# ChorusOS man pages section 3M: Mathematical Libraries

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3332
December 10, 1999

# Contents

# PREFACE

## Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man`(1) for more information about man pages in general.

| | |
|---|---|
| NAME | This section gives the names of the commands or functions documented, followed by a brief description of what they do. |
| SYNOPSIS | This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required. |

The following special characters are used in this section:

| | |
|---|---|
| [ ] | The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified. |
| . . . | Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ' "`filename` . . ." . |
| \| | Separator. Only one of the arguments separated by this character can be specified at time. |
| { } | Braces. The options and/or arguments enclosed within braces are |

|  |  |
|---|---|
|  | interdependent, such that everything enclosed must be treated as a unit. |
| FEATURES | This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured. |
| DESCRIPTION | This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES.. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE. |
| OPTIONS | This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied. |
| OPERANDS | This section lists the command operands and describes how they affect the actions of the command. |
| OUTPUT | This section describes the output - standard output, standard error, or output files - generated by the command. |
| RETURN VALUES | If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES. |
| ERRORS | On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code. |

| | |
|---|---|
| USAGE | This section is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:<br>Commands<br>Modifiers<br>Variables<br>Expressions<br>Input Grammar |
| EXAMPLES | This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%` or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections. |
| ENVIRONMENT VARIABLES | This section lists any environment variables that the command or function affects, followed by a brief description of the effect. |
| EXIT STATUS | This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values other than zero for various error conditions. |
| FILES | This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation. |
| SEE ALSO | This section lists references to other man pages, in-house documentation and outside publications. |
| DIAGNOSTICS | This section lists diagnostic messages with a brief explanation of the condition causing the error. |
| WARNINGS | This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics. |
| NOTES | This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here. |

## BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

# Mathematical Library

14

**NAME**

acos – arc cosine function

**SYNOPSIS**

#include <math.h>
double **acos**(double *x*);

**DESCRIPTION**

The *acos* function computes the principal value of the arc cosine of *x*. A domain error occurs for arguments not in the range [-1, +1]. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *acos* function returns the arc cosine in the range [0, Pi] radians. If | *x* | > 1, *errno* is set to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`sin`(3M), `cos`(3M), `tan`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `sinh`(3M), `cosh`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *acos* function conforms to `ANSI-C.`

**NAME**

acosh – inverse hyperbolic cosine function

**SYNOPSIS**

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

**DESCRIPTION**

The *acosh* function computes the inverse hyperbolic cosine of the real argument *x*. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *acosh* function returns the inverse hyperbolic cosine of *x*. If the argument is less than one, *acosh* sets *errno* to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`asinh`(3M), `atanh`(3M), `exp`(3M), `infnan`(3M), `math`(3M)

**NAME**

asin – arc sine function

**SYNOPSIS**

#include <math.h>
double **asin**(double *x*);

**DESCRIPTION**

The *asin* function computes the principal value of the arc sine of *x*. A domain error occurs for arguments not in the range [-1, +1]. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *asin* function returns the arc sine in the range [-Pi/2, +Pi/2] radians. If | *x* | > 1, *errno* is set to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `sinh`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *asin* function conforms to `ANSI-C.`

**NAME**

asinh – inverse hyperbolic sine function

**SYNOPSIS**

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

**DESCRIPTION**

The *asinh* function computes the inverse hyperbolic sine of the real argument *x*. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *asinh* function returns the inverse hyperbolic sine of *x*.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acosh`(3M), `atanh`(3M), `exp`(3M), `infnan`(3M), `math`(3M)

**NAME**

atan2 – arc tangent function of two variables

**SYNOPSIS**

#include <math.h>
double **atan2**(double *y*, double *x*);

**DESCRIPTION**

The atan2 function computes the principal value of the arc tangent of *y/x*, using the signs of both arguments to determine the quadrant of the return value.

