Contents

Chapter 1

Mathematical Complex Functions (`complex.h`)

This chapter groups the complex mathematical functions. The corresponding definitions and declarations are in `complex.h`. Functions and documentations are taken from NetBSD.

1.1 cabs

cabs, cabsf, cabsl — complex absolute-value

Synopsis

`#include <complex.h>`

double **cabs**(double complex z);
float **cabsf**(float complex z);
long double **cabsl**(long double complex z);

Description

These functions compute compute the complex absolute value (also called norm, modulus, or magnitude) of `z`.

`cabsf` is identical to `cabs`, except that it performs its calculations on `float complex`.

`cabsl` is identical to `cabs`, except that it performs its calculations on `long double complex`.

Returns

The cabs* functions return the complex absolute value.

Portability

`cabs`, `cabsf` and `cabsl` are ISO C99

1.2 cacos

cacos, cacosf — complex arc cosine

Synopsis

```
#include <complex.h>
```

double complex **cacos**(double complex z);
float complex **cacosf**(float complex z);

Description

These functions compute the complex arc cosine of `z`, with branch cuts outside the interval [-1, +1] along the real axis.

`cacosf` is identical to `cacos`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, pi] along the real axis.

Portability

`cacos` and `cacosf` are ISO C99

1.3   cacosh

cacosh, cacoshf — complex arc hyperbolic cosine

Synopsis

```
#include <complex.h>
```

double complex **cacosh**(double complex z);
float complex **cacoshf**(float complex z);

Description

These functions compute the complex arc hyperbolic cosine of `z`, with a branch cut at values less than 1 along the real axis.

`cacoshf` is identical to `cacosh`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [-i * pi, +i * pi] along the imaginary axis.

Portability

`cacosh` and `cacoshf` are ISO C99

1.4 carg

carg, cargf — argument (phase angle)

Synopsis

```
#include <complex.h>
```

double **carg**(double complex z);
float **cargf**(float complex z);

Description

These functions compute the argument (also called phase angle) of z, with a branch cut along the negative real axis.

cargf is identical to carg, except that it performs its calculations on floats complex.

Returns

The carg functions return the value of the argument in the interval [-pi, +pi]

Portability

carg and cargf are ISO C99

1.5 casin

casin, casinf — complex arc sine

Synopsis

```
#include <complex.h>
```

double complex **casin**(double complex z);
float complex **casinf**(float complex z);

Description

These functions compute the complex arc sine of z, with branch cuts outside the interval [-1, +1] along the real axis.

casinf is identical to casin, except that it performs its calculations on floats complex.

Returns

These functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

Portability

casin and casinf are ISO C99

1.6   casinh

casinh, casinhf — complex arc hyperbolic sine

Synopsis

```
#include <complex.h>
```

double complex **casinh**(double complex z);
float complex **casinhf**(float complex z);

Description

These functions compute the complex arc hyperbolic sine of z, with branch cuts outside the interval [-i, +i] along the imaginary axis.

casinhf is identical to casinh, except that it performs its calculations on floats complex.

Returns

These functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i*p/2, +i*p/2] along the imaginary axis.

Portability

casinh and casinhf are ISO C99

1.7   catan

catan, catanf — complex arc tangent

Synopsis

```
#include <complex.h>
```

double complex **catan**(double complex z);
float complex **catanf**(float complex z);

Description

These functions compute the complex arc tangent of z, with branch cuts outside the interval [-i, +i] along the imaginary axis.

catanf is identical to catan, except that it performs its calculations on floats complex.

Returns

These functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

Portability

`catan` and `catanf` are ISO C99

1.8 catanh

catanh, catanhf — complex arc hyperbolic tangent

Synopsis

```
#include <complex.h>
```

double complex **catanh**(double complex z);
float complex **catanhf**(float complex z);

Description

These functions compute the complex arc hyperbolic tan of `z`, with branch cuts outside the interval [-1, +1] along the real axis.

`catanhf` is identical to `catanh`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i*p/2, +i*p/2] along the imaginary axis.

Portability

`catanh` and `catanhf` are ISO C99

1.9 ccos

ccos, ccosf — complex cosine

Synopsis

```
#include <complex.h>
```

double complex **ccos**(double complex z);
float complex **ccosf**(float complex z);

Description

These functions compute the complex cosine of `z`.

`ccosf` is identical to `ccos`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex cosine value.

Portability

`ccos` and `ccosf` are ISO C99

1.10 ccosh

ccosh, ccoshf — complex hyperbolic cosine

Synopsis

```
#include <complex.h>
```

double complex **ccosh**(double complex z);
float complex **ccoshf**(float complex z);

Description

These functions compute the complex hyperbolic cosine of `z`.

`ccoshf` is identical to `ccosh`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex hyperbolic cosine value.

Portability

`ccosh` and `ccoshf` are ISO C99

1.11 cexp

cexp, cexpf — complex base-e exponential

Synopsis

```
#include <complex.h>
```

double complex **cexp**(double complex z);
float complex **cexpf**(float complex z);

Description

These functions compute the complex base-`e` exponential of `z`.

