The Red Hat newlib C Math Library

`libm` 1.18.0
December 2008

Steve Chamberlain
Roland Pesch
Red Hat Support
Jeff Johnston

Red Hat Support
sac@cygnus.com
pesch@cygnus.com
jjohnstn@redhat.com

1 Mathematical Functions (‘math.h’)

This chapter groups a wide variety of mathematical functions. The corresponding definitions and declarations are in ‘math.h’. Two definitions from ‘math.h’ are of particular interest.

1. The representation of infinity as a `double` is defined as `HUGE_VAL`; this number is returned on overflow by many functions. The macro `HUGE_VALF` is a corresponding value for `float`.
2. The structure `exception` is used when you write customized error handlers for the mathematical functions. You can customize error handling for most of these functions by defining your own version of `matherr`; see the section on `matherr` for details.

Since the error handling code calls `fputs`, the mathematical subroutines require stubs or minimal implementations for the same list of OS subroutines as `fputs`: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. See Section “System Calls” in *The Red Hat newlib C Library*, for a discussion and for sample minimal implementations of these support subroutines.

Alternative declarations of the mathematical functions, which exploit specific machine capabilities to operate faster—but generally have less error checking and may reflect additional limitations on some machines—are available when you include ‘fastmath.h’ instead of ‘math.h’.

1.1 Error Handling

There are four different versions of the math library routines: IEEE, POSIX, X/Open, or SVID. The version may be selected at runtime by setting the global variable `_LIB_VERSION`, defined in '`math.h`'. It may be set to one of the following constants defined in '`math.h`': `_IEEE_`, `_POSIX_`, `_XOPEN_`, or `_SVID_`. The `_LIB_VERSION` variable is not specific to any thread, and changing it will affect all threads.

The versions of the library differ only in how errors are handled.

In IEEE mode, the `matherr` function is never called, no warning messages are printed, and `errno` is never set.

In POSIX mode, `errno` is set correctly, but the `matherr` function is never called and no warning messages are printed.

In X/Open mode, `errno` is set correctly, and `matherr` is called, but warning message are not printed.

In SVID mode, functions which overflow return 3.40282346638528860e+38, the maximum single-precision floating-point value, rather than infinity. Also, `errno` is set correctly, `matherr` is called, and, if `matherr` returns 0, warning messages are printed for some errors. For example, by default '`log(-1.0)`' writes this message on standard error output:

```
log: DOMAIN error
```

The library is set to X/Open mode by default.

The aforementioned error reporting is the supported Newlib libm error handling method. However, the majority of the functions are written so as to produce the floating-point exceptions (e.g. "invalid", "divide-by-zero") as required by the C and POSIX standards, for floating-point implementations that support them. Newlib does not provide the floating-point exception access routines defined in the standards for fenv.h, though, which is why they are considered unsupported. It is mentioned in case you have separately-provided access routines so that you are aware that they can be caused.

1.2 Standards Compliance And Portability

Most of the individual function descriptions describe the standards to which each function complies. However, these descriptions are mostly out of date, having been written before C99 was released. One of these days we'll get around to updating the rest of them. (If you'd like to help, please let us know.)

"C99" refers to ISO/IEC 9899:1999, "Programming languages–C". "POSIX" refers to IEEE Standard 1003.1. POSIX® is a registered trademark of The IEEE.

1.3 acos, acosf—arc cosine

Synopsis

```
#include <math.h>
double acos(double x);
float acosf(float x);
```

Description

`acos` computes the inverse cosine (arc cosine) of the input value. Arguments to `acos` must be in the range -1 to 1.

`acosf` is identical to `acos`, except that it performs its calculations on `floats`.

Returns

`acos` and `acosf` return values in radians, in the range of `0` to π.

If x is not between -1 and 1, the returned value is NaN (not a number) the global variable `errno` is set to `EDOM`, and a `DOMAIN error` message is sent as standard error output.

You can modify error handling for these functions using `matherr`.

1.4 `acosh`, `acoshf`—inverse hyperbolic cosine

Synopsis

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
```

Description

`acosh` calculates the inverse hyperbolic cosine of x. `acosh` is defined as

$$ln\left(x + \sqrt{x^2 - 1}\right)$$

x must be a number greater than or equal to 1.

`acoshf` is identical, other than taking and returning floats.

Returns

`acosh` and `acoshf` return the calculated value. If x less than 1, the return value is NaN and `errno` is set to `EDOM`.

You can change the error-handling behavior with the non-ANSI `matherr` function.

Portability

Neither `acosh` nor `acoshf` are ANSI C. They are not recommended for portable programs.

1.5 asin, asinf—arc sine

Synopsis

```
#include <math.h>
double asin(double x);
float asinf(float x);
```

Description

`asin` computes the inverse sine (arc sine) of the argument x. Arguments to `asin` must be in the range -1 to 1.

`asinf` is identical to `asin`, other than taking and returning floats.

You can modify error handling for these routines using `matherr`.

Returns

`asin` returns values in radians, in the range of $-\pi/2$ to $\pi/2$.

If x is not in the range -1 to 1, `asin` and `asinf` return NaN (not a number), set the global variable `errno` to `EDOM`, and issue a `DOMAIN error` message.

You can change this error treatment using `matherr`.

1.6 `asinh`, `asinhf`—inverse hyperbolic sine

Synopsis

