The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.2.4.2.2 Characteristics of floating types <float.h>

floating types characteristics

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[16)] 330

Commentary

Floating-point types are an alternative to integer representation of numeric values. Some of the floating types occupy the same amount of storage as the larger integer types. Floating-point types trade-off some accuracy representation bits to hold an exponent value. This exponent is used to increase the range of values that can be represented at the cost of a reduced number of significant digits.

The C90 Standard made no mention of ISO/IEC 60599 floating-point arithmetic (although IEEE-754 is mentioned in an example). At that time this standard was only just emerging as the computer industry-wide model of choice. There were still other formats in widespread use. The usage of these formats continues to decline and is now limited to a small number of markets.

In at least one case, it is possible for a program to check which floating-point representation is being used. __STDC_IEC_559__ macro The __STDC_IEC_559__ macro has the value 1 if the implementation conforms to annex F (IEC 60559 floating-point arithmetic).

This model assumes its components have a fixed width during program execution. Models based on a variable number of bits in the exponent and significand (adjusted as the value represented gets larger or smaller) have been proposed.[71] Such variable-width models have yet to be used by commercial processors.

Rationale

The characterization of floating point follows, with minor changes, that of the Fortran standardization Committee. The C89 Committee chose to follow the Fortran model in some part out of a concern for Fortran-to-C translation, and in large part out of deference to the Fortran Committee's greater experience with fine points of floating point usage. Note that the floating point model adopted permits all common representations, including sign-magnitude and two's-complement, but precludes a logarithmic implementation.

The C89 Committee also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which were contrary to that standard. IEEE 754 is now an international standard, IEC 60559; and that is how it is referred to in C99.

C++

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**

Other Languages

Many languages contain floating-point types. Ada gets involved in very low-level details of the representation model. Fortran (90) contains inquiry functions that return values representing properties of the real types used by an implementation. Lisp defines a bignum package, allowing calculations to precisions greater than that supported by most C implementations **long double**. However, the performance is not as great. One of the design aims of the Java language was to ensure that the behavior of a program did not vary across implementations. While the Java language does specify support for a floating-point standard:

The Java types **float** *and* **double** *are IEEE 754 32-bit single-precision and 64-bit double-precision binary floating-point values, respectively.*

it is necessary to remember that this standard permits some degree of implementation leeway. The Java designers attempt to exactly specify how an implementation should follow the floating-point standard (in order to prevent differences in generated results) has met with some criticism.[38]

Common Implementations

Most implementations use the floating-point format supported by the processor on which the program is executed.

Writing a C translator does not require the use of floating-point types. A translator can output floating-point literals in some canonical representation, which is converted to the host representation on program startup (before control is transferred to `main`). However, many translators do make use of knowledge of the representation used at execution time and output, to object files, floating-point literals in this format.

The Unisys e-@ction Application Development Solutions (formerly known as the Universal Compiling System [UCS])[68] (see Table 330.1) has a word size that is not an integer multiple of the types defined by IEC 60559.

Table 330.1: Range of representable floating-point values for the Unisys e-@ction Application Development Solutions Compiling System.

Type	Bits	Decimal Range
`float`	36	1.4693680E-39 . . . 1.7014118E+38
`double`	72	2.7813423E-309 . . . 8.9884657E+307
`long double`	72	2.7813423E-309 . . . 8.9884657E+307

Some implementations emulate the larger representations by joining together several smaller representations. For instance, the Apple numerics implementation on the PowerPC uses two doubles, each using 64 bits, to represent a **`long double`**, occupying 128 bits.[4] The PowerPC processor does not support operations on 128-bit floating-point types; they are implemented with some software support. The exponents are adjusted so that the *least significant* component has an exponent that is at least 54 (representing a value of 2 to 54) less than the *most significant* component. This arrangement increases the precision of the significand (unless it is near the smallest normal value) but does not change the range of possible exponents. The original IBM 390 floating-point format[59] (base 16) also used two doubles to represent a **`long double`**. In some cases the sequence of value bits, in the object representation, may be disjoint. For instance, the Motorola 68000 only uses 80 bits of the possible 96 bits in its double extended object representation.[49]

For those applications requiring even greater precision, use of four doubles (providing 212 bits of significand) has been proposed.[29]

Coding Guidelines

Goldberg[25] is frequently recommended reading for developers working with floating-point types. The following provides a rough-and-ready introduction to error analysis.

Assume alg represents the value returned by the C algorithm for the mathematically exact function f, the error is (for simplicity assume a single argument x):

$$error = alg(x) - f(x) \tag{330.1}$$

If alg is a good approximation to f, another way to look at this is to say the result returned by alg at x corresponds to the exact value at the point $x + e$, where e is a very small displacement from x:

$$alg(x) \approx f(x + e) \tag{330.2}$$

Taking the first two terms from a Taylor series expansion about x (assuming f does not contain any discontinuities):

$$f(x + e) \approx f(x) + e * f'(x) \tag{330.3}$$

Combining the preceding three equations we get:

$$\begin{aligned} error &= alg(x) - f(x) && (330.4)\\ &\approx f(x+e) - f(x) && (330.5)\\ &\approx f(x) + e * f'(x) - f(x) && (330.6)\\ &\approx e * f'(x) && (330.7) \end{aligned}$$

showing that the error is proportional to the derivative of the function. To within the approximations used, the algorithm used is not a factor in the error analysis; the mathematical function used to solve the application-domain problem is the important factor.

The formula given in mathematical textbooks usually assume infinite precision and are not well-behaved when less precision is used.[39] A well-known example is computing the area of a triangle using Heron's formula:

$$Area = sqrt(s(s-x)(s-y)(s-z)) \tag{330.8}$$

where: x, y, z are the length of the sides, and $s = (x+y+z)/2$.

For narrow, pointed, triangles this formula can give incorrect results, even when every floating-point operation is correctly rounded. Table 330.2 gives an example where the values are rounded to five significant digits. The final result can either be 0.0 or 1.5813 (the correct area is 1.000025).

Table 330.2: Area of triangle, using Heron's formula, calculated using different rounding directions.

	Correct	Rounding Down	Rounding Up
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x+(y+z))/2$	100.015	100.01	100.02
Area	1.000025	0.0000	1.5813

By recognizing that intermediate results have a finite accuracy, the preceding formula can be reorganized. First the relative values of the length of the sides is important. They need to be sorted so that $x \leq y \leq z$. The formula:

$$A = sqrt(\frac{(x+(y+z))(z-(x-y))(x+(y-z))}{4}) \tag{330.9}$$

then gives a good, numerically stable approximation to the area. Note that it is only guaranteed to work correctly if the parentheses are not removed (they tell the translator that operands cannot be reordered). On processors that do not support denormal numbers, flushing very small values to zero, this formula fails because a subtraction, $(p-q)$, can underflow. For a more detailed analysis of this problem, see Kahan.[40]

Rev 330.1 Developers who are not familiar with floating-point error analysis shall not be involved in designing or implementing algorithms that use floating-point types.

Usage

Many of the following identifiers were referenced from one program, `enquire.c`, whose job was to deduce the characteristics of a host's floating-point support.

Table 330.3: Number of identifiers defined as macros in **<float.h>** (see Table **??** for information on the number of identifiers appearing in the source) appearing in the visible form of the `.c` and `.h` files.

Name	.c file	.h file	Name	.c file	.h file	Name	.c file	.h file
DBL_MIN	9	21	FLT_MAX	5	15	FLT_ROUNDS	18	14
DBL_MAX	20	19	FLT_DIG	5	15	FLT_RADIX	20	14
DBL_DIG	41	17	LDBL_MIN_EXP	4	14	FLT_MIN_EXP	4	14
FLT_EPSILON	4	16	LDBL_MIN	4	14	FLT_MIN_10_EXP	4	14
DBL_MIN_EXP	4	16	LDBL_MIN_10_EXP	4	14	FLT_MAX_EXP	4	14
DBL_MIN_10_EXP	4	16	LDBL_MAX_EXP	4	14	FLT_MAX_10_EXP	4	14
DBL_MAX_EXP	27	16	LDBL_MAX	4	14	FLT_MANT_DIG	8	14
DBL_MAX_10_EXP	14	16	LDBL_MAX_10_EXP	4	14	FLT_EVAL_METHOD	0	0
DBL_MANT_DIG	14	16	LDBL_MANT_DIG	4	14	DECIMAL_DIG	0	0
DBL_EPSILON	4	16	LDBL_EPSILON	4	14			
FLT_MIN	5	15	LDBL_DIG	4	14			

331 The following parameters are used to define the model for each floating-point type:

Commentary

These parameters are not C-language specific. They are the values needed to fully define, mathematically, a floating-point representation. The values in this model are used, by developers, to:

- ensure that implementations support the range of values likely to be needed by an application,
- perform error analysis, and
- calculate the bit pattern representation of hexadecimal floating constants.

floating constant syntax

Common Implementations

Most implementations divide the representation of a floating-point type into groups of contiguous sequences of bits, each representing a different component of the model presented here.

332 s sign (± 1)

Commentary

One of the few operations that can be performed on a floating-point value, without fear of loss of accuracy, is to change its sign. Information on the sign of the value is held separately from the significand, unlike two's complement notation where it changes the encoding of the value. A consequence of this representation is that it is possible to represent both +0.0 and -0.0 (this possibility also occurs for integers in sign and magnitude representation; and one's complement, but for a different reason). The copysign library function can be used to directly access the sign of a floating-point number.

Common Implementations

Some floating-point formats, e.g., VAX and the HP–was DEC– Alpha in VAX mode, do not support -0.0 (it is considered to be a NaN). While IBM S/360 supports -0.0, any use of it is treated as +0.0. Also -0.0 cannot be generated by normal floating-point operations. The representation of NaNs have a sign bit, but it is not always possible to change this sign bit.

Processors using two's complement floating-point formats have been built in the past. In this representation –MAX is not representable.

b base or radix of exponent representation (an integer > 1) 333

Commentary

FLT_RADIX 366 The FLT_RADIX macro specifies the radix used by an implementation.

exponent

e exponent (an integer between a minimum e_{min} and a maximum e_{max}) 334

Commentary

The range of possible exponent values is governed by the radix and the number of bits used in its representation. Discussions on how many bits of accuracy to trade-off against exponent range have now resolved themselves, or at least the Standard's Committee has published generally agreed-on numbers. The *_MIN_EXP and *_MAX_EXP macros, along with FLT_RADIX define the possible range of values.

Common Implementations

Most implementations use a biased exponent representation (the TMS320C3x[63] is one of the few that uses two's complement). Here a fixed constant is added to the exponent as it appears in its human-readable form, so the value actually held in the bit pattern representation is always positive (hence the term *biased exponent*; this notation is also known as *excess-N*). By using such a notation for the exponent, it is possible to perform relational comparisons on floating-point values by treating their bit pattern as an *integer* type. This considerably simplifies the hardware implementation of floating-point comparison operations.

Some of the bit patterns representing possible exponent values are also given special meaning by IEC 60559 (e.g., all bits 1 is used to indicate one of several possible states). The Unisys A Series[67] represents the value of the exponent using 6 bits to represent the magnitude and 1 bit to represent the sign. The CDC 6600[10] and 7600 used an 11-bit one's complement representation for the exponent.

Usage

The range of exponent values that can occur within programs may depend on the application domain. For instance, astronomy programs may contain ranges of very large values and subatomic particle programs contain ranges of very small values. A study of software for automotive control systems[15] showed (see Table 334.1) a relatively small range of exponents, close to zero.

Table 334.1: Dynamic distribution of decimal exponents, as a percentage, for operands of various floating point operations. Adapted from Connors, Yamada, and Hwu[15] (thanks to Connors for supplying the raw data).

Exponent	Compare	Add	Multiply	Divide	Exponent	Compare	Add	Multiply	Divide
0	15.60	11.4	6.7	3.0					
-1	2.5	2.5	1.9	0.0	1	10.80	9.3	1.6	1.0
-2	0.7	1.2	0.6	1.0	2	5.20	2.6	1.3	3.0
-3	0.1	0.0	0.7	0.0	3	8.50	4.3	0.7	0.0
-4	0.0	0.1	0.2	1.0	4	0.50	0.0	0.5	0.0
-5	0.0	0.0	0.5	0.0					
-6	0.0	0.6	1.4	0.0					

precision floating-point

p precision (the number of base-b digits in the significand) 335

Commentary

The larger the value of p, the greater the precision that can be represented. This value is ultimately responsible for the values of the *_DIG macros. * DIG 369 macros

The points in the real continuum that can be represented by a floating-point value are not as self-evident as they are for the integer values. Because of the presence of an exponent and a normalized significand, the distance between representable values increases as the numbers get bigger. For a radix of 2, for instance, the number of representable values between each power of two is the same. This means there are as many representable floating-point values between 2 and 4 as there are between 1,048,576 and 2,097,152.

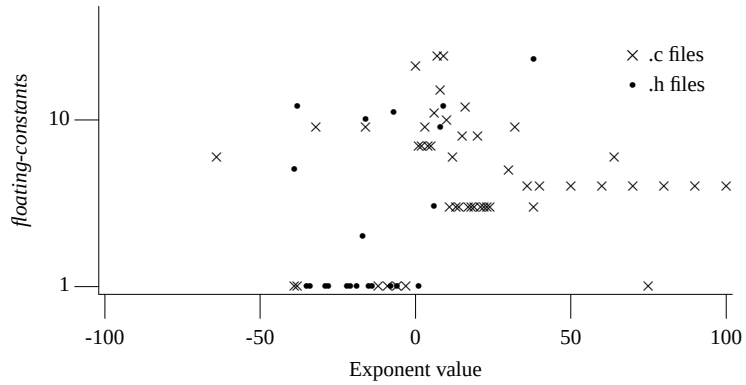

Figure 334.1: Number of *floating-constant*s (that included an *exponent-part*) having a given exponent value. Based on the visible form of the `.c` and `.h` files.

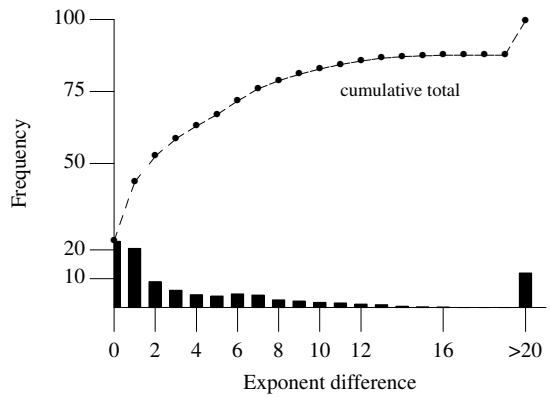

Figure 334.2: Difference in the value of the exponents (in powers of 2) of the two operands of floating-point addition and subtraction operations, obtained by executing the SPECfp92 benchmarks. Adapted from Oberman.[54]

C++

The term *significand* is not used in the C++ Standard.

Common Implementations

The fact that there is a sign bit gives away the representation of the significand in IEC 60559. It is held in sign and magnitude format. This representation was chosen primarily on the grounds of ease of implementation in hardware.

The IEC 60559 representation also uses an additional hidden bit. The bits of the significand are shifted such that the most significant bit that is set, always occurs immediately to the left of the bits actually stored (the value of the exponent is updated to ensure that the actual value being represented remains unchanged), a process called *normalization*. Because this bit is always known to be set, there is no need to store it. Thus there is always one more effective bit of accuracy than is stored in the significand. Implementations that do not use a radix of 2 cannot make use of this hidden-bit technique to obtain more accuracy. For instance, with a radix of 16, bits in the significand need to be shifted by 4 for every incremental change in value of the exponent. The IEC 60559 technique of using a hidden bit can incur a significant performance penalty for software emulations of floating-point operations. Using a radix other than 2 reduces the performance overhead associated with normalizing the significand.[15]

On the WE DSP32[6] the least-significant bit of the significand representation can optionally be used as a parity (odd) bit. Floating-point operations treat this parity bit as part of the value representation. This

WE DSP32

ULP 346 introduces an error or 1 ULP in 50% of cases.

