NAME

ieee - IEEE standard 754 for floating-point arithmetic

DESCRIPTION

The IEEE Standard 754 for Binary Floating-Point Arithmetic defines representations of floating-point numbers and abstract properties of arithmetic operations relating to precision, rounding, and exceptional cases, as described below.

IEEE STANDARD 754 Floating-Point Arithmetic

Radix: Binary.

Overflow and underflow:

Overflow goes by default to a signed infinity. Underflow is *gradual*.

Zero is represented ambiguously as +0 or -0.

Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x-x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and **copysign**(*x, +-0*). In particular, comparison (x > y, x >= y, etc.) cannot be affected by the sign of zero; but if finite x = y then infinity = 1/(x-y) != -1/(y-x) = -infinity.

Infinity is signed.

It persists when added to itself or to any finite number. Its sign transforms correctly through multiplication and division, and (finite)/+-infinity = +-0 (nonzero)/0 = +-infinity. But infinity-infinity, infinity*0 and infinity/infinity are, like 0/0 and sqrt(-3), invalid operations that produce NaN. ...

Reserved operands (NaNs):

An NaN is (*N*ot *a N*umber). Some NaNs, called Signaling NaNs, trap any floating-point operation performed upon them; they are used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet NaNs; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x != x then x is NaN; every other predicate (x > y, x = y, x < y, ...) is FALSE if NaN is involved.

Rounding:

Every algebraic operation (+, -, *, /, <sqrt>) is rounded by default to within half an *ulp*, and when the rounding error is exactly half an *ulp* then the rounded value's least significant bit is zero. (An *ulp* is one *U*nit in the *L*ast *P*lace.) This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, we find (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite that both the quotients and the products have been rounded. Only rounding like IEEE 754 can do that. But no single kind of rounding can be proved best for every circumstance, so

IEEE 754 provides rounding towards zero or towards +infinity or towards -infinity at the programmer's option.

Exceptions:

IEEE 754 recognizes five kinds of floating-point exceptions, listed below in declining order of probable importance.

Exception	Default Result
Invalid Operation	NaN, or FALSE
Overflow	+-infinity
Divide by Zero	+-infinity
Underflow	Gradual Underflow
Inexact	Rounded value

NOTE: An Exception is not an Error unless handled badly. What makes a class of exceptions exceptional is that no single default response can be satisfactory in every instance. On the other hand, if a default response will serve most instances satisfactorily, the unsatisfactory instances cannot justify aborting computation every time the exception occurs.

Data Formats

Single-precision:

Type name: *float*

Wordsize: 32 bits.

Precision: 24 significant bits, roughly like 7 significant decimals.

If x and x' are consecutive positive single-precision numbers (they differ by 1 *ulp*), then 5.9e-08 < 0.5**24 < (x'-x)/x <= 0.5**23 < 1.2e-07.

Range: Overflow threshold = 2.0**128 = 3.4e38
Underflow threshold = 0.5**126 = 1.2e-38

Underflowed results round to the nearest integer multiple of 0.5**149 = 1.4e-45.

Double-precision:

Type name: *double* (On some architectures, *long double* is the same as *double*)

Wordsize: 64 bits.

Precision: 53 significant bits, roughly like 16 significant decimals.

If x and x' are consecutive positive double-precision numbers (they differ by 1 *ulp*), then
1.1e-16 < 0.5**53 < (x'-x)/x <= 0.5**52 < 2.3e-16.

Range: Overflow threshold = 2.0**1024 = 1.8e308
Underflow threshold = 0.5**1022 = 2.2e-308

Underflowed results round to the nearest integer multiple of
0.5**1074 = 4.9e-324.

Extended-precision:

Type name: *long double* (when supported by the hardware)

Wordsize: 96 bits.

Precision: 64 significant bits, roughly like 19 significant decimals.

If x and x' are consecutive positive extended-precision numbers (they differ by 1 *ulp*), then
1.0e-19 < 0.5**63 < (x'-x)/x <= 0.5**62 < 2.2e-19.

Range: Overflow threshold = 2.0**16384 = 1.2e4932
Underflow threshold = 0.5**16382 = 3.4e-4932

Underflowed results round to the nearest integer multiple of
0.5**16445 = 5.7e-4953.

Quad-extended-precision:

Type name: *long double* (when supported by the hardware)

Wordsize: 128 bits.

Precision: 113 significant bits, roughly like 34 significant decimals.