```
#include <math.h>
double asinh(double x);
float asinhf(float x);
```

Description
`asinh` calculates the inverse hyperbolic sine of x. `asinh` is defined as

$$sign(x) \times ln\Big(|x| + \sqrt{1 + x^2}\Big)$$

`asinhf` is identical, other than taking and returning floats.

Returns
`asinh` and `asinhf` return the calculated value.

Portability
Neither `asinh` nor `asinhf` are ANSI C.

1.7 atan, atanf—arc tangent

Synopsis

```
#include <math.h>
double atan(double x);
float atanf(float x);
```

Description

`atan` computes the inverse tangent (arc tangent) of the input value.
`atanf` is identical to `atan`, save that it operates on `floats`.

Returns
`atan` returns a value in radians, in the range of $-\pi/2$ to $\pi/2$.

Portability
`atan` is ANSI C. `atanf` is an extension.

1.8 atan2, atan2f—arc tangent of y/x

Synopsis

```
#include <math.h>
double atan2(double y,double x);
float atan2f(float y,float x);
```

Description

`atan2` computes the inverse tangent (arc tangent) of y/x. `atan2` produces the correct result even for angles near $\pi/2$ or $-\pi/2$ (that is, when x is near 0).

`atan2f` is identical to `atan2`, save that it takes and returns `float`.

Returns

`atan2` and `atan2f` return a value in radians, in the range of $-\pi$ to π.

You can modify error handling for these functions using `matherr`.

Portability

`atan2` is ANSI C. `atan2f` is an extension.

1.9 atanh, atanhf—inverse hyperbolic tangent

Synopsis

```
#include <math.h>
double atanh(double x);
float atanhf(float x);
```

Description

`atanh` calculates the inverse hyperbolic tangent of x.

`atanhf` is identical, other than taking and returning `float` values.

Returns

`atanh` and `atanhf` return the calculated value.

If $|x|$ is greater than 1, the global `errno` is set to `EDOM` and the result is a NaN. A `DOMAIN error` is reported.

If $|x|$ is 1, the global `errno` is set to `EDOM`; and the result is infinity with the same sign as `x`. A `SING error` is reported.

You can modify the error handling for these routines using `matherr`.

Portability

Neither `atanh` nor `atanhf` are ANSI C.

1.10 `jN`, `jNf`, `yN`, `yNf`—Bessel functions

Synopsis

```
#include <math.h>
double j0(double x);
float j0f(float x);
double j1(double x);
float j1f(float x);
double jn(int n, double x);
float jnf(int n, float x);
double y0(double x);
float y0f(float x);
double y1(double x);
float y1f(float x);
double yn(int n, double x);
float ynf(int n, float x);
```

Description

The Bessel functions are a family of functions that solve the differential equation

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - p^2)y = 0$$

These functions have many applications in engineering and physics.

`jn` calculates the Bessel function of the first kind of order n. `j0` and `j1` are special cases for order 0 and order 1 respectively.

Similarly, `yn` calculates the Bessel function of the second kind of order n, and `y0` and `y1` are special cases for order 0 and 1.

`jnf`, `j0f`, `j1f`, `ynf`, `y0f`, and `y1f` perform the same calculations, but on `float` rather than `double` values.

Returns

The value of each Bessel function at x is returned.

Portability

None of the Bessel functions are in ANSI C.

1.11 cbrt, cbrtf—cube root

Synopsis

```
#include <math.h>
double cbrt(double x);
float  cbrtf(float x);
```

Description
`cbrt` computes the cube root of the argument.

Returns
The cube root is returned.

Portability
`cbrt` is in System V release 4. `cbrtf` is an extension.

1.12 `copysign`, `copysignf`—sign of *y*, magnitude of *x*

Synopsis

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

Description

`copysign` constructs a number with the magnitude (absolute value) of its first argument, *x*, and the sign of its second argument, *y*.

`copysignf` does the same thing; the two functions differ only in the type of their arguments and result.

Returns

`copysign` returns a `double` with the magnitude of *x* and the sign of *y*. `copysignf` returns a `float` with the magnitude of *x* and the sign of *y*.

Portability

`copysign` is not required by either ANSI C or the System V Interface Definition (Issue 2).

1.13 cosh, coshf—hyperbolic cosine

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf(float x)
```

Description

`cosh` computes the hyperbolic cosine of the argument x. `cosh(x)` is defined as

$$\frac{(e^x + e^{-x})}{2}$$

Angles are specified in radians. `coshf` is identical, save that it takes and returns `float`.

Returns

The computed value is returned. When the correct value would create an overflow, `cosh` returns the value `HUGE_VAL` with the appropriate sign, and the global value `errno` is set to `ERANGE`.

You can modify error handling for these functions using the function `matherr`.

Portability

`cosh` is ANSI. `coshf` is an extension.

1.14 `erf`, `erff`, `erfc`, `erfcf`—error function

Synopsis

```
#include <math.h>
double erf(double x);
float erff(float x);
double erfc(double x);
float erfcf(float x);
```

Description

`erf` calculates an approximation to the "error function", which estimates the probability that an observation will fall within x standard deviations of the mean (assuming a normal distribution). The error function is defined as

$$\frac{2}{\sqrt{\pi}} \times \int_0^x e^{-t^2} dt$$

`erfc` calculates the complementary probability; that is, `erfc(x)` is `1 - erf(x)`. `erfc` is computed directly, so that you can use it to avoid the loss of precision that would result from subtracting large probabilities (on large x) from 1.

`erff` and `erfcf` differ from `erf` and `erfc` only in the argument and result types.

Returns

For positive arguments, `erf` and all its variants return a probability—a number between 0 and 1.

Portability

None of the variants of `erf` are ANSI C.

