for floating-point needs per se. All default states for FPCE pragmas conform to the C standard.

Implementations on architectures based on efficient wide registers have special problems in both exploiting the architecture's potential and also conforming to the C standard. 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. 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 the Standard C requirements, but specifies pragmas that allow optional wide representation of floating-point parameters and return values, and in the same spirit, local variables.

Also in the interest of efficiency on wide-register architectures, an FPCE library provides typedefs called `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.

Even perfect binary-decimal conversion surprises the uninitiated user. Imperfect binary-decimal conversion is a serious impediment to predictability. The IEEE floating-point standards demand slightly less of binary-decimal conversion than the correct rounding they require of basic operations. In response to the development of improved algorithms, FPCE recommends correctly-rounded binary-decimal conversion for all implementations and requires it for IEEE ones.

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 a minor extension to the syntax accepted by `strtod` and related functions.

IEEE Support

This section reviews FPCE's support of IEEE arithmetic. Although by and large motivated by the objective of IEEE support, some of the specification discussed here is intended for all implementations, either because non-IEEE systems have features similar enough to the IEEE ones, or for portability between IEEE and non-IEEE implementations.

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

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

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

FPCE's direct support for IEEE floating-point includes:

- 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

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

331

- representation, I/O support, and inquiry for special values—infinities (+∞ and -∞), NaNs (indicating Not-a-Number), and signed zeros (+0 and -0)—mostly handled through new library macros and functions, but also through minor extensions for `strtod`, `fprintf`, etc.

- comparison operators to handle NaN cases

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

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

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

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

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. Of course, in some instances other approaches would have been more beneficial in certain respects though less so in others. And some details appear rather arbitrary—for example names. Although a vendor already providing IEEE extensions might prefer conformance with its particular implementation, the attempt for FPCE was to consider the diverse, incompatible existing implementations as prior art, but to decide 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 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 still does not require correct rounding. The C committee is evaluating a separate proposal for IEEE-compatible complex arithmetic.

The general approach for developing FPCE was to accept IEEE arithmetic as is. In certain cases, FPCE allows but does not specially support IEEE features:

- IEEE's optional traps, and more broadly the whole area of exception handling, over and above access to exception flags (which FPCE covers), was placed outside the scope of FPCE. Specifying exception handling was foreseen as a longer term, separable effort.

- Though providing strong support for quiet NaNs, the kind of NaN produced by invalid operations, FPCE does not specify support for signaling NaNs, which must be specially constructed and whose utility depends on exception handling facilities. Attempted support for signaling NaNs proved tedious and pervasive, inhibited optimization, and offered little portable functionality.

- FPCE does not specify support for precision control per se, though implementations could adhere to IEEE requirements by providing an option for evaluate-to-type expression evaluation, or by mimicking FPCE protocol for rounding directions to allow access to rounding precision modes in underlying arithmetic.

NaNs and Relational Operators

FPCE's new relational operators have been notably controversial. This section contains rationale.

That the traditional relational and equality operators are not sufficient to handle NaNs is non-controversial. First, for any ordered pair of numeric values exactly one of the relationships—*less*, *equal*, and *greater*—is true, but for a NaN and a numeric value, or for two NaNs, just the *unordered* relationship is true. Second, a NaN operand for a traditional relational operator (<, <=, >, >=) raises the invalid exception, as the lone indication that a path taken was probably erroneous unless the code had been written with attention to NaNs.

To neatly handle both complications, FPCE extends the allowable combinations of ordering symbols to include <>= and extends the notion of negated operators, as in the familiar not-equal-to operator `!=`, to the rest of the comparison operators, creating new relational operators such as not-less-than `!<`. With the understanding that `!` indicates an awareness of the unordered case, hence requires no exception, this small extension handles the entirety of floating-point representations, runs unexceptional old code as before, supports exceptional old code by raising exceptions where crashes might be anticipated, and provides exception-free comparisons for new NaN-aware code. On systems without NaNs, the new operators can be viewed as notational alternatives (like `!=`).

Opponents argue that the new relational operators won't be used much and hence library macros would be a more appropriate device than new operator tokens.