**RETURN VALUES**

The atan2 function, if successful, returns the arc tangent of *y/x* in radians. If both *x* and *y* are zero, *errno* is set to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `sinh`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *atan2* function conforms to `ANSI-C.`

**NAME**

atan – arc tangent function of one variable

**SYNOPSIS**

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

**DESCRIPTION**

The *atan* function computes the principal value of the arc tangent of *x*. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *atan* function returns the arc tangent in radians.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `sinh`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *atan* function conforms to `ANSI-C.`

**NAME**

atanh – inverse hyperbolic tangent function

**SYNOPSIS**

#include <math.h>
double **atanh**(double *x*);

**DESCRIPTION**

The *atanh* function computes the inverse hyperbolic tangent of the real argument *x*. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *atanh* function returns the inverse hyperbolic tangent of *x*, if successful. If the argument has an absolute value greater than or equal to 1, *atanh* sets *errno* to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acosh`(3M), `asinh`(3M), `exp`(3M), `infnan`(3M), `math`(3M)

**NAME**

hypot, cabs – euclidean distance and complex absolute value functions

**SYNOPSIS**

#include <math.h>
double **hypot**(double *x*, double *y*);

double **cabs**(struct {double *x* ; double *y* ;} *z* ;);

**DESCRIPTION**

The *hypot* and *cabs* functions compute the square root of (x*x+y*y) in such a way that underflow will not occur, and overflow occurs only if the final result justifies it.

*hypot* (∞ , v) = *hypot* (v, ∞) = +Infinity

for all values of *v* , including **NaN** .

**ERRORS (due to Roundoff)**

Less than 0.97 **ulp** s. Consequently, *hypot(5.0, 12.0)* = 13.0 exactly; in general, hypot and cabs return an integer whenever an integer is expected.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `sqrt`(3M)

**NAME**

sqrt, cbrt – cube root and square root functions

**SYNOPSIS**

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

double sqrt(double x);
```

**DESCRIPTION**

The *cbrt* function computes the cube root of *x* .

The *sqrt* function computes the non-negative square root of x.

**RETURN VALUES**

The *cbrt* function returns the requested cube root. The *sqrt* function returns the requested square root unless an error occurs. An attempt to take the *sqrt* of a negative value of *x* causes an error; in this event, *errno* is set to EDOM and a system-dependent notification is performed.

**ERRORS (due to Roundoff)**

The *cbrt* function is accurate to within 0.7 `ulps` .

On a machine that conforms to `IEEE` 754 sqrt is correctly rounded in accordance with the rounding mode in force; the error is less than half a `ulp` in the default mode (round–to–nearest). A `ulp` is one Unit in the Last Place carried.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**STANDARDS**

The *sqrt* function conforms to `ANSI-C.`

**NAME**

ceil – smallest integral value not less than x

**SYNOPSIS**

```
#include <math.h>
double ceil(double x);
```

**DESCRIPTION**

The *ceil* function computes the smallest integral value not less than *x*.

**RETURN VALUES**

The *ceil* function returns the smallest integral value expressed as a double.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`floor`(3M), `rint`(3M), `ieee`(3M), `math`(3M) `abs`(3STDC) `fabs`(3STDC)

**STANDARDS**

The *ceil* function conforms to `ANSI-C.`

## NAME

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

## SYNOPSIS

#include <math.h>
double **copysign**(double *x*, double *y*);

double **drem**(double *x*, double *y*);

int **finite**(double *x*);

double **logb**(double *x*);

double **scalb**(double *x*, int *n*);

## DESCRIPTION

These functions are required for the IEEE 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n***y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

## RETURN VALUES

The IEEE 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

IEEE 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

## ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

## SEE ALSO

floor(3M) , math(3M) , infnan(3M)

**NAME**

cos – cosine function

**SYNOPSIS**