If x and x' are consecutive positive quad-extended-precision numbers (they differ by 1 *ulp*), then
9.6e-35 < 0.5**113 < (x'-x)/x <= 0.5**112 < 2.0e-34.

Range: Overflow threshold = 2.0**16384 = 1.2e4932
Underflow threshold = 0.5**16382 = 3.4e-4932

Underflowed results round to the nearest integer multiple of
0.5**16494 = 6.5e-4966.

Additional Information Regarding Exceptions

For each kind of floating-point exception, IEEE 754 provides a Flag that is raised each time its exception is signaled, and stays raised until the program resets it. Programs may also test, save and restore a flag. Thus, IEEE 754 provides three ways by which programs may cope with exceptions for which the default result might be unsatisfactory:

1. Test for a condition that might cause an exception later, and branch to avoid the exception.

2. Test a flag to see whether an exception has occurred since the program last reset its flag.

3. Test a result to see whether it is a value that only an exception could have produced.

   CAUTION: The only reliable ways to discover whether Underflow has occurred are to test whether products or quotients lie closer to zero than the underflow threshold, or to test the Underflow flag. (Sums and differences cannot underflow in IEEE 754; if x != y then x-y is correct to full precision and certainly nonzero regardless of how tiny it may be.) Products and quotients that underflow gradually can lose accuracy gradually without vanishing, so comparing them with zero (as one might on a VAX) will not reveal the loss. Fortunately, if a gradually underflowed value is destined to be added to something bigger than the underflow threshold, as is almost always the case, digits lost to gradual underflow will not be missed because they would have been rounded off anyway. So gradual underflows are usually *provably* ignorable. The same cannot be said of underflows flushed to 0.

At the option of an implementor conforming to IEEE 754, other ways to cope with exceptions may be provided:

1. ABORT. This mechanism classifies an exception in advance as an incident to be handled by means traditionally associated with error-handling statements like "ON ERROR GO TO ...". Different languages offer different forms of this statement, but most share the following characteristics:

   - No means is provided to substitute a value for the offending operation’s result and resume computation from what may be the middle of an expression. An exceptional result is abandoned.

   - In a subprogram that lacks an error-handling statement, an exception causes the subprogram to abort within whatever program called it, and so on back up the chain of calling subprograms until an error-handling statement is encountered or the whole task is aborted and memory is

dumped.

2. STOP. This mechanism, requiring an interactive debugging environment, is more for the programmer than the program. It classifies an exception in advance as a symptom of a programmer's error; the exception suspends execution as near as it can to the offending operation so that the programmer can look around to see how it happened. Quite often the first several exceptions turn out to be quite unexceptionable, so the programmer ought ideally to be able to resume execution after each one as if execution had not been stopped.

3. ... Other ways lie beyond the scope of this document.

Ideally, each elementary function should act as if it were indivisible, or atomic, in the sense that ...

1. No exception should be signaled that is not deserved by the data supplied to that function.

2. Any exception signaled should be identified with that function rather than with one of its subroutines.

3. The internal behavior of an atomic function should not be disrupted when a calling program changes from one to another of the five or so ways of handling exceptions listed above, although the definition of the function may be correlated intentionally with exception handling.

The functions in **libm** are only approximately atomic. They signal no inappropriate exception except possibly ...

Over/Underflow
when a result, if properly computed, might have lain barely within range, and

Inexact in **cabs**(), **cbrt**(), **hypot**(), **log10**() and **pow**()
when it happens to be exact, thanks to fortuitous cancellation of errors.

Otherwise, ...

Invalid Operation is signaled only when
any result but NaN would probably be misleading.

Overflow is signaled only when
the exact result would be finite but beyond the overflow threshold.

Divide-by-Zero is signaled only when
a function takes exactly infinite values at finite operands.

Underflow is signaled only when
the exact result would be nonzero but tinier than the underflow threshold.

Inexact is signaled only when
greater range or precision would be needed to represent the exact result.

SEE ALSO

fenv(3), ieee_test(3), math(3)

An explanation of IEEE 754 and its proposed extension p854 was published in the IEEE magazine MICRO in August 1984 under the title "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic" by W. J. Cody et al. The manuals for Pascal, C and BASIC on the Apple Macintosh document the features of IEEE 754 pretty well. Articles in the IEEE magazine COMPUTER vol. 14 no. 3 (Mar. 1981), and in the ACM SIGNUM Newsletter Special Issue of Oct. 1979, may be helpful although they pertain to superseded drafts of the standard.

STANDARDS

IEEE Std 754-1985