Domain errors happen. NaNs offer a way of dealing with them that is more useful than the old alternatives of preflighting or crashing. IEEE arithmetic is designed for NaNs to pass through many calculations without requiring awkward or inefficient code—though useful, NaNs are not usually the most interesting aspect of the code. IEEE's basic arithmetic operators (+, *, etc.) propagate NaNs in a predictable fashion. Although comparisons don't propagate NaNs, they should handle them in a manner consistent with the IEEE approach. The programmer needs to be able to direct NaNs through branches, without having to resort to awkward or inefficient code. Some codes, for example equation solvers, can exploit NaNs directly for robustness and efficiency. More generally, robust functions will be expected to deal with invalid and NaN input in a reasonable way. Thus programmers' attention to NaNs, and comparisons involving NaNs, should be rather common.

A library macro such as a Boolean

isrelation(x, FPUNORD | FPLESS | FPEQUAL, y)

instead of `x !> y` would provide equivalent functionality without change to the language, but would be arguably more awkward, would still require compiler changes if it were to be efficient, and at any rate would undermine programmer expectations of efficiency—perhaps fulfilling the prophecy of little use.

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. This specification is essential for the predictability of both numeric results and also IEEE side effects. The optimizer is constrained by requirements for execution-time semantics only under the effect of an enabling fenv_access pragma.

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`, `remainder`, `remquo`, `rint`, `rinttol`, `nearbyint`, `remainder`, `remquo`, `copysign`, `scalb`, `logb`, `nextafter`, `isfinite`, `isnan`, `fpclassify`, `signbit`)
- 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`)

`<fp.h>` is intended to supersede `<math.h>`. For IEEE implementations, 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 floating-point environment, comprising the execution time control modes and status flags. For IEEE implementations, these are (at least) the rounding direction mode and the exception flags (invalid, overflow, divide by zero, underflow, and inexact). For implementations not supporting access to modes or flags, implementation of `<fenv.h>` is trivial.

334

Overloading

The most salient new feature of <fp.h> as a whole is overloading. FPCE specifies overloaded names for most of the functions in <fp.h>. Depending on the argument type (and the expression evaluation method), the implementation will invoke a float, double, or long double version of the named function. The specification is consistent with what could be achieved in C++ with a set of overloaded prototypes. However, FPCE does not dictate the manner of implementation, and, in particular, does not require a general purpose overloading mechanism.

Overloading obsoletes most explicit suffixing. With overloading, all calls to a sine function would use the name sin, with the type of the function actually invoked being determined by the type of the argument. With explicit suffixing, the name used in the call—sin, sinf, or sinl—would determine the function invoked, regardless of the argument type. The advantages of overloading are twofold.

First, FPCE's overloading bestows certain advantage on the <fp.h> functions previously enjoyed only by C's built-in operators. Note that C already has overloading for its basic operators. The type and evaluation format of a + operation are determined by its operands and the expression evaluation method. This is widely regarded as preferable to the cumbersome alternative of explicitly identifying the type of each addition. And allowing the implementation's evaluation method to determine the evaluation format means the one natural articulation of the operation can be optimally efficient on the various floating-point architectures.

Second, without overloading, the number of public names would proliferate, especially with the advent of extensions for complex arithmetic. For example, the sine functions would require sin, sinf, sinl, csin, csinf, and csinl, whereas with overloading a single sin would suffice.

Also, FPCE's overloading allows more compatibility between C and C++.

The great majority of existing programs should run straight-away if they include <fp.h> instead of <math.h>. An overloaded function differs from what might be a <math.h> counterpart only in the semantic types of its calls.

C Integration

All told, FPCE's impact on Standard C is rather 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 the C standard.
- Hexadecimal floating constants require extension of C's syntax for floating constants and (to a slight extent) preprocessor numbers.
- New relational operators entail new operator tokens and straightforward extension to the grammar for relational expressions.
- The pragmas fenv_access, fp_contract, fp_wide_function_returns, fp_wide_function_parameters, and fp_wide_variables require recognition

by the translator (though response may 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.

The C standardization committee must decide what parts of FPCE to incorporate and what parts to sanction through other means. A reasonable approach might be to incorporate FPCE's basic specification, which is for all general purpose C implementations, into the standard and embody the IEEE specification in an official technical report.

Summary

As part of the Standard C revision effort, the ANSI C committee will be considering an NCEG-approved technical report for Floating-Point C Extensions. Addressing long-standing problems, the specification offers (1) more predictable, yet still efficient, arithmetic, even for legacy implementations, and (2) program-level access to the features already built into IEEE systems, the predominant industry standard. The language definition and implementation costs are low, relative to the benefits for programmers and users. Standardization will compound the benefits for those working on multiple implementations.

References

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

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

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

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

[5] Jim Thomas. *Floating-Point C Extensions* (WG14/N319, X3J11/94-003). January 6, 1994.

336