1.15 exp, expf—exponential

Synopsis

```
#include <math.h>
double exp(double x);
float expf(float x);
```

Description

exp and expf calculate the exponential of x, that is, e^x (where e is the base of the natural system of logarithms, approximately 2.71828).

You can use the (non-ANSI) function matherr to specify error handling for these functions.

Returns

On success, exp and expf return the calculated value. If the result underflows, the returned value is 0. If the result overflows, the returned value is HUGE_VAL. In either case, errno is set to ERANGE.

Portability

exp is ANSI C. expf is an extension.

1.16 exp2, exp2f—exponential, base 2

Synopsis

```
#include <math.h>
double exp2(double x);
float exp2f(float x);
```

Description

exp2 and exp2f calculate 2 ^ x, that is, 2^x

You can use the (non-ANSI) function matherr to specify error handling for these functions.

Returns

On success, exp2 and exp2f return the calculated value. If the result underflows, the returned value is 0. If the result overflows, the returned value is HUGE_VAL. In either case, errno is set to ERANGE.

Portability

ANSI C, POSIX.

1.17 expm1, expm1f—exponential minus 1

Synopsis

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
```

Description

`expm1` and `expm1f` calculate the exponential of x and subtract 1, that is, $e^x - 1$ (where e is the base of the natural system of logarithms, approximately 2.71828). The result is accurate even for small values of x, where using `exp(x)-1` would lose many significant digits.

Returns

e raised to the power x, minus 1.

Portability

Neither `expm1` nor `expm1f` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.18 `fabs`, `fabsf`—absolute value (magnitude)

Synopsis

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
```

Description

`fabs` and `fabsf` calculate $|x|$, the absolute value (magnitude) of the argument x, by direct manipulation of the bit representation of x.

Returns

The calculated value is returned. No errors are detected.

Portability

`fabs` is ANSI. `fabsf` is an extension.

1.19 fdim, fdimf—positive difference

Synopsis

```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
```

Description

The fdim functions determine the positive difference between their arguments, returning:

x - y if x > y, or

+0 if $x \leq y$, or

NAN if either argument is NAN.

A range error may occur.

Returns

The fdim functions return the positive difference value.

Portability

ANSI C, POSIX.

1.20 `floor`, `floorf`, `ceil`, `ceilf`—floor and ceiling

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
double ceil(double x);
float ceilf(float x);
```

Description

`floor` and `floorf` find $\lfloor x \rfloor$, the nearest integer less than or equal to x. `ceil` and `ceilf` find $\lceil x \rceil$, the nearest integer greater than or equal to x.

Returns

`floor` and `ceil` return the integer result as a double. `floorf` and `ceilf` return the integer result as a float.

Portability

`floor` and `ceil` are ANSI. `floorf` and `ceilf` are extensions.

1.21 fma, fmaf–floating multiply add

Synopsis

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
```

Description

The `fma` functions compute (*x* * *y*) + *z*, rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS. That is, they are supposed to do this: see below.

Returns

The `fma` functions return (*x* * *y*) + *z*, rounded as one ternary operation.

Bugs

This implementation does not provide the function that it should, purely returning "(*x* * *y*) + *z*;" with no attempt at all to provide the simulated infinite precision intermediates which are required. DO NOT USE THEM.

If double has enough more precision than float, then `fmaf` should provide the expected numeric results, as it does use double for the calculation. But since this is not the case for all platforms, this manual cannot determine if it is so for your case.

Portability

ANSI C, POSIX.

1.22 `fmax`, `fmaxf`—maximum

Synopsis

```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
```

Description
The `fmax` functions determine the maximum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the `fmax` functions choose the numeric value.

Returns
The `fmax` functions return the maximum numeric value of their arguments.

Portability
ANSI C, POSIX.

1.23 fmin, fminf—minimum

Synopsis

```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
```

Description
The fmin functions determine the minimum numeric value of their arguments. NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the fmin functions choose the numeric value.

Returns
The fmin functions return the minimum numeric value of their arguments.

Portability
ANSI C, POSIX.

1.24 `fmod`, `fmodf`—floating-point remainder (modulo)

Synopsis

```
#include <math.h>
double fmod(double x, double y)
float fmodf(float x, float y)
```

Description

The `fmod` and `fmodf` functions compute the floating-point remainder of x/y (x modulo y).

Returns

The `fmod` function returns the value $x - i \times y$, for the largest integer i such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

`fmod(x,0)` returns NaN, and sets `errno` to `EDOM`.

You can modify error treatment for these functions using `matherr`.

Portability

`fmod` is ANSI C. `fmodf` is an extension.

1.25 frexp, frexpf—split floating-point number

Synopsis

```
#include <math.h>
double frexp(double val, int *exp);
float frexpf(float val, int *exp);
```

Description

All nonzero, normal numbers can be described as *m* * 2**p*. `frexp` represents the double *val* as a mantissa *m* and a power of two *p*. The resulting mantissa will always be greater than or equal to `0.5`, and less than `1.0` (as long as *val* is nonzero). The power of two will be stored in `*`*exp*.

m and *p* are calculated so that $val = m \times 2^p$.

`frexpf` is identical, other than taking and returning floats rather than doubles.

Returns

`frexp` returns the mantissa *m*. If *val* is `0`, infinity, or Nan, `frexp` will set `*`*exp* to `0` and return *val*.

Portability

`frexp` is ANSI. `frexpf` is an extension.

