NAME

intro – introduction to mathematical library functions and constants

SYNOPSIS

#include <sys/ieeefp.h>

#include <floatingpoint.h>

#include <math.h>

DESCRIPTION

The include file **<math.h>** contains declarations of all the functions described in Section 3M that are implemented in the math library, **libm**. C programs should be linked with the the **–lm** option in order to use this library.

<sys/ieeefp.h> and **<floatingpoint.h>** define certain types and constants used for **libm** exception handling, conforming to ANSI/IEEE Std 754-1985, the *IEEE Standard for Binary Floating-Point Arithmetic*.

ACKNOWLEDGEMENT

The Sun version of **libm** is based upon and developed from ideas embodied and codes contained in 4.3 BSD, which may not be compatible with earlier BSD or UNIX implementations.

IEEE ENVIRONMENT

The IEEE Standard specifies modes for rounding direction, precision, and exception trapping, and status reflecting accrued exceptions. These modes and status constitute the IEEE run-time environment. On Sun-2 and Sun-3 systems without 68881 floating-point co-processors, only the default rounding direction to nearest is available, only the default non-stop exception handling is available, and accrued exception bits are not maintained.

IEEE EXCEPTION HANDLING

The IEEE Standard specifies exception handling for **aint**, **ceil**, **floor**, **irint**, **remainder**, **rint**, and **sqrt**, and suggests appropriate exception handling for **fp_class**, **copysign**, **fabs**, **finite**, **fmod**, **isinf**, **isnan**, **ilogb**, **ldexp**, **logb**, **nextafter**, **scalb**, **scalbn** and **signbit**, but does not specify exception handling for the other **libm** functions.

For these other unspecified functions the spirit of the IEEE Standard is generally followed in **libm** by handling invalid operand, singularity (division by zero), overflow, and underflow exceptions, as much as possible, in the same way they are handled for the fundamental floating-point operations such as addition and multiplication.

These unspecified functions are usually not quite correctly rounded, may not observe the optional rounding directions, and may not set the inexact exception correctly.

SYSTEM V EXCEPTION HANDLING

The *System V Interface Definition* (SVID) specifies exception handling for some **libm** functions: **j0( )**, **j1( )**, **jn( )**, **y0( )**, **y1( )**, **yn( )**, **exp( )**, **log( )**, **log10( )**, **pow( )**, **sqrt( )**, **hypot( )**, **lgamma( )**, **sinh( )**, **cosh( )**, **sin( )**, **cos( )**, **tan( )**, **asin( )**, **acos( )**, and **atan2( )**. See **matherr**(3M) for a discussion of the extent to which Sun's implementation of **libm** follows the SVID when it is consistent with the IEEE Standard and with hardware efficiency.

LIST OF MATH LIBRARY FUNCTIONS

Name	**Appears on Page**	**Description**
–	**bessel(3M)**	Bessel functions
–	**frexp(3M)**	floating-point analysis
–	**hyperbolic(3M)**	hyperbolic functions
–	**ieee_functions(3M)**	IEEE classification
–	**ieee_test(3M)**	IEEE tests for compliance
–	**ieee_values(3M)**	returns double-precision IEEE infinity
–	**trig(3M)**	trigonometric functions
acos()	**trig(3M)**	inverse trigonometric functions
acosh()	**hyperbolic(3M)**	inverse hyperbolic function
aint()	**rint(3M)**	convert to integral value in floating-point format
anint()	**rint(3M)**	convert to integral value in floating-point format
asin()	**trig(3M)**	inverse trigonometric function
asinh()	**hyperbolic(3M)**	inverse hyperbolic function
atan()	**trig(3M)**	inverse trigonometric function
atan2()	**trig(3M)**	rectangular to polar conversion
atanh()	**hyperbolic(3M)**	inverse hyperbolic function
cbrt()	**sqrt(3M)**	cube root
ceil()	**rint(3M)**	ceiling function
copysign()	**ieee_functions(3M)**	copy sign bit
cos()	**trig(3M)**	trigonometric function
cosh()	**hyperbolic(3M)**	hyperbolic function
erf()	**erf(3M)**	error function
erfc()	**erf(3M)**	complementary error function
exp()	**exp(3M)**	exponential function
expm1()	**exp(3M)**	exp(X)-1
exp2()	**exp(3M)**	2**X
exp10()	**exp(3M)**	10**X
fabs()	**ieee_functions(3M)**	absolute value function
finite()	**ieee_functions(3M)**	test for finite number
floor()	**rint(3M)**	floor function
fmod()	**ieee_functions(3M)**	floating-point remainder
fp_class()	**ieee_functions(3M)**	classify operand
frexp()	**frexp(3M)**	floating-point analysis
hypot()	**hypot(3M)**	Euclidean distance
ieee_flags()	**ieee_flags(3M)**	IEEE modes and status
ieee_handler()	**ieee_handler(3M)**	IEEE trapping
ilogb()	**ieee_functions(3M)**	exponent extraction
infinity()	**ieee_values(3M)**	returns double-precision IEEE infinity
irint()	**rint(3M)**	convert to integral value in integer format
isinf()	**ieee_functions(3M)**	IEEE classification
isnan()	**ieee_functions(3M)**	IEEE classification
isnormal()	**ieee_functions(3M)**	IEEE classification
issubnormal()	**ieee_functions(3M)**	IEEE classification
iszero()	**ieee_functions(3M)**	IEEE classification
j0()	**bessel(3M)**	Bessel function
j1()	**bessel(3M)**	Bessel function
jn()	**bessel(3M)**	Bessel function
ldexp()	**frexp(3M)**	exponent adjustment
lgamma()	**lgamma(3M)**	log gamma function
log()	**exp(3M)**	natural logarithm