The Motorola DSP563CCC[48] uses a 24-bit two's complement representation for the significand (no hidden bit). The CDC 6600[10] and 7600 used a 49-bit one's complement representation for the significand. The radix point was at the least significant end.

value profiling Research on hardware support for reusing the results of previously executed sequences of instructions usually requires that values input to an instruction sequence match those of previous evaluations, before the previous computed result can be reused. A study by Álvarev, Corbal, Salamí, and Valero,[2] using multimedia applications (e.g., JPEG encoding), investigated what they called *tolerant reuse*. By ignoring some of the least significant bits (up to four) of floating-point values, they were able to significantly increase the number of previously computed results that could be reused (because the input values matched at lower precisions). The number of instructions executed by programs was reduced by between 25% to 40%.

A study by Tong, Rutenbar, and Nagle[64] analyzed the performance of four floating-point intensive applications. They found that it was possible to significantly reduce the number of bits in the significand representation (a speech-recognition program to 5 bits; a fingerprint classification program to 11 bits; an image-processing benchmark to 9 bits; a neural network trainer to 5 bits), without unduly affecting final program accuracy. Customizing the design of floating-point units (i.e., the number of bit in the exponent and significand) to match the accuracy/performance/cost requirements of specific applications has also been proposed.[23]

Coding Guidelines

The term commonly used by developers is *mantissa*, not *significand*. There is probably little to be gained by attempting to change developers' common terminology.

---

f_k nonnegative integers less than b (the significand digits) 336

Commentary

If the radix is 2 (as specified by IEC 60559), possible values for this quantity are 0 and 1.

Common Implementations

Most implementations now follow IEC 60559. The IBM 360 originally used a radix of 16; possible values of f range from 0 to 15 for this implementation. Its successor, the IBM 390, also includes support for the IEC 60559 Standard.[59]

---

floating-point model A *floating-point number* (x) is defined by the following model: 337

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

Commentary

This defines the term *floating-point number*. This model applies to a broad range of floating-point implementations. It differs from the model for signed magnitude integer types in that it contains an exponent part (which enables the decimal point to *float*) and offers support for a radix other than two.

The presence of a decimal point is not unique to floating-point numbers. Both fixed-point and logarithmic representations provide support for a decimal point. In the fixed-point representation the position of the decimal point in the number is fixed (i.e., only two digits after the decimal point are supported). Support for fixed-point types in C is one of the constructs specified in the embedded C TR. In a logarithmic number system a numeric value is represented by its logarithm. This means that operations, such as multiply and divide, are very fast; however, addition and subtraction are much slower. Recent research,[13] using base 2 logarithms, has shown a factor of two speed improvement (over comparable floating-point implementations) when the ratio of floating-point add to multiple operations was 40% to 60%. A processor using this logarithmic number system has also been built.[14]

Embedded C TR

C90

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{min} \leq e \leq e_{max}$$

The C90 Standard did not explicitly deal with subnormal or unnormalized floating-point numbers.

C++

The C++ document does not contain any description of a floating-point model. But, Clause 18.2.2 explicitly refers the reader to ISO C90 subclause 5.2.4.2.2

Coding Guidelines

This model specifies the maximum range of values and precision of a floating-point type. These, and other quantities, are exposed to the developer via macros defined in the header **<float.h>**. The relevant guidelines are covered where these identifiers are discussed.

---

338 In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ($x \neq 0$, $e = e_{min}$, $f_1 = 0$) and unnormalized floating-point numbers ($x \neq 0$, $e > e_{min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs.

floating types can represent

Commentary

This terminology and choice of values of x, f_1, and e originates with IEC 60559 (or to be exact IEEE-754 and IEEE-854 standards that eventually became the ISO/IEC Standard). The ability to support subnormal floating-point numbers, unnormalized floating-point numbers, and NaNs is not unique to ISO 60599, although they may be referred to by other names. A terminological note: The 854 Standard uses the term *subnormal* to refer to entities that the 754 Standard calls *denormals* (the term *denormal* and *denormalized* is not only seen in older books and papers but is still in use).

In a normalized floating-point number the value is always held in a form such that the significand has no leading zeros (i.e., the first digit is always one). The exponent is adjusted appropriately. The minimum values for floating-point quantities given elsewhere are required to be normalized numbers.

330 floating types characteristics

Subnormal numbers provide support for gradual underflow to zero. The difference in value between the smallest representable normalized number closest to zero and zero is much larger than the difference between the last two smallest adjacent representable normalized numbers. Rounding to zero is thus seen as a big jump. Subnormal numbers populate this chasm with values that provide a more gradual transition to zero. Their representation is such that most significant bits of the significand can be zero, hence the name *subnormal*. As the value moves toward zero, more and more of the significant bits of the significand become zero. There is obviously a decrease in precision for subnormal numbers, the exponent already having its minimum value and there being fewer representable bits available in the significand.

subnormal numbers

For implementations that support subnormals, the test `x != y` implies that `(x-y) != 0` (which need not be true if subnormals are not supported and enables code such as `if (x != y) z=1/(x-y)` to be written).

Floating-point infinities are used to handle overflows in arithmetic operations. The presence of a sign bit means that implementations often include representations for positive and negative infinity. The infinity values (positive and negative) are not a finite floating-point value. An infinity returned as the result of a subexpression does not necessarily percolate through the remainder of the evaluation of an expression. For instance, a nonzero value divided by infinity returns a zero value.

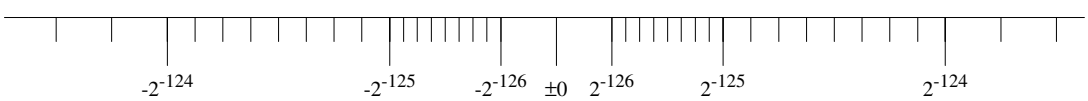

Figure 338.1: Range of normalized numbers about zero, including subnormal numbers.

The `fpclassify` macro returns information on the classification of a floating-point value.

C90

The C90 Standard does not mention these kinds of floating-point numbers. However, the execution environments for C90 programs are likely to be the same as C99 in terms of their support for IEC 60559.

C++

The C++ Standard does not go into this level of detail.

Other Languages

The class `java.lang.Float` contains the members:

```
1  public static final float NEGATIVE_INFINITY = -1.0f/0.0f;
2  public static final float POSITIVE_INFINITY = 1.0f/0.0f;
3  public static final float NaN = 0.0f/0.0f;
```

The class `java.lang.Double` contains the same definitions, but using literals having type **`double`**.

Common Implementations

Handling subnormals in hardware would introduce a lot of complexity and because they are expected to occur very infrequently[37] operations on them are often handled in software (which the hardware traps to when such a value is encountered). In applications where subnormals are quiet common the overhead of software implementation can have a huge impact on performance.[45]

The first digit of a normalized floating-point number is always 1. The IEC 60559 Standard makes use of this property by not holding the 1 in the stored value. An IEC 60559 single-precision, 23-bit significand actually represents 24 bits, because there is an implicit leading digit with value 1. How is zero represented? A significand with all bits zero does not represent a value of zero because of the implicit, leading 1. To represent a value of zero, one of the values of the exponent is required (all bits zero in both the significand and the biased exponent is used).

guard digit

One of the consequences of having an implicit leading digit is that the significand needs to right-shifted by one digit (and the leading 1 used to fill the vacated bit) before any operations can be performed on it. This shift operation would remove one (binary) digit of accuracy unless the least significant digit was saved. The IEC 60559 Standard specifies that implementations use a so-called *guard digit* to hold this shifted value, preventing accuracy from being lost. Before returning the result of an arithmetic operation, the significand is left-shifted. This guard digit is described here because there are some implementations that do not support it (Cray is well-known for having some processors that don't).

Other information that needs to be maintained by a conforming IEC 60559 implementation, during arithmetic operations on floating-point types, include a *rounding digit* and a *sticky bit*. Carter[9] discusses some of the consequences of implementations that support a subset of this information.

Figure 338.2: Single-precision IEC 60559 format.

Table 338.1: Format Parameters of IEC 60559 representation. All widths measured in bits. Intel's *extended-precision* format is a conforming IEC 60559 format derived from that standards *extended double-precision* format.

Parameter	Single	Single Extended	Double	Double Extended	Intel x86 Extended
Precision, p, (apparent mantissa width)	24	32	53	64	64
Actual mantissa width	23	31	52	63	64
Mantissa's MS-Bit	hidden bit	unspecified	hidden bit	unspecified	explicit bit
Decimal digits of precision, $p/\log_2(10)$	7.22	9.63	15.95	19.26	19.26
E_{max}	+127	+1023	+1023	+16383	+16383
E_{min}	-126	-1022	-1022	-16382	-16382
Exponent bias	+127	unspecified	+1023	unspecified	+16383
Exponent width	8	11	11	15	15
Sign width	1	1	1	1	1
Format width (9) + (8) + (4)	32	43	64	79	80
Maximum value, $2^{E_{max}+1}$	3.4028E+38	1.7976E+308	1.7976E+308	1.1897E+4932	1.1897E+4932
Minimum value, $2^{E_{min}}$	1.1754E-38	2.2250E-308	2.2250E-308	3.3621E-4932	3.3621E-4932
Denormalized minimum value, $2^{E_{min}-(4)}$	1.4012E-45	1.0361E-317	4.9406E-324	3.6451E-4951	1.8225E-4951

Unnormalized numbers can only occur in the extended formats. No extended formats, known to your author, uses unnormals. While the Intel 80387 (and later) double-extended format allows for unnormals, they are treated as signaling NaNs. The early Intel 80287 (produced before the IEEE-754 standard was finalized) used unnormals. The IBM S/360 supports them, but does not produce them as the result of any normalized floating-point operation. Use of an unnormalized representation can be used to provide an indication of the degree of precision available in a value[5] and reduce the time taken to perform floating-point operations. Unnormalized numbers are usually created by poor quality implementations.

Table 338.2: List of some results of operations on infinities and NaNs. Also see: "Expression transformations" in annex F.8.2 of the C Standard.

$Operation \Longrightarrow Result$	$Operation \Longrightarrow Result$
$x/(+\infty) \Longrightarrow +0$	$x/(+0) \Longrightarrow +\infty$
$x/(-\infty) \Longrightarrow -0$	$x/(-0) \Longrightarrow -\infty$
$(+\infty) + x \Longrightarrow +\infty$	$x + NaN \Longrightarrow NaN$
$(+\infty) \times x \Longrightarrow +\infty$	$\infty \times 0 \Longrightarrow NaN$
$(+\infty)/x \Longrightarrow +\infty$	$0/0 \Longrightarrow NaN$
$(+\infty) - (+\infty) \Longrightarrow NaN$	$NaN - NaN \Longrightarrow NaN$

Some processors do not support subnormal numbers in hardware. A solution sometimes adopted is for an occurrence of such a value to cause a trap to a software routine that handles it. On such processors operations on subnormals can execute significantly more slowly than on normalized values.[19]

- The Cray T90 and IBM S/360 do not handle subnormal numbers in hardware or software. Such a value input to a floating-point functional unit is forced to zero. Underflow results from arithmetic operations are forced to zero. Such an implementation choice is sometimes made to favor performance over accuracy.
- The AMD 3DNow! extensions[21] to the Intel x86 instruction set include support for an IEC 60559 single-precision, floating-point format that does not include NaNs, infinities, or subnormal numbers.
- The Intel SSE extensions[33] to the Intel x86 instruction set include a status bit that, when set, causes subnormal numbers to be treated as zero. Setting this status bit speeds up operations and can be used by those applications where the loss of accuracy is not significant.

- The Motorola DSP563CCC[48] does not support NaNs or infinities. Floating-point arithmetic operations do not overflow to infinity; they saturate at the maximum representable value.
- The HP Precision Architecture[28] supports a quad format (113-bit precision with a 15-bit exponent).

Example

```
#include <float.h>
#include <stdio.h>

double x, y;

int main(void)
{
double inverse;

if ((DBL_MIN / 2.0) > 0.0)
   printf("This implementation supports extended precision or subnormal numbers\n");
if ((double)(DBL_MIN / 2.0) > 0.0)
   printf("This implementation supports subnormal numbers\n");

if (x != y) /* Only works if subnormals are supported. */
   inverse = 1.0 / (x - y);
}
```

Table 338.3: Example of gradual underflow. * Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Variable or Operation	Value	Biased Exponent	Comment
A0	$1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-125}$	2	
A1 = A0 / 2	$1.100\ 1100\ 1100\ 1100\ 1100\ 1101 \times 2^{-126}$	1	
A2 = A1 / 2	$0.110\ 0110\ 0110\ 0110\ 0110\ 0110 \times 2^{-126}$	0	Inexact*
A3 = A2 / 2	$0.011\ 0011\ 0011\ 0011\ 0011\ 0011 \times 2^{-126}$	0	Exact result
A4 = A3 / 2	$0.001\ 1001\ 1001\ 1001\ 1001\ 1010 \times 2^{-126}$	0	Inexact*
.			
.			
.			
A23 = A22 / 2	$0.000\ 0000\ 0000\ 0000\ 0000\ 0011 \times 2^{-126}$	0	Exact result
A24 = A23 / 2	$0.000\ 0000\ 0000\ 0000\ 0000\ 0010 \times 2^{-126}$	0	Inexact*
A25 = A24 / 2	$0.000\ 0000\ 0000\ 0000\ 0000\ 0001 \times 2^{-126}$	0	Exact result
A26 = A25 / 2	0.0	0	Inexact*

NaN

A *NaN* is an encoding signifying Not-a-Number.

339

Commentary

The encoding for NaNs specified in IEC 60559 requires them to have the maximum exponent value (just like infinity) and a nonzero significand. There is no specification for how different kinds of NaN might be represented. In IEC 60559 the expression `0.0/0.0` returns a NaN value.

A NaN always compares unequal to any other value, even the identical NaN. Such a comparison can also lead to an exception being raised.

Rationale

However, C support for signaling NaNs, or for auxiliary information that could be encoded in NaNs, is problematic. Trap handling varies widely among implementations. Implementation mechanisms may trigger signaling NaNs, or fail to, in mysterious ways. The IEC 60559 floating-point standard recommends that NaNs propagate; but it does not require this and not all implementations do. And the floating-point standard fails to specify the contents of NaNs through format conversion. Making signaling NaNs predictable imposes optimization restrictions that anticipated benefits don't justify. For these reasons this standard does not define the behavior of signaling NaNs nor specify the interpretation of NaN significands.

Rationale

IEC 60559 (International version of IEEE-754) requires two kinds of NaNs: Quiet NaNs and Signaling NaNs. Standard C only adopted Quiet NaNs. It did not adopt Signaling NaNs because it was believed that they are of too limited utility for the amount of work required.

C90

This concept was not described in the C90 Standard.

C++

Although this concept was not described in C90, C++ does include the concept of NaN.

18.2.1.2p34 Template class numeric_limits

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

18.2.2.1p37

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."[194)]

Other Languages

As standards for existing languages are revised, they are usually updated to support the operations and properties described in IEC 60559.

Common Implementations

Support for NaNs has been available on a variety of hardware platforms and therefore C90 implementations since the late 1980s. Some implementations encode additional diagnostic information in the don't care bits of a NaN representation (e.g., the operation on the Apple PowerPC which created the NaN provides the offset of instruction that created it).