1.26 gamma, gammaf, lgamma, lgammaf, gamma_r, gammaf_r, lgamma_r, lgammaf_r, tgamma, and tgammaf–logarithmic and plain gamma functions

Synopsis

```
#include <math.h>
double gamma(double x);
float gammaf(float x);
double lgamma(double x);
float lgammaf(float x);
double gamma_r(double x, int *signgamp);
float gammaf_r(float x, int *signgamp);
double lgamma_r(double x, int *signgamp);
float lgammaf_r(float x, int *signgamp);
double tgamma(double x);
float tgammaf(float x);
```

Description

`gamma` calculates $ln(\Gamma(x))$, the natural logarithm of the gamma function of x. The gamma function (`exp(gamma(x))`) is a generalization of factorial, and retains the property that $\Gamma(N) \equiv N \times \Gamma(N-1)$. Accordingly, the results of the gamma function itself grow very quickly. `gamma` is defined as $ln(\Gamma(x))$ rather than simply $\Gamma(x)$ to extend the useful range of results representable.

The sign of the result is returned in the global variable `signgam`, which is declared in math.h.

`gammaf` performs the same calculation as `gamma`, but uses and returns `float` values.

`lgamma` and `lgammaf` are alternate names for `gamma` and `gammaf`. The use of `lgamma` instead of `gamma` is a reminder that these functions compute the log of the gamma function, rather than the gamma function itself.

The functions `gamma_r`, `gammaf_r`, `lgamma_r`, and `lgammaf_r` are just like `gamma`, `gammaf`, `lgamma`, and `lgammaf`, respectively, but take an additional argument. This additional argument is a pointer to an integer. This additional argument is used to return the sign of the result, and the global variable `signgam` is not used. These functions may be used for reentrant calls (but they will still set the global variable `errno` if an error occurs).

`tgamma` and `tgammaf` are the "true gamma" functions, returning $\Gamma(x)$, the gamma function of x–without a logarithm. (They are apparently so named because of the prior existence of the old, poorly-named `gamma` functions which returned the log of gamma up through BSD 4.2.)

Returns

Normally, the computed result is returned.

When x is a nonpositive integer, `gamma` returns `HUGE_VAL` and `errno` is set to `EDOM`. If the result overflows, `gamma` returns `HUGE_VAL` and `errno` is set to `ERANGE`.

You can modify this error treatment using `matherr`.

Portability

Neither `gamma` nor `gammaf` is ANSI C. It is better not to use either of these; use `lgamma` or

`tgamma` instead.
`lgamma`, `lgammaf`, `tgamma`, and `tgammaf` are nominally C standard in terms of the base return values, although the `matherr` error-handling is not standard, nor is the *signgam* global for `lgamma`.

1.27 `hypot`, `hypotf`—distance from origin

Synopsis

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
```

Description

`hypot` calculates the Euclidean distance $\sqrt{x^2 + y^2}$ between the origin (0,0) and a point represented by the Cartesian coordinates (x,y). `hypotf` differs only in the type of its arguments and result.

Returns

Normally, the distance value is returned. On overflow, `hypot` returns `HUGE_VAL` and sets `errno` to `ERANGE`.

You can change the error treatment with `matherr`.

Portability

`hypot` and `hypotf` are not ANSI C.

1.28 ilogb, ilogbf—get exponent of floating-point number

Synopsis

```
#include <math.h>
int ilogb(double val);
int ilogbf(float val);
```

Description

All nonzero, normal numbers can be described as *m* * 2**p*. `ilogb` and `ilogbf` examine the argument *val*, and return *p*. The functions `frexp` and `frexpf` are similar to `ilogb` and `ilogbf`, but also return *m*.

Returns

`ilogb` and `ilogbf` return the power of two used to form the floating-point argument. If *val* is `0`, they return `FP_ILOGB0`. If *val* is infinite, they return `INT_MAX`. If *val* is NaN, they return `FP_ILOGBNAN`. (`FP_ILOGB0` and `FP_ILOGBNAN` are defined in math.h, but in turn are defined as INT_MIN or INT_MAX from limits.h. The value of FP_ILOGB0 may be either INT_MIN or -INT_MAX. The value of FP_ILOGBNAN may be either INT_MAX or INT_MIN.)

Portability
C99, POSIX

1.29 `infinity`, `infinityf`–representation of infinity

Synopsis

```
#include <math.h>
double infinity(void);
float infinityf(void);
```

Description

`infinity` and `infinityf` return the special number IEEE infinity in double- and single-precision arithmetic respectively.

Portability

`infinity` and `infinityf` are neither standard C nor POSIX. C and POSIX require macros HUGE_VAL and HUGE_VALF to be defined in math.h, which Newlib defines to be infinities corresponding to these archaic infinity() and infinityf() functions in floating-point implementations which do have infinities.

1.30 isgreater, isgreaterequal, isless, islessequal, islessgreater, and isunordered–comparison macros

Synopsis

```
#include <math.h>
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

Description

`isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater`, and `isunordered` are macros defined for use in comparing floating-point numbers without raising any floating-point exceptions.

The relational operators (i.e. `<`, `>`, `<=`, and `>=`) support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships–less, greater, and equal–is true. Relational operators may raise the "invalid" floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true (i.e., if one or both of the arguments a NaN, the relationship is called unordered). The specified macros are quiet (non floating-point exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the "invalid" floating-point exception. In the synopses shown, "real-floating" indicates that the argument is an expression of real floating type.

Please note that saying that the macros do not raise floating-point exceptions, it is referring to the function that they are performing. It is certainly possible to give them an expression which causes an exception. For example:

`NaN < 1.0` causes an "invalid" exception,

`isless(NaN, 1.0)`
: does not, and

`isless(NaN*0., 1.0)`
: causes an exception due to the "NaN*0.", but not from the resultant reduced comparison of isless(NaN, 1.0).