logb()	**ieee_test**(3M)	exponent extraction
log1p()	**exp**(3M)	log(1+X)
log2()	**exp**(3M)	log base 2
log10()	**exp**(3M)	common logarithm
matherr()	**matherr**(3M)	math library exception-handling routines
max_normal()	**ieee_values**(3M)	double-precision IEEE largest positive normalized number
max_subnormal()	**ieee_values**(3M)	double-precision IEEE largest positive subnormal number
min_normal()	**ieee_values**(3M)	double-precision IEEE smallest positive normalized number
min_subnormal()	**ieee_values**(3M)	double-precision IEEE smallest positive subnormal number
modf()	**frexp**(3M)	floating-point analysis
nextafter()	**ieee_functions**(3M)	IEEE nearest neighbor
nint()	**rint**(3M)	convert to integral value in integer format
pow()	**exp**(3M)	power X**Y
quiet_nan()	**ieee_values**(3M)	returns double-precision IEEE quiet NaN
remainder()	**ieee_functions**(3M)	floating-point remainder
rint()	**rint**(3M)	convert to integral value in floating-point format
scalb()	**ieee_test**(3M)	exponent adjustment
scalbn()	**ieee_functions**(3M)	exponent adjustment
signaling_nan()	**ieee_values**(3M)	returns double-precision IEEE signaling NaN
signbit()	**ieee_functions**(3M)	IEEE sign bit test
significand()	**ieee_test**(3M)	scalb(x,-ilogb(x))
sin()	**trig**(3M)	trigonometric function
sincos()	**trig**(3M)	simultaneous sin and cos
single_precision()	**single_precision**(3M)	single-precision libm access
sinh()	**hyperbolic**(3M)	hyperbolic function
sqrt()	**sqrt**(3M)	square root
tan()	**trig**(3M)	trigonometric function
tanh()	**hyperbolic**(3M)	hyperbolic function
y0()	**bessel**(3M)	Bessel function
y1()	**bessel**(3M)	Bessel function
yn()	**bessel**(3M)	Bessel function

NAME

j0, j1, jn, y0, y1, yn – Bessel functions

SYNOPSIS

```
#include <math.h>

double j0(x)
double x;

double j1(x)
double x;

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

double y0(x)
double x;

double y1(x)
double x;

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

DESCRIPTION

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

SEE ALSO

exp(3M)

DIAGNOSTICS

The functions *y0*, *y1*, and *yn* have logarithmic singularities at the origin, so they treat zero and negative arguments the way *log* does, as described in exp(3M). Such arguments are unexceptional for *j0*, *j1*, and *jn*.

NAME

erf, erfc – error functions

SYNOPSIS

#include <math.h>

double erf(x)
double x;

double erfc(x)
double x;

DESCRIPTION

erf(x) returns the error function of *x*; where **erf** (x):= $(2/\sqrt{\pi})\int_0^x$ **exp**$(-t^2)$ dt.

erfc(x) returns 1.0–erf (x), computed however by other methods that avoid cancellation for large *x*.

NAME

exp, expm1, exp2, exp10, log, log1p, log2, log10, pow – exponential, logarithm, power

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double expm1(x)
double x;

double exp2(x)
double x;

double exp10(x)
double x;

double log(x)
double x;

double log1p(x)
double x;

double log2(x)
double x;

double log10(x)
double x;

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

DESCRIPTION

exp() returns the exponential function *e**x*.

expm1() returns *e**x–1* accurately even for tiny *x*.

exp2() and **exp10()** return *2**x* and *10**x* respectively.

log() returns the natural logarithm of *x*.

log1p() returns log(1+x) accurately even for tiny *x*.

log2() and **log10()** return the logarithm to base 2 and 10 respectively.

pow() returns *x**y*. **pow(*x*,0.0)** is 1 for all x, in conformance with 4.3BSD, as discussed in the *Floating Point Programmers Guide*.

SEE ALSO

matherr(3M)

DIAGNOSTICS

All these functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. Thus for x == ±0, log(*x*) is –∞ with a division by zero exception; for x < 0, including –∞, log(*x*) is a quiet NaN with an invalid operation exception; for x == +∞ or a quiet NaN, log(*x*) is x without exception; for x a signaling NaN, log(*x*) is a quiet NaN with an invalid operation exception; for x == 1, log(*x*) is 0 without exception; for any other positive x, log(*x*) is a normalized number with an inexact exception.

In addition, **exp,exp2,exp10, log,log2,log10,** and **pow()** may also set **errno** and call **matherr(3M)**.

NAME

frexp, modf, ldexp – traditional UNIX functions

SYNOPSIS

```
#include <math.h>

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(x,n)
double x;
int n;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

These functions are provided for compatibility with other UNIX system implementations. They are not used internally in **libm** or **libc**. Better ways to accomplish similar ends may be found in **ieee_functions**(3M) and **rint**(3M).

ldexp(*x*,*n*) returns *x* * *2***n* computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication. Note: **ldexp**(*x*,*n*) differs from **scalbn**(*x*,*n*), defined in **ieee_functions**(3M), only that in the event of IEEE overflow and underflow, **ldexp**(*x*,*n*) sets **errno** to ERANGE.