```
#include <math.h>
double cos(double x);
```

**DESCRIPTION**

The *cos* function computes the cosine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance. For a discussion of errors due to rounding off, see math(3M).

**RETURN VALUES**

The *cos* function returns the cosine value.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`sin`(3M), `tan`(3M), `asin`(3M), `acos`(3M), `atan`(3M), `atan2`(3M), `sinh`(3M), `cosh`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *cos* function conforms to `ANSI-C.`

**NAME**

cosh – hyperbolic cosine function

**SYNOPSIS**

#include <math.h>
double **cosh**(double *x*);

**DESCRIPTION**

The *cosh* function computes the hyperbolic cosine of *x*.

**RETURN VALUES**

The *cosh* function returns the hyperbolic cosine, unless the magnitude of *x* is too large; in this event, *errno* is set to ERANGE and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `sin`(3M), `sinh`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *cosh* function conforms to `ANSI-C.`

**NAME**

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

**SYNOPSIS**

#include <math.h>
double **copysign**(double *x*, double *y*);

double **drem**(double *x*, double *y*);

int **finite**(double *x*);

double **logb**(double *x*);

double **scalb**(double *x*, int *n*);

**DESCRIPTION**

These functions are required for the IEEE 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n***y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

**RETURN VALUES**

The IEEE 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

IEEE 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

**ATTRIBUTES**

See attributes(5) for descriptions of the following attributes:

| **ATTRIBUTE TYPE** | **ATTRIBUTE VALUE** |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

floor(3M) , math(3M) , infnan(3M)

**NAME**

erf, erfc – error function operators

**SYNOPSIS**

#include <math.h>
double **erf**(double *x*);

double **erfc**(double *x*);

**DESCRIPTION**

These functions calculate the error function of *x* .

The *erf* function calculates the error function of x; where

erf(x) = 2 / sqrt(pi)*integral from 0 to x of exp(-t*t) dt.

The *erfc* function calculates the complementary error function of *x* ; that is, *erfc* subtracts the result of the error function *erf(x)* from 1.0. This is useful when *x* is a large value, as decimal places can be lost.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M)

**NAME**

erf, erfc – error function operators

**SYNOPSIS**

#include <math.h>
double **erf**(double *x*);

double **erfc**(double *x*);

**DESCRIPTION**

These functions calculate the error function of *x* .

The *erf* function calculates the error function of x; where

erf(x) = 2 / sqrt(pi)*integral from 0 to x of exp(-t*t) dt.

The *erfc* function calculates the complementary error function of *x* ; that is, *erfc* subtracts the result of the error function *erf(x)* from 1.0. This is useful when *x* is a large value, as decimal places can be lost.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M)

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x)*, *log(x)*, *expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log*, unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x)*, *log(x)*, *expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log*, unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO** `math`(3M) , `infnan`(3M)

**NAME**

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

**SYNOPSIS**

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

double **`drem`**(double *x*, double *y*);

int **`finite`**(double *x*);

double **`logb`**(double *x*);

double **`scalb`**(double *x*, int *n*);

**DESCRIPTION**

These functions are required for the `IEEE` 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n***y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

**RETURN VALUES**

The `IEEE` 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

`IEEE` 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`floor`(3M) , `math`(3M) , `infnan`(3M)

**NAME**

floor – largest integral value not greater than x

**SYNOPSIS**