Coding Guidelines

Additional information encoded in a NaN value by an implementation may be of use in providing diagnostic information. The availability of such information and its encoding is not specified by the standard and such usage is making use of an extension.

Example

```
1  double d_max(double val1, double val2)
2  {
3  /*
4   * IEC 60559 requires an exception to be raised if
5   * either of the following operands is a NaN.
6   */
7  if (val1 < val2) /* Result is 0 (false) if either operand is a NaN. */
```

```
8       return val2;
9    else
10      return val1;
11   }
```

NaN raising an exception

A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; 340

Commentary

Once a calculation yields a result of quiet NaN all other arithmetic operations having that value as an operand also return a result of quiet NaN. The advantage of the quiet NaN value is that it can be tested for explicitly at known points in a program chosen by the developer. There disadvantage is that the developer has to insert the checks explicitly. A quiet NaN can still cause an exception to be raised. If it is the operand of a relational or equality operator, IEC 60559 requires that an invalid floating-point exception be raised.

C90

The concept of NaN was not discussed in the C90 Standard.

C++

18.2.1.2p34

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

Common Implementations

Most, if not all, implementations represent quiet NaN by setting the most significant bit of a value's significand.

Coding Guidelines

Checking the result of all floating-point operations against NaN enables problems to be caught very quickly, while a lot of context information is available. However, such pervasive checking could complicate the source significantly, may require context information to be passed back through calling functions, and may impact performance.

It is often more practical to view the implementation of some algorithm, involving floating-point values, as an atomic entity from the point of view of NaN handling. In this scenario the implementation of the algorithm does no checking against NaN. However, it is the responsibility of the user of that function, or inlined source code, to check that the input values are valid and to check that the output values are valid. This approach has the advantage of simplicity and efficiency, but the potential disadvantage of loss of context information on exactly where any NaNs were generated.

Designers of third-party libraries need to consider how information is passed back to the caller. A function may be capable of handling the complete range of possible floating-point values, plus the infinities. But NaN is likely to be an invalid input value. Provided that such an input value does not take part in any comparison operations, it may be possible to allow this value to percolate through to the result. If the input value does appear as the operand of a comparison operator, the possibility of a signal being raised means that either the floating-point comparison macros need to be used or an explicit check for NaN needs to be made.

Example

```
1  #include <math.h>
2
3  enum FLT_CMP {cmp_less, cmp_greater, cmp_equal, cmp_nan};
```

```
4
5  enum FLT_CMP cmp_float(float val1, float val2)
6  {
7  if (isnan(val1) || isnan(val2))
8     return cmp_nan;
9
10 if (val1 < val2)
11    return cmp_less;
12 if (val1 > val2)
13    return cmp_greater;
14
15 return cmp_equal;
16 }
```

341 a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.[17)]

Commentary

A signaling NaN raises an exception at the point it is created. A signaling NaN cannot be generated from any arithmetic operation. Signaling NaNs can only be generated by extensions outside those specified in C99, and then only by the `scanf` and `strtod` functions. A signaling NaN cannot be generated from an initialization of an object with static storage duration (see annex F.7.5). Assignment of a signaling NaN may, or may not, raise an exception. It depends on how the implementation copies the value and the checks made by the host processor when objects are moved.

There is no requirement that any of the floating-point exceptions cause a signal to be raised. It is permissible for an implementation's behavior, when a signaling NaN is triggered, to crash the program (with no method being provided to prevent this from occurring). However, if floating-point exceptions are turned into signals, the behavior should be equivalent to `raise(SIGFPE)`.

Signaling NaNs have the advantage of removing the need for the developer to insert explicit checks in the source code. Their disadvantage is that they can be difficult to recover from gracefully. The code that generated the exception may be unknown to the function handling the exception, and returning to continue execution within the previous program flow of control can be almost impossible. The state of the abstract machine is largely undefined after the signal is raised and the standard specifies that the behavior is undefined.

The `fegetexceptflag` function returns information that may enable a program to distinguish between an exception raised by a signaling NaN and an exception raised by other floating-point operations.

C++

18.2.1.2p37

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."[194]

Common Implementations

Most, if not all, implementations represent signaling NaN by a zero in the most significant bit of a value's significand. Some processors (e.g., the Motorola 68000 family[49]) have configuration flags that control whether the signaling NaNs raise an exception.

Coding Guidelines

If a choice is available, is it better to use quiet NaNs or signaling NaNs? They each have their advantages and disadvantages. The exclusive use of signaling NaNs guarantees that, if one is an operand of an operator that treats it as a floating number (except assignment), a signal will be raised. Using the different kinds of NaNs requires use of different control flow structuring of a program. The type of application and how it

handles data will also be important factors in selecting which kind of NaN is best suited. The choice, if one is available, of the type of NaN to use is a design issue that is outside the scope of these coding guidelines. Hauser discusses these issues in some detail.[27]

Example

```
1  #include <setjmp.h>
2  #include <signal.h>
3  #include <stdio.h>
4
5  static jmp_buf start_again;
6
7  void handle_sig_NaN(int sig_info)
8  {
9  longjmp(start_again, 2);
10 }
11
12 void calculate_something(void)
13 { /* ... */ }
14
15 int main(void)
16 {
17 if (setjmp(start_again))
18    printf("Let's start again\n");
19 signal(SIGFPE, handle_sig_NaN);
20
21 calculate_something();
22 }
```

signed of non-numeric values

An implementation may give zero and non-numeric values (such as infinities and NaNs) a sign or may leave them unsigned. 342

Commentary

This sentence was added by the response to DR #218, which also added the following wording to the Rationale.

Rationale

The committee has been made aware of at least one implementation (VAX and Alpha in VAX mode) whose floating-point format does not support signed zeros. The hardware representation that one thinks would represent `-0.0` is in fact treated as a non-numeric value similar to a NaN. Therefore, `copysign(+0.0,-1.0)` returns `+0.0`, not the expected `-0.0`, on this implementation. Some places that mention (or might have) signed zero results and the sign might be different than you expect:

The complex functions, in particular with branch cuts;

```
ceil()
conj()
copysign()
fmod()
modf()
fprintf()
fwprintf()
nearbyint()
nextafter()
nexttoward()
remainder()
```

```
remquo()
rint()
round()
signbit()
strtod()
trunc()
wcstod()
```

Underflow: In particular: `ldexp()`, `scalbn()`, `scalbln()`.

343 Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

Commentary

This sentence was added by the response to DR #218.

344 15) See 6.2.5. footnote 15

Commentary

Clause 6.2.5 deals with types. types

345 16) The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical. footnote 16

Commentary

Most translators take what they are given in terms of processor hardware support, as the starting point for implementing floating-point arithmetic. This wording is intended to justify just such a decision.

C++

The C++ Standard does not explicitly describe a floating-point model. However, it does include the template class `numeric_limits`. This provides a mechanism for accessing the values of many, but not all, of the characteristics used by the C model to describe its floating-point model.

Common Implementations

Most modern floating-point hardware is based on the IEEE-754 (now IEC 60559) standard. Some supercomputer designers made the design decision of improved performance over accuracy (which was degraded) for some arithmetic operations. Users of such machines tend to be supported by knowledgeable developers who take care to ensure that the final, applications answers are within acceptable tolerances. Users in a mass market have no such backup service and the IEEE floating-point standards have therefore aimed at providing reliable accuracy.

For economic reasons many DSPs implement fixed-point arithmetic rather than a floating-point. In some cases, because of the application-specific nature of the problems they are used in, the parameters (such as implied decimal point) of the fixed-point model do not need to be flexible; different processors can use different parameters. The C99 Standard does not support fixed-point data types, although they are described in a Technical Report.[36]

One solution to implementing floating-point types on processors that support fixed-point types is to convert the source containing floating-point data operations to make calls to a fixed-point library. A tool that automatically performs such a conversion is described by Kum, Kang, and Sung.[44] It works by first monitoring the range of values taken on by a floating-point object. This information is then used to perform the appropriate scaling of values in the transformed source. An analysis of the numerical errors introduced by such a conversion is given in.[1]

The Motorola 68000 processor[49] supports a packed decimal real format. All digits are represented in base 10 and are held one per byte. The 24-byte types consists of a 3-digit exponent, a 17-digit significand, two don't care bytes, a byte holding the two separate sign bits (one for the exponent, the other for the mantissa), and an extra exponent (for overflows that can occur when converting from the extended-precision real format to the packed decimal real format). The Motorola 68000 processor extended precision floating-point type contains 16 don't care bits in its object representation.

Coding Guidelines

The availability of IEC 60559 Standard compatible floating-point support is sufficiently widespread that concern for implementations that don't support it is not considered to be sufficient to warrant a guideline recommendation. Of greater concern is the widespread incorrect belief, among developers, that conformance to IEC 60559 guarantees the same results. This issue is discussed elsewhere.

IEC 60559

Testing scripts that operate by differencing the output from a program with its expected output, often show differences between processors when floating-point values are compared. The extent to which these differences are significant can only be decided by developers. A percentage change in a value is often a more important consideration that its absolute change. For small values close to zero it might also be necessary to take into account likely error bounds, which can lead to small values exhibiting a large percentage change that is not significant because the absolute difference is still within the bounds of error.

---

floating-point operations accuracy

The accuracy of the floating-point operations (+, -, *, /) and of the library functions in <math.h> and <complex.h> that return floating-point results is implementation-defined , as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library routine in <stdio.h>, <stdlib.h> and <wchar.h>. 346

Commentary

Accuracy has two edges— poor accuracy and excessive accuracy. The standard permits (but does not require) an implementation to return as little as one digit of accuracy, after the decimal point, for the result of arithmetic operations. In practice poor levels of accuracy are unlikely to be tolerated by developers. From the practical point of view, it is possible to achieve accuracies of 1 ULP. A more insidious problem, in some cases, can be caused by additional accuracy; for instance, on a host that performs floating-point operations in some extended precision:

```
1  #include <stdio.h>
2
3  extern double a, b;
4
5  void f(void)
6  {
7  double x;
8
9  x = a + b;
10 if (x != a + b)
11    printf("x != a + b\n");
12 }
```

any extended precision bits will be lost in the first calculation of a+b when it is assigned to x. The result of the second calculation of a+b may be held in a working register at the extended precision and potentially contain additional value bits not held in x, the result of the equality test then being false.

Accuracy does not just relate to individual operations. The operator sequence of multiply followed by addition is sufficiently common that some vendors have created a fused multiply/add instruction. Such fused instructions not only execute more quickly than the sum of two instruction timings, but often deliver results containing greater accuracy (than would have been obtained by executing separate instructions performing the same mathematical operations).

How is the accuracy of a floating-point operation measured? There are two definitions in common use— relative error and units in the last place (ULP).

1. Relative error is the difference between the two values divided by the actual value; for instance, if the actual result of a calculation should be 3.14159, but the result obtained was 3.14×10^0, the relative error is $0.00159/3.14159 \Rightarrow 0.0005$.
2. If the floating-point number z is approximated by $d.d.d\ldots d\times b^e$ (where b is the base and p the number of digits in the significand), the ULP error is $\mid d.d.d\ldots d-(z/b^e)\mid b^{p-1}$. When a floating-point value is in error by n ULP the number of contaminated (possibly incorrect) digits in that value is $\log_b n$. For IEC arithmetic b is 2 and each contaminated digit corresponds to a bit in the representation of the value. See Muller[51] for a comprehensive discussion. The error in ULPs depends on the radix and the precision used in the representation of a floating-point number, but not the exponent. For instance, if the radix is 10 and there are three digits of precision, the difference between 0.314e+1 and 0.31416e+1 is 0.16 ULPs, the same as the difference between 0.314e+10 and 0.31416e+10. When the two numbers being compared span a power of the radix, the two possible error calculations differ by a factor of the radix. For instance, consider the two values 9.99e2 and 1.01e3, with a radix of 10 and three digits of precision. These two values are adjacent to the value 1.00e3, a power of the radix. If 9.99e2 is the correct value and 1.01e3 is the computed value, the error is 11 ULPs. But, if 1.01e3 is the correct value and 0.999e3 is the computed value, then the error is 1.1 ULPs.

ULP

A floating-point operation usually delivers more bits of accuracy than are held in the representation. The least significant digit is the result of rounding those other digits. A possible error of 0.5 ULP is inherent in any floating-point operation. If a variety of calculations all have the same relative error, their error expressed in ULP can vary (*wobble* is the proper technical term) by a factor of b. Similarly, if a set of calculations all have the same ULP, their relative error can vary by a factor of b.

Several programs have been written to test the accuracy of a processor's floating-point operations.[58] Various mathematical identities can be used, making it possible for these programs to be written in a high-level language (many are written in C) and be processor-independent. For a description of the latest and most thorough testing tool, as of 2001, see Verdonk, Cuyt, and Verschaeren.[69] For a detailed, mathematical discussion of floating-point accuracy, see Priest.[56]

The basic formula for error analysis of an operation, assuming $fl(a\ op\ b)$ does not overflow or underflow, is:

error analysis

$$fl(a\ op\ b) = (1+\epsilon)\times(a\ op\ b) \tag{346.1}$$

where: $(a\ op\ b)$ is the exact result of the operation, where op is one of $+$, $-$, $*$ and $/$; $fl(a\ op\ b)$ is the floating-point result; $\mid \epsilon \mid \leq macheps$.

To incorporate underflow (but not overflow), we can write:

$$fl(a\ op\ b) = (1+\epsilon) * (a\ op\ b) + \eta \tag{346.2}$$

where:

$$\mid \eta \mid \leq macheps * underflow\ threshold \tag{346.3}$$

On machines that do not implement gradual underflow (including some IEEE machines, which have an option to perform flush-to-zero arithmetic instead), the corresponding error formula is:

$$\mid \eta \mid \leq underflow\ threshold \tag{346.4}$$

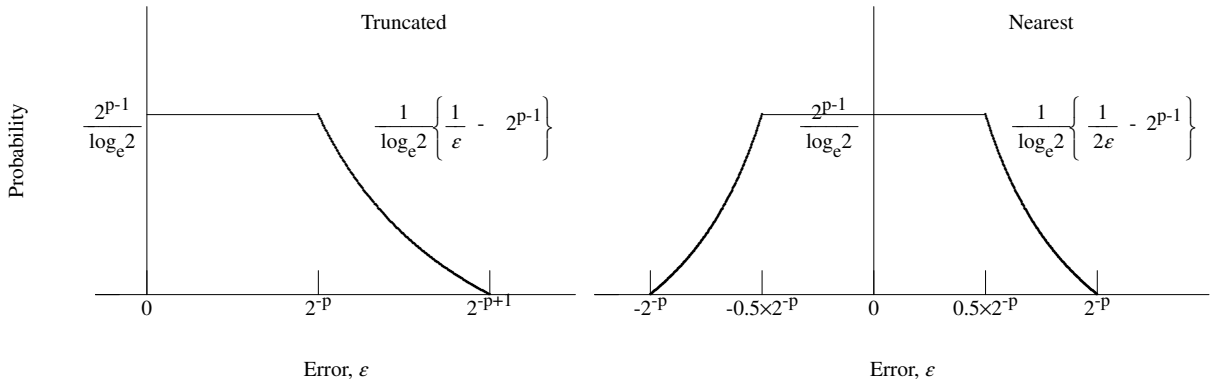

Figure 346.1: Probability of a floating-point operation having a given error (ϵ) for two kinds of rounding modes (truncated and to-nearest); p is the number of digits in the significand. Adapted from Tsao.[42]

(i.e., the error is $1/macheps$ times larger).

Tsao[42] performed an analysis of the distribution of round-off errors, based on Benford's law,[30] to estimate the probability of ϵ having particular values.