Every non-zero number can be written uniquely as *x* * *2***n*, where the significand *x* is in the range *0.5* <= */x/* < *1.0* and the exponent *n* is an integer. The function **frexp**( ) returns the significand of a double *value* as a double quantity, *x*, and stores the exponent *n*, indirectly through *eptr*. If *value* == 0, both results returned by **frexp**( ) are 0.

modf( ) returns the fractional part of *value* and stores the integral part indirectly through *iptr*. Thus the argument *value* and the returned values **modf**( ) and **iptr* satisfy

(**iptr* + *modf*) == *value*

and both results have the same sign as *value*. The definition of **modf**( ) varies among UNIX system implementations, so avoid **modf**( ) in portable code.

The results of **frexp**( ) and **modf**( ) are not defined when *value* is an IEEE infinity or NaN.

SEE ALSO

ieee_functions(3M), **rint**(3M)

NAME

sinh, cosh, tanh, asinh, acosh, atanh – hyperbolic functions

SYNOPSIS

#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;

DESCRIPTION

These functions compute the designated direct and inverse hyperbolic functions for real arguments. They inherit much of their roundoff error from **expm1( )** and **log1p**, described in **exp**(3M).

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. Thus **sinh( )** and **cosh( )** return ±∞ on overflow, **acosh( )** returns a NaN if its argument is less than 1, and **atanh( )** returns a NaN if its argument has absolute value greater than 1. In addition, **sinh,cosh**, and **tanh( )** may also set **errno** and call **matherr**(3M).

SEE ALSO

exp(3M), **matherr**(3M)

NAME

hypot – Euclidean distance

SYNOPSIS

#include <math.h>

double hypot(x, y)
double x, y;

DESCRIPTION

hypot() returns

sqrt(x*x + y*y) ,

taking precautions against unwarranted IEEE exceptions. On IEEE overflow, **hypot()** may also set **errno** and call **matherr**(3M). **hypot**(±∞, y) is +∞ for any y, even a NaN, and is exceptional only for a signaling NaN.

hypot(x,y) and **atan2**(3M) convert rectangular coordinates (x,y) to polar (r,θ); **hypot()** computes r, the modulus or radius.

SEE ALSO

matherr(3M)

NAME

ieee_flags – mode and status function for IEEE standard arithmetic

SYNOPSIS

#include <sys/ieeefp.h>

int ieee_flags(action,mode,in,out)
char *action, *mode, *in, **out;

DESCRIPTION

This function provides easy access to the modes and status required to fully exploit ANSI/IEEE Std 754-1985 arithmetic in a C program. All arguments are pointers to strings. Results arising from invalid arguments and invalid combinations are undefined for efficiency.

There are four types of *action*: "get", "set", "clear", and "clearall". There are three valid settings for *mode*, two corresponding to modes of IEEE arithmetic:

"direction",	... current rounding direction mode
"precision",	... current rounding precision mode

and one corresponding to status of IEEE arithmetic:

"exception".	... accrued exception-occurred status

There are 14 types of *in* and *out* :

"nearest",	... round toward nearest
"tozero",	... round toward zero
"negative",	... round toward negative infinity
"positive",	... round toward positive infinity
"extended",	
"double",	
"single",	
"inexact",	
"division",	... division by zero exception
"underflow",	
"overflow",	
"invalid",	
"all",	... all five exceptions above
"common".	... invalid, overflow, and division exceptions

Note: "all" and "common" only make sense with "set" or "clear".

For "clearall", ieee_flags() returns 0 and restores all default modes and status. Nothing will be assigned to *out*. Thus

```
char *mode, *out, *in;
ieee_flags("clearall",mode, in, &out);
```

set rounding direction to "nearest", rounding precision to "extended", and all accrued exception-occurred status to zero.

For "clear", ieee_flags() returns 0 and restores the default mode or status. Nothing will be assigned to *out*. Thus

```
char *out, *in;
ieee_flags("clear","direction", in, &out);     ... set rounding direction to round to nearest.
```

For "set", ieee_flags( ) returns 0 if the action is successful and 1 if the corresponding required status or mode is not available (for instance, not supported in hardware). Nothing will be assigned to *out*. Thus

```
char *out, *in;
ieee_flags ("set","direction","tozero",&out);        ... set rounding direction to round toward zero;
```

For "get", we have the following cases:

Case 1: *mode* is "direction". In that case, *out* returns one of the four strings "nearest", "tozero", "positive", "negative"; and ieee_flags( ) returns a value corresponding to *out* according to the enum *fp_direction_type* defined in <sys/ieeefp.h>.

Case 2: *mode* is "precision". In that case, *out* returns one of the three strings "extended", "double", "single"; and ieee_flags( ) returns a value corresponding to *out* according to the enum *fp_precision_type* defined in <sys/ieeefp.h>.