```
#include <math.h>
double floor(double x);
```

**DESCRIPTION**

The *floor* function computes the largest integral value not greater than *x*.

**RETURN VALUES**

The *floor* function returns the largest integral value expressed as a double.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

, `ieee`(3M), `rint`(3M), `math`(3M), `abs`(3STDC), `fabs`(3STDC)

**STANDARDS**

The *floor* function conforms to `ANSI-C.`

**NAME**

fmod – floating-point remainder function

**SYNOPSIS**

#include <math.h>
double **fmod**(double *x*, double *y*);

**DESCRIPTION**

The *fmod* function computes the floating-point remainder of $x / y$.

**RETURN VALUES**

The *fmod* function returns the value $x - i * y$, for the integer `i`. If *y* is non-zero, the result has the same sign as *x* and a magnitude less than the magnitude of *y*. If *y* is zero, *errno* is set to EDOM and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M)

**STANDARDS**

The *fmod* function conforms to `ANSI-C.`

**NAME**

lgamma, gamma – log gamma function, gamma function

**SYNOPSIS**

#include <math.h>; extern int signgam
double **lgamma**(double *x*);

double **gamma**(double *x*);

**DESCRIPTION**

*lgamma(x)* returns:

ln | |~(x)|

where

|~(x) = integral from 0 to +∞ of pow(t, x-1)*exp(-t) dt for x > 0

and

|~(x) = Pi/(|~(1-x)sin(pi*x)) for x < 0

The external integer *signgam* returns the sign of |~(x).

*gamma(x)* returns |~(x) , with no effect on *signgam* .

**IDIOSYNCRASIES**

Do not use the expression

signgam*exp(lgamma(x)) to compute g := |~(x).

Instead, use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

*signgam* will only be correct after *lgamma* has returned.

For arguments within its range, *gamma* is preferable , as for positive arguments it is accurate to within one unit in the last place. Exponentiation of *lgamma* will lose up to 10 significant bits.

**Note -** The *lgamma* function is not thread safe.

**RETURN VALUES**

The *gamma* and *lgamma* functions return appropriate values unless an argument is out of range. Overflow will occur for sufficiently large positive values, and non-positive integers. In this case, infinity is returned, *errno* is set to ERANGE , and a system-dependent notification is performed. For large non-integer negative values, *gamma* will underflow.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NOTES**

Due to *signgam* being a global variable, this routine is not reentrant.

**NAME**

hypot, cabs – euclidean distance and complex absolute value functions

**SYNOPSIS**

#include <math.h>
double **hypot**(double *x*, double *y*);

double **cabs**(struct {double *x* ; double *y* ;} *z* ;);

**DESCRIPTION**

The *hypot* and *cabs* functions compute the square root of (x*x+y*y) in such a way that underflow will not occur, and overflow occurs only if the final result justifies it.

*hypot* (∞ , v) = *hypot* (v, ∞) = +Infinity

for all values of *v* , including **NaN** .

**ERRORS (due to Roundoff)**

Less than 0.97 **ulp** s. Consequently, *hypot(5.0, 12.0)* = 13.0 exactly; in general, hypot and cabs return an integer whenever an integer is expected.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `sqrt`(3M)

## NAME

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

## SYNOPSIS

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

double **`drem`**(double *x*, double *y*);

int **`finite`**(double *x*);

double **`logb`**(double *x*);

double **`scalb`**(double *x*, int *n*);

## DESCRIPTION

These functions are required for the `IEEE` 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n*y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

## RETURN VALUES

The `IEEE` 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

`IEEE` 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

## ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

## SEE ALSO

`floor`(3M) , `math`(3M) , `infnan`(3M)

**NAME**

infnan – signals invalid floating point operations

**SYNOPSIS**

#include <math.h>
double **infnan**(int *iarg*);

**DESCRIPTION**

Invalid, Overflow and Divide–by–Zero events are notified to the application by calls to *infnan* in appropriate places in *libm*. As exception–handling depends upon the operating system, *infnan* does not necessarily alter the current flow of control. Users of libm can design their own *infnan*.

Whenever an elementary function code in libm runs into an exceptional situation, or has to return an invalid result, it calls *infnan(iarg)* with an appropriate value of *iarg* ( ERANGE or EDOM ). The *infnan* function assigns the corresponding value to *errno* and triggers whatever exception mechanism is available. If given back control, it returns a non-finite value, which allows computation to resume, and prompts the user to consult the *errno* file.

ERANGE and EDOM are defined in `errno.h`.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M)

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

lgamma, gamma – log gamma function, gamma function

**SYNOPSIS**

#include <math.h>; extern int signgam
double **lgamma**(double *x*);

double **gamma**(double *x*);

**DESCRIPTION**

*lgamma(x)* returns:

ln | |~(x)|

where

|~(x) = integral from 0 to +∞ of pow(t, x-1)*exp(-t) dt for x > 0

and

|~(x) = Pi/(|~(1-x)sin(pi*x)) for x < 0

The external integer *signgam* returns the sign of |~(x).

*gamma(x)* returns |~(x) , with no effect on *signgam* .

**IDIOSYNCRASIES**

Do not use the expression

signgam*exp(lgamma(x)) to compute g := |~(x).

Instead, use a program like this (in C):