Rationale

Because of the practical difficulty involved in defining a uniform metric that all vendors would be willing to follow (just computing the accuracy reliably could be a significant), and because the importance of floating point accuracy differs greatly among users, the standard allows a great deal of latitude in how an implementation documents the accuracy of the real and complex floating point operations and functions.

. . .

If an implementation documents worst-case error, there is no requirement that it be the minimum worst-case error. That is, if a vendor believes that the worst-case error for a function is around 5 ULPs, they could document it as 7 ULPs to be safe.

The Committee could not agree on upper limits on accuracy that all conforming implementations must meet, for example, "addition is no worse than 2 ULPs for all implementations.". This is a quality-of-implementation issue.

Implementations that conform to IEC 60559 have one half ULP accuracy in round-to-nearest mode, and one ULP accuracy in the other three rounding modes, for the basic arithmetic operations and square root. For other floating point arithmetics, it is a rare implementation that has worse than one ULP accuracy for the basic arithmetic operations.

. . .

The C99 Committee discussed the idea of allowing the programmer to find out the accuracy of floating point operations and math functions during compilation (say, via macros) or during execution (with a function call), but neither got enough support to warrant the change to the Standard. The use of macros would require over one hundred symbols to name every math function, for example, `ULP_SINF`, `ULP_SIN`, and `ULP_SINL` just for the real-valued sin function. One possible function implementation might be a function that takes the name of the operation or math function as a string, `ulp_err("sin")` for example, that would return a double such as 3.5 to indicate the worst case error, with `-1.0` indicating unknown error. But such a simple scheme would likely be of very limited use given that so many functions have accuracies that differ significantly across their domains. Constrained to worst-case error across the entire domain, most implementations would wind up reporting either unknown error or else a uselessly large error for a very large percentage of functions. This would be useless because most programs that care about accuracy are written in the first place to try to compensate for

accuracy problems that typically arise when pushing domain boundaries; and implementing something more useful like the worst case error for a user-specified partition of the domain would be excessively difficult.

Some of the issues of how representation used can impact accuracy is discussed elsewhere. The issue of the accuracy of decimal string to/from *binary* (non-decimal) floating-point conversions was raised by DR #211. The resolution of this DR resulted in the change of wording highlighted in the preceding C99 sentence. 368 DECIMAL_DIG macro

C90

In response to DR #063 the Committee stated (while the Committee did revisit this issue during the C99 revision of the C Standard, there was no change of requirements):

DR #063

Probably the most useful response would be to amend the C Standard by adding two requirements on implementations:

Require that an implementation document the maximum errors it permits in arithmetic operations and in evaluating math functions. These should be expressed in terms of "units in the least-significant position" (ULP) or "lost bits of precision."

Establish an upper bound for these errors that all implementations must adhere to. The state of the art, as the Committee understands it, is:

correctly rounded results for arithmetic operations (no loss of precision)

1 ULP for functions such as sqrt, sin, and cos (loss of 1 bit of precision)

4–6 ULP (loss of 2–3 bits of precision) for other math functions.

Since not all commercially viable machines and implementations meet these exacting requirements, the C Standard should be somewhat more liberal.

The Committee would, however, suggest a requirement no more liberal than a loss of 3 bits of precision, out of kindness to users. An implementation with worse performance can always conform by providing a more conservative version of **<float.h>***, even if that is not a desirable approach in the general case. The Committee should revisit this issue during the revision of the C Standard.*

C++

The C++ Standard says nothing on this issue.

Common Implementations

The accuracy of floating-point operations is usually at the mercy of the host floating-point processor hardware. Many implementations try to achieve an accuracy of 1 ULP in their floating-point operations and **<math.h>** library functions. 346 ULP

Use of the logarithmic number system, to represent real numbers, enables multiplication and division to be performed with no rounding error.[13]

There is an Open Source implementation of the IEEE-754 standard available in software.[26] This provides floating-point operations to a known, high level of accuracy, but somewhat slowly.

In IEC 60559 arithmetic the floating-point result $fl(a\ op\ b)$ of the exact operation $(a\ op\ b)$ is the nearest floating-point number to $(a\ op\ b)$, breaking ties by rounding to the floating-point number whose least significant bit is zero (an *even* number). Taking underflow into account requires an additional term to be added to the right side of the basic equation given earlier. For processors that make use of gradual underflow the value is bounded: $|\ \epsilon\ | \leq macheps \times underflow\_threshold$. On machines that do not implement gradual underflow (including some IEC 60559 base processors, which have an option to perform flush-to-zero arithmetic instead), the corresponding error formula is:

$$|\ \epsilon\ | \leq underflow\ threshold \tag{346.5}$$

Cray had a line of processors (the C90 and Cray 2) on which speed of execution was given priority over accuracy. For this processor the error analysis formula for addition and subtraction was:

$$fl(a \pm b) = ((1 + \epsilon_1) * a) \pm ((1 + \epsilon_2) * b) \tag{346.6}$$

Subtracting two values that are very close to each other can lead to a much larger than normal, in IEC 60559, error. On some Cray processors divide was emulated by multiplying by a reciprocal, leading to $x/x \neq 1.0$ in some cases.

The AMD 3DNow! extensions[21] to the Intel x86 processor include a reciprocal instruction that delivers a result accurate to 14 bits (instead of the normal 23 bits). Developers can choose, with suitable translator support, to accept this level of accuracy or to have two additional instructions generated which return a result more slowly, but is accurate to 1 ULP.

The accuracy of the trigonometric functions $\sin$, $\cos$, and $\tan$ depends on how well their argument is reduced, modulo $\pi/2$, and by the approximations used on the small domain near zero. Some implementations do a very poor job of argument reduction, so values near a multiple of $\pi/2$ often have few, if any, bits correct.[52]

The Intel x86 processor family contains an instruction for computing the sine (and cosine and tangent) of its operand. This instruction requires the absolute value of its operand, in radians, to be less than 2^{63}. Values outside this range are returned as the result of the instruction and a processor flag, C2, is set. Such a result is a surprise in the sense that `sin` (and sine and tangent) is expected to return a value between -1 and 1.

Another accuracy problem with the Intel x86 instruction for computing the sine (and cosine) is accuracy near π ($\pi/2$ for cosine). When using 64-bit mode the 53-bit significand result may only be accurate in the first 15 bits and the 80-bit mode the 64-bit significand result may only be accurate in the first 5 bits (the problem is caused by insufficient accuracy, 68 bits, in the approximation of π used; 126 bits are actually needed, and used by AMD in their x86 processors).

Given the difficulty in performing mathematical error analysis on algorithms, let alone coded implementations running on imperfect hardware, one solution is to have the translator generate machine code to estimate round-off errors during program execution. One such translator, Fortran-based, has been developed.[7]

double rounding

Processors that perform extended-based arithmetic operations do not always produce more accurate results. There is a source of error known as *double rounding* that occurs when the rounding mode is round-to-nearest. In the default precision mode, an extended-based system will initially round the result of an arithmetic operation to extended double-precision. If that result needs to be stored in double-precision, it has to be rounded again. The combination of these two rounding operations can yield a value that is different from what would have been obtained by rounding the result of the arithmetic operation directly to double-precision. This can happen when the result as rounded, to extended double-precision, is a *halfway case* (i.e., it lies exactly halfway between two double-precision numbers) so the second rounding is determined by the round-ties-to-even rule. If this second rounding rounds in the same direction as the first, the net rounding error will exceed half a unit in the last place. It can be shown[22] that the sum, difference, product, or quotient of two p-bit numbers, or the square root of a p-bit number, rounded first to q bits and then to p bits gives the same value as rounding the result of the operation just once to p bits provided $q \geq 2p + 2$.

Double-rounding can also prevent execution-time measurement of rounding errors in calculations. The formula:

```
1  t = s + y;
2  e = (s - t) + y;
```

recovers the roundoff error in computing `t`, provided double-rounding does not occur.

Coding Guidelines

There are three sources of inaccuracies in the output from a program that uses floating-point data types. Those caused by the implementation, those intrinsic to the algorithm used, and those caused by how the

developer coded the algorithm. For instance, assume that `raise_to_power(valu, power)` is a function that returns `valu` raised to `power` (e.g., `raise_to_power(3.0, 2)` returns 9.0). If we want to raise a value to a negative power, the obvious solution is to use `raise_to_power(1.0/valu, power)`. However, there in a rounding error introduced by the calculation `1.0/valu` that will be compounded each time it is used inside the library function. The expression `1.0/raise_to_power(valu, power)` avoids this compounding of the rounding error; it has a single rounding error introduced by the final divide.

Minimizing the rounding error in floating-point calculations can require some theoretical knowledge. For instance, the simple loop for summing the elements of an array:

```
1  double sum_array(int num_elems, double a[num_elems])
2  {
3  double result = 0.0;
4
5  for (int a_index=0; a_index < num_elems; a_index++)
6     result += a[a_index];
7
8  return result;
9  }
```

can have the rounding error in the final result significantly reduced by replacing the body of the function by the Kahan summation formula:[25]

```
1  double sum_array(int num_elems, double a[num_elems])
2  {
3  double result = a[0],
4         C = 0.0,
5         T, Y;
6
7  for (int a_index=1; a_index < num_elems; a_index++)
8     {
9     Y = a[a_index] - C;
10    T = result + Y;
11    C = (T - result) - Y;
12    result = T;
13    }
14
15 return result;
16 }
```

which takes account of the rounding properties of floating-point operations to deliver a final result that contains significantly less rounding error than the simple iterative addition. The correct operation of this summation formula also requires the translator to be aware that algebraic identities that might apply to integer objects, allowing most of the body of the loop to be optimized away, do not apply to floating-point objects.

Usage

In theory it is possible to measure the accuracy required/expected by an application. However, it is not possible to do this automatically — it requires detailed manual analysis. Consequently, there are no usage figures for this sentence (because no such analyses have been carried out by your author for any of the programs in the measurement set).

---

347 The implementation may state that the accuracy is unknown.

Commentary

It is not always possible for an implementation to give a simple formula for the accuracy of its floating-point arithmetic operations, or the accuracy of the **<math.h>** functions. A variety of different implementation techniques may be applied to each floating-point operator[54, 55] and to the **<math.h>** functions. The accuracy in individual cases may be known. But a complete specification for all cases may be sufficiently complex that a vendor chooses not to specify any details.

C++

The C++ Standard does not explicitly give this permission.

Other Languages

Most languages don't get involved in this level of detail. However, Ada specifies a sophisticated model of accuracy requirements on floating point operations.

Common Implementations

No vendors known to your author state that the accuracy is unknown.

Coding Guidelines

How such a statement in an implementation's documentation should be addressed is a management issue and is outside the scope of these guidelines.

---

float.h suitable for #if

All integer values in the **<float.h>** header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; 348

Commentary

This is a requirement on the implementation. It is more restrictive than simply requiring an integer constant expression. The cast and **sizeof** operators cannot appear in **#if** preprocessing directives, although they can appear within a constant expression.

FLT_ROUNDS may be an expression whose value varies at execution time, through use of the `fesetround` library function.

C90

*Of the values in the **<float.h>** header, FLT_RADIX shall be a constant expression suitable for use in **#if** preprocessing directives;*

C99 requires a larger number of values to be constant expressions suitable for use in a **#if** preprocessing directive and in static and aggregate initializers.

C++

The requirement in C++ only applies if the header **<cfloat>** is used (17.4.1.2p3). While this requirement does not apply to the contents of the header **<float.h>**, it is very likely that implementations will meet it and no difference is flagged here. The namespace issues associated with using **<cfloat>** do not apply to names defined as macros in C (17.4.1.2p4)

17.4.4.2p2 *All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in **#if** preprocessing directives, unless explicitly stated otherwise.*

The C++ wording does not specify the C99 Standard and some implementations may only support the requirements specified in C90.

Common Implementations

This statement was true of some C90 implementations. But, there were many implementations that defined the *_MAX macros in terms of external variables, so the *_MAX macros were not usable in static or aggregate initializers. Exact figures are hard to come by, but perhaps 50% of C90 implementations did not use constant expressions.[65]

Coding Guidelines

It will take time for implementations to migrate to C99. The extent to which a program relies on a C99 translator being available is outside the scope of these coding guidelines.

Example

```
#include <float.h>

#if FLT_RADIX == 10 /* A suitable constant in C90 and C99. */
#endif

#if FLT_DIG == 5 /* Not required to be conforming in C90. */
#error This is not a C99 conforming implementation
#endif
```

---

349 all floating values shall be constant expressions.

Commentary

This is a requirement on the implementation. It ensures that floating values can be used as initializers for objects defined with static storage duration.

address constant

C90

all other values need not be constant expressions.

This specification has become more restrictive, from the implementations point of view, in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Common Implementations

This C99 requirement was met by many C90 implementations.

Coding Guidelines

It will take time for implementations to migrate to C99. The extent to which a program relies on a C99 translator being available is outside the scope of these coding guidelines.

---

350 All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types.

Commentary

DECIMAL_DIG applies to " . . . the widest supported floating type . . . ". One of the floating types has to be selected; there cannot be three widest types. It is the type **long double**, even if an implementation supports a floating type with more precision as an extension (the standard requires it to be one of the types it specifies, and is silent on the issue of additional floating-point types provided by the implementation, as an extension).

The fact that the three FLT_* macros have a single value for all three floating-point types implies a requirement that the floating-point evaluation method, radix, and rounding behavior be the same for these floating-point types, which is also required in IEC 60559.

C90

Support for DECIMAL_DIG and FLT_EVAL_METHOD is new in C99. The FLT_EVAL_METHOD macro appears to add functionality that could cause a change of behavior in existing programs. However, in practice it provides access to information on an implementation's behavior that was not previously available at the source code level. Implementations are not likely to change their behavior because of this macro, other than to support it.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Common Implementations

For those processors containing more than one floating-point unit (e.g., Intel Pentium with SSE extensions,[32] AMD Athlon with 3Dnow!,[21] and PowerPC with AltiVec support[50]) implementations may chose to evaluate some expressions in different units. Such implementations may then support more than one name/value for some of these macros; for instance, the evaluation method each representing the characteristics of a different floating-point unit.

multiply always truncate

The floating-point model representation is provided for all values except **FLT_EVAL_METHOD** and **FLT_ROUNDS**. 351

Commentary

floating types 330 characteristics 354 FLT_EVAL_METHOD 352 FLT_ROUNDS

That is, the characteristics of the model used to define floating types does not include FLT_EVAL_METHOD or FLT_ROUNDS as one of its parameters. The standard lists the possible values for these macros and their meaning separately. The actual values used are implementation-defined. These two macros correspond to two classes of implementation-defined behavior described by the IEC 60559 Standard.

The IEC 60559 Standard also allows implementations latitude in how some constructs are performed. These macros give software developers explicit control over some aspects of the evaluation of arithmetic operations.

C90

Support for FLT_EVAL_METHOD is new in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

Coding Guidelines

Educating developers about the availability of these options and there implications is outside the scope of these coding guidelines. A statement of the form "It is possible to obtain different results, using the same expression, operand values and compiler, between different IEC 60559 implementations or even the same implementation operating in a different mode." might be used as an introductory overview of floating point.

FLT_ROUNDS

The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:[18)] 352

-1 indeterminable
0 toward zero
1 to nearest
2 toward positive infinity
3 toward negative infinity

All other values for **FLT_ROUNDS** characterize implementation-defined rounding behavior.

Commentary

One form of rounding that would be classified as indeterminable is known as ROM-rounding.[43] Here a small number of the least significant bits of the value are used as an index into a table (usually held in ROM on the hardware floating-point unit). The output from this table lookup is appended to the other digits to give the rounded result. In the boundary case where there would normally be a ripple add up through to the next significant bit the result is the same as a round toward zero (thus simplifying the hardware and possible improving performance at the cost of an inconsistent rounding operation for a small percentage of values).