`cexpf` is identical to `cexp`, except that it performs its calculations on `floats complex`.

Returns

The cexp functions return the complex base-`e` exponential value.

Portability

`cexp` and `cexpf` are ISO C99

1.12 cimag

cimag, cimagf, cimagl — imaginary part

Synopsis

```
#include <complex.h>
```

double **cimag**(double complex z);
float **cimagf**(float complex z);
long double **cimagl**(long double complex z);

Description

These functions compute the imaginary part of `z`.

`cimagf` is identical to `cimag`, except that it performs its calculations on `float complex`.

`cimagl` is identical to `cimag`, except that it performs its calculations on `long double complex`.

Returns

The cimag* functions return the imaginary part value (as a real).

Portability

`cimag`, `cimagf` and `cimagl` are ISO C99

1.13 clog

clog, clogf — complex base-e logarithm

Synopsis

```
#include <complex.h>
```

double complex **clog**(double complex z);
float complex **clogf**(float complex z);

Description

These functions compute the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

`clogf` is identical to `clog`, except that it performs its calculations on `floats complex`.

Returns

The clog functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [-i*pi , +i*pi] along the imaginary axis.

Portability

`clog` and `clogf` are ISO C99

1.14 conj

conj, conjf — complex conjugate

Synopsis

```
#include <complex.h>
```

double complex **conj**(double complex z);
float complex **conjf**(float complex z);

Description

These functions compute the complex conjugate of `z`, by reversing the sign of its imaginary part.

`conjf` is identical to `conj`, except that it performs its calculations on `floats complex`.

Returns

The conj functions return the complex conjugate value.

Portability

`conj` and `conjf` are ISO C99

1.15 cpow

cpow, cpowf — complex power

Synopsis

```
#include <complex.h>
```

double complex **cpow**(double complex x, double complex y);
float complex **cpowf**(float complex x, float complex y);

Description

The cpow functions compute the complex power function xˆy power, with a branch cut for the first parameter along the negative real axis.

cpowf is identical to cpow, except that it performs its calculations on `floats complex`.

Returns

The cpow functions return the complex power function value.

Portability

cpow and cpowf are ISO C99

1.16 cproj

cproj, cprojf — Riemann sphere projection

Synopsis

```
#include <complex.h>
```

double complex **cproj**(double complex z);
float complex **cprojf**(float complex z);

Description

These functions compute a projection of z onto the Riemann sphere: z projects to z except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If z has an infinite part, then cproj(z) is equivalent to

INFINITY + I * copysign(0.0, cimag(z))

cprojf is identical to cproj, except that it performs its calculations on `floats complex`.

Returns

The cproj functions return the value of the projection onto the Riemann sphere.

Portability

cproj and cprojf are ISO C99

1.17 creal

creal, crealf, creall — real part

Synopsis

```
#include <complex.h>
```

double **creal**(double complex z);
float **crealf**(float complex z);
double long **creall**(long double complex z);

Description

These functions compute the real part of `z`.

`crealf` is identical to `creal`, except that it performs its calculations on `float complex`.

`creall` is identical to `creal`, except that it performs its calculations on `long double complex`.

Returns

The creal* functions return the real part value.

Portability

`creal`, `crealf` and `creall` are ISO C99

1.18   csin

csin, csinf — complex sine

Synopsis

```
#include <complex.h>
```

double complex **csin**(double complex z);
float complex **csinf**(float complex z);

Description

These functions compute the complex sine of `z`.

`csinf` is identical to `csin`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex sine value.

Portability

`csin` and `csinf` are ISO C99

1.19 csinh

csinh, csinhf — complex hyperbolic sine

Synopsis

```
#include <complex.h>
```

double complex **csinh**(double complex z);
float complex **csinhf**(float complex z);

Description

These functions compute the complex hyperbolic sine of z.

ccoshf is identical to ccosh, except that it performs its calculations on floats complex.

Returns

These functions return the complex hyperbolic sine value.

Portability

csinh and csinhf are ISO C99

1.20 csqrt

csqrt, csqrtf — complex square root

Synopsis

```
#include <complex.h>
```

double complex **csqrt**(double complex z);
float complex **csqrtf**(float complex z);

Description

These functions compute the complex square root of z, with a branch cut along the negative real axis.

csqrtf is identical to csqrt, except that it performs its calculations on floats complex.

Returns

The csqrt functions return the complex square root value, in the range of the right halfplane (including the imaginary axis).

Portability

csqrt and csqrtf are ISO C99

1.21 ctan

ctan, ctanf — complex tangent

Synopsis

```
#include <complex.h>
```

double complex **ctan**(double complex z);
float complex **ctanf**(float complex z);

Description

These functions compute the complex tangent of `z`.

`ctanf` is identical to `ctan`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex tangent value.