Case 3: *mode* is "exception". In that case, *out* returns

(a) "not available" if information on exception is not available,
(b) "no exception" if no accrued exception,
(c) the accrued exception that has the highest priority according to the list below

(1) the exception named by *in*,
(2) "invalid",
(3) "overflow",
(4) "division",
(5) "underflow",
(6) "inexact".

In this case ieee_flags( ) returns a five bit value where each bit (cf. enum *fp_exception_type* in <sys/ieeefp.h>) corresponds to an exception-occurred accrued status flag: 0 = off, 1 = on. The bit corresponding to a particular exception varies among architectures.

Example:

```
char *out; int k, ieee_flags( );
ieee_flags ("clear","exception","all",&out);        /* clear all accrued exceptions */
...
... (code that generates three exceptions: overflow, invalid, inexact)
...
k = ieee_flags("get","exception","overflow",&out);
```

then out = "overflow", and on a Sun-3, k=25.

FILES

/usr/include/sys/ieeefp.h
/usr/lib/libm.a

NAME

ieee_functions, fp_class, finite, ilogb, isinf, isnan, isnormal, issubnormal, iszero, signbit, copysign, fabs, fmod, nextafter, remainder, scalbn – appendix and related miscellaneous functions for IEEE arithmetic

SYNOPSIS

```
#include <math.h>

enum fp_class_type fp_class(x)
double x;

int finite(x)
double x;

int ilogb(x)
double x;

int isinf(x)
double x;

int isnan(x)
double x;

int isnormal(x)
double x;

int issubnormal(x)
double x;

int iszero(x)
double x;

int signbit(x)
double x;

double copysign(x,y)
double x, y;

double fabs(x)
double x;

double fmod(x,y)
double x, y;

double nextafter(x,y)
double x, y;

double remainder(x,y)
double x, y;

double scalbn(x,n)
double x; int n;
```

DESCRIPTION

Most of these functions provide capabilities required by ANSI/IEEE Std 754-1985 or suggested in its appendix.

fp_class(x) corresponds to the IEEE's class() and classifies x as zero, subnormal, normal, ∞, or quiet or signaling *NaN*; <**floatingpoint.h**> defines *enum fp_class_type*. The following functions return 0 if the indicated condition is not satisfied:

finite(x)	returns 1 if x is zero, subnormal or normal
isinf(x)	returns 1 if x is ∞
isnan(x)	returns 1 if x is *NaN*
isnormal(x)	returns 1 if x is normal
issubnormal(x)	returns 1 if x is subnormal
iszero(x)	returns 1 if x is zero
signbit(x)	returns 1 if x's sign bit is set

ilogb(x) returns the unbiased exponent of x in integer format. **ilogb**(±∞) = +MAXINT and **ilogb**(0) = −MAXINT; <**values.h**> defines MAXINT as the largest int. **ilogb**(x) never generates an exception. When x is subnormal, **ilogb**(x) returns an exponent computed as if x were first normalized.

copysign(x,y) returns x with y's sign bit.

fabs(x) returns the absolute value of x.

nextafter(x,y) returns the next machine representable number from x in the direction y.

remainder(x,y) and **fmod**(x,y) return a remainder of x with respect to y; that is, the result r is one of the numbers that differ from x by an integral multiple of y. Thus (x-r)/y is an integral value, even though it might exceed MAXINT if it were explicitly computed as an int. Both functions return one of the two such r smallest in magnitude. **remainder**(x,y) is the operation specified in ANSI/IEEE Std 754-1985; the result of **fmod**(x,y) may differ from **remainder**'*s* result by ±y. The magnitude of **remainder**'*s* result can not exceed half that of y; its sign might not agree with either x or y. The magnitude of **fmod**'*s* result is less than that of y; its sign agrees with that of x. Neither function can generate an exception as long as both arguments are normal or subnormal. **remainder**(x , 0), **fmod**(x , 0), **remainder**(∞, y), and **fmod**(∞, y) are invalid operations that produce a *NaN*.

scalbn(x,n) returns x* 2**n computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication. Thus

$$1 \le \mathbf{scalbn}(\mathbf{fabs}(x), -\mathbf{ilogb}(x)) < 2$$

for every x except 0, ∞, and *NaN*.

FILES

/usr/include/floatingpoint.h
/usr/include/math.h
/usr/include/values.h
/usr/lib/libm.a

SEE ALSO

floatingpoint(3), **ieee_environment**(3M), **matherr**(3M)

NAME

ieee_handler – IEEE exception trap handler function

SYNOPSIS

#include <floatingpoint.h>

int ieee_handler(action,exception,hdl)
char action[ ], exception[ ];
sigfpe_handler_type hdl;

DESCRIPTION

This function provides easy exception handling to exploit ANSI/IEEE Std 754-1985 arithmetic in a C program. All arguments are pointers to strings. Results arising from invalid arguments and invalid combinations are undefined for efficiency.