Rounding to zero, also known as *chopping*, occurs when floating-point values are converted to integers. Of the methods listed, it generates the largest rounding error in a series of calculations.

Rounding to nearest has minimum rounding error, of the known methods, in a series of calculations. The C Standard does not specify the result in the case where there are two nearest representable values. The IEC 60559 behavior is to round to the even value (i.e., the value with the least significant bit set to zero).

Rounding to positive and negative infinity is used to implement interval arithmetic. In this arithmetic, values are not discrete points; they are represented as a range, with a minimum and maximum limit. Adding

two such intervals, for instance, generates another interval whose lower and upper bounds delimit the possible range of values represented by the addition. Calculating the lower bound requires rounding to negative infinity, the upper bound requires rounding to positive infinity.

FLT_ROUNDS is not required to be an lvalue. It could be an internal function that is called. It is generally assumed, but not required, that the rounding mode used for other floating-point operators is the same as that for addition.

The FENV_ACCESS pragma may need to be used in source that uses the FLT_ROUNDS macro. The value of a particular occurrence of the FLT_ROUNDS macro is only valid at the point at which it occurs. The value of this macro could be different on the next statement (or even within the same expression if it contains multiple calls to the fesetround library function).

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

The C++ header **<limits>** also contains the enumerated type:

18.2.1.3

```
namespace std {
   enum float_round_style {
     round_indeterminable       = -1,
     round_toward_zero          = 0,
     round_to_nearest           = 1,
     round_toward_infinity      = 2,
     round_toward_neg_infinity  = 3
   };
}
```

which is referenced by the following member, which exists for every specialization of an arithmetic type (in theory this allows every floating-point type to support a different rounding mode):

18.2.1.2p62

Meaningful for all floating point types.

```
static const float_round_style round_style;
```

The rounding style for the type.[206)]

Other Languages

Most other language specifications say nothing on this subject. Java specifies round-to-nearest for floating-point arithmetic.

Common Implementations

floating value converted not representable

On most implementations the rounding mode does not just affect addition. It is applied equally to all operations. It can also control how overflow is handled. Some applications may choose a round-to-zero rounding behavior because overflow then returns a saturated result, not infinity. Most implementations follow the IEC 60559 recommendation and use *to nearest* as the default rounding mode.

Almost all processors (the INMOS T800[31] does not) allow the rounding mode of floating-point operations to be changed during program execution (often using two bits in one of the processor control registers). The HP–was DEC– Alpha processor[60] includes both instructions that take the rounding mode from a setable control register and instructions that encode the floating-point rounding mode within their bit-pattern (a translator can generate either form of instruction). The translator for the Motorola DSP563CCC[48] only supports *round-to-nearest*.

Some of the older Cray processors make use of implementation-defined rounding modes for some operators, resulting in greater than 1 ULP errors.

The HP–was DEC– VAX processor[16] has two rounding modes. Round-to-nearest, with the tie break being to round to the larger of the absolute values (i.e., away from zero, as does the HP 2000 series). The other is

called *chopped* (i.e., round toward zero). The HP–was DEC– Alpha processor[60] contains instructions that emulate this behavior for compatibility with existing code.

Some processors have two sets of floating-point instructions. For instance AMD Athlon processors containing the 3DNow! extensions[21] have instructions that perform operations on IEC 60559 compatible single and double floating-point values. The extensions operate on a single-precision type only and do not support full IEC 60559 functionality (only one rounding mode— underflow saturates to the minimum normalized value or negative infinity; overflow saturates to the maximum normalized value or positive infinity; no support for infinities or NaN as operands). The advantage of using these floating-point instructions is that they operate on two sets of operands at the same time (each operand is two 32-bit floating-point values packed into 64 bits), offering twice the number-crunching rate for the right kind of algorithms.

multiply
always truncate

Coding Guidelines

Applications that need to change the rounding mode are rare. For instance, a set of library functions implementing interval arithmetic[11,41] needs to switch between rounding to positive and negative infinity. The C Standard lists the programming conventions that its model of the floating-point environment is based on (in the library section). These conventions can be used by implementation's to limit the number of possible different permutations of floating-point status they need to correctly handle.

Example

Table 352.1: Effect of rounding mode (FLT_ROUNDS taking on values 0, 1, 2, or 3) on the result of a single precision value (given in the left column).

	0	1	2	3
1.00000007	1.0	1.00000012	1.00000012	1.0
1.00000003	1.0	1.0	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012

floating operands
evaluation format

~~The~~ Except for assignment and cast (which remove all extra range and precision), the values of operations with floating operands and values 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. 353

Commentary

Operands of both the arithmetic, comparison, and relational operators are subject to the usual arithmetic conversions. Operations that do not cause their operands to be the subject of the usually arithmetic conversions are assignment (both through the assignment operator and the passing of a value as an argument) and casts.

A floating-point expression has both a semantic type (as defined by the usual arithmetic conversion rules) and an evaluation format (as defined by the FLT_EVAL_METHOD). The need for an evaluation format arises because of historical practice and because of how floating-point arithmetic is implemented on some processors. Including floating constants in this list means that, for instance, `0.1f` will be represented using **float** only if FLT_EVAL_METHOD is 0; one guaranteed way of obtaining a **float** representation of this value is:

FLT_EVAL_METHOD 354

```
1  static const float tenth_f = 0.1f;
```

However, this is not a floating constant (in C, it is in C++).

The wording was changed by the response to DR #290.

C90

The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type;

This wording allows wider representations to be used for floating-point operands and expressions. It could also be interpreted (somewhat liberally) to support the idea that C90 permitted floating constants to be represented in wider formats where the usual arithmetic conversions applied.
Having the representation of floating constants change depending on how an implementation chooses to specify FLT_EVAL_METHOD is new in C99.

C++

Like C90, the FLT_EVAL_METHOD macro is not available in C++.

Other Languages

Java requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision.

Most other language specifications do not get involved in this level of detail.

Common Implementations

Early implementations of C evaluated all floating-point expressions in double-precision. This practice has continued in some implementations because developers have become used to the extra precision in calculations (also some existing code relies on it).

Processor designers have taken a number of different approaches to the architecture of hardware floating-point units.

floating-point architectures

- *Extended based.* Here all floating-point operands are evaluated in an extended format. For instance, the Intel x86 operates on an 80-bit extended floating-point format. Other processors whose arithmetic engine operates on an extended format include the Motorola 6888x and Intel 960.
- *Double based.* Here floating-point operands are evaluated in the double format. Processors whose arithmetic engine operates on double format includes IBM RS/6000 and Cray (which has no IEC 60559 single format at all).
- *Single/double based.* Here the processor instructions operate on floating-point values according to their type. Operations on the single format are often faster than those on double. Processors whose arithmetic engine operates like this include MIPS, SUN SPARC, and HP PA-RISC.
- *Single/double/extended based.* Here the processor provides instructions that can operate on three different floating-point types. Processors whose arithmetic engine operates like this include the IBM/370.

Coding Guidelines

The existence of evaluation formats are a cause for concern (i.e., they are a potential cost). Porting a program, which uses floating-point types, to an implementation that is identical except for a difference in FLT_EVAL_METHOD can result in considerably different output.

Many floating-point expressions are exact, or correctly rounded, when evaluated using twice the number of digits of precision as the data. For instance:

correctly rounded result

```
1  float cross_product(float w, float x, float y, float z)
2  {
3  return w * x - y * z;
4  }
```

yields a correctly rounded result if the expression is evaluated to the range and precision of type **double** (and the type **double** has at least twice the precision of the type **float**). By a happy coincidence, the developer may have obtained the most accurate result. Writing code that has less dependence of happy coincidences requires more effort:

```
1  float cross_product(float w, float x, float y, float z)
2  {
3  return ((double)w) * x - ((double)y) * z;
4  }
```

These coding guidelines are not intended as an introductory course on the theory and practicalities of coding floating-point expressions. But, it is hoped that readers will be sufficiently worried by the issues pointed out to either not use floating-point types in their programs or to become properly trained in the subject.

Example

```
1  #include <tgmath.h>
2
3  static float glob;
4
5  float f(void)
6  {
7  /*
8   * If FLT_EVAL_METHOD is zero, we want to invoke the float version of
9   * the following function to avoid the double rounding that would occur
10  * if the double sqrt function were invoked.
11  */
12 return hypot(+glob, 0.0);
13 }
```

FLT_EVAL_METHOD

The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:[19)] 354

Commentary

This specification is based on existing practice and provides a mechanism to make evaluation format information available to developers (many C90 implementations were already using the concept of an evaluation format, even though that standard did not explicitly mention it)— existing practice is affected by processor characteristics. Some processors do not provide hardware support to perform some floating-point operations in specific formats. This means that either an available format is used, or additional instruction is executed to convert the result value (slowing the performance of the application).

Demmel and Hida

The actual types denoted by the typedef names `float_t` and `double_t` are dependent on the value of the `FLT_EVAL_METHOD` macro. Demmel and Hida give an error analysis for summing a series when the intermediate results are held to greater precision than the values in the series.

C90

Support for the `FLT_EVAL_METHOD` macro is new in C99. Its significant attendant baggage was also present in C90 implementations, but was explicitly not highlighted in that standard.

C++

Support for the `FLT_EVAL_METHOD` macro is new in C99 and it is not available in C++. However, it is likely that the implementation of floating point in C++ will be the same as in C.

Other Languages

Most language implementations use the support provided by their hosts' floating-point unit, if available. They may also be influenced by the representation of floating types used by other language implementations

(e.g., C). Even those languages that only contain a single floating-point type are not immune to evaluation format issues, they simply have fewer permutations to consider. Fortran has quite a complex floating-point expression evaluation specification. This also includes a concept called *widest-need*.

Common Implementations

Implementations often follow the default behavior of the hardware floating point unit, if one is available. In some cases the time taken to perform a floating-point operation does not depend on the evaluation format.[33,49] In these cases the widest format is often the default. When floating-point operations have to be emulated in software, performance does depend on evaluation format and implementations often choose the narrowest format.

The Intel IA64[35] provides a mechanism for software emulation, as does the HP–was DEC– Alpha.[60]

Coding Guidelines

Calculations involving floating-point values often involve large arrays of these values. Before the late 1990s, the availability of storage to hold these large arrays was often an important issue. Many applications used arrays of floats, rather than arrays of doubles, to save storage space. These days the availability of storage is rarely an important issue in selecting the base type of an array.

The choice of floating-point type controls the precision to which results are rounded when they are assigned to an object. Depending on the value of FLT_EVAL_METHOD, this choice may also have some influence on the precision to which operations, on floating-point operands, are performed.

Example

```
1  #include <float.h>
2
3  #if FLT_EVAL_METHOD < 1
4  #error Program won't give the correct answers using this implementation
5  #endif
```

355-1 indeterminable;

Commentary

Some processors support more than one evaluation format and provide a mechanism for encoding the choice within the instruction encoding (rather than in a separate configuration register), and some support both mechanisms.[60] A program executing on such a processor, may have been built from two translation units—one that was translated using one evaluation format and one using a different evaluation format.

Common Implementations

An example of a processor where a translator might want to support a FLT_EVAL_METHOD macro value of less than zero is the Acorn ARM processor. The ARM machine code generated for the following expression statement:

```
1  float a;
2  double b, c;
3  a = b + c;
```

might be (if the FLT_EVAL_METHOD macro had a value of 0 or 1):

```
1  adfd    f0, f0, f1   # f0 = f0 + f1, rounded to double
2  mvfs    f0, f0       # f0 = f0, rounded to single
```

however, the following would be a more efficient sequence:

```
1  adfs    f0, f0, f1   # f0 = f0 + f1, rounded to single
```

but the addition operation is performed to less precision than is available in the operands. An implementation can only generate this instruction sequence if the FLT_EVAL_METHOD macro has a value less than zero.

Coding Guidelines

At first sight, having to use an implementation that specified a value of -1 for FLT_EVAL_METHOD would appear to be very bad news. It means that developers cannot depend on any particular evaluation format being used consistently when a program is translated. In this case, the responsible developer is forced to ensure that explicit casts are used before and after every arithmetic operation. Casting an operand before an operation guarantees a minimum level of precision. Casting the result of an operation ensures that any extra precision is not passed on to the next evaluation. The end result is a program that gives much more consistent results when ported.

The extent to which an indeterminable value will affect the irresponsible developer cannot be estimated (i.e., will the differences in behavior be significant; will an indeterminate value for FLT_EVAL_METHOD have made any difference compared to some other value?) For source written by that developer, perhaps not. But the developer may have copied some code from an expert who assumed a particular evaluation method. There is little these coding guidelines can do to address the issue of the irresponsible developer. Given that implementations using an indeterminate value appears to be rare a guideline recommendations is not considered worthwhile.

---

0 evaluate all operations and constants just to the range and precision of the type; 356

Commentary

usual arithmetic conversions

In many contexts, operands will have been subject to the usual arithmetic conversions, so objects having type **float** will have been converted to type **double**. The difference between this and an evaluation format value of 1 is that constants may be represented to greater precision. In:

```
1  float f1, f2;
2  if (f1 < f2)
```

as-if rule

an implementation may choose to perform the relational test without performing the usual arithmetic conversions by invoking the as-if rule. No recourse to the value of FLT_EVAL_METHOD is needed. But in:

```
1  float f1;
2  if (f1 < 0.1f)
```

the value of FLT_EVAL_METHOD needs to be taken into account because of a possible impact on the representation used for 0.1f.

Other Languages

This might be thought to be the default specification for many languages that do not aim to get as close to the hardware as C does. However, few languages define the behavior of operations on floating-point values with the precision needed to make this the only implementation option. Performance of the translated program is an issue for all languages and many follow, like C, the default behavior of the underlying hardware (even Fortran).

Common Implementations

This form of evaluation is most often used by implementations that perform floating-point operations in software; it being generally felt that the execution-time penalty of using greater precision is not compensated for by improvements in the accuracy of results. (If a developer wants the accuracy of **double**, let them insert explicit casts).

This evaluation format is not always used on hosts whose floating-point hardware performs operations in **float** and **double** types. The desire for compatibility with behavior seen on other processors may have a higher priority. Some translators provide an option to select possible evaluation modes.

Coding Guidelines

This evaluation format is probably what the naive developer thinks occurs in all implementations. Developers who have been using C for many years may be expecting the behavior specified by an evaluation format value of 1. As such, this evaluation format contains the fewest surprises for a developer expecting the generated machine code to mimic the type of operations in the source code. An implementation is doing exactly what it is asked to do.

---

357 17) IEC 60559:1989 specifies quiet and signaling NaNs. footnote 17

Commentary

The IEEE 754R committee is tentatively planning to remove the requirement of support for signaling NaNs from the floating-point standard and to not recommend such support. The collected experience of the 754R committee is that use of signaling NaNs has been vanishingly small and in most of those cases alternate methods would suffice. The minimal utility does not justify the considerable cost required of system and tool implementors to support signaling NaNs in a coherent manner nor the cost to users in dealing with overburdened tools and overly complicated specification. Liaison report to WG14, Sep 2002

---

358 For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

Commentary

The standard does not require that such encoding be provided, only that, if there are encodings with similar behavior, the terms quiet NaN and signaling NaN can be applied to them.

C90

The concept of NaN is new, in terms of being explicitly discussed, in C99.

C++

18.2.1.2p34

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

Meaningful for all floating point types.

Shall be true for all specializations in which `is_iec559 != false`.

18.2.1.2p37

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."194)

Meaningful for all floating point types.