Returns

No floating-point exceptions are raised for any of the macros.
The `isgreater` macro returns the value of (x) > (y).
The `isgreaterequal` macro returns the value of (x) >= (y).
The `isless` macro returns the value of (x) < (y).
The `islessequal` macro returns the value of (x) <= (y).
The `islessgreater` macro returns the value of (x) < (y) || (x) > (y).
The `isunordered` macro returns 1 if either of its arguments is NaN and 0 otherwise.

Portability
C99, POSIX.

1.31 fpclassify, isfinite, isinf, isnan, and isnormal–floating-point classification macros; finite, finitef, isinf, isinff, isnan, isnanf–test for exceptional numbers

Synopsis

```
[C99 standard macros:]
#include <math.h>
int fpclassify(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);

[Archaic SUSv2 functions:]
#include <ieeefp.h>
int isnan(double arg);
int isinf(double arg);
int finite(double arg);
int isnanf(float arg);
int isinff(float arg);
int finitef(float arg);
```

Description

`fpclassify`, `isfinite`, `isinf`, `isnan`, and `isnormal` are macros defined for use in classifying floating-point numbers. This is a help because of special "values" like NaN and infinities. In the synopses shown, "real-floating" indicates that the argument is an expression of real floating type. These function-like macros are C99 and POSIX-compliant, and should be used instead of the now-archaic SUSv2 functions.

The `fpclassify` macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument. The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument:

`FP_INFINITE`
: *x* is either plus or minus infinity;

`FP_NAN`
: *x* is "Not A Number" (plus or minus);

`FP_NORMAL`
: *x* is a "normal" number (i.e. is none of the other special forms);

`FP_SUBNORMAL`
: *x* is too small be stored as a regular normalized number (i.e. loss of precision is likely); or

`FP_ZERO`
: *x* is 0 (either plus or minus).

The "`is`" set of macros provide a useful set of shorthand ways for classifying floating-point numbers, providing the following equivalent relations:

`isfinite(x)`
: returns non-zero if x is finite. (It is equivalent to (`fpclassify`(x) != FP_INFINITE && `fpclassify`(x) != FP_NAN).)

`isinf(x)`
: returns non-zero if x is infinite. (It is equivalent to (`fpclassify`(x) == FP_INFINITE).)

`isnan(x)`
: returns non-zero if x is NaN. (It is equivalent to (`fpclassify`(x) == FP_NAN).)

`isnormal(x)`
: returns non-zero if x is normal. (It is equivalent to (`fpclassify`(x) == FP_NORMAL).)

The archaic SUSv2 functions provide information on the floating-point argument supplied.

There are five major number formats ("exponent" referring to the biased exponent in the binary-encoded number):

`zero`
: A number which contains all zero bits, excluding the sign bit.

`subnormal`
: A number with a zero exponent but a nonzero fraction.

`normal`
: A number with an exponent and a fraction.

`infinity`
: A number with an all 1's exponent and a zero fraction.

`NAN`
: A number with an all 1's exponent and a nonzero fraction.

`isnan` returns 1 if the argument is a nan. `isinf` returns 1 if the argument is infinity. `finite` returns 1 if the argument is zero, subnormal or normal. The `isnanf`, `isinff` and `finitef` functions perform the same operations as their `isnan`, `isinf` and `finite` counterparts, but on single-precision floating-point numbers.

It should be noted that the C99 standard dictates that `isnan` and `isinf` are macros that operate on multiple types of floating-point. The SUSv2 standard declares `isnan` as a function taking double. Newlib has decided to declare them both as macros in math.h and as functions in ieeefp.h to maintain backward compatibility.

Returns

The fpclassify macro returns the value corresponding to the appropriate FP_ macro.
The isfinite macro returns nonzero if x is finite, else 0.
The isinf macro returns nonzero if x is infinite, else 0.
The isnan macro returns nonzero if x is an NaN, else 0.
The isnormal macro returns nonzero if x has a normal value, else 0.

Portability

math.h macros are C99, POSIX.

ieeefp.h funtions are outdated and should be avoided.

1.32 ldexp, ldexpf—load exponent

Synopsis

```
#include <math.h>
double ldexp(double val, int exp);
float ldexpf(float val, int exp);
```

Description

`ldexp` calculates the value $val \times 2^{exp}$. `ldexpf` is identical, save that it takes and returns `float` rather than `double` values.

Returns

`ldexp` returns the calculated value.

Underflow and overflow both set `errno` to `ERANGE`. On underflow, `ldexp` and `ldexpf` return 0.0. On overflow, `ldexp` returns plus or minus `HUGE_VAL`.

Portability

`ldexp` is ANSI. `ldexpf` is an extension.

1.33 `log`, `logf`—natural logarithms

Synopsis

```
#include <math.h>
double log(double x);
float logf(float x);
```

Description

Return the natural logarithm of x, that is, its logarithm base e (where e is the base of the natural system of logarithms, 2.71828...). `log` and `logf` are identical save for the return and argument types.

You can use the (non-ANSI) function `matherr` to specify error handling for these functions.

Returns

Normally, returns the calculated value. When x is zero, the returned value is `-HUGE_VAL` and `errno` is set to `ERANGE`. When x is negative, the returned value is NaN (not a number) and `errno` is set to `EDOM`. You can control the error behavior via `matherr`.

Portability

`log` is ANSI. `logf` is an extension.

1.34 log10, log10f—base 10 logarithms

Synopsis

```
#include <math.h>
double log10(double x);
float log10f(float x);
```

Description

`log10` returns the base 10 logarithm of x. It is implemented as `log(x) / log(10)`.