There are three types of *action* : "get", "set", and "clear". There are five types of *exception* :

"inexact"	
"division"	... division by zero exception
"underflow"	
"overflow"	
"invalid"	
"all"	... all five exceptions above
"common"	... invalid, overflow, and division exceptions

Note: "all" and "common" only make sense with "set" or "clear".

hdl contains the address of a signal-handling routine. **<floatingpoint.h>** defines *sigfpe_handler_type*.

"get" will get the location of the current handler routine for *exception* in **hdl** . "set" will set the routine pointed at by **hdl** to be the handler routine and at the same time enable the trap on *exception*, except when **hdl == SIGFPE_DEFAULT** or **SIGFPE_IGNORE**; then **ieee_handler()** will disable the trap on *exception*. When **hdl == SIGFPE_ABORT**, any trap on *exception* will dump core using **abort**(3). "clear" "all" disables trapping on all five exceptions.

Two steps are required to intercept an IEEE-related **SIGFPE** code with **ieee_handler**:

1) Set up a handler with **ieee_handler**.

2) Perform a floating-point operation that generates the intended IEEE exception.

Unlike **sigfpe**(3), **ieee_handler()** also adjusts floating-point hardware mode bits affecting IEEE trapping. For "clear", "set" **SIGFPE_DEFAULT**, or "set" **SIGFPE_IGNORE**, the hardware trap is disabled. For any other "set", the hardware trap is enabled.

SIGFPE signals can be handled using **sigvec**(2), **signal**(3), **signal**(3F), **sigfpe**(3), or **ieee_handler**(3M). In a particular program, to avoid confusion, use only one of these interfaces to handle **SIGFPE** signals.

DIAGNOSTICS

ieee_handler() normally returns 0. In the case of "set", 1 will be returned if the action is not available (for instance, not supported in hardware).

EXAMPLE

A user-specified signal handler might look like this:

```
void sample_handler( sig, code, scp, addr)
int sig ;          /* sig == SIGFPE always */
int code ;
struct sigcontext *scp ;
char *addr ;
{
        /*
          Sample user-written sigfpe code handler.
          Prints a message and continues.
          struct sigcontext is defined in <signal.h>.
         */
        printf("ieee exception code %x occurred at pc %X \n",code,scp->sc_pc);
}
```

and it might be set up like this:

```
extern void sample_handler();
main()
{
        sigfpe_handler_type hdl, old_handler1, old_handler2;
/*
 * save current overflow and invalid handlers
 */
        ieee_handler("get","overflow",old_handler1);
        ieee_handler("get","invalid", old_handler2);
/*
 * set new overflow handler to sample_handler() and set new
 * invalid handler to SIGFPE_ABORT (abort on invalid)
 */
        hdl = (sigfpe_handler_type) sample_handler;
        if(ieee_handler("set","overflow",hdl) != 0)
                printf("ieee_handler can't set overflow \n");
        if(ieee_handler("set","invalid",SIGFPE_ABORT) != 0)
                printf("ieee_handler can't set invalid \n");
        ...
/*
 * restore old overflow and invalid handlers
 */
        ieee_handler("set","overflow", old_handler1);
        ieee_handler("set","invalid", old_handler2);
}
```

FILES

/usr/include/floatingpoint.h
/usr/include/signal.h
/usr/lib/libm.a

SEE ALSO

sigvec(2), abort(3), floatingpoint(3), sigfpe(3), signal(3), signal(3F)

NAME

ieee_test, logb, scalb, significand – IEEE test functions for verifying standard compliance

SYNOPSIS

#include <math.h>

double logb(x)
double x;

double scalb(x,y)
double x; double y;

double significand(x)
double x;

DESCRIPTION

These functions allow users to verify compliance to ANSI/IEEE Std 754-1985 by running certain test vectors distributed by the University of California. Their use is not otherwise recommended; instead use **scalbn**(*x*,*n*) and **ilogb**(*x*) described in **ieee_functions**(3M). See the *Floating Point Programmers Guide* for details.

logb(*x*) returns the unbiased exponent of *x* in floating-point format, for exercising the logb(L) test vector. **logb**(±∞) = +∞; **logb(0)** = –∞ with a division by zero exception. **logb**(*x*) differs from **ilogb**(*x*) in returning a result in floating-point rather than integer format, in sometimes signaling IEEE exceptions, and in not normalizing subnormal *x*.

scalb(*x*,**(double)n**) returns *x* * *2****n* computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication, for exercising the scalb(S) test vector. Thus

0 ≤ scalb(fabs(*x*),–logb(*x*)) < 2

for every *x* except 0, ∞ and *NaN*. **scalb**(*x*,*y*) is not defined when *y* is not an integral value. **scalb**(*x*,*y*) differs from **scalbn**(*x*,**n**) in that the second argument is in floating-point rather than integer format.

significand(*x*) computes just

scalb(*x*, (double) -ilogb(*x*)),

for exercising the fraction-part(F) test vector.