Shall be true for all specializations in which is_iec559 != false.

18.2.1.2p52

```
static const bool is_iec559;
```

True if and only if the type adheres to IEC 559 standard.[201)]

The C++ Standard requires NaNs to be supported if IEC 60559 is supported, but says nothing about the situation where that standard is not supported by an implementation.

Other Languages

Few languages get involved in this level of detail.

Common Implementations

HP–was DEC– VAX has a bit representation for reserved, Cray a bit representation for indefinite. They behave like NaNs.

Coding Guidelines

If developers only make use of the functionality specified by the standard, this representation detail is unlikely to be of significance to them.

---

footnote 18

18) Evaluation of **FLT_ROUNDS** correctly reflects any execution-time change of rounding mode through the function **fesetround** in **<fenv.h>**. 359

Commentary

float.h 348 suitable for #if

This footnote clarifies the intended behavior for the FLT_ROUNDS (an earlier requirement also points out that this macro need not be a constant expression), at least for values between 0 and 3. It is possible for FLT_ROUNDS to have a value of -1 (or perhaps a value indicating an implementation-defined rounding behavior) and be unaffected by changes in rounding mode through calls to the function fesetround. The evaluation of FLT_ROUNDS also needs to correctly reflect any translation-time change of rounding mode through, for instance, use of a **#pragma** directive.

FLT_ROUNDS 352 Like errno FLT_ROUNDS provides a method of accessing information in the execution environment.

C90

Support for the header **<fenv.h>** is new in C99. The C90 Standard did not provide a mechanism for changing the rounding direction.

C++

Support for the header **<fenv.h>** and the fesetround function is new in C99 and is not specified in the C++ Standard.

---

footnote 19

19) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. 360

Commentary

complex types

Floating types are made up of the real and complex types

C90

Support for complex types is new in C99.

C++

The complex types are a template class in C++. The definitions of the instantiation of these classes do not specify that the evaluation format shall be the same as for the real types. But then, the C++ Standard does not specify the evaluation format for the real types.

Other Languages

The complex types in Fortran and Ada follow the same evaluation rules as the real types in those languages.

361 For example, if **FLT_EVAL_METHOD** is 1, then the product of two **float _Complex** operands is represented in the **double _Complex** format, and its parts are evaluated to **double**.

Commentary

This restatement of the specification makes doubly sure that there is no possible ambiguity of the intent.

C++

The C++ Standard does not specify a FLT_EVAL_METHOD mechanism.

362 **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;

Commentary

Historically, this is the behavior many experienced C developers have come to expect.

Common Implementations

This is how translators are affected (i.e., existing source expects this behavior and a vendor wants their product to handle existing source) by the original K&R behavior.

Coding Guidelines

Although this behavior is implicitly assumed in much existing source, inexperienced developers may not be aware of it. As such it is an educational rather than a coding guidelines issue.

363 **2** evaluate all operations and constants to the range and precision of the **long double** type.

Commentary

This choice reflects the fact that some hardware floating-point units (e.g., the Intel x86) support a single evaluation format for all floating-point operations. Continually having to convert the intermediate results, during the evaluation of an expression, to another format imposes an execution-time overhead that translator vendors may feel their customers will not tolerate.

Common Implementations

The Intel x86[33] performs floating-point operations in an 80-bit extended representation (later versions of this processor contained additional instructions that operated on a 32-bit representation only). Some translators, for this processor, use a **long double** type that has 96 bits in its object representation and 80 bits in its value representation.

object representation
value representation

364 All other negative values for **FLT_EVAL_METHOD** characterize implementation-defined behavior.

Commentary

The standard does not say anything about other possible positive values, not even reserving them for future revisions.

Common Implementations

One form of evaluation format seen in some implementations is widest-need.[4] This format was also described in the Floating-point C Extensions technical report[53] produced by the NCEG (Numerical C Extensions Group) subcommittee of X3J11. Widest-need examines a full expression for the widest real type in that expression. All of the floating-point operations in that expression are then evaluated to that widest type.

X3J11

Coding Guidelines

The issues behind a widest-need evaluation strategy are the same as those for integer types. A change of type of one operand can affect the final result of an expression. Recommending the use of a single floating type is not such an obvious solution (as it is for integer types). Producing a correctly rounded result requires the use of two different floating types. This is a complex issue and your author knows of no simple guideline recommendation that might be applicable.

correctly rounded result

Example

```
1  extern float f1, f2;
2  extern double d1, d2;
3  extern long double ld1, ld2;
4
5  void f(void)
6  {
7  f1 = f1 * f2; /* Widest type is float. */
8  /*
9   * Widest type is double.
10  * If FLT_EVAL_METHOD is 0 f1 * f2 would be evaluated in float.
11  */
12 f1 = d1 + f1 * f2;
13 }
```

The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign: 365

Commentary

The values listed below are all integers. The descriptions imply an integer type, but this is not explicitly specified (other requirements apply to integer values). One likely use of the values listed here are as the size in an array definition.

float.h 348 suitable for #if

C90

In C90 the only expression that was required to be a constant expression was FLT_RADIX. It was explicitly stated that the others need not be constant expressions; however, in most implementations, the values were constant expressions.

C++

18.2.1p3 *For all members declared* `static const` *in the* `numeric_limits` *template, specializations shall define these values in such a way that they are usable as integral constant expressions.*

Other Languages

Java specifies the exact floating-point format to use, so many of the following values are already implicitly defined in that language. It also defines the classes `java.lang.Float` and `java.lang.Double` which contain a minimal set of related information.

Coding Guidelines

Like other identifiers, which are constant expressions, defined by the C Standard, they represent a symbolic property. As discussed elsewhere, making use of the property rather than a particular arithmetic value has many advantages.

symbolic name

FLT_RADIX — radix of exponent representation, b 366

FLT_RADIX **2**

Commentary

A poorly worded way of saying the radix of the significand. The value of this macro is an input parameter to many formulas used in the calculation of rounding errors. The constants used in some numerical algorithms

may depend on the value of FLT_RADIX. For instance, minimizing the error in the calculation of sine requires using different constant values for different radixs.

The radix used by humans is 10. A fraction that has a finite representation in one radix may have an infinitely repeating representation in another radix. This only occurs when there is some prime number, p, that divides one radix but not the other. There is no prime divisor of 2, 8, or 16 that does not also divide 10. Therefore, it is always possible to exactly represent binary, octal, or hexadecimal fractions exactly as base 10 fractions. The prime number 5 divides 10, but not 2, 8, or 16. Therefore, there are decimal fractions that have no exact representation in binary, octal, or hexadecimal. For instance, 0.1_{10} has an infinitely recurring representation in base 2 (i.e., $1.10011001100110011\ldots_2 \times 2^{-4}$).

When FLT_RADIX has a value other than 2, the precision wobbles. For instance, with a FLT_RADIX of 16 for the type **float**, there are 4×6 or 24 bits of significand, but only 21 bits of precision that can be counted on. The shifting of the bits in the significand, to ensure correct alignment, occurs in groups of 4, meaning that 3 bits can be 0. A large radix has the advantages of a large range (of representable values) and high speed (of execution). The price paid for this advantage may be less accuracy and larger errors in the results of arithmetic operations. A radix of 2 provides the best accuracy (it minimizes the root mean square round-off error[8]).

The base of the exponent affects the significand if it has a value other than 2. In these cases the significand will contain leading zeros for some floating-point values. A discussion of the trade-off involved in choosing the base and the exponent range is given by Richman.[57]

C++

```
static const int radix;
```

18.2.1.2p16

For floating types, specifies the base or radix of the exponent representation (often 2).[185)]

Other Languages

A numeric inquiry function, RADIX, which returns the radix of a real or integer argument was added to Fortran 90. Few other languages get involved in exposing such details to developers.

Common Implementations

Because of the widespread use of IEC 60559 by processor vendors, the most commonly encountered value is 2. Values of 8 and 16 are still sometimes encountered, but such usage is becoming rarer as the machines supporting them are scrapped (or move to support IEC 60559, like the IBM 390, and existing software is migrated); although software emulation of floating-point operators sometimes uses a non-2 value.[15]

The Unisys A Series[67] uses a FLT_RADIX of 8. The Motorola 68000 processor[49] supports a packed decimal real format that has a FLT_RADIX of 10. This processor also supports IEC 60559 format floating-point types. The original IBM 390 floating-point format uses a FLT_RADIX of 16. Later versions of this processor also support the IEC 60559 format.[59] The Data General Nova[20] also uses a FLT_RADIX of 16. The Los Alamos MANIAC II used a radix of 256, only one was every built, using valves, in 1955. The Moscow State University SETUN experimental computer[70] used a radix of 3.

The advantage of a larger radix is that it increases the probability[62] that the exponents of two floating-point values, of a binary operator, will be the same, reducing the need to align the value significands. A larger radix is also likely to require less work to normalise result of an operation.

The advantage of a smaller radix is that it has a smaller maximum and average representation error[12] (for the same total number of bits in the floating-point representation).

A radix of 2 introduces a significant performance penalty for software implementations of floating-point operations and a radix of 8 has been found[15] to be a better choice.

Hardware support for a radix of 10 has started to take off in environments where accurate calculations involving values represented in *human form* is important[18] (Cowlishaw[17] provides a specification of IEC

60559 conforming decimal arithmetic). There is a strong possibility that use of radix 10 will eventually become the norm, as the economic (e.g., impact on chip fabrication costs) and technical (e.g., runtime efficiency) costs become less important than the benefits, to end-users, of decimal arithmetic.

Example

Malcolm[46] provides Fortran source that computes the radix used by the processor on which the code executes.

```
#include <float.h>

#if FLT_RADIX != 16
#error Go away!  This program only translates on old cranky hosts.
#endif
```

FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG

Commentary

Assumed to be an integer (which it is for almost all implementations).

Rationale The term FLT_MANT_DIG stands for "float mantissa digits." The Standard now uses the more precise term *significand* rather than *mantissa*.

Other Languages

Few languages get involved in exposing such details to the developer. A numeric inquiry function, DIGITS, which returns the number of base b digits in the significand of the argument, was added in Fortran 90.

Common Implementations

The IEC 60559 Standard defines single-precision as 24 bits (base-2 digits) and double-precision as 53 bits. It also defines a single extended as having 32 or more bits and a double extended as having 64 or more bits. In the actual bit representation (for single and double) there is always 1 less bit. This is because normalized numbers contain an implied 1 in the most significant bit and subnormals an implied 0 in that position.

subnormal numbers 338

In many freestanding environments the host processor does not support floating-point operations in hardware; they are emulated via library calls. To reduce the significant execution-time overhead of performing floating-point operations in software, a 16-bit significand is sometimes used (the exponent and sign occupy 8 bits, giving a total of 24 bits).

Some processors (e.g., the ADSP–2106x[3] and Intel XScale microarchitecture[34]) support single extended by adding an extra byte to the type representation, giving a 40-bit significand. Some processors also support a floating-point format, in hardware, that cannot represent the full range of values supported by IEC 60559. For instance, the ADSP–2106x[3] has what it calls a *short word* floating-point format— occupying 16 bits with a 4-bit exponent, sign bit and an 11-bit significand.

Example

Malcolm[46] provides Fortran source that computes the number of mantissa digits used by the processor on which the code executes.

```
#include <float.h>

#if (DBL_MANT_DIG != 53) || (FLT_MANT_DIG != 24)
#error Please rewrite the hex floating-point literals
#endif
```

368 — number of decimal digits, n, such that any floating-point number in the widest supported floating type with p_{max} radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

DECIMAL_DIG macro

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

DECIMAL_DIG **10**

Commentary

The conversion process here is base-FLT_RADIX⇒base-10⇒base-FLT_RADIX (the opposite conversion ordering is discussed in the following C sentence). The binary fraction 0.00011001100110011001100110011 is exactly equal to the decimal fraction 0.099999999627470970153808593 75, which might suggest that a DECIMAL_DIG value of 10 is not large enough. However, the second part of the above requirement is that the number be converted back without change, not that the exact decimal representation be used. If FLT_RADIX is less than 10, then for the same number of digits there will be more than one decimal representation for each binary number. The number N of radix-B digits required to represent an n-digit FLT_RADIX floating-point number is given by the condition (after substituting C Standard values):[47]

$$10^{N-1} > FLT\_RADIX^{LDBL\_MANT\_DIG} \tag{368.1}$$

Minimizing for N, we get:

$$N = 2 + \frac{LDBL\_MANT\_DIG}{\log_{FLT\_RADIX} 10} \tag{368.2}$$

When FLT_RADIX is 2, this simplifies to:

$$N = 2 + \frac{LDBL\_MANT\_DIG}{3.321928095} \tag{368.3}$$

By using fewer decimal digits, we are accepting that the decimal value may not be the one closest to the binary value. It is simply a member of the set of decimal values having the same binary value that is representable in DECIMAL_DIG digits. For instance, the decimal fraction 0.1 is closer to the preceding binary fraction than any other nearby binary fractions.[61]

379 DECIMAL_DIG conversion recommended practice

When b is not a power of 10, this value will be larger than the equivalent *_DIG macro. But not all of the possible combinations of DECIMAL_DIG decimal digits can be generated by a conversion. The number of representable values between each power of the radix is fixed. However, each successive power of 10 supports a greater number of representable values (see Figure 368.1). Eventually the number of representable decimal values, in a range, is greater than the number of representable p radix values. The value of DECIMAL_DIG denotes the power of 10 just before this occurs.

335 precision floating-point

C90

Support for the DECIMAL_DIG macro is new in C99.

C++

Support for the DECIMAL_DIG macro is new in C99 and specified in the C++ Standard.

Other Languages

Few other languages get involved in exposing such details to the developer.

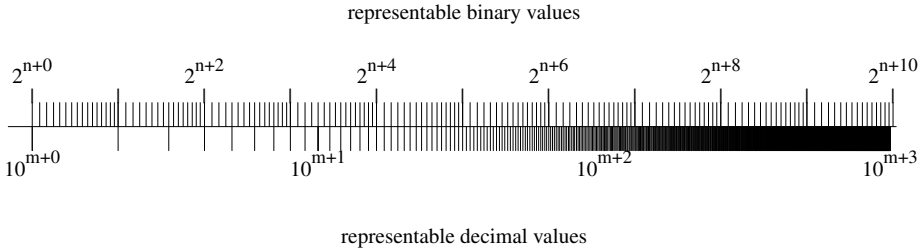

Figure 368.1: Representable powers of 10 and powers of 2 on the real line.

Common Implementations

A value of 17 would be required to support IEC 60559 double precision. A value of 9 is sufficient to support IEC 60559 single precision.

long double Apple

The format used by Apple on the PowerPC[4] to represent the **long double** type is the concatenation of two **double** types. Apple recommends that the difference in exponents, between the two doubles, be 54. However, it is not possible to guarantee this will always be the case, giving the representation an indefinite precision. The number of decimal digits needed to ensure that a cycle of conversions delivers the original value is proportional to the difference between the exponents in the two doubles. When the least significant double has a value of zero, the difference can be very large.

The following example is based on the one given in the Apple Macintosh PowerPC numerics documentation.[4] If an object with type **double**, having the value `1.2`, is assigned to an object having **long double** type, the least significant bits of the significand are given the zero value. In hexadecimal (and decimal to 34 digits of accuracy):

```
1        0x3FF33333 33333333 00000000 00000000
2          1.19999999999999995591079014993738
```

The decimal form is the closest 34-digit approximation of the long double number (represented using double-double). It is also the closest 34-decimal digit approximation to an infinitely precise binary value whose exponent is 0 and whose fractional part is represented by 13 sequences of `0011` followed by 52 binary zeros, followed by some nonzero bits. Converting this decimal representation back to a binary representation, the Apple PowerPC Numerics library returns the closest double-double approximation of the infinitely precise value, using all of the bits of precision available to it. It will use all 53 bits in the head and 53 bits in the tail to store nonzero values and adjust the exponent of the tail accordingly. The result is:

```
1        0x3FF33333 33333333 xxxyzzzz zzzzzzzz
```

where `xxx` represents the sign and exponent of the tail, and `yzzz...` represents the start of a nonzero value. Because the tail is always nonzero, this value is guaranteed to be not equal to the original value.