```
lg = lgamma(x); g = signgam*exp(lg);
```

*signgam* will only be correct after *lgamma* has returned.

For arguments within its range, *gamma* is preferable , as for positive arguments it is accurate to within one unit in the last place. Exponentiation of *lgamma* will lose up to 10 significant bits.

**Note -** The *lgamma* function is not thread safe.

**RETURN VALUES**

The *gamma* and *lgamma* functions return appropriate values unless an argument is out of range. Overflow will occur for sufficiently large positive values, and non-positive integers. In this case, infinity is returned, *errno* is set to ERANGE , and a system-dependent notification is performed. For large non-integer negative values, *gamma* will underflow.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NOTES**

Due to *signgam* being a global variable, this routine is not reentrant.

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x), log(x), expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log,* unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x), log(x), expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log,* unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x), log(x), expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log,* unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| **ATTRIBUTE TYPE** | **ATTRIBUTE VALUE** |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO** `math`(3M) , `infnan`(3M)

**NAME**

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

**SYNOPSIS**

#include <math.h>
double **copysign**(double *x*, double *y*);

double **drem**(double *x*, double *y*);

int **finite**(double *x*);

double **logb**(double *x*);

double **scalb**(double *x*, int *n*);

**DESCRIPTION**

These functions are required for the IEEE 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n***y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

**RETURN VALUES**

The IEEE 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

IEEE 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`floor`(3M) , `math`(3M) , `infnan`(3M)

**NAME**

math – introduction to mathematical library functions

**DESCRIPTION**

These functions constitute the C math library, *libm*. The link editor searches this library under the *–lm* option. Declarations for these functions may be obtained from the include file <`math.h`>. References of the form *name*(3M) refer to pages in this section of this document.

**LIST OF FUNCTIONS**

| Name | Appears on Page | Description | Error Bound (ULPs) |
|---|---|---|---|
| acos | acos.3 | inverse trigonometric function | 3 |
| acosh | acosh.3 | inverse hyperbolic function | 3 |
| asin | asin.3 | inverse trigonometric function | 3 |
| asinh | asinh.3 | inverse hyperbolic function | 3 |
| atan | sin.3 | inverse trigonometric function | 1 |
| atanh | atanh.3 | inverse hyperbolic function | 3 |
| atan2 | atan2.3 | inverse trigonometric function | 2 |
| cabs | hypot.3 | complex absolute value | 1 |
| cbrt | sqrt.3 | cube root | 1 |
| ceil | ceil.3 | integer no less than | 0 |
| copysign | ieee.3 | copy sign bit | 0 |
| cos | cos.3 | trigonometric function | 1 |
| cosh | cosh.3 | hyperbolic function | 3 |
| drem | ieee.3 | remainder | 0 |
| erf | erf.3 | error function | ??? |
| erfc | erf.3 | complementary error function | ??? |
| exp | exp.3 | exponential | 1 |
| expm1 | exp.3 | exp(x)--1 | 1 |
| finite | ieee.3 | Is the value a valid finite number? | |
| floor | floor.3 | integer no greater than | 0 |
| fmod | fmod.3 | floating-point remainder | ??? |
| gamma | lgamma.3 | gamma function | 4 |
| hypot | hypot.3 | Euclidean distance | 1 |
| infnan | infnan.3 | signals exceptions | |
| j0 | j0.3 | bessel function | ??? |
| j1 | j0.3 | bessel function | ??? |
| jn | j0.3 | bessel function | ??? |
| lgamma | lgamma.3 | log gamma function | 2 for positive arguments |
| log | exp.3 | natural logarithm | 1 |
| logb | ieee.3 | exponent extraction | 0 |
| log10 | exp.3 | logarithm to base 10 | 3 |
| log1p | exp.3 | log(1+x)     1 | |
| pow | exp.3 | exponential x**y | 60–500 |
| rint | rint.3 | round to nearest integer | 0 |
| scalb | ieee.3 | exponent adjustment | 0 |
| sin | sin.3 | trigonometric function | 1 |
| sinh | sinh.3 | hyperbolic function | 3 |
| sqrt | sqrt.3 | square root | 1 |
| tan | tan.3 | trigonometric function | 3 |
| tanh | tanh.3 | hyperbolic function | 3 |
| y0 | j0.3 | bessel function | ??? |
| y1 | j0.3 | bessel function | ??? |
| yn | j0.3 | bessel function | ??? |

An **ulp** is one **U**nit in the **L**ast **P**lace.

## NOTES

The foregoing functions assume double–precision arithmetic conforming to the IEEE Standard 754 for Binary Floating–Point Arithmetic.

### Properties of IEEE 754 Double–Precision

Wordsize:

64 bits, 8 bytes

Radix:
Binary

Precision:
53 significant bits, approximate to 16 significant decimals

If x and x' are consecutive positive Double–Precision numbers (they differ by 1 **ulp**), then 1.1e–16 < 0.5**53 < (x'–x) / x ≤ 0.5**52 < 2.3e–16.

Range:
Overflow threshold = 2.0**102 = 1.8e308

Underflow threshold = 0.5**1022 ≈ 2.2e–308

Overflow goes by default to a signed ∞

Underflow is *Gradual*, rounding to the nearest integer multiple of

0.5**1074 ≈ 4.9e–324.

Zero is represented ambiguously as +0 or –0:
Its sign transforms correctly through multiplication or division, and is preserved by addition of zeros with like signs; but x–x yields +0 for every finite x. The only operations that reveal zero's sign are division by zero and copysign(x,±0). In particular, comparison (x > y, x ≥ y) cannot be affected by the sign of zero; but if finite x = y then ∞ = 1 / (x–y) ≠ –1 / (y–x) = –∞

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

Reserved operands:
There are 2**53–2 of them, all called **NaN** (*N*ot *a* *N*umber). Some, called Signaling **NaN**s, trap any floating–point operation performed upon them; they are used to mark missing or uninitialized values, or nonexistent elements of arrays. The rest are Quiet **NaN**s; they are the default results of Invalid Operations, and propagate through subsequent arithmetic operations. If x ≠ x then x is **NaN**; every other predicate (x > y, x = y, x < y, ...) is FALSE if **NaN** is involved.

NOTE: Trichotomy is violated by **NaN**. Besides being FALSE, predicates that entail ordered comparison, rather than mere (in)equality, signal Invalid Operation when **NaN** is involved.

Rounding:

Every algebraic operation (+, –, *, /, sqrt) is rounded by default to within half a **ulp**, and when the rounding error is exactly half a **ulp**, the rounded value's least significant bit is zero. This kind of rounding is usually the best kind, sometimes provably so; for instance, for every x = 1.0, 2.0, 3.0, 4.0, ..., 2.0**52, both (x/3.0)*3.0 == x and (x/10.0)*10.0 == x and ... despite both the quotients and the products having been rounded. No single kind of rounding can be proved best for every circumstance, IEEE 754 therefore provides rounding towards zero, or towards +∞, or towards –∞, at the programmer's option. The same kinds of rounding are specified for Binary–Decimal Conversions, at least for magnitudes between roughly 1.0e–10 and 1.0e37.

Exceptions:

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

| Exception | Default Result |
|---|---|
| Invalid Operation | NaN, or FALSE |
| Overflow | ±∞ |
| Divide by Zero | ±∞ |
| Underflow | Gradual Underflow |
| Inexact | Rounded value |

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

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

1. Test for a condition that might cause an exception later, and branch to avoid the exception.
2. Test a flag to see whether an exception has occurred since the program last reset its flag.

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

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

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

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

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

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

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

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

## RESTRICTIONS

This library implements very little of the above-mentionned IEEE 754 signaling requirements. As it does not rely upon the floating-point unit / operating system cooperation to signal errors asynchronously, it catches most errors explicitly. However, some **NaN** or results may be issued by the floating-point unit and be returned as such to the application without any warning better than the value of the result. Detected errors are reported by setting *errno* to either ERANGE or EDOM, performing a system-dependant notification, and returning either + , - or **NaN**, whichever best suits the nature of the error. The above system-dependent notification is non-operational in ChorusOS products for which this library is currently distributed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**NAME**

exp, expm1, log, log10, log1p, pow – exponential, logarithm, power functions

**SYNOPSIS**

#include <math.h>
double **exp**(double *x*);

double **expm1**(double *x*);

double **log**(double *x*);

double **log10**(double *x*);

double **log1p**(double *x*);

double **pow**(double *x*, double *y*);

**DESCRIPTION**

The *exp* function computes the exponential value of the given argument *x* .

The *expm1* function computes the value exp(x)–1 accurately even for extremely small values of *x* .

The *log* function computes the natural logarithm of the argument x.

The *log10* function computes the logarithm of argument *x* to base 10.

The *log1p* function computes the value of log(1+x) accurately even for extremely small values of *x* .

The *pow* computes the value of to the exponent