FILES

/usr/include/math.h
/usr/lib/libm.a

SEE ALSO

floatingpoint(3), **ieee_values**(3M), **ieee_functions**(3M), **matherr**(3M)

NAME

ieee_values, min_subnormal, max_subnormal, min_normal, max_normal, infinity, quiet_nan, signaling_nan, HUGE, HUGE_VAL – functions that return extreme values of IEEE arithmetic

SYNOPSIS

#include <math.h>

double min_subnormal()

double max_subnormal()

double min_normal()

double max_normal()

double infinity()

double quiet_nan(n)
long n;

double signaling_nan(n)
long n;

#define HUGE (infinity())

#define HUGE_VAL (infinity())

DESCRIPTION

These functions return special values associated with ANSI/IEEE Std 754-1985 double-precision floating-point arithmetic: the smallest and largest positive subnormal numbers, the smallest and largest positive normalized numbers, positive infinity, and a quiet and signaling NaN. The long parameters *n* to **quiet_nan**(*n*) and **signaling_nan**(*n*) are presently unused but are reserved for future use to specify the significand of the returned NaN.

None of these functions are affected by IEEE rounding or trapping modes or generate any IEEE exceptions.

The macro **HUGE** returns $+\infty$ in accordance with previous SunOS releases. The macro **HUGE_VAL** returns $+\infty$ in accordance with the System V Interface Definition.

FILES

/usr/include/math.h
/usr/lib/libm.a

SEE ALSO

ieee_functions(3M)

NAME

lgamma – log gamma function

SYNOPSIS

#include <math.h>

extern int signgam;

double lgamma(x)
double x;

DESCRIPTION

lgamma( ) returns

$$\ln|\Gamma(x)|$$

where

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

for $x > 0$ and

$$\Gamma(x) = \pi/(\Gamma(1-x)\sin(\pi x))$$

for $x < 1$.

The external integer **signgam** returns the sign of $\Gamma(x)$.

IDIOSYNCRASIES

Do *not* use the expression **signgam*exp(lgamma(x))** to compute '$g := \Gamma(x)$'. Instead compute **lgamma( )** first:

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

only after **lgamma( )** has returned can **signgam** be correct. Note: $\Gamma(x)$ must overflow when x is large enough, underflow when $-x$ is large enough, and generate a division by zero exception at the singularities x a nonpositive integer. In addition, **lgamma( )** may also set **errno** and call **matherr**(3M).

SEE ALSO

matherr(3M)

NAME

matherr – math library exception-handling function

SYNOPSIS

#include <math.h>

int matherr(exc)
struct exception *exc;

DESCRIPTION

The SVID (*System V Interface Definition*) specifies that certain **libm** functions call **matherr()** when exceptions are detected. Users may define their own mechanisms for handling exceptions, by including a function named **matherr()** in their programs. **matherr()** is of the form described above. When an exception occurs, a pointer to the exception structure *exc* will be passed to the user-supplied **matherr()** function. This structure, which is defined in the **<math.h>** header file, is as follows:

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

The element **type** is an integer describing the type of exception that has occurred, from the following list of constants (defined in the header file):

DOMAIN	argument domain exception
SING	argument singularity
OVERFLOW	overflow range exception
UNDERFLOW	underflow range exception

The element **name** points to a string containing the name of the function that incurred the exception. The elements **arg1** and **arg2** are the arguments with which the function was invoked. **retval** is set to the default value that will be returned by the function unless the user's **matherr()** sets it to a different value.

If the user's **matherr()** function returns non-zero, no exception message will be printed, and **errno** will not be set.

If **matherr()** is not supplied by the user, the default matherr exception-handling mechanisms, summarized in the table below, will be invoked upon exception:

DOMAIN==fp_invalid
: An IEEE NaN is usually returned, **errno** is set to EDOM, and a message is printed on standard error. **pow(*x*,0.0)** for any *x* and **atan2(0.0,0.0)** return numerical default results but set **errno** and print the message.

SING==fp_division
: An IEEE ∞ of appropriate sign is returned, **errno** is set to EDOM, and a message is printed on standard error.

OVERFLOW==fp_overflow
: In the default rounding direction, an IEEE ∞ of appropriate sign is returned. In optional rounding directions, **±MAXDOUBLE**, the largest finite double-precision number, is sometimes returned instead of ±∞. **errno** is set to ERANGE.

UNDERFLOW==fp_underflow
: An appropriately-signed zero, subnormal number, or smallest normalized number is returned, and **errno** is set to ERANGE.

The facilities provided by **matherr()** are not available in situations such as compiling on a Sun-3 system with **/usr/lib/f68881/libm.il** or **/usr/lib/ffpa/libm.il**, in which case some **libm** functions are converted to atomic hardware operations. In these cases setting **errno** and calling **matherr()** are not worth the adverse performance impact, but regular ANSI/IEEE Std 754-1985 exception handling remains available. In any case **errno** is not a reliable error indicator in that it may be unexpectedly set by a function in a handler for an asynchronous signal.