Implementations add additional bits to the exponent and significand to support a greater range of values and precision, and most keep the bits representing the various components contiguous. The Unisys A Series[67] represents the type **double** using the same representation as type **float** in the first word, and by having additional exponent and significand bits in a second word. The external effect is the same. But it is an example of how developer assumptions about representation, in this case bits being contiguous, can be proved wrong.

Coding Guidelines

One use of this macro is in calculating the amount of storage needed to hold the character representation, in decimal, of a floating-point value. The definition of the macro excludes any characters (digits or otherwise) that may be used to represent the exponent in any printed representation.

The only portable method of transferring data having a floating-point type is to use a character-based representation (e.g., a list of decimal floating-point numbers in character form). For a given implementation, this macro gives the minimum number of digits that must be written if that value is to be read back in without change of value.

```
printf("%#.*g", DECIMAL_DIG, float_valu);
```

Space can be saved by writing out fewer than DECIMAL_DIG digits, provided the floating-point value contains less precision than the widest supported floating type. Trailing zeros may or may not be important; the issues involved are discussed elsewhere.

Converting a floating-point number to a decimal value containing more than DECIMAL_DIG digits may, or may not, be meaningful. The implementation of the printf function may, or may not, choose to convert to the decimal value closest to the internally represented floating-point value, while other implementations simply produce garbage digits.[61]

Example

```
#include <float.h>

/*
 * Array big enough to hold any decimal representation (at least for one
 * implementation).  Extra characters needed for sign, decimal point,
 * and exponent (which could be six, E-1234, or perhaps even more).
 */
#if LDBL_MAX_10_EXP < 10000

char float_digits[DECIMAL_DIG + 1 + 1 + 6 + 1];

#else
#error Looks like we need to handle this case
#endif
```

---

369 — number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits, *_DIG macros

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

FLT_DIG	**6**
DBL_DIG	**10**
LDBL_DIG	**10**

Commentary

The conversion process here is base-10⇒base-FLT_RADIX⇒base-10 (the opposite ordering is described elsewhere). These macros represent the maximum number of decimal digits, such that each possible digit sequence (value) maps to a different (it need not be exact) radix b representation (value). If more than one decimal digit sequence maps to the same radix b representation, it is possible for a different decimal sequence (value) to be generated when the radix b form is converted back to its decimal representation. 368 DECIMAL_DIG macro

C++

18.2.1.2p9

```
static const int digits10;
```

Number of base 10 digits that can be represented without change.

Footnote 184 *Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.*

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

The Fortran 90 `PRECISION` inquiry function is defined as `INT((p-1) * LOG10(b)) + k`, where `k` is 1 is `b` is an integer power of 10 and 0 otherwise.

Example

```
#include <limits.h>
#include <stdio.h>

float real_float;

int main(void)
{
for (int findex = INT_MAX; findex > INT_MAX-20; findex--)
   {
   real_float=(float)findex; /* Just in case we have a 'clever' optimizer. */
   printf("I=%d, F1=%f, F2=%f\n", findex, (float)findex, real_float);
   }
}
```

---

*_MIN_EXP — minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, e_{min} 370

FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP

Commentary

*_MIN macros 378 These values are essentially the minimum value of the exponent used in the internal floating-point representation. The *_MIN macros provide constant values for the respective minimum normalized floating-point value. No minimum values are given in the standard. The possible values can be calculated from the following:

$$FLT\_MIN\_EXP = \frac{FLT\_MIN\_10\_EXP}{\log(FLT\_RADIX)} \pm 1 \tag{370.1}$$

C++

18.2.1.2p23

```
static const int min_exponent;
```

Minimum negative integer such that `radix` *raised to that power is in the range of normalised floating point numbers.*[189)]

Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP. Footnote 189

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2p4

Other Languages

Fortran 90 contains the inquiry function MINEXPONENT which performs a similar function.

Common Implementations

In IEC 60559 the value for single-precision is -125 and for double-precision -1021. The two missing values (available in the biased notation used to represent the exponent) are used to represent 0.0, subnormals, infinities, and NaNs. 338 floating types can represent

Coding Guidelines

The usage of these macros in existing code is so rare that reliable information on incorrect usage is not available, making it impossible to provide any guideline recommendations. (The rare usage could also imply that a guideline recommendation would not be worthwhile).

---

371 — minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{min}-1} \rceil$ *_MIN_10_EXP

FLT_MIN_10_EXP	**-37**
DBL_MIN_10_EXP	**-37**
LDBL_MIN_10_EXP	**-37**

Commentary

Making this information available as an integer constant allows it to be accessed in a **#if** preprocessing directive.

These are the exponent values for normalized numbers. If subnormal numbers are supported, the smallest representable value is likely to have an exponent whose value is FLT_DIG, DBL_DIG, and LDBL_DIG less than (toward negative infinity) these values, respectively. 338 subnormal numbers

The Committee is being very conservative in specifying the minimum values for the exponents of the types **double** and **long double**. An implementation is permitted to define the same range of exponents for all floating-point types. There may be normalized numbers whose respective exponent value is smaller than the values given for these macros; for instance, the exponents appearing in the *_MIN macros. The power of 10 exponent values given for these *_MIN_10_EXP macros can be applied to any normalized significand.

C++

```
static const int min_exponent10;
```

18.2.1.2p25

Minimum negative integer such that 10 raised to that power is in the range of normalised floating point numbers.[190)]

Footnote 190 *Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.*

18.2.2p4 *Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.*

Common Implementations

The value of DBL_MIN_10_EXP is usually the same as FLT_MIN_10_EXP or LDBL_MIN_10_EXP. In the latter case a value of -307 is often seen.

---

*_MAX_EXP — maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, e_{max} 372

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

Commentary

*_MIN_EXP 370 FLT_RADIX to the power *_MAX_EXP is the smallest large number that cannot be represented (because of limited exponent range).

C++

18.2.1.2p27

```
static const int max_exponent;
```

Maximum positive integer such that `radix` *raised to the power one less than that integer is a representable finite floating point number.*[191)]

Footnote 191 *Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.*

18.2.2p4 *Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.*

Other Languages

Fortran 90 contains the inquiry function MAXEXPONENT which performs a similar function.

Common Implementations

In IEC 60559 the value for single-precision is 128 and for double-precision 1024.

---

*_MAX_10_EXP — maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p})b^{e_{max}}) \rfloor$ 373

FLT_MAX_10_EXP	**+37**
DBL_MAX_10_EXP	**+37**
LDBL_MAX_10_EXP	**+37**

Commentary

As in choosing the *_MIN_10_EXP values, the Committee is being conservative. 371 *_MIN_10_EXP

C++

```
static const int max_exponent10;
```

Maximum positive integer such that 10 raised to that power is in the range of normalised floating point numbers. 18.2.1.2p29

Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP. Footnote 192

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2

Common Implementations

The value of DBL_MAX_10_EXP is usually the same as FLT_MAX_10_EXP or LDBL_MAX_10_EXP. In the latter case a value of 307 is often seen.

---

374 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown: floating values listed

Commentary

This is a requirement on the implementation. The requirement that they be constant expressions ensures that they can be used to initialize an object having static storage duration.

The values listed represent a floating-point number. Their equivalents in the integer domain are required to have appropriate promoted types. There is no such requirement specified for these floating-point values. symbolic name integer types sizes

C90

C90 did not contain the requirement that the values be constant expressions.

C++

This requirement is not specified in the C++ Standard, which refers to the C90 Standard by reference.

---

375 — maximum representable finite floating-point number, $(1 - b^{-p})b^{e_{max}}$

FLT_MAX	**1E+37**
DBL_MAX	**1E+37**
LDBL_MAX	**1E+37**

Commentary

There is no requirement that the type of the value of these macros match the real type whose maximum they denote. Although some implementations include a representation for infinity, the definition of these macros require the value to be finite. These values correspond to a FLT_RADIX value of 10 and the exponent values given by the *_MAX_10_EXP macros. 373 *_MAX_10_EXP

The HUGE_VAL macro value may compare larger than any of these values.

C++

18.2.1.2p4

```
static T max() throw();
```

Maximum finite value.[182]

Footnote 182 *Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.*

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

The class `java.lang.Float` contains the member:

```
1  public static final float MAX_VALUE = 3.4028235e+38f
```

The class `java.lang.Double` contains the member:

```
1  public static final double MAX_VALUE = 1.7976931348623157e+308
```

Fortran 90 contains the inquiry function `HUGE` which performs a similar function.

Common Implementations

Many implementations use a suffix to give the value a type corresponding to what the macro represents. The IEC 60559 values of these macros are:

```
single float FLT_MAX        3.40282347e+38
double float DBL_MAX        1.7976931348623157e+308
```

EXAMPLE 380 minimum floating-point representation

EXAMPLE 381 IEC 60559 floating-point

Coding Guidelines

How many calculations ever produce a value that is anywhere near `FLT_MAX`? The known Universe is thought to be 3×10^{29}mm in diameter, 5×10^{19} milliseconds old, and contain 10^{79} atoms, while the Earth is known to have a mass of 6×10^{24}Kg.

Floating-point values whose magnitude approaches `DBL_MAX`, or even `FLT_MAX` are only likely to occur as the intermediate results of calculating a final value. Very small numbers are easily created from values that do not quite cancel. Dividing by a very small value can lead to a very large value. Very large values are thus more often a symptom of a problem, rounding errors or poor handling of values that almost cancel, than of an application meaningful value.

Some processors saturate to the maximum representable value on overflow, while others return an infinity. Testing whether an operation will overflow is one use for these identifiers. However, in C99, tests such as (`x > LDBL_MAX`) can now be replaced by the `isinf` macro.

Example

```
1  #include <float.h>
2
3  #define FALSE 0
4  #define TRUE 1
5
6  extern float f_glob;
7
```

```
8   _Bool f(float p1, float p2)
9   {
10  if (f_glob > (FLT_MAX / p1))
11     return FALSE;
12
13  f_glob *= p1;
14
15  if (f_glob > (FLT_MAX - p2))
16     return FALSE;
17
18  f_glob += p2;
19
20  return TRUE;
21  }
```

376 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

Commentary

The previous discussion is applicable here.

374 floating values listed

377 — the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p} *_EPSILON

FLT_EPSILON	**1E-5**
DBL_EPSILON	**1E-9**
LDBL_EPSILON	**1E-9**

Commentary

The Committee is being very conservative in specifying these values. Although IEC 60559 arithmetic is in common use, there are several major floating-point implementations of it that do not support an extended precision. The Committee could not confidently expect implementations to support the type **long double** containing greater accuracy than the type **double**. IEC 60559

Like the *_DIG macros more significand digits are required for the types **double** and **long double**. 369 *_DIG macros

C++

18.2.1.1p20

```
static T epsilon() throw();
```

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[187)]

Footnote 187

Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

Other Languages

Fortran 90 contains the inquiry function EPSILON, which performs a similar function.

Common Implementations

long double 368
Apple

Some implementations (e.g., Apple) use a contiguous pair of objects having type **double** to represent an object having type **long double**. Such a representation creates a second meaning for LDBL_EPSILON. This is because, in such a representation, the least value greater than 1.0 is 1.0+LDBL_MIN, a difference of LDBL_MIN (which is not the same as $b^{(1-p)}$)— the correct definition of *_EPSILON. Their IEC 60559 values are:

```
FLT_EPSILON     1.19209290e-7 /* 0x1p-23 */
DBL_EPSILON     2.2204460492503131e-16 /* 0x1p-52 */
```

Coding Guidelines

It is a common mistake for these values to be naively used in equality comparisons:

```
1  #define EQUAL_DBL(x, y) ((((x)-DBL_EPSILON) < (y)) && \
2                           (((x)+DBL_EPSILON) > (y)))
```

ULP 346

This test will only work as expected when x is close to 1.0. The difference value not only needs to scale with x, (x + x*DBL_EPSILON), but the value DBL_EPSILON is probably too small (equality within 1 ULP is a very tight bound):

```
1  #define EQUAL_DBL(x, y) ((((x)*(1.0-MY_EPSILON)) < (y)) && \
2                           (((x)*(1.0+MY_EPSILON)) > (y)))
```

Even this test fails to work as expected if x and y are subnormal values. For instance, if x is the smallest subnormal and y is just 1 ULP bigger, y is twice x.

Another, less computationally intensive, method is to subtract the values and check whether the result is within some scaled approximation of zero.

```
1  #include <math.h>
2
3  _Bool equalish(double f_1, double f_2)
4  {
5  int exponent;
6  frexp(((fabs(f_1) > fabs(f_2)) ? f_1 : f_2), &exponent);
7  return (fabs(f_1-f_2) < ldexp(MY_EPSILON, exponent));
8  }
```

---

*_MIN macros

— minimum normalized positive floating-point number, $b^{e_{min}-1}$ 378

FLT_MIN	**1E-37**
DBL_MIN	**1E-37**
LDBL_MIN	**1E-37**

Commentary

These values correspond to a FLT_RADIX value of 10 and the exponent values given by the *_MIN_10_EXP macros. There is no requirement that the type of these macros match the real type whose minimum they denote. Implementations that support subnormal numbers will be able to represent smaller quantities than these.

*_MIN_10_EXP 371
subnormal numbers 338

C++

18.2.1.2p1

```
static T min() throw();
```

Maximum finite value.[181)]

Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc. Footnote 181

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2p4

Other Languages

The class `java.lang.Float` contains the member:

```
1  public static final float MIN_VALUE = 1.4e-45f;
```

The class `java.lang.Double` contains the member:

```
1  public static final double MIN_VALUE = 5e-324;
```

which are the smallest subnormal, rather than normal, values.

Fortran 90 contains the inquiry function `TINY` which performs a similar function.

Common Implementations

Their IEC 60559 values are:

```
FLT_MIN         1.17549435e-38f
DBL_MIN         2.2250738585072014e-308
```

Implementations without hardware support for floating point sometimes chose the minimum required limits because of the execution-time overhead in supporting additional bits in the floating-point representation.

Coding Guidelines

How many calculations ever produce a value that is anywhere near as small as `FLT_MIN`? The hydrogen atom weighs 10^{-26}Kg and has an approximate radius of 5×10^{-11} meters, well within limits. But current theories on the origin of the Universe start at approximately 10^{-36} seconds, a very small number. However, writers of third-party libraries might not know whether their users are simulating the Big Bang, or weighing groceries. They need to ensure that all cases are handled.

Given everyday physical measurements, which don't have very small values, where can very small numbers originate? Subtracting two floating-point quantities that differ by 1 ULP, for instance, produces a value that is approximately 10^{-5} smaller. Such a difference can result from random fluctuations in the values input to a program, or because of rounding errors in calculations. Producing a value that is close to `FLT_MIN` invariably requires either a very complex calculation, or an iterative algorithm using values from previous iterations. Intermediate results that are expected to produce a value of zero may in fact deliver a very small value. Subsequent tests against zero fail and the very small value is passed through into further calculations. One solution to this problem is to have a relatively wide test of zeroness. In many physical systems a value that is a factor of 10^{-6} smaller than the smallest measurable quantity would be considered to be zero.

Rev 378.1

Floating-point comparisons against zero shall take into account the physical properties or engineering tolerances of the system being controlled or simulated.