`log10f` is identical, save that it takes and returns `float` values.

Returns

`log10` and `log10f` return the calculated value.

See the description of `log` for information on errors.

Portability

`log10` is ANSI C. `log10f` is an extension.

1.35 `log1p`, `log1pf`—log of `1 + x`

Synopsis

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
```

Description

`log1p` calculates $ln(1 + x)$, the natural logarithm of `1+x`. You can use `log1p` rather than '`log(1+x)`' for greater precision when x is very small.

`log1pf` calculates the same thing, but accepts and returns `float` values rather than `double`.

Returns

`log1p` returns a `double`, the natural log of `1+x`. `log1pf` returns a `float`, the natural log of `1+x`.

Portability

Neither `log1p` nor `log1pf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.36 log2, log2f—base 2 logarithm

Synopsis

```
#include <math.h>
double log2(double x);
float log2f(float x);
```

Description

The log2 functions compute the base-2 logarithm of x. A domain error occurs if the argument is less than zero. A range error occurs if the argument is zero.

The Newlib implementations are not full, intrinisic calculations, but rather are derivatives based on log. (Accuracy might be slightly off from a direct calculation.) In addition to functions, they are also implemented as macros defined in math.h:

```
#define log2(x) (log (x) / _M_LOG2_E)
#define log2f(x) (logf (x) / (float) _M_LOG2_E)
```

To use the functions instead, just undefine the macros first.

You can use the (non-ANSI) function matherr to specify error handling for these functions, indirectly through the respective log function.

Returns

The log2 functions return $log_2(x)$ on success. When x is zero, the returned value is -HUGE_VAL and errno is set to ERANGE. When x is negative, the returned value is NaN (not a number) and errno is set to EDOM. You can control the error behavior via matherr.

Portability

C99, POSIX, System V Interface Definition (Issue 6).

1.37 `logb`, `logbf`—get exponent of floating-point number

Synopsis

```
#include <math.h>
double logb(double x);
float logbf(float x);
```

Description

The `logb` functions extract the exponent of x, as a signed integer value in floating-point format. If x is subnormal it is treated as though it were normalized; thus, for positive finite x, $1 \leq (x \cdot FLT\_RADIX^{-logb(x)}) < FLT\_RADIX$. A domain error may occur if the argument is zero. In this floating-point implementation, FLT_RADIX is 2. Which also means that for finite x, `logb`(x) = `floor`(`log2`(`fabs`(x))).

All nonzero, normal numbers can be described as $m \cdot 2^p$, where $1.0 \leq m < 2.0$. The `logb` functions examine the argument x, and return p. The `frexp` functions are similar to the `logb` functions, but returning m adjusted to the interval [.5, 1) or 0, and p+1.

Returns

When x is:
+inf or -inf, +inf is returned;
NaN, NaN is returned;
0, -inf is returned, and the divide-by-zero exception is raised;
otherwise, the `logb` functions return the signed exponent of x.

Portability

ANSI C, POSIX

See Also

frexp, ilogb

1.38 lrint, lrintf, llrint, llrintf—round to integer

Synopsis

```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long long int llrint(double x);
long long int llrintf(float x);
```

Description

The `lrint` and `llrint` functions round their argument to the nearest integer value, using the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of x is too large. The "inexact" floating-point exception is raised in implementations that support it when the result differs in value from the argument (i.e., when a fraction actually has been truncated).

Returns

x rounded to an integral value, using the current rounding direction.

See Also

`lround`

Portability

ANSI C, POSIX

1.39 `lround`, `lroundf`, `llround`, `llroundf`–round to integer, to nearest

Synopsis

```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long long int llround(double x);
long long int llroundf(float x);
```

Description
The `lround` and `llround` functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified (depending upon the floating-point implementation, not the library). A range error may occur if the magnitude of x is too large.

Returns
x rounded to an integral value as an integer.

See Also
See the `round` functions for the return being the same floating-point type as the argument. `lrint`, `llrint`.

Portability
ANSI C, POSIX

1.40 matherr—modifiable math error handler

Synopsis

```
#include <math.h>
int matherr(struct exception *e);
```

Description

`matherr` is called whenever a math library function generates an error. You can replace `matherr` by your own subroutine to customize error treatment. The customized `matherr` must return 0 if it fails to resolve the error, and non-zero if the error is resolved.

When `matherr` returns a nonzero value, no error message is printed and the value of `errno` is not modified. You can accomplish either or both of these things in your own `matherr` using the information passed in the structure `*e`.

This is the `exception` structure (defined in ‘`math.h`’):

```
struct exception {
        int type;
        char *name;
        double arg1, arg2, retval;
int err;
};
```

The members of the exception structure have the following meanings:

`type` The type of mathematical error that occured; macros encoding error types are also defined in ‘`math.h`’.

`name` a pointer to a null-terminated string holding the name of the math library function where the error occurred.

`arg1, arg2`
The arguments which caused the error.

`retval` The error return value (what the calling function will return).

`err` If set to be non-zero, this is the new value assigned to `errno`.

The error types defined in ‘`math.h`’ represent possible mathematical errors as follows:

`DOMAIN` An argument was not in the domain of the function; e.g. `log(-1.0)`.

`SING` The requested calculation would result in a singularity; e.g. `pow(0.0,-2.0)`

`OVERFLOW` A calculation would produce a result too large to represent; e.g. `exp(1000.0)`.

`UNDERFLOW`
A calculation would produce a result too small to represent; e.g. `exp(-1000.0)`.

`TLOSS` Total loss of precision. The result would have no significant digits; e.g. `sin(10e70)`.

`PLOSS` Partial loss of precision.

Returns

The library definition for `matherr` returns `0` in all cases.

You can change the calling function's result from a customized `matherr` by modifying `e->retval`, which propagates backs to the caller.

If `matherr` returns `0` (indicating that it was not able to resolve the error) the caller sets `errno` to an appropriate value, and prints an error message.

Portability
`matherr` is not ANSI C.

1.41 modf, modff—split fractional and integer parts

Synopsis