**ERRORS (due to Roundoff)**

The *exp(x), log(x), expm1(x)* and *log1p(x)* functions are accurate to within a `ulp` , and *log10(x)* to within approximately 2 `ulp` s. A `ulp` is one `Unit` in the `Last Place` . The error in *pow(x, y)* is below about 2 `ulp` s when its magnitude is moderate, but increases as *pow(x, y)* approaches the over / underflow thresholds. Almost as many bits as are occupied by the floating–point format's exponent field could be lost; that is 11 bits for `IEEE` 754 Double. The worst errors observed during testing have been under 300 `ulp` s for `IEEE` 754 Double. Moderate values of *pow* are sufficiently accurate that *pow(integer, integer)* is precise until it is greater than 2**53 for `IEEE` 754.

**RETURN VALUES**

These functions will return the appropriate computation unless an error occurs or an argument is out of range. The functions *exp* , *expm1* and *pow* detect if the computed value will overflow, set *errno* to ERANGE and a system-dependent notification is performed. The *pow(x, y)* function checks whether *x* < 0 and *y* is not an integer; if this is true, *errno* is set to EDOM and a system-dependent notification is performed. *errno* is set to EDOM and a system-dependent notification is performed by *log,* unless *x* > 0, by *log1p* unless *x* > –1.

**NOTES**

The function *pow(x, 0)* returns x**0 = 1 for all x including x = 0, , and **NaN** .

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

rint – round to closest integer functions

**SYNOPSIS**

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

**DESCRIPTION**

The *rint* function finds the integer (represented as a double precision number) nearest to *x* in the direction of the prevailing rounding mode.

**NOTES**

In the default rounding mode on a machine that conforms to IEEE 754, *rint(x)* is the integer closest to *x*, with the additional stipulation that if |rint(x)–x|=1/2, then *rint(x)* will be even. Other rounding modes can make *rint* perform like *floor*, or like *ceil*, or round up to zero.

Another way to obtain an integer near *x* is to declare (in C): double x; int k;k=x;

Most C compilers round *x* up to 0 to get the integer *k*, but not all. Use *floor*, *ceil*, or *rint* first, depending on the result required. Note that if x is larger than *k* can accommodate, the value of *k* and the presence or absence of an integer overflow are unpredictable.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`ceil`(3M), `floor`(3M), `ieee`(3M), `math`(3M) `abs`(3STDC) `fabs`(3STDC)

## NAME

ieee, copysign, drem, finite, logb, scalb – IEEE 754 floating point support

## SYNOPSIS

#include <math.h>
double **copysign**(double *x*, double *y*);

double **drem**(double *x*, double *y*);

int **finite**(double *x*);

double **logb**(double *x*);

double **scalb**(double *x*, int *n*);

## DESCRIPTION

These functions are required for the IEEE 754 standard for floating–point arithmetic.

The *copysign ()* function returns *x* with its sign changed to that of *y* .

The *drem ()* function returns the remainder *r* := *x* – *n***y* where *n* is the integer closest to the exact value of *x/y* . If | *n* - *x/y* | = 1/2, *n* is even. Consequently, the remainder is computed exactly and | *r* | <= | *y* | / 2. Note that *drem* (x, *0)* is exceptional.

The *finite ()* function returns true (1) if the argument *x* is neither inifinity nor *NaN* value. Otherwise it returns false (0).

The *logb ()* function computes the exponent of x, which is the integral part of log2 |x|, as a signed floating point value for non-zero x.

The *scalb ()* function returns *x* * (2**n) computed, for integer n, without first computing 2**n.

## RETURN VALUES

The IEEE 754 standard defines *drem(x, 0)* and *drem(* infinity , *y)* to be invalid operations that produce a **NaN** .

IEEE 754 defines *logb* ( ± infinity) = + infinity, and *logb* (0) = -HUGE_VAL, and requires the latter to signal Division–by–Zero. Upon successful completion, logb() returns the exponent of x.

## ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

## SEE ALSO

floor(3M) , math(3M) , infnan(3M)

**NAME**

sin – sine function

**SYNOPSIS**

#include <math.h>
double **sin**(double *x*);

**DESCRIPTION**

The *sin* function computes the sine of *x* (measured in radians). A large magnitude argument may yield a result with little or no significance.

**RETURN VALUES**

The *sin* function returns the sine value.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sinh`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *sin* function conforms to `ANSI-C`.

**NAME**

sinh – hyperbolic sine function

**SYNOPSIS**

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

**DESCRIPTION**

The *sinh* function computes the hyperbolic sine of *x*.

**RETURN VALUES**

The *sinh* function returns the hyperbolic sine value unless the magnitude of *x* is too large. In this event, *errno* is set to ERANGE and a system-dependent notification is performed.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `tan`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *sinh* function conforms to `ANSI-C.`

## NAME

sqrt, cbrt – cube root and square root functions

## SYNOPSIS

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

double sqrt(double x);
```