DEFAULT ERROR HANDLING PROCEDURES

	Types of Errors			
<math.h> type	DOMAIN	SING	OVERFLOW	UNDERFLOW
errno	EDOM	EDOM	ERANGE	ERANGE
IEEE Exception	Invalid Operation	Division by Zero	Overflow	Underflow
<floatingpoint.h> type	fp_invalid	fp_division	fp_overflow	fp_underflow
ACOS, ASIN:	M, NaN	–	–	–
ATAN2(0,0):	M, ±0.0 or ±π	–	–	–
BESSEL:				
y0, y1, yn (x < 0)	M, NaN	–	–	–
y0, y1, yn (x = 0)	–	M, –∞	–	–
COSH, SINH:	–	–	IEEE Overflow	–
EXP:	–	–	IEEE Overflow	IEEE Underflow
HYPOT:	–	–	IEEE Overflow	–
LGAMMA:	–	M, +∞	IEEE Overflow	–
LOG, LOG10:				
(x < 0)	M, NaN	–	–	–
(x = 0)	–	M, –∞	–	–
POW:				
usual cases	–	–	IEEE Overflow	IEEE Underflow
(x < 0) ** (y not an integer)	M, NaN	–	–	–
0 ** 0	M, 1.0	–	–	–
0 ** (y < 0)	–	M, ±∞	–	–
SQRT:	M, NaN	–	–	–

ABBREVIATIONS

M	Message is printed (EDOM exception).
NaN	IEEE NaN result and invalid operation exception.
∞	IEEE ∞ result and division-by-zero exception.
IEEE Overflow	IEEE Overflow result and exception.
IEEE Underflow	IEEE Underflow result and exception.
π	Closest machine-representable approximation to pi.

The interaction of IEEE arithmetic and **matherr( )** is not defined when executing under IEEE rounding modes other than the default round to nearest: **matherr( )** may not be called on overflow or underflow, and the Sun-provided **matherr( )** may return results that differ from those in this table.

EXAMPLE

```
#include <math.h>

int
matherr(x)
register struct exception *x;
{
        switch (x->type) {
        case
                DOMAIN:
                /* change sqrt to return sqrt(-arg1), not NaN */
                if (!strcmp(x->name, "sqrt")) {
                        x->retval = sqrt(-x->arg1);
                        return (0); /* print message and set errno */
        } /* fall through */
        case
                SING:
                /* all other domain or sing exceptions, print message and abort */
                fprintf(stderr, "domain exception in %s\n", x->name);
                abort( );
                break;
        }
        return (0); /* all other exceptions, execute default procedure */
}
```

NAME

aint, anint, ceil, floor, rint, irint, nint – round to integral value in floating-point or integer format

SYNOPSIS

```
#include <math.h>

double aint(x)
double x;

double anint(x)
double x;

double ceil(x)
double x;

double floor(x)
double x;

double rint(x)
double x;

int irint(x)
double x;

int nint(x)
double x;
```

DESCRIPTION

aint, **anint**, **ceil**, **floor**, and **rint()** convert a double value into an integral value in double format. They vary in how they choose the result when the argument is not already an integral value. Here an "integral value" means a value of a mathematical integer, which however might be too large to fit in a particular computer's int format. All sufficiently large values in a particular floating-point format are already integral; in IEEE double-precision format, that means all values >= 2**52. Zeros, infinities, and quiet NaNs are treated as integral values by these functions, which always preserve their argument's sign.

aint() returns the integral value between *x* and 0, nearest *x*. This corresponds to IEEE rounding toward zero and to the Fortran generic intrinsic function **aint**.

anint() returns the nearest integral value to *x*, except halfway cases are rounded to the integral value larger in magnitude. This corresponds to the Fortran generic intrinsic function **anint**.

ceil() returns the least integral value greater than or equal to *x*. This corresponds to IEEE rounding toward positive infinity.

floor() returns the greatest integral value less than or equal to *x*. This corresponds to IEEE rounding toward negative infinity.

rint() rounds *x* to an integral value according to the current IEEE rounding direction.

irint converts *x* into int format according to the current IEEE rounding direction.

nint() converts *x* into int format rounding to the nearest int value, except halfway cases are rounded to the int value larger in magnitude. This corresponds to the Fortran generic intrinsic function **nint**.

NAME

single_precision - Single-precision access to math library functions

SYNOPSIS

#include <math.h>

FLOATFUNCTIONTYPE r_acos_ (x)
FLOATFUNCTIONTYPE r_acosh_ (x)
FLOATFUNCTIONTYPE r_aint_ (x)
FLOATFUNCTIONTYPE r_anint_ (x)
FLOATFUNCTIONTYPE r_asin_ (x)
FLOATFUNCTIONTYPE r_asinh_ (x)
FLOATFUNCTIONTYPE r_atan_ (x)
FLOATFUNCTIONTYPE r_atanh_ (x)
FLOATFUNCTIONTYPE r_atan2_ (x,y)
FLOATFUNCTIONTYPE r_cbrt_ (x)
FLOATFUNCTIONTYPE r_ceil_ (x)
enum fp_class_type ir_fp_class_ (x)
FLOATFUNCTIONTYPE r_copysign_ (x,y)
FLOATFUNCTIONTYPE r_cos_ (x)
FLOATFUNCTIONTYPE r_cosh_ (x)
FLOATFUNCTIONTYPE r_erf_ (x)
FLOATFUNCTIONTYPE r_erfc_ (x)
FLOATFUNCTIONTYPE r_exp_ (x)
FLOATFUNCTIONTYPE r_expm1_ (x)
FLOATFUNCTIONTYPE r_exp2_ (x)
FLOATFUNCTIONTYPE r_exp10_ (x)
FLOATFUNCTIONTYPE r_fabs_ (x)
int ir_finite_ (x)
FLOATFUNCTIONTYPE r_floor_ (x)
FLOATFUNCTIONTYPE r_fmod_ (x,y)
FLOATFUNCTIONTYPE r_hypot_ (x,y)
int ir_ilogb_ (x)
int ir_irint_ (x)
int ir_isinf_ (x)
int ir_isnan_ (x)
int ir_isnormal_ (x)
int ir_issubnormal_ (x)
int ir_iszero_ (x)
int ir_nint_ (x)
FLOATFUNCTIONTYPE r_infinity_ ()
FLOATFUNCTIONTYPE r_j0_ (x)
FLOATFUNCTIONTYPE r_j1_ (x)
FLOATFUNCTIONTYPE r_jn_ (n,x)
FLOATFUNCTIONTYPE r_lgamma_ (x)
FLOATFUNCTIONTYPE r_logb_ (x)
FLOATFUNCTIONTYPE r_log_ (x)
FLOATFUNCTIONTYPE r_log1p_ (x)
FLOATFUNCTIONTYPE r_log2_ (x)
FLOATFUNCTIONTYPE r_log10_ (x)
FLOATFUNCTIONTYPE r_max_normal_ ()
FLOATFUNCTIONTYPE r_max_subnormal_ ()
FLOATFUNCTIONTYPE r_min_normal_ ()
FLOATFUNCTIONTYPE r_min_subnormal_ ()
FLOATFUNCTIONTYPE r_nextafter_ (x,y)