```
#include <math.h>
double modf(double val, double *ipart);
float modff(float val, float *ipart);
```

Description

`modf` splits the double *val* apart into an integer part and a fractional part, returning the fractional part and storing the integer part in `*ipart`. No rounding whatsoever is done; the sum of the integer and fractional parts is guaranteed to be exactly equal to *val*. That is, if *realpart* = modf(*val*, *&intpart*); then ‘`realpart+intpart`’ is the same as *val*. `modff` is identical, save that it takes and returns `float` rather than `double` values.

Returns

The fractional part is returned. Each result has the same sign as the supplied argument *val*.

Portability

`modf` is ANSI C. `modff` is an extension.

1.42 nan, nanf—representation of "Not a Number"

Synopsis

```
#include <math.h>
double nan(const char *);
float nanf(const char *);
```

Description

`nan` and `nanf` return an IEEE NaN (Not a Number) in double- and single-precision arithmetic respectively. The argument is currently disregarded.

1.43 nearbyint, nearbyintf—round to integer

Synopsis

```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
```

Description

The `nearbyint` functions round their argument to an integer value in floating-point format, using the current rounding direction and (supposedly) without raising the "inexact" floating-point exception. See the `rint` functions for the same function with the "inexact" floating-point exception being raised when appropriate.

Bugs

Newlib does not support the floating-point exception model, so that the floating-point exception control is not present and thereby what may be seen will be compiler and hardware dependent in this regard. The Newlib `nearbyint` functions are identical to the `rint` functions with respect to the floating-point exception behavior, and will cause the "inexact" exception to be raised for most targets.

Returns

x rounded to an integral value, using the current rounding direction.

Portability

ANSI C, POSIX

See Also

`rint`, `round`

1.44 nextafter, nextafterf—get next number

Synopsis

```
#include <math.h>
double nextafter(double val, double dir);
float nextafterf(float val, float dir);
```

Description

`nextafter` returns the double-precision floating-point number closest to *val* in the direction toward *dir*. `nextafterf` performs the same operation in single precision. For example, `nextafter(0.0,1.0)` returns the smallest positive number which is representable in double precision.

Returns

Returns the next closest number to *val* in the direction toward *dir*.

Portability

Neither `nextafter` nor `nextafterf` is required by ANSI C or by the System V Interface Definition (Issue 2).

1.45 pow, powf—x to the power y

Synopsis

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
```

Description

pow and powf calculate x raised to the exponent y. (That is, x^y.)

Returns

On success, pow and powf return the value calculated.

When the argument values would produce overflow, pow returns HUGE_VAL and set errno to ERANGE. If the argument x passed to pow or powf is a negative noninteger, and y is also not an integer, then errno is set to EDOM. If x and y are both 0, then pow and powf return 1.

You can modify error handling for these functions using matherr.

Portability

pow is ANSI C. powf is an extension.

1.46 `remainder`, `remainderf`—round and remainder

Synopsis

```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
```

Description

`remainder` and `remainderf` find the remainder of x/y; this value is in the range -$y/2$ .. +$y/2$.

Returns

`remainder` returns the integer result as a double.

Portability

`remainder` is a System V release 4. `remainderf` is an extension.

1.47 remquo, remquof–remainder and part of quotient

Synopsis

```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
```

Description

The remquo functions compute the same remainder as the remainder functions; this value is in the range -$y/2$ ... +$y/2$. In the object pointed to by quo they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2**n to the magnitude of the integral quotient of x/y. (That is, quo is given the n lsbs of the quotient, not counting the sign.) This implementation uses n=31 if int is 32 bits or more, otherwise, n is 1 less than the width of int.

For example:

```
remquo(-29.0, 3.0, &quo)
```

returns -1.0 and sets *quo*=10, and

```
remquo(-98307.0, 3.0, &quo)
```

returns -0.0 and sets *quo*=-32769, although for 16-bit int, *quo*=-1. In the latter case, the actual quotient of -(32769=0x8001) is reduced to -1 because of the 15-bit limitation for the quotient.

Returns

When either argument is NaN, NaN is returned. If y is 0 or x is infinite (and neither is NaN), a domain error occurs (i.e. the "invalid" floating point exception is raised or errno is set to EDOM), and NaN is returned. Otherwise, the remquo functions return x REM y.

Bugs

IEEE754-2008 calls for remquo(subnormal, inf) to cause the "underflow" floating-point exception. This implementation does not.

Portability

C99, POSIX.

1.48 `rint`, `rintf`–round to integer

Synopsis

```
#include <math.h>
double rint(double x);
float rintf(float x);
```

Description
The `rint` functions round their argument to an integer value in floating-point format, using the current rounding direction. They raise the "inexact" floating-point exception if the result differs in value from the argument. See the `nearbyint` functions for the same function with the "inexact" floating-point exception never being raised. Newlib does not directly support floating-point exceptions. The `rint` functions are written so that the "inexact" exception is raised in hardware implementations that support it, even though Newlib does not provide access.

Returns
x rounded to an integral value, using the current rounding direction.