## DESCRIPTION

The *cbrt* function computes the cube root of *x* .

The *sqrt* function computes the non-negative square root of x.

## RETURN VALUES

The *cbrt* function returns the requested cube root. The *sqrt* function returns the requested square root unless an error occurs. An attempt to take the *sqrt* of a negative value of *x* causes an error; in this event, *errno* is set to EDOM and a system-dependent notification is performed.

## ERRORS (due to Roundoff)

The *cbrt* function is accurate to within 0.7 `ulps` .

On a machine that conforms to `IEEE` 754 sqrt is correctly rounded in accordance with the rounding mode in force; the error is less than half a `ulp` in the default mode (round–to–nearest). A `ulp` is one Unit in the Last Place carried.

## ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

## SEE ALSO

`math`(3M) , `infnan`(3M)

## STANDARDS

The *sqrt* function conforms to `ANSI-C.`

**NAME**

tan – tangent function

**SYNOPSIS**

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

**DESCRIPTION**

The *tan* function computes the tangent of *x* (measured in radians). An argument of high magnitude may yield a result with little or no significance. For a discussion of errors due to rounding up, see math(3).

**RETURN VALUES**

The *tan* function returns the tangent value.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `sinh`(3M), `tanh`(3M), `math`(3M)

**STANDARDS**

The *tan* function conforms to `ANSI-C.`

**NAME**

tanh – hyperbolic tangent function

**SYNOPSIS**

#include <math.h>
double **tanh**(double *x*);

**DESCRIPTION**

The *tanh* function computes the hyperbolic tangent of *x*. For a discussion of errors due to rounding up, see math(3).

**RETURN VALUES**

The *tanh* function returns the hyperbolic tangent value.

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`acos`(3M), `asin`(3M), `atan`(3M), `atan2`(3M), `cos`(3M), `cosh`(3M), `sin`(3M), `sinh`(3M), `tan`(3M), `math`(3M)

**STANDARDS**

The *tanh* function conforms to `ANSI-C.`

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

**NAME**

j0, j1, jn, y0, y1, yn – Bessel functions of the first and second kind

**SYNOPSIS**

#include <math.h>
double **j0**(double *x*);

double **j1**(double *x*);

double **jn**(int *n*, double *x*);

double **y0**(double *x*);

double **y1**(double *x*);

double **yn**(int *n*, double *x*);

**DESCRIPTION**

The *j0* and *j1* functions compute the Bessel function of the first type of the order 0 and the order 1, respectively, for the real value *x* . The *jn* function computes the Bessel function of the first type of the integer order *n* for the real value *x* .

**RETURN VALUES**

If these functions are successful, the computed value is returned. Upon successful completion, *j0()* , *j1()* and *jn()* return the relevant Bessel value of x of the first type.

Upon successful completion, *y0()* , *y1()* and *yn()* return the relevant Bessel value of x of the second type. If the x argument of *y0()* , *y1()* or *yn()* is negative, `-HUGE_VAL` is returned and errno is set errno is set to EDOM. If x is 0.0, `-HUGE_VAL` is returned and errno is set to ERANGE. If the correct result would cause overflow, `-HUGE_VAL` or `HUGE_VAL` is returned and errno is set to ERANGE

**ATTRIBUTES**

See `attributes`(5) for descriptions of the following attributes:

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| Interface Stability | Evolving |

**SEE ALSO**

`math`(3M) , `infnan`(3M)

# Index