Portability

`ctan` and `ctanf` are ISO C99

1.22 ctanh

ctanh, ctanf, ctanhf — complex hyperbolic tangent

Synopsis

```
#include <complex.h>
```

double complex **ctanh**(double complex z);
float complex **ctanhf**(float complex z);

Description

These functions compute the complex hyperbolic tangent of `z`.

`ctanhf` is identical to `ctanh`, except that it performs its calculations on `floats complex`.

Returns

These functions return the complex hyperbolic tangent value.

Portability

`ctanh` and `ctanhf` are ISO C99

Chapter 2

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`. @xref{syscalls,,System Calls, libc.info, 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`.

2.1 acos

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 pi.

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.

2.2   acosh

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

```
log(x + sqrt(x*x-1))
```

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.

2.3   asin

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.

2.4   asinh

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

```
 sgn(x)  *  log(abs(x)  +  sqrt(1+x*x))
```

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.

2.5   atan

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.

2.6   atan2

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 pi.

You can modify error handling for these functions using matherr.

Portability

atan2 is ANSI C. atan2f is an extension.

2.7 atanh

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.

2.8 jN

jN, jNf, yN, yNf, j0, j0f, j1, j1f, jn, jnf, y0, y0f, y1, y1f, 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

```
  2                   2    2
 x   y'' + xy' + (x  - p )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.

2.9 cbrt

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.

2.10 copysign

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).

2.11 cosh

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

```
 (exp(x) + exp(-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.

2.12 erf

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).

`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.

2.13 exp

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 raised to the power 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.

2.14   exp10

exp10, exp10f — exponential, base 10

Synopsis

```
#include <math.h>
```

double **exp10**(double x);
float **exp10f**(float x);

Description

`exp10` and `exp10f` calculate `10 ^ x`, that is, 10 raised to the power x.

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

Returns

On success, `exp10` and `exp10f` 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

`exp10` and `exp10f` are GNU extensions.

2.15   exp2

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 raised to the power 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.

2.16 expm1

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 raised to the power x minus 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).

2.17 fabs

fabs, fabsf — absolute value (magnitude)

Synopsis

```
#include <math.h>
```

double **fabs**(double x);
float **fabsf**(float x);

Description

`fabs` and `fabsf` calculate 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.

2.18 fdim

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 <= 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.

2.19 floor

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 the nearest integer less than or equal to `x`. `ceil` and `ceilf` find 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.

2.20 fma

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.

2.21   fmax

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.

2.22   fmin

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.

2.23   fmod

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*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.

2.24 frexp

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` is `m` times 2 to the power `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.

2.25 gamma

gamma, gammaf, lgamma, lgammaf, gamma_r, gammaf_r, lgamma_r, lgammaf_r, tgamma, 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 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 exp(gamma(N)) is equivalent to N*exp(gamma(N-1)). Accordingly, the results of the gamma function itself grow very quickly. gamma is defined as the natural log of the gamma function, rather than the gamma function itself, 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 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 math err error-handling is not standard, nor is the signgam global for lgamma.

2.26 hypot

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*x + y*y) 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.

2.27 ilogb

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

2.28 infinity

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.

2.29 isgreater

isgreater, isgreaterequal, isless, islessequal, islessgreater, 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.

2.30 fpclassify

fpclassify, isfinite, isinf, isnan, isnormal, finite, finitef, isinf, isinff, isnan, isnanf — floating-point classification macros, 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.

2.31 ldexp

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 to the power `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.

2.32 log

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.

2.33   log10

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.

2.34 log1p

log1p, log1pf — log of `1 + x`

Synopsis

```
#include <math.h>
```

double **log1p**(double x);
float **log1pf**(float x);

Description

`log1p` calculates 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).

2.35 log2

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_LN2)
#define log2f(x) (logf (x) / (float) _M_LN2)
```

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 base-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).

2.36 logb

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 <= (x * FLT_RADIX to the power (-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 * 2**p, where 1.0 <= 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

2.37 lrint

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

2.38 lround

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

2.39   matherr

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.

2.40 modf

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.

2.41 nan

nan, nanf — representation of ``Not a Number''

Synopsis

```
#include <math.h>
```

double **nan**(const char *unused);
float **nanf**(const char *unused);

Description

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

2.42 nearbyint

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

2.43 nextafter

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).

2.44 pow

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.

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.

2.45 pow10

pow10, pow10f — base 10 power functions

Synopsis

```
#include <math.h>
```

double **pow10**(double x);
float **pow10f**(float x);

Description

pow10 and pow10f calculate 10 ^ x, that is, 10 raised to the power x.

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

Returns

On success, pow10 and pow10f 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

pow10 and pow10f are GNU extensions.

2.46 remainder

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.

2.47   remquo

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.

2.48 rint

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`

2.49 round

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`

2.50 scalbn

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` times FLT_RADIX to the power `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`

2.51 signbit

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.

2.52 sin

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.

2.53 sinh

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

```
(exp(x) - exp(-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.

2.54 sqrt

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.

2.55 tan

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.

2.56 tanh

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.

2.57 trunc

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

Chapter 3

Index