There might be some uncertainty in the interpretation of the test `(abs(x) < FLT_MIN);` is it an approximate test against zero, or a test for a subnormal value? The C Standard now includes the `fpclassify` macro for obtaining the classification of its argument, including subnormal.

Example

```
1  #include <math.h>
2
3  #define MIN_TOLERANCE (1e-9)
4
5  _Bool inline effectively_zero(float valu)
6  {
7  return (abs(valu) < MIN_TOLERANCE);
8  }
```

Recommended practice

DECIMAL_DIG conversion recommended practice

Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back should be the identity function. 379

Commentary

Why is this a recommended practice? Unfortunately many existing implementations of `printf` and `scanf` do a poor job of base conversions, and they are not the identity functions.

To claim conformance to both C99 and IEC 60559 (Annex F in force), the requirements of F.5 Binary-decimal conversion must be met. Just making use of IEC 60559 floating-point hardware is not sufficient. The I/O library can still be implemented incorrectly and the conversions be wrong.

Rationale When the radix b is not a power of 10, it can be difficult to find a case where a decimal number with $p \times \log_{10} b$ digits fails. Consider a four-bit mantissa system (that is, base $b = 2$ and precision $p = 4$) used to represent one-digit decimal numbers. While four bits are enough to represent one-digit numbers, they are not enough to support the conversions of decimal to binary and back to decimal in all cases (but they are enough for most cases). Consider a power of 2 that is just under 9.5e21, for example, 2^{73} = 9.44e21. For this number, the three consecutive one-digit numbers near that special value and their round-to-nearest representations are:

```
  9e21    1e22    2e22
0xFp69  0x8p70  0x8p71
```

No problems so far; but when these representations are converted back to decimal, the values as three-digit numbers and the rounded one-digit numbers are:

```
8.85e21  9.44e21  1.89e22
   9e21     9e21     2e22
```

and we end up with two values the same. For this reason, four-bit mantissas are not enough to start with any one-digit decimal number, convert it to a binary floating-point representation, and then convert back to the same one-digit decimal number in all cases; and so p radix b digits are (just barely) not enough to allow any decimal numbers with $p \times \log_{10} b$ digits to do the round-trip conversion. p radix b digits are enough, however, for $(p-1) \times \log_{10} b$ digits in all cases.

The issues involved in performing correctly rounded decimal-to-binary and binary-to-decimal conversions are discussed mathematically by Gay.[24]

C90

The *Recommended practice* clauses are new in the C99 Standard.

C++

There is no such macro, or requirement specified in the C++ Standard.

Other Languages

The specification of the Java base conversions is poor.

Common Implementations

Experience with testing various translators shows that the majority don't, at the time of this publication, implement this Recommended Practice. The extent to which vendors will improve their implementations is unknown.

There is a publicly available set of tests for testing binary to decimal conversions.[66]

Coding Guidelines

A Recommended Practice shall not be relied on to be followed by an implementation.

---

380 EXAMPLE 1 The following describes an artificial floating-point representation that meets the minimum requirements of this International Standard, and the appropriate values in a **<float.h>** header for type **float**:

EXAMPLE minimum floating-point representation

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
FLT_RADIX                  16
FLT_MANT_DIG                6
FLT_EPSILON        9.53674316E-07F
FLT_DIG                     6
FLT_MIN_EXP               -31
FLT_MIN            2.93873588E-39F
FLT_MIN_10_EXP            -38
FLT_MAX_EXP               +32
FLT_MAX            3.40282347E+38F
FLT_MAX_10_EXP            +38
```

Commentary

Note that this example has a FLT_RADIX of 16, not 2.

---

381 EXAMPLE 2 The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in IEC 60559,[20] and the appropriate values in a **<float.h>** header for types **float** and **double**:

EXAMPLE IEC 60559 floating-point

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                  2
DECIMAL_DIG               17
FLT_MANT_DIG              24
FLT_EPSILON        1.19209290E-07F // decimal constant
FLT_EPSILON              0X1P-23F // hex constant
FLT_DIG                    6
FLT_MIN_EXP             -125
FLT_MIN            1.17549435E-38F // decimal constant
FLT_MIN                0X1P-126F // hex constant
FLT_MIN_10_EXP            -37
FLT_MAX_EXP              +128
FLT_MAX            3.40282347E+38F // decimal constant
```

```
FLT_MAX           0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                +38

DBL_MANT_DIG                  53
DBL_EPSILON 2.2204460492503131E-16 // decimal constant
DBL_EPSILON               0X1P-52 // hex constant
DBL_DIG                       15
DBL_MIN_EXP                -1021
DBL_MIN    2.2250738585072014E-308 // decimal constant
DBL_MIN                  0X1P-1022 // hex constant
DBL_MIN_10_EXP                -307
DBL_MAX_EXP                +1024
DBL_MAX    1.7976931348623157E+308 // decimal constant
DBL_MAX     0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP                +308
```

If a type wider than **double** were supported, then **DECIMAL_DIG** would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then **DECIMAL_DIG** would be 21.

Commentary

The values given here are important in that they are the most likely values to be provided by a conforming implementation using IEC 60559, which is what the majority of modern implementations use. These values correspond to the IEC 60559 single- and double-precision formats. This standard also defines extended single and extended double formats, which contain more bits in the significand and greater range in the exponent.

Note that this example gives the decimal and hexadecimal floating-constant representation for some of the macro definitions. A real header will only contain one of these definitions.

C90

The C90 wording referred to the ANSI/IEEE-754–1985 standard.

---

footnote 20

20) The floating-point model in that standard sums powers of b from zero, so the values of the exponent limits are one less than shown here. 382

Commentary

Fortran counts from 1, not 0 and the much of the contents of **<float.h>**, in C90, came from Fortran.

Forward references: conditional inclusion (6.10.1), complex arithmetic **<complex.h>** (7.3), extended multibyte and wide character utilities **<wchar.h>** (7.24), floating-point environment **<fenv.h>** (7.6), general utilities **<stdlib.h>** (7.20), input/output **<stdio.h>** (7.19), mathematics **<math.h>** (7.12). 383

References

1. T. Aamodt and P. Chow. Numerical error minimizing floating-point to fixed-point ANSI C compilation. In 1^{st} *Workshop on Media Processors and Digital Signal Processing*, Nov. 1999.
2. C. Álvarez, J. Corbal, E. Salamí, and M. Valero. On the potential of tolerant region reuse for multimedia applications. In *Proceedings of the* 15^{th} *international conference on Supercomputing*, pages 218–228. ACM Press, Apr. 2001.
3. Analog Devices, Inc. *ADSP-21065L SHARC DSP Technical Reference*. Analog Devices, Inc, 2.0 edition, July 2003.
4. Apple Computer. *Inside Macintosh: PowerPC Numerics*. Addison–Wesley, 1994.
5. R. L. Ashenhurst and N. Metropolis. Unnormalized floating point arithmetic. *Journal of the ACM*, 6(3):415–428, July 1959.
6. AT&T Document Management. *WE DSP32 Digital Signal Processor Information Manual*, 1988.
7. B. E. Bliss. Instrumentation of FORTRAN programs for automatic roundoff error analysis and performance evaluation. Thesis (m.s.), University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1990.
8. R. P. Brent. On the precision attainable with various floating-point number systems. *IEEE Transactions on Computers*, C-22(6):601–607, June 1973.
9. R. Carter. Y-MP floating point and Cholesky factorization. Technical Report RND-90-007, NASA Ames Research Center, 1990.
10. CDC. *6600 Central Processor. Volume II Functional Units*. Control Data Corporation, publication number 60239700 edition, 1966.
11. D. Chiriaev and G. W. Walster. Interval arithmetic specification. Technical report, Marquette University, May 1998.
12. W. J. Cody Jr. Static and dynamic numerical characteristics of floating-point arithmetic. *IEEE Transactions on Computers*, C-22(6):598–601, June 1973.
13. J. N. Coleman and E. I. Chester. A 32-bit logarithmic arithmetic unit and its performance compared to floating-point. In *Proceedings of the* 14^{th} *Symposium on Computer Arithmetic*, pages 142–151, 1999.
14. J. N. Coleman, C. I. Softley, J. Kadlec, R. Matoušek, Z. Pohl, and A. Heřmáek. The european logarithmic microprocessor - a QR RLS application. Technical Report No. 2038, Institute of Information Theory and Automation, Czech Republic, Dec. 2001.
15. D. A. Connors, Y. Yamada, and W. mei W. Hwu. A software-oriented floating-point format for automotive control systems. In *Workshop on Compiler and Architecture Support for Embedded Computing Systems (CASES'98)*, 1998.
16. D. E. Corporation. *VAX11 780 Architecture Handbook*. Digital Equipment Corporation, 1977.
17. M. Cowlishaw. General decimal arithmetic specification. Technical Report Version 1.30, IBM UK Laboratories, June 2003.
18. M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In 16^{th} *IEEE Symposium on Computer Arithmetic (ARITH-16'03)*, pages 104–111, June 2003.
19. J. D. Darcy and D. Gay. FLECKmarks measuring floating point performance using a FullL IEEE Compliant arithmetic BenchmarK. Technical report, U.C. Berkeley, Aug. 1996.
20. Data General Corporation. *Programmer's Reference Manual: Nova Line Computers*. Data General Corporation, 2 edition, Sept. 1975.
21. A. M. Devices. *3DNow! Technology Manual*. Advanced Micro Devices, Inc, 2000.
22. S. A. Figueroa del Cid. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, New York University, Jan. 2000.
23. A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In M. Glesner, P. Zipf, and M. Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 523–533. Springer-Verlag, Apr. 2002.
24. D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, Murray Hill, NJ, USA, Nov. 1990.
25. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, Mar. 1991.
26. J. Hauser. Softfloat-2b. www.jhauser.us/arithmetic/SoftFloat.html, 2002.
27. J. R. Hauser. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems*, 18(2):139–174, Mar. 1996.
28. Hewlett-Packard. *PA-RISC 2.0*. Hewlett-Packard, 2.0 edition, 1995.
29. Y. Hida, X. S. Li, and D. H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, 2000.
30. T. P. Hill. The first-digit phenomenon. *American Scientist*, 86:358–363, July-Aug. 1998.
31. INMOS Limited. *Transputer Reference Manual*. Prentice–Hall, 1988.
32. Intel. *Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*. Intel Corporation, 1.1 edition, Jan. 1999.
33. Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel, Inc, 2000.
34. Intel. *Intel XScale Microarchitecture Programmers Reference Manual*. Intel, Inc, 2001.
35. Intel, Inc. *Intel IA-64 Architecture Software Developer's Manual*, 2000. Instruction Set Reference.
36. ISO. *Programming languages, their environments and system software interfaces —Extensions for the programming language C to support embedded processors*. ISO, 2004.
37. W. Kahan. IEEE standard 754 for binary floating-point arithmetic. Lecture Notes in progress, May 1995.
38. W. Kahan. How Java's floating-point hurts everyone. In *ACM 1998 Workshop on Java for High-Performance Computing*, 1998.
39. W. Kahan. The improbability of PROBABILISTIC ERROR ANALYSIS for numerical computation. University of California at Berkeley, 1998.

40. W. Kahan. Miscalculating area and angles of a needle-like triangle. Lecture Notes for Introductory Numerical Analysis Classes, Mar. 2000.
41. R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2(1):95–112, 1996.
42. N. kuan Tsao. On the distribution of significant digits and roundoff errors. *Communications of the ACM*, 17(5):269–271, May 1974.
43. D. J. Kuck, D. S. Parker Jr., and A. H. Sameh. Analysis of rounding methods in floating-point arithmetic. *IEEE Transactions on Computers*, C-26(7):643–650, July 1977.
44. K.-I. Kum, J. Kang, and W. Sung. A floating-point to fixed-point C converter for fixed-point digital signal processors. In *Proceedings of the Second SUIF Compiler Workshop*, Aug. 1997.
45. O. Lawlor, H. Govind, I. Dooley, M. Breitenfeld, and L. Kale. Performance degradation in the presence of subnormal floating-point values. In *Proceedings of the International Workshop on Operating System Interference in High Performance Applications*, page ???, Sept. 2005.
46. M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Communications of the ACM*, 15(11):949–951, 1972.
47. D. W. Matula. In and out conversions. *Communications of the ACM*, 11:47–50, 1968.
48. Motorola, Inc. *DSP563CCC Motorola DSP56300 Family Optimizing C Compiler User's Manual*. Motorola, Inc, Austin, TX, USA, 19??
49. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
50. Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.
51. J.-M. Muller. On the definition of ulp (x). Technical Report No. 5504, Unité de recherche INRIA Rhône-Alpes, Feb. 2005.
52. K. C. Ng and FP group of SunPro. Argument reduction for huge arguments: Good to the last bit. Work in progress, Mar. 1992.
53. Numerical C Extensions Group. Float-point C extensions. Technical Report X3J11.1/93-028, American National Standards Institute, Aug. 1993.
54. S. F. Oberman. Design issues in high performance floating point arithmetic units. Technical Report CSL-TR-96-711, Stanford University, Dec. 1996.
55. A. R. Omondi. *Computer Arithmetic Systems: Algorithms, architecture, and implementation*. Prentice Hall, Inc, 1994.
56. D. M. Priest. *On Properties of Floating-Point Arithmetic: Numerical Stability and the Cost of Accurate Computations*. PhD thesis, University of California at Berkeley, Nov. 1992.
57. P. L. Richman. Floating-point number representations: base choice versus exponent range. Technical Report CS-TR-67-64, Stanford University, Department of Computer Science, Apr. 1967.
58. N. L. Schryer. A test of a computer's floating-point arithmetic unit. Technical Report Computing Science Technical Report No. 89, Bell Laboratories, Murray Hill, NJ, USA, Feb. 1981.
59. E. M. Schwarz and C. A. Krygowski. The S/390 G5 floating point unit. *IBM Journal of Research and Development*, 43(5/6):707–721, Sept.-Nov. 1999.
60. R. L. Sites and R. L. Witek. *Alpha AXP architecture reference manual*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, second edition, 1995.
61. G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. In *Proceeding of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.
62. D. W. Sweeney. An analysis of floating-point addition. *IBM Systems Journal*, 4(1):31–42, 1965.
63. Texas Instruments. *TMS320C3x User's Guide*. Texas Instruments, Inc, spru031e, revision l edition, July 1997.
64. Y. F. Tong, R. A. Rutenbar, and D. F. Nagle. Minimizing floating-point power dissipation via bit-width reduction. In *Proceedings of the 1998 International Symposium on Computer Architecture Power Driven Microarchitecture Workshop*, June 1998.
65. F. Tydeman. Private email from Fred. www.tybor.com, 2002.
66. F. Tydeman. Sample C99+FPCE tests. www.tybor.com, 2005.
67. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.
68. Unisys Corporation. *C Compiler Programming Reference Manual Volume 1: C Language and Library*. Unisys Corporation, 7831 0422■006, release level 8R1A edition, 2001.
69. B. Verdonk, A. Cuyt, and D. Verschaeren. A precision and range independent tool for testing floating-point arithmetic I: basic operations, square root and remainder. *ACM Transactions on Mathematical Software*, 27(1):92–118, 2001.
70. W. H. Ware, S. N. Alexander, P. Armer, M. M. Astrahan, T. Biers, H. H. Goode, H. D. Huskey, and M. Rebinoff. Soviet computer technology – 1959. *Communications of the ACM*, 3(3):131–166, 1960.
71. H. Yokoo. Overflow/underflow-free floating-point number representations with self-delimiting variable-length exponent field. *IEEE Transactions on Computers*, 41(8):1033–1039, Aug. 1992.