FLOATFUNCTIONTYPE r_pow_ (x,y)
FLOATFUNCTIONTYPE r_quiet_nan_ (n)
FLOATFUNCTIONTYPE r_remainder_ (x,y)
FLOATFUNCTIONTYPE r_rint_ (x)
FLOATFUNCTIONTYPE r_scalb_ (x,y)
FLOATFUNCTIONTYPE r_scalbn_ (x,n)
FLOATFUNCTIONTYPE r_signaling_nan_ (n)
int ir_signbit_ (x)
FLOATFUNCTIONTYPE r_significand_ (x)
FLOATFUNCTIONTYPE r_sin_ (x)
void r_sincos_ (x,s,c)
FLOATFUNCTIONTYPE r_sinh_ (x)
FLOATFUNCTIONTYPE r_sqrt_ (x)
FLOATFUNCTIONTYPE r_tan_ (x)
FLOATFUNCTIONTYPE r_tanh_ (x)
FLOATFUNCTIONTYPE r_y0_ (x)
FLOATFUNCTIONTYPE r_y1_ (x)
FLOATFUNCTIONTYPE r_yn_ (n,x)

float *x, *y, *s, *c
int *n

DESCRIPTION

These functions are single-precision versions of certain **libm** functions. Primarily for use by Fortran programmers, these functions may also be used in other languages. The single-precision floating-point results are deviously declared to avoid C's automatic type conversion to double.

FILES

/usr/lib/libm.a

NAME

sqrt, cbrt – cube root, square root

SYNOPSIS

#include <math.h>

double cbrt(x)
double x;

double sqrt(x)
double x;

DESCRIPTION

sqrt(*x*) returns the square root of *x*, correctly rounded according to ANSI/IEEE 754-1985. In addition, **sqrt**( ) may also set **errno** and call **matherr**(3M).

cbrt(*x*) returns the cube root of *x*. **cbrt**( ) is accurate to within 0.7 *ulps*.

SEE ALSO

matherr(3M)

NAME

sin, cos, tan, asin, acos, atan, atan2 – trigonometric functions

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

void sincos(x, s, c)
double x, *s, *c;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(y, x)
double y, x;
```

DESCRIPTION

sin, **cos**, **sincos**, and **tan**( ) return trigonometric functions of radian arguments. The values of trigonometric functions of arguments exceeding $\pi/4$ in magnitude are affected by the precision of the approximation to $\pi/2$ used to reduce those arguments to the range $-\pi/4$ to $\pi/4$. Argument reduction may occur in hardware or software; if in software, the variable **fp_pi** defined in **<math.h>** allows changing that precision at run time. Trigonometric argument reduction is discussed in the *Floating Point Programmers Guide*. Note that **sincos(x,s,c)** allows simultaneous computation of ***s = sin(x)** and ***c = cos(x)**.

asin( ) returns the arc sin in the range $-\pi/2$ to $\pi/2$.

acos( ) returns the arc cosine in the range 0 to π.

atan( ) returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.

atan2(y,x) and **hypot**(3M) convert rectangular coordinates (x,y) to polar (r,θ); **atan2**( ) computes θ, the argument or phase, by computing an arc tangent of y/x in the range $-\pi$ to π. **atan2**(0.0,0.0) is ± 0.0 or $\pm\pi$, in conformance with 4.3BSD, as discussed in the *Floating Point Programmers Guide*.

DIAGNOSTICS

These functions handle exceptional arguments in the spirit of ANSI/IEEE Std 754-1985. **sin**($\pm\infty$), **cos**($\pm\infty$), **tan**($\pm\infty$), or **asin**(x) or **acos**(x) with $|x|>1$, return NaN. In addition, **asin**, **acos**, and **atan2**( ) may also set **errno** and call **matherr**(3M).

SEE ALSO

hypot(3M), **matherr**(3M)