Portability
ANSI C, POSIX

See Also
`nearbyint`, `round`

1.49 round, roundf—round to integer, to nearest

Synopsis

```
#include <math.h>
double round(double x);
float roundf(float x);
```

Description

The round functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction. (While the "inexact" floating-point exception behavior is unspecified by the C standard, the round functions are written so that "inexact" is not raised if the result does not equal the argument, which behavior is as recommended by IEEE 754 for its related functions.)

Returns

x rounded to an integral value.

Portability

ANSI C, POSIX

See Also

nearbyint, rint

1.50 scalbn, scalbnf, scalbln, scalblnf–scale by power of FLT_RADIX (=2)

Synopsis

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
```

Description

The `scalbn` and `scalbln` functions compute $x \cdot FLT\_RADIX^n$. efficiently. The result is computed by manipulating the exponent, rather than by actually performing an exponentiation or multiplication. In this floating-point implementation FLT_RADIX=2, which makes the `scalbn` functions equivalent to the `ldexp` functions.

Returns

x times 2 to the power n. A range error may occur.

Portability

ANSI C, POSIX

See Also

`ldexp`

1.51 signbit–Does floating-point number have negative sign?

Synopsis

```
#include <math.h>
int signbit(real-floating x);
```

Description

The `signbit` macro determines whether the sign of its argument value is negative. The macro reports the sign of all values, including infinities, zeros, and NaNs. If zero is unsigned, it is treated as positive. As shown in the synopsis, the argument is "real-floating," meaning that any of the real floating-point types (float, double, etc.) may be given to it.

Note that because of the possibilities of signed 0 and NaNs, the expression "$x < 0.0$" does not give the same result as `signbit` in all cases.

Returns

The `signbit` macro returns a nonzero value if and only if the sign of its argument value is negative.

Portability

C99, POSIX.

1.52 `sin`, `sinf`, `cos`, `cosf`—sine or cosine

Synopsis

```
#include <math.h>
double sin(double x);
float  sinf(float x);
double cos(double x);
float cosf(float x);
```

Description

`sin` and `cos` compute (respectively) the sine and cosine of the argument x. Angles are specified in radians.

`sinf` and `cosf` are identical, save that they take and return `float` values.

Returns

The sine or cosine of x is returned.

Portability

`sin` and `cos` are ANSI C. `sinf` and `cosf` are extensions.

1.53 sinh, sinhf—hyperbolic sine

Synopsis

```
#include <math.h>
double sinh(double x);
float  sinhf(float x);
```

Description

`sinh` computes the hyperbolic sine of the argument x. Angles are specified in radians. `sinh`(x) is defined as

$$\frac{e^x - e^{-x}}{2}$$

`sinhf` is identical, save that it takes and returns `float` values.

Returns

The hyperbolic sine of x is returned.

When the correct result is too large to be representable (an overflow), `sinh` returns `HUGE_VAL` with the appropriate sign, and sets the global value `errno` to `ERANGE`.

You can modify error handling for these functions with `matherr`.

Portability

`sinh` is ANSI C. `sinhf` is an extension.

1.54 `sqrt`, `sqrtf`—positive square root

Synopsis

```
#include <math.h>
double sqrt(double x);
float  sqrtf(float x);
```

Description

`sqrt` computes the positive square root of the argument. You can modify error handling for this function with `matherr`.

Returns

On success, the square root is returned. If x is real and positive, then the result is positive. If x is real and negative, the global value `errno` is set to `EDOM` (domain error).

Portability

`sqrt` is ANSI C. `sqrtf` is an extension.

1.55 tan, tanf—tangent

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

Description

`tan` computes the tangent of the argument x. Angles are specified in radians.

`tanf` is identical, save that it takes and returns `float` values.

Returns

The tangent of x is returned.

Portability

`tan` is ANSI. `tanf` is an extension.

1.56 `tanh`, `tanhf`—hyperbolic tangent

Synopsis

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

Description

`tanh` computes the hyperbolic tangent of the argument x. Angles are specified in radians.

`tanh(x)` is defined as

```
sinh(x)/cosh(x)
```

`tanhf` is identical, save that it takes and returns `float` values.

Returns

The hyperbolic tangent of x is returned.

Portability

`tanh` is ANSI C. `tanhf` is an extension.

1.57 trunc, truncf—round to integer, towards zero

Synopsis

```
#include <math.h>
double trunc(double x);
float truncf(float x);
```

Description

The `trunc` functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument, regardless of the current rounding direction. (While the "inexact" floating-point exception behavior is unspecified by the C standard, the `trunc` functions are written so that "inexact" is not raised if the result does not equal the argument, which behavior is as recommended by IEEE 754 for its related functions.)

Returns

x truncated to an integral value.

Portability

ANSI C, POSIX

2 Reentrancy Properties of `libm`

When a libm function detects an exceptional case, `errno` may be set, the `matherr` function may be called, and a error message may be written to the standard error stream. This behavior may not be reentrant.

With reentrant C libraries like the Red Hat newlib C library, `errno` is a macro which expands to the per-thread error value. This makes it thread safe.

When the user provides his own `matherr` function it must be reentrant for the math library as a whole to be reentrant.

In normal debugged programs, there are usually no math subroutine errors—and therefore no assignments to `errno` and no `matherr` calls; in that situation, the math functions behave reentrantly.

3 The long double function support of libm

Currently, the full set of long double math functions is only provided on platforms where long double equals double. For such platforms, the long double math functions are implemented as calls to the double versions.

Library Index

M

N

O

P

R

S

T

Y

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents