```
L O G ( X )  :=  T H E  L O G A R I T H M  O F  X  ( B A S E  E )
IEEE double extended precision (64 bits)
```


WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions:

```
scalb(x,n)
copysign(x,y)
logb(x)
finite(x)
```

Required kernel function:

```
log__L(s)
```

Method (Due to Dr. K.C. Ng, UCB) :

1. Argument Reduction: find k and f such that

   $$x = 2^k * (1+f),$$

   where $\sqrt{2}/2 < 1+f < \sqrt{2}$.

2. Let $s = f/(2+f)$; based on $\log(1+f) = \log(1+s) - \log(1-s)$
   $= 2s + 2/3\ s^3 + 2/5\ s^5 + ...,$
   log(1+f) is computed by

   $$\log(1+f) = 2s + s*\log\_\_L(s)$$

   where
   log__L(s) appoximates $(\log(1+f)-2s)/s$.

3. Finally, $\log(x) = k*\log2 + \log(1+f)$. (k*log2 will be stored
   stored in two floating point number: k * log2hi + k * log2lo,
   k * log2hi is exact since the last 17 bits of log2hi are 0.)

Special cases:

log(x) is NAN with signal if x < 0 (including -INF) ;
log(+INF) is +INF; log(0) is -INF with signal;
log(NAN) is that NAN with no signal.

Accuracy:

log(x) returns the exact log(x) nearly rounded. In a test run with
288,000 random arguments, the maximum observed error was 0.82 ulps.

Implementation:

```
                <----------  hex  --------->
LOG2HI  =  2 ** -0001 * 1.62e4 2fef a3a0 0000  =  hi  part log 2
LOG2LO  = -2 ** -0031 * 1.0ca8 6c38 98cf f81a  =  low part log 2
SQRT2   =  2 **  0000 * 1.6a09 e667 f3bc c908  =  sqrt 2

   if  finite(X)  then
   {
       if  X > 0   then
       {
```

Perform the argument reduction.

```
           k := logb(X) ;
```

```
            x := scalb(X, -k) ;
            if  k = -16383  then  ... X is subnormal
            {
                n := logb(x) ;
                x := scalb(x,-n) ;
                k := k + n ;
            }
            if  x >= SQRT2  then
            {
                k := k + 1 ;
                x := x * 0.5 ;
            }
            x := x - 1 ;


    Compute  log(1+x)  and return.

            s := x / (2 + x) ;
            t := x * x * 0.5 ;
            z := k * LOG2LO + s * (t + log__L(s));
            log(X) :=
                 k * LOG2HI + (x + (k * LOG2LO + s * (t + log__L(s)) - t)) ;
        }


        else  ... X is finite but non-positive
            log(X) := -1 / 0  if  X = 0 ,  else
                   :=  0 / 0 ;  ... NaN with invalid signal for  X < 0

    else  ... X is NaN or INF
        log(X) :=  0 / 0  if  X NOT(?>=) 0 ,  else
               :=  X ;  ... +INF or NaN
```

L O G 1 P (X) := T H E L O G A R I T H M O F 1 + X (base e)
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions:
```
scalb(x,n)
copysign(x,y)
logb(x)
finite(x)
```

Required kernel function:
```
log__L(s)
```

Method (due to Dr. K.C. Ng, UCB) :

1. Argument Reduction: find k and f such that
   1 + x = 2 ^ k * (1+f),
   where 0.5 sqrt 2 < 1+f < sqrt 2 . See remarks (i & iii).

2. Let s = f /(2+f) ; based on log(1+f) = log(1+s) - log(1-s)
   = 2s + 2/3 s**3 + 2/5 s**5 + ...,
   log(1+f) is computed by

   log(1+f) = 2s + s*log__L(s)

   where
   log__L(s) approximates (log(1+f)-2s)/s .

3. Finally, log(1+x) = k * log 2 + log(1+f). See remark (ii).

Remarks

(i) f may not be representable. A correction term c for f is computed. It follows that the correction term for f - t , the leading term of log(1+f) , is c - c * x . We add this correction term to k * (low part of log 2) to compensate the error.

(ii) k * log 2 will be represented as the sum of two floating point numbers
   k * (high part of log 2) + k * (low part of log 2) ,
where (high part of log 2) is chosen with enough trailing zeros (bits) so that
   k * (hi part of log 2)
is exactly representable; for compatibility with other architectures, at least two more than the width of the widest exponent field is used for the number of trailing zeros.

(iii) To compute log1p(2x) , even when 2x overflows, a special entry log1p_r7 into the the log1p code is used. The entry permits k to be incremented by one after the argument reduction.

Special cases:

```
     log1p(x) is NaN with signal if x < -1; log1p(NaN) is NaN ;
     log1p(INF) is +INF; log1p(-1) is -INF with signal;
     only log1p(0)=0 is exact for finite arguments.

Accuracy:
     log1p(x) returns the exact log(1+x) nearly rounded. In a test run
     with 288K random arguments, the max. observed error was 0.82 ulps.

Implementation:       <----------  hex  --------->
     LOG2HI  =  2 ** -0001 * 1.62e4 2fef a3a0 0000  =  hi  part log 2
     LOG2LO  = -2 ** -0031 * 1.0ca8 6c38 98cf f81a  =  low part log 2
     SQRT2   =  2 **  0000 * 1.6a09 e667 f3bc c908  =  sqrt 2

        if finite(X) then
        {
            if X > -1 then
            {

     Perform the argument reduction.

     Save the sticky flags; save the trap enables;
     lower the sticky inexact flag; leave the inexact trap as is;
     disable all other traps.

                k := logb(1 + X) ;
                z := scalb(X, -k) ;
                t := scalb(1, -k) ;
                if z + t >= SQRT2 then
                {
                    k := k + 1 ;
                    z := z * 0.5 ;
                    t := t * 0.5 ;
                }

     At this point, modify the assembly code so that  k  is
     incremented by one if the entry is by  log1p_r7 .

                t := t - 1 ;
                x := z + t ;


     Compute the correction term for  x .

                z := z + (t - x) ;


     Return  log(1 + X) .

                s := x / (2 + x) ;
                t := x * x * 0.5 ;
                z := s * (t + log__L(s)) + (z + (k * LOG2LO - z * x)) ;

     Restore the saved flags or'ed with the sticky
     inexact flag upon return; restore the trap enables.

                log1p(X) := k * LOG2HI + (x + (z - t)) ;

            }  ... end of X > -1

            else  ... finite(X)  and  X =< -1
                log1p(X) := -1/0  if  X = -1 ,  else
                         :=  0/0 ;
```

```
        }  ... end of finite(X)

        else  ... X is NaN or INF
            log1p(X) := 0/0  if  X NOT(?>=) 0 , else
                     := X ;  ... +INF or NaN
```

L O G _ _ L (s) returns (log(1+x)-2s)/s , where s = x/(2+x) and
IEEE double extended precision (64 bits) |x| =< sqrt(2) - 1 .

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Method :

1. Save the divide-by-zero's sticky flag and trap status; disable its trap.

2. Using a continued fraction approximation based on

$$\log(1+x) = 2\,\operatorname{atanh} s\,, \quad \text{where} \quad s = x/(2+x),$$

(log(1+x)-2s)/s is approximated by

$$\cfrac{2}{z + \cfrac{a_3}{z + a_2 + \cfrac{a_5}{z + a_4 + \cfrac{a_7}{z + a_6 + \cfrac{}{z + a_8}}}}}$$

where $z = 3 / s^2$.

3. Restore the divide-by-zero's sticky flag and trap status.

Accuracy:

Assuming no rounding error, the maximum magnitude of the approximation error (absolute) is 2**(-79.32) .

Implementation:

```
A( 2) = -2 **  0000 * 1.cccc cccc cccc cd98 =~ -9/5       =~ -1.8
A( 3) = -2 ** -0001 * 1.3bfa 2608 c6e8 0050 =~ -108/175   =~ -0.62
A( 4) = -2 **  0000 * 1.8888 8888 9f56 de96 =~ -23/15     =~ -1.5
A( 5) = -2 ** -0001 * 1.2786 d548 7541 7322 =~ -400/693   =~ -0.58
A( 6) = -2 **  0000 * 1.8348 5aea 5e37 05e8 =~ -59/39     =~ -1.5
A( 7) = -2 ** -0001 * 1.2360 4356 c206 1f38 =~ -5292/9295=~ -0.56
A( 8) = -2 **  0000 * 1.87f6 19f9 e8a2 8cd8 =~ -333/221   =~ -1.5
```

Save the divide-by-zero's sticky flag and trap status; disable the trap.

```
z := 3 / (s * s) ;
```

Restore the divide-by-zero's sticky flag and trap status upon return.

```
log__L(s) :=  2 / (A(2)+A(3)/(A(4)+A(5)/(A(6)+A(7)/(A(8)+z)+z)+z)+z) ;
```

L O G 1 0 (X) := T H E L O G A R I T H M O F X (B A S E 1 0)
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required kernel function:
log(x)

Method:

$$\log10(x) = \frac{\log x}{\log 10}$$

Note:
[log(10)] rounded to 64 bits has error 1/16 ulps,
[1/log(10)] rounded to 64 bits has error 3/16 ulps;
therefore, for better accuracy, division is preferred
over multiplication.

Special cases:
log10(x) is NAN with signal if x < 0;
log10(+INF) is +INF with no signal; log10(0) is -INF with signal;
log10(NAN) is that NAN with no signal.

Accuracy:
log10(x) returns the exact log10(x) nearly rounded. In a test run
with ??? random arguments, the maximum observed error was ??? ulps.

Implementation:
LOG10 = 2 ** 0001 * 1.26bb 1bbb 5551 582e = log 10

log10(x) := log(x) / LOG10 ;

A S I N H (X) := A R C H Y P E R B O L I C S I N E O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
```
    copysign(x,y)
    fabs(x)
    sqrt(x)
    log1p(x)        ... log(1+x)
```

Method:
```
            s  :=  copysign(1,x);
            z  :=  | x |;
            t  :=  1/z + \/1 + (1/z)^2  ignoring under/overflow and /0;
    asinh(x)  :=  s * log1p(2z)  if  t = 1 ,  else
              :=  s * log1p(z + z / t)  ignoring underflow.
```
To compute log1p(2z) , even when 2z overflows,
a special entry log1p_r7 into the log1p code is used.

The entry permits k to be incremented by one
after the argument reduction
$$1 + z = 2 \wedge k * (1+f) ,$$ where $$\sqrt{1/2} < 1+f < \sqrt{2} ,$$
occurs in log1p .

Special cases:
```
    asinh(x)   is NaN with invalid exception for  x < 1 ;
    asinh(NaN) is NaN.
```

Accuracy:
ASINH has not been proven monotonic; however, it is if log1p is.
ASINH obeys ATRIGH(x) := atrigh(x) nearly rounded ;

In a test run with ??? random arguments, the maximum observed
error was ???1.58 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
After the input argument has been referenced,
save the sticky flags; save the trap enables;
lower the sticky inexact flag; leave the inexact trap as is;
disable all other traps.

```
            s := copysign(1,x) ;
            z := fabs(x) ;
            t := 1/z + sqrt(1 + (1/z)^2) ;
    asinh(x) := s * log1p_r7(z,1)  if  t = 1 ,  else
             := s * log1p(z + z / t) ;
```

Before calling log1p or log1p_r7 , restore the saved flags or'ed with

the sticky inexact flag; restore the trap enables.

A C O S H (X) := A R C H Y P E R B O L I C C O S I N E O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
sqrt(x)
log1p(x) ... log(1+x)

Method:
acosh(x) := +log1p(2x) if x - 1 == x , else

:= +log1p($\sqrt{x - 1}$ * ($\sqrt{x - 1}$ + $\sqrt{x + 1}$)) .

To compute log1p(2x) , even when 2x overflows,
a special entry log1p_r7 into the log1p code is used.

The entry permits k to be incremented by one
after the argument reduction
1 + x = 2 ^ k * (1+f) , where $\sqrt{1/2}$ < 1+f < $\sqrt{2}$,
occurs in log1p .

Special cases:
acosh(x) is NaN with invalid exception for x < 1 ;
acosh(NaN) is NaN.

Accuracy:
ACOSH has not been proven monotonic; however, it is if log1p is.
ACOSH obeys ATRIGH(x) := atrigh(x) nearly rounded ;

In a test run with ??? random arguments, the maximum observed
error was ???3.20 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:

```
acosh(x) := log1p_r7(x,1)  if  x - 1 = x ,  else
         := log1p(sqrt(x - 1) * (sqrt(x - 1) + sqrt(x + 1))) ;
```

A T A N H (X) := A R C H Y P E R B O L I C T A N G E N T O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
```
copysign(x,y)
fabs(x)
log1p(x)    ... log(1 + x)
```

Method:
```
        z := | x |;
        s := copysign(1,x) ... = +-1;
atanh(x) := s * log1p(2 * z / (1 - z)) / 2 .
```

Special cases:
```
atanh(x)   is NaN with invalid exception for  | x |  >  1 ;
atanh(NaN) is NaN;
atanh(+-1) is +-INF with /0 exception.
```

Accuracy:
ATANH has not been proven monotonic; however, it is if log1p is.
ATANH obeys ATRIGH(x) := atrigh(x) nearly rounded ;

In a test run with ??? random arguments, the maximum observed error was ???1.45 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
```
    s := copysign(1/2, x) ;
    z := fabs(x) ;
    atanh(x) := s * log1p((z / (1 - z)) * 2);
```

Make sure the division occurs before the doubling to prevent a spurious overflow when twice z would otherwise overflow.

E X P (X) := T H E E X P O N E N T I A L O F X .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions:

```
fabs(x)
fscalb(x,an) ... scalb for floating integers  an
finite(x)
fint(x)      ... round to floating integer
frem(x,y)    ... x REM y
```

Kernel function:

```
exp__E(z,c)  ... exp(r) - 1 - r , where  r = z + c
```

Method:

1. Argument Reduction: given the input x , find r and integer k such that

   $$x = k \log 2 + r, \quad |r| \le 0.5 \log 2 .$$

   r will be represented by z + c for better accuracy.

2. Compute E(r) = exp(r) - 1 by

   $$E(r=z+c) := z + exp\_\_E(z,c)$$

3. exp(x) := 2 ^ k * (E(r) + 1) .

Remark

(i) To compute exp(x) / 2 , even when exp(x) overflows, a special entry exp_r7 into the the exp code is used. The entry permits k to be decremented by one prior to the final scaling.

Special cases:

```
exp(INF) is INF, exp(NAN) is NAN;
exp(-INF) = 0;
for finite arguments, only exp(0) = 1 is exact.
```

Accuracy:

exp(x) returns the exponential of x nearly rounded. In a test run with ??? random arguments, the maximum observed error was ??? ulps.

Implementation:

```
                <--------- hex ------------>
LOG2HI  =  2 ** -0001 * 1.62e4 2fef a3a0 0000  =  hi  part log 2
LOG2LO  = -2 ** -0031 * 1.0ca8 6c38 98cf f81a  =  low part log 2
LOGHUGE =  2 **  0e * 1.bb a0 02 = (1 + 5 * 2 ^ (exp. width - 2)) log 2

   if  fabs(x) NOT(?>=) LOGHUGE  then
   {
Argument reduction:  z + c  :=  x REM (LOG2HI + LOG2LO) .
       hi := frem(x, LOG2HI) ;
       k  := fint((x - hi) / LOG2HI) ;  ... keep  k  in floating point
       c  := k * LOG2LO ;
       z  := hi - c ;
```

```
        c  := (hi - z) - c ;
```

Prior to the next addition, save the sticky flags and trap enables,
then disable the underflow and denormalized traps, perform the addition,
then restore the flags and traps to their previous settings.

```
        z := z + exp__E(z, c) ;
        z := z + 1 ;
```

At this point, modify the assembly code so that k is
decremented by 1.0 when the entry is via exp_r7 .

```
        exp(x) := fscalb(z, k) ;  ... return  2^k (E(x) + 1) .
    }
    else if  not finite(x)  then  ... return  2 ^ x .
        exp(x) := fscalb(1, x) ;
    else ... return INF (or 0) and signal overflow (or underflow) & inexact
    {
        z := fint(LOGHUGE / LOG2) ;
        exp(x) := fscalb(1,  z)  if  x > 0 ,  else
               := fscalb(1, -z) ;
    }
```

E X P M 1 (X) := T H E E X P O N E N T I A L O F X , M I N U S O N E
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions:
```
scalb(x,n)
fscalb(x,an) ... scalb for floating integers  an
finite(x)
frem(x,y)
fint(x)      ... round to floating integer
fabs(x)
```

Kernel function:
```
exp__E(z,c)  ... exp(r) - 1 - r , where  r = z + c
```

Method: (Due to Dr. K.C. Ng, UCB)

1. Argument Reduction: given the input x, find r and integer k such that

$$x = k \log 2 + r, \quad |r| \leq 0.5 \log 2 .$$

   r will be represented by $z + c$ for better accuracy.

2. Compute $\mathrm{expm1}(r) := \exp(r) - 1$ by

$$\mathrm{expm1}(z + c) := z + \mathrm{exp\_\_E}(z, c) .$$

$$\mathrm{expm1}(x) := 2^k (\mathrm{expm1}(r) + 1 - 2^{-k}) .$$

Remarks:

1. When $k = 1$ and $z < -0.25$, use the formula

$$\mathrm{expm1}(x) = 2 ((z + 1/2) + \mathrm{exp\_\_E}(z, c))$$

   for better accuracy.

2. To avoid a rounding error in $1 - 2^{-k}$ when k is large, use

$$\mathrm{expm1}(x) = 2^k ((z + (\mathrm{exp\_\_E}(z,c) - 2^{-k})) + 1)$$

   when $k > 64$.

Special cases:
```
expm1(+INF) is +INF;
expm1(-INF) is  -1;
expm1(NAN)  is NAN;
for finite arguments, only  expm1(0) = 0  is exact.
```

Accuracy:
```
expm1(x)  returns the exact  exp(x) - 1  nearly rounded.
In a test run with 144,000 random arguments, the maximum
observed error was  0.769 ulps .
```

Implementation:
```
                <----------hex----------->
LOG2HI  =  2 ** -0001 * 1.62e4 2fef a3a0 0000  = hi  part log 2
LOG2LO  = -2 ** -0031 * 1.0ca8 6c38 98cf f81a  = low part log 2
```

```
LOGHUGE =  2 **  0e * 1.bb 9d 3c  =  5 * 2^(exp. width - 2) log 2

   if  fabs(x) NOT(?>=) LOGHUGE  then
   {

Argument Reduction:  z `+' c  :=  x REM (LOG2HI `+' LOG2LO) ,
and                          k  :=  nearest f.p. integer to  x / LOG2HI .
       z := frem(x, LOG2HI) ;
       c := fint((z - x) / LOG2HI) ;
       k := -c ;   ... keep as a floating point integer.
       c := c * LOG2LO ;
       t := z ;
       z := z + c ;
       c := c + (t - z) ;

Prior to the addition in the  k = 0  case,
save the sticky flags and trap enables, then
disable the underflow and denormalized traps, perform the addition,
then restore the flags and traps to their previous settings.

       expm1(x)
            := z + exp__E(z, c)  if  k = 0 ,  else
            := 2 * ((z + 1/2) + exp__E(z, c))  if  k=1 & z < -1/4 ,  else
            := 2 * ((z + exp__E(z, c)) + 1/2)  if  k=1 & z >= -1/4 ,  else
            := fscalb( (1 - scalb(1,-k)) + (z + exp__E(z, c)) , k )
                                               if  fabs(k) <= 64 ,  else
            := fscalb( ((exp__E(z, c) - scalb(1,-k)) + z) + 1 , k )
                                               if  fabs(k) < 200 ,  else
            := fscalb( (exp__E(z, c) + z) + 1 , k )  if  k > 0 ,  else
            := -1 + LOG2LO ;  ... return  -1  and signal inexact

   }
   else  ... | x |  >=  LOGHUGE
   {
       expm1(x)
            := fscalb(1, x) - 1  if  not finite(x) ,  else
            := -1 + LOG2LO  if  x < 0 , else ... overflow to INF inexactly
            := fscalb(1, fint(LOGHUGE / LOG2HI)) ;
   }

The constants  64  and  200  are, respectively, the precision and
thrice the precision plus slop.
```

E X P __ E (X , C) returns exp(x + c) - 1 - x ,
where |x| < 0.5 log 2 and |c| < 0.5 ulp of x ,
ignoring all exceptions except INEXACT.

IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Method:

1. Save the sticky flags; save the trap enables; lower the sticky inexact flag; leave the inexact trap as is; disable all other traps.

2. Using a continued fraction approximation based on

$$\exp(x) - 1 = 2 / (\coth(x/2) - 1)$$

and

$$\tanh(x/2) = x/2 - (x/2) / CF(12/x^2) ,$$

exp(x+c)-1-x is computed by

$$x*x/2 + (c + x\,(c + \frac{(x/2)W - (1+x/2)/CF}{1 - W})) ,$$

where $W = (x/2) - (x/2) / CF = \tanh(x/2)$.

The continued fraction CF is approximated by

$$z + a_2 + \cfrac{a_3}{\cfrac{a_5}{z + a_4 + \cfrac{a_6}{z + a_2}}}$$

where $z = 12 / x^2$.

3. Restore the saved flags or'ed with the sticky inexact flag; restore the trap enables.

Approximation error:

$$\left| \frac{\exp(x) - 1}{x} - (\text{exp\_\_E}(x,0)+x)/x \right| <= 2^{-74}, \text{ (IEEE extended)}$$

Implementation:

A2 = 2 ** 0000 * 1.3333 3333 3333 37be =~ 6/5 =~ 1.2
A3 = -2 ** -0006 * 1.18de 5ab2 7b17 54e6 =~ -3/175 =~ -0.017
A4 = 2 ** -0003 * 1.1111 1126 ddd8 6ed0 =~ 2/15 =~ 0.13
A5 = -2 ** -000a * 1.7a4a 86b7 ff7d 9cda =~ -1/693 =~ -0.0014

```
A6 =  2 ** -0005 * 1.b174 1997 d7a4 3a80  =~  2/39  =~  0.053
```

After the input argument has been referenced,
save the sticky flags; save the trap enables;
lower the sticky inexact flag; leave the inexact trap as is;
disable all other traps.

X and x are different variables. In fact, x is half X .

```
   x := X ;
   x := x * (1/2) ;
   xX := x * X ;
   cf := 6 / xX ;
   cf := A(2)+A(3)/(A(4)+A(5)/(A(6)+cf)+cf)+cf ;
   w  := x - x / cf ;
   exp(X, c)
      := -(-xX + (-c - X * (c + (x * w - (1 + x) / cf) / (1 - w)))) ;
```

Restore the saved flags or'ed with the sticky
inexact flag upon return; restore the trap enables.

S I N H (X) := H Y P E R B O L I C S I N E O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
copysign(x,y)
fabs(x)
exp(x)
expm1(x) ... exp(x) - 1 ; abbreviated as E(x)

Method:

$$z := |x|;$$
$$s := \mathrm{copysign}(1,x) \ldots = \pm 1;$$
$$\sinh(x) := s * \left(E(z) + \frac{E(z)}{1 + E(z)}\right) / 2 \quad \text{if } z < \log(2^{64}+1) \text{ else,}$$
$$:= s * \exp(z) / 2 \quad \text{provided } \exp(z) \text{ doesn't overflow.}$$

To compute exp(z) / 2 , even when exp(z) overflows,
a special entry exp_r7 into the exp code is used.

The entry permits k to be decremented by one prior
to the final scaling
exp(x) := 2 ^ k * (E(r) + 1)
occuring in exp .

Special cases:
sinh(non-finite) is that non-finite;
sinh(x) is exact only for x = 0 and non-finite x .

Accuracy:
SINH has not been proven monotonic; however, it is if expm1 is.
SINH obeys TRIGH(x) := trigh(x) nearly rounded ;

In a test run with ??? random arguments, the maximum observed
error was ???1.93 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
LOG2_64 = 2 ** 05 * 1.62 e4 30 = float ceiling log(2^64+1)

halfs := copysign(0.5, x) ;
z := fabs(x) ;
sinh(x)
:= (expm1(z) / (expm1(z) + 1) + expm1(z)) * halfs
if z NOT(?>=) LOG2_64 , else
:= 2 * (exp_r7(z, 1) * halfs) ;

C O S H (X) := H Y P E R B O L I C C O S I N E O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
fabs(x)
exp(x)

Method:
z := | x |;
cosh(x) := 0.5 exp(z) + 0.25 / (0.5 exp(z)) ignoring underflow and denormalized during the divide and add;
To compute 0.5 exp(z) , even when exp(z) overflows,
a special entry exp_r7 into the exp code is used.

The entry permits k to be decremented by one prior
to the final scaling
exp(x) := 2 ^ k * (E(r) + 1)
occuring in exp .

Special cases:
cosh(NaN) is NaN ;
cosh(INF) is | INF | ;
cosh(x) is exact only for x = 0 and non-finite x .

Accuracy:
COSH has not been proven monotonic; however, it is if exp is.
COSH obeys TRIGH(x) := trigh(x) nearly rounded;

In a test run with ??? random arguments, the maximum observed
error was ???1.23 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:

```
z := fabs(x) ;
z := exp_r7(z, 1);
```

Prior to the next divide, save the sticky flags and trap enables;
lower the overflow and inexact sticky flags; leave their traps as is;
disable all other traps.

```
cosh(x) := (1/4) / z + z ;
```

Upon return, restore the saved flags or'ed with the overflow and
inexact sticky flags; restore the trap enables.

T A N H (X) := H Y P E R B O L I C T A N G E N T O F X
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
```
copysign(x,y)
fabs(x)
expm1(x)    ... exp(x) - 1
```

Method:
```
        z := | x |;
        s := copysign(1,x) ... = +-1;
tanh(x) := -s * expm1(-2z) / (2 + expm1(-2z))  ignoring overflow.
```

Special cases:
```
tanh(NaN) is NaN;
tanh(x)   is exact only for  | x | = 0, INF .
```

Accuracy:
TANH has not been proven monotonic; however, it is if expm1 is.
TANH obeys TRIGH(x) := trigh(x) nearly rounded ;

In a test run with ??? random arguments, the maximum observed error was ???2.22 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:

```
    s := -copysign(1, x) ;  ... single precision  s
```

Prior to the next doubling, save the overflow sticky flag and trap; disable the trap; perform the doubling; then restore the saved settings.

```
    t := expm1(-2*fabs(x)) ;
    tanh(x) := s * t / (2 + t) ;
```

S I N (X) := T H E S I N E O F X R A D I A N S
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
```
frem(x,y)
fint(x)      ... round to floating integer
```

Required kernel function:
```
tan__T(x) := 2 tan(x / 2)  ... abbreviated as T(x)
```

Method:

1. theta := x REM [pi/2] , where [pi/2] is pi/2 rounded to 64 bits;
   n := the least significant two bits of the quotient, signed.

2. Sine or cosine of theta can be calculated from t := T(theta) fairly accurately for all | theta | =< pi/4 by using the following procedure:

```
t := T(theta);  q := t * t;  sin(theta) := t - t /(1+4/q);
if  q =< 4/15
      then  cos(theta) := 1 - 2/(1+4/q);
      else  cos(theta) := 3/4 + ((1-2q) + q/4)/(4+q);
```

3. Using (1) and (2), sine or cosine of x is computed by:

n	sin(x)	cos(x)
n = -3 or 1	cos(theta)	-sin(theta)
n = -2 or 2	-sin(theta)	-cos(theta)
n = -1 or 3	-cos(theta)	sin(theta)
n = 0	sin(theta)	cos(theta)

provided just prior to executing "q := t * t" you

(i) Save the sticky flags; save the trap enables; lower the sticky inexact flag; leave the inexact trap as is; disable all other traps.

and you

(ii) Restore the saved flags or'ed with the sticky inexact flag; restore the trap enables.

upon return.

Special cases:
```
sin(INF) is NaN and invalid exception;
sin(NaN) is NaN;
sin(x)   is exact only for representable multiples of  [pi]/4 ,
         i.e.  | x | =  2**n * [pi]/4  &  0 .
```

```
Accuracy:
     SIN(x) returns  sin(x)  to within 1.9 ulps according to a test run
            with 320,000 random arguments.
     SIN(x) is provably monotonic.
     SIN(x) obeys    TRIG(x) := trig(x*pi/[pi])  nearly rounded ,
            where    pi =  2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
                    [pi]=  2 ** 2 * .c90f daa2 2168 c235 .

References:
     Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
     On the Monotonicity of Some Computed Functions,  W. Kahan.

Implementation:
     TWOPI      =  2 **  0002 * 1.921f b544 42d1 846a  =  2[pi]
     HALFPI     =  2 **  0000 * 1.921f b544 42d1 846a  =  [pi]/2

     Since it is implementation dependent how the remainder operation
     returns the least significant few bits of the quotient, the double
     REM trick from the August 1984 issue of IEEE Micro, p.92 , is used to
     obtain      t := x REM [pi/2] ,
     and         q := the least significant two bits of the quotient, signed .
        q := frem(x, TWOPI) ;
        t := frem(q, HALFPI) ;
        q := fint((q - t) / HALFPI) ;

        t := tan__T(t) ;

     Save the sticky flags; save the trap enables;
     lower the sticky inexact flag; leave the inexact
     trap as is; disable all other traps.

        n := q  truncated to an integer ;
        q := t * t ;
        sin(x) := 1 - 2 / (1 + 4 / q)  if  n = -3 or 1  and  q <= 4/15 ,  else
               := 3/4 + ((1 - 2*q) + q/4) / (4 + q)  if  n = -3 or 1 ,  else
               := t / (1 + 4 / q) - t  if  n = -2 or 2 ,  else
               := 2 / (1 + 4 / q) - 1  if  n = -1 or 3  and  q <= 4/15 ,  else
               := -(3/4 + ((1 - 2*q) + q/4) / (4 + q))  if  n = -1 or 3 ,  else
               := t - t / (1 + 4 / q)  if  n = 0 ;

     Upon return, restore the saved flags or'ed with the
     sticky inexact flag;  restore the trap enables.
```

C O S (X) := T H E C O S I N E O F X R A D I A N S
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
```
frem(x,y)
fint(x)      ... round to floating integer
```

Required kernel function:
```
tan_T(x) := 2 tan(x / 2)  ... abbreviated as T(x)
```

Method:

1. theta := x REM [pi/2] , where [pi/2] is pi/2 rounded to 64 bits;
   n := the least significant two bits of the quotient, signed.

2. Sine or cosine of theta can be calculated from t := T(theta) fairly accurately for all | theta | =< pi/4 by using the following procedure:

```
t := T(theta);  q := t * t;  sin(theta) := t - t /(1+4/q);
if  q =< 4/15
      then  cos(theta) := 1 - 2/(1+4/q);
      else  cos(theta) := 3/4 + ((1-2q) + q/4)/(4+q);
```

3. Using (1) and (2), sine or cosine of x is computed by:

n	sin(x)	cos(x)
n = -3 or 1	cos(theta)	-sin(theta)
n = -2 or 2	-sin(theta)	-cos(theta)
n = -1 or 3	-cos(theta)	sin(theta)
n = 0	sin(theta)	cos(theta)

provided just prior to executing "q := t * t" you

(i) Save the sticky flags; save the trap enables; lower the sticky inexact flag; leave the inexact trap as is; disable all other traps.

and you

(ii) Restore the saved flags or'ed with the sticky inexact flag; restore the trap enables.

upon return.

Special cases:
```
cos(INF) is NaN and invalid exception;
cos(NaN) is NaN;
cos(x)   is exact only for representable multiples of  [pi]/4 ,
         i.e.  | x | =  2**n * [pi]/4  &  0 .
```

```
Accuracy:
     COS(x) returns  cos(x)  to within 1.9 ulps according to a test run
            with 320,000 random arguments.
     COS(x) is provably monotonic.
     COS(x) obeys    TRIG(x) := trig(x*pi/[pi])  nearly rounded ,
            where    pi =  2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
                    [pi]=  2 ** 2 * .c90f daa2 2168 c235 .

References:
     Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
     On the Monotonicity of Some Computed Functions,  W. Kahan.

Implementation:
     TWOPI      =  2 **  0002 * 1.921f b544 42d1 846a  =  2[pi]
     HALFPI     =  2 **  0000 * 1.921f b544 42d1 846a  =  [pi]/2

     Since it is implementation dependent how the remainder operation
     returns the least significant few bits of the quotient, the double
     REM trick from the August 1984 issue of IEEE Micro, p.92 , is used to
     obtain      t := x REM [pi/2] ,
     and         q := the least significant two bits of the quotient, signed .
        q := frem(x, TWOPI) ;
        t := frem(q, HALFPI) ;
        q := fint((q - t) / HALFPI) ;

        t := tan__T(t) ;

     Save the sticky flags; save the trap enables;
     lower the sticky inexact flag; leave the inexact
     trap as is; disable all other traps.

        n := q  truncated to an integer ;
        q := t * t ;
        cos(x) := t / (1 + 4 / q) - t  if  n = -3 or 1 ,  else
               := 2 / (1 + 4 / q) - 1  if  n = -2 or 2  and  q <= 4/15 ,  else
               := -(3/4 + ((1 - 2*q) + q/4) / (4 + q))  if  n = -2 or 2 ,  else
               := t - t / (1 + 4 / q)  if  n = -1 or 3 ,  else
               := 1 - 2 / (1 + 4 / q)  if  n = 0  and  q <= 4/15 ,  else
               := 3/4 + ((1 - 2*q) + q/4) / (4 + q)  if  n = 0 ;

     Upon return, restore the saved flags or'ed with the
     sticky inexact flag;  restore the trap enables.
```

T A N (X) := T H E T A N G E N T O F X R A D I A N S
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
frem(x,y) ... x REM y

Required kernel function:
tan__T(x) := 2 tan(x / 2) ... abbreviated as T(x)

Method:
1. h := x REM [pi] , where [pi] is pi rounded to 64 bits.

2. If | h | < [pi]/8
then return tan := T(2h)/2 ;
If | h | >= 3[pi]/8
then return tan := 2/T([pi]sign(h)-2h)
else
t := T(2 | h | - [pi]/2) ;
return tan := sign(h)(2+t)/(2-t) .

Special cases:
tan(inf) is NaN with invalid flag raised;
invalid trap taken, if enabled;
tan(NaN) is NaN;
tan(x) is exact only for representable multiples of [pi]/4 ,
i.e. | x | = 2**n * [pi]/4 & 0 .

Accuracy:
TAN(x) returns tan(x) to within ???1.5 ulps.
TAN(x) is provably monotonic.
TAN(x) obeys TRIG(x) := trig(x*pi/[pi]) nearly rounded,
where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

Tests:
TAN's worst observed error on -[pi]/2 to [pi]/2 was ??? ulps
for ??? random arguments.

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
```
PI      =  2 **  0001 * 1.921f b544 42d1 846a  =  [pi]
PIOVER2 =  2 **  0000 * 1.921f b544 42d1 846a  =  [pi]/2
PIOVER8 =  2 ** -0002 * 1.921f b544 42d1 846a  =  [pi]/8
```

```
t := frem(x, PI) ;
```

Save the sticky flags and trap enables just prior to the divide,
then disable the integer overflow trap and the underflow trap, then

restore the flags and traps immediately after the convert to integer.

```
   n :=  t / PIOVER8  truncated to an integer ;

   tan(x) := 2 / tan__T(-(PI+2*t))  if  n = -4 or -3 ,  else
          := -(2 + tan__T(-2*t-PIOVER2)) / (2 - tan__T(-2*t-PIOVER2))
                                    if  n = -2 or -1 ,  else
          := (2 + tan__T(2*t-PIOVER2)) / (2 - tan__T(2*t-PIOVER2))
                                    if  n = 2 or 1 ,  else
          := tan__T(2*t) * 0.5  if  n = 0 ,  else
          := 2 / tan__T(PI-2*t)  if  n = 3 or 4 ;
```

T A N __ T (X) := 2 T A N (X / 2) , where |x| =< [pi]/4 .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Method:

1. Save the sticky flags; save the trap enables; lower the sticky inexact flag; leave the inexact trap as is; disable all other traps.
2. $z := a_1^{x_2}$.
3. $$\mathrm{tan\_\_T}(x) := x + \cfrac{x}{z + a_2 + \cfrac{a_3}{z + a_4 + \cfrac{a_5}{z + a_6}}}$$.
4. Restore the saved flags or'ed with the sticky inexact flag; restore the trap enables.

Accuracy:

Assuming no rounding error, the maximum magnitude of the approximation error (absolute) is 2**(-66.14) .
tan__T(x) is provably monotonic.
tan__T(x) obeys TRIG(x) := trig(x*pi/[pi]) nearly rounded,
where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

References:

Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:

A(1) = 2 ** 0003 * 1.8000 0000 0000 021a =~ 12 =~ 12.
A(2) = -2 ** 0000 * 1.3333 3333 3334 7090 =~ -6/5 =~ -1.2
A(3) = -2 ** -0006 * 1.18de 5ab2 5d5b e362 =~ -3/175 =~ -0.017
A(4) = -2 ** -0003 * 1.1111 112f 8c57 78dc =~ -2/15 =~ -0.13
A(5) = -2 ** -000a * 1.7a45 0166 8187 fdfa =~ -1/693 =~ -0.0014
A(6) = -2 ** -0005 * 1.a501 80bf 4236 08c2 =~ -2/39 =~ -0.051

Save the sticky flags; save the trap enables;
lower the sticky inexact flag; leave the inexact trap as is;
disable all other traps.

```
z := A(1) / (x*x) ;
```

Restore the saved flags or'ed with the sticky

```
inexact flag upon return; restore the trap enables.

   tan__T(x) := x + x / (A(2)+A(3)/(A(4)+A(5)/(A(6)+z)+z)+z) ;
```

A S I N (X) := A R C S I N E O F X R A D I A N S .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
atan(x)
fabs(x)
sqrt(x)

Method:

```
                                                  2
If  | x |  ?<=  1 / 2  then     r := 1 - x    ... ignoring underflow
                         else  { y := 1 - | x |  ... exactly ;
                                 r := 2y - y^2   };
```

asin(x) := atan(x / \/ r) ignoring divide-by-zero.

Special cases:
asin(x) is NaN with invalid exception for | x | > 1 .

Accuracy:
ASIN has not been proven monotonic; however, it is if ATAN is.
ASIN obeys ARCTRIG(x) := [pi]/pi*arctrig(x) nearly rounded ,
where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

In a test run with ??? random arguments, the maximum observed error was ???2.06 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
After the input argument has been referenced,
save the sticky flags; save the trap enables;
lower the inexact and invalid sticky flags;
leave the inexact and invalid traps as is; disable all other traps.

```
asin(x) := atan(x / sqrt(1 - x * x))  if  fabs(x) ?<= 1/2 ,  else
        := atan(x / sqrt(2 * y - y * y))  where  y := 1 - fabs(x) ;
```

Before calling atan , restore the trap enables and restore the saved flags or'ed with the inexact and invalid sticky flags.

A C O S (X) := A R C C O S I N E O F X R A D I A N S .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
atan(x)
sqrt(x)

Method:

$$\text{acos}(x) := 2\,\text{atan}\left(\sqrt{\frac{1-x}{1+x}}\right) \quad \text{ignoring divide-by-zero.}$$

Special cases:
acos(x) is NaN with invalid exception for | x | > 1 .

Accuracy:
ACOS has not been proven monotonic; however, it is if ATAN is.
ACOS obeys ARCTRIG(x) := [pi]/pi*arctrig(x) nearly rounded ,
where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

In a test run with ??? random arguments, the maximum observed error was ???2.07 ulps .

References:
Elementary Functions from Kernels, Prof. W. Kahan, U.C.Berkeley
On the Monotonicity of Some Computed Functions, W. Kahan.

Implementation:
After the input argument has been referenced,
save the sticky flags; save the trap enables;
lower the inexact and invalid sticky flags;
leave the inexact and invalid traps as is; disable all other traps.

acos(x) := 2 atan(sqrt((1 - x) / (1 + x))) ;

Before calling atan , restore the trap enables and restore the saved flags or'ed with the inexact and invalid sticky flags.

A T A N (X) := A R C T A N G E N T O F X R A D I A N S .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:
copysign(x,y)
fabs(x)

Method: (Due to Dr. K.C. Ng, U.C.Berkeley)

1. Reduce x to the positive case by atan(-x) = -atan(x) .

2. According to the truncated integer 4(x+1/16) select one of the following intervals and evaluate atan(x) using the corresponding formula.

```
[0,7/16]       atan(x) = x-x/(a2+a3/(a4+a5/(a6+a7/(a8+a9/(a10+z)
                         where  z = a1/x^2          +z)+z)+z)+z)
[7/16,11/16]   atan(x) = atan(1/2) + atan((x-1/2)/(1+x/2))
[11/16,19/16]  atan(x) = atan( 1 ) + atan((x- 1 )/(1+x  ))
[19/16,39/16]  atan(x) = atan(3/2) + atan((x-3/2)/(1+3x/2))
[39/16,INF]    atan(x) = atan(INF) + atan(-1/x)
```

Special cases:
atan(NaN) is NaN;
atan(-0) is -0;
atan(x) is exact only for | x | = 0,1,INF .

Accuracy:
ATAN returns atan(x) to within better than 0.89 ulps, according to an error analysis done by Dr. Ng;
ATAN has not been proved monotonic;
ATAN obeys ARCTRIG(x) := [pi]/pi*arctrig(x) nearly rounded , where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

Tests:
ATAN's worst error on -524,297 to 524,297 was 0.86 ulps for 1,312,000 random arguments. No monotonicity failures occured.

References:
ATAN (for computers that conform to IEEE standard 754)
by Dr. K.C. Ng, U.C. Berkeley.

Implementation:

```
A1  =  2 **  0001 * 1.8000 0000 0000 021c =~  3          =~  3.00
A2  =  2 **  0000 * 1.cccc cccc ccca 81f6 =~  9/5        =~  1.80
A3  = -2 ** -0001 * 1.3bfa 2608 c357 b5f0 =~ -108/175    =~ - .617
A4  =  2 **  0000 * 1.8888 8887 4061 c1d0 =~  23/15      =~  1.53
A5  = -2 ** -0001 * 1.2786 d4d4 f5e8 7498 =~ -400/693    =~ - .577
A6  =  2 **  0000 * 1.8348 1a77 d068 6434 =~  59/39      =~  1.51
A7  = -2 ** -0001 * 1.237a 8123 9828 9f48 =~ -5292/9295  =~ - .569
```

```
A8   =  2 **  0000 * 1.815e 503c 6b5b 4810 =~  333/221      =~  1.51
A9   = -2 ** -0001 * 1.1982 5c9a 5461 7902 =~ -15552/27455 =~ - .566
A10  =  2 **  0000 * 1.4ed9 f09c 4ceb 3d8e =~  179/119      =~  1.50
ATAN12HI =  2 ** -0002 * 1.dac6 7056 1bb4 f68c = [pi]/pi atan(1/2) hi part
ATAN12LO = -2 ** -0043 * 1.28bb 83f3 597a 57ec = [pi]/pi atan(1/2) lo part
PIOVER4  =  2 ** -0001 * 1.921f b544 42d1 846a = [pi]/4
ATAN32HI =  2 ** -0001 * 1.f730 bd28 1f69 b202 = [pi]/pi atan(3/2) hi part
ATAN32LO = -2 ** -0043 * 1.eae0 d654 3812 74c0 = [pi]/pi atan(3/2) lo part
PIOVER2  =  2 **  0000 * 1.921f b544 42d1 846a = [pi]/2
                 <hex>   <------- hex ------->
```

After the input argument has been referenced,
save the sticky flags; save the trap enables;
lower the sticky inexact flag; leave the inexact trap as is;
disable all other traps.

```
   sign := copysign(1,x) ;
   y := fabs(x) ;
   (head,tail,y) := (PIOVER2 ,    0   ,     -1/y      )  if  y>=39/16,else
                 := (   0    ,    0   ,       y       )  if  n = 0,1, else
                 := (ATAN12HI,ATAN12LO,(y-1/2)/(1+y/2))  if  n = 2 ,  else
                 := (PIOVER4 ,    0   ,(y- 1 )/(1+ y ))  if  n = 3,4, else
                 := (ATAN32HI,ATAN32LO,(y-3/2)/(1+3/2*y)) ,
   where n := 4 * (y + 1/16) truncated to an integer ;

   atan(x) := sign * (head + (y + (tail - y / cf(A1/y^2)))) ,  where
     cf(z) := A2+A3/(A4+A5/(A6+A7/(A8+A9/(A10+z)+z)+z)+z)+z ;
```

Restore the saved flags or'ed with the sticky
inexact flag upon return; restore the trap enables.

Note: If truncation to an integer can signal inexact on your system,
disable the inexact trap just prior to the conversion;
immediately afterwards, clear the sticky inexact flag and
restore the inexact trap to its previous setting.

A T A N 2 (Y , X) := A R G (X + I Y) .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions :
copysign(x,y)
scalb(x,n)
logb(x)

Method: (Due to K.C. Ng, U.C.Berkeley)
1. Reduce y to positive case by atan2(y,x) = -atan2(-y,x) .
2. Reduce x to positive case by
ARG (x+iy) = arctan(y/x) ... if x > 0,
ARG (x+iy) = pi - arctan[y/(-x)] ... if x < 0,
provided x and y are unexceptional.
3. According to the truncated integer 4(x+1/16) select one of the following intervals and evaluate atan(x) using the corresponding formula.

```
[0,7/16]      atan(y/x) = x-x/(a2+a3/(a4+a5/(a6+a7/(a8+a9/(a10
                          where  z = a1/x^2      +z)+z)+z)+z)+z)
[7/16,11/16]  atan(y/x) = atan(1/2) + atan( (y-x/2)/(x+y/2) )
[11/16,19/16] atan(y/x) = atan( 1 ) + atan( (y-x)/(x+y) )
[19/16,39/16] atan(y/x) = atan(3/2) + atan( (y-1.5x)/(x+1.5y) )
[39/16,INF]   atan(y/x) = atan(INF) + atan( -x/y )
```

Special cases:
Notations: atan2(y,x) == ARG (x+iy) == ARG(x,y).

ARG(NaN , (anything)) is NaN;
ARG((anything), NaN) is NaN;
ARG(+(anything but NaN), +-0) is +-0 ;
ARG(-(anything but NaN), +-0) is +-PI ;
ARG(0, +-(anything but 0 and NaN)) is +-PI/2;
ARG(+INF,+-(anything but INF and NaN)) is +-0 ;
ARG(-INF,+-(anything but INF and NaN)) is +-PI;
ARG(+INF,+-INF) is +-PI/4 ;
ARG(-INF,+-INF) is +-3PI/4;
ARG((anything but,0,NaN, and INF),+-INF) is +-PI/2;

Accuracy:
ATAN2 has not been proved monotonic;
ATAN2 obeys ARCTRIG(y,x) := [pi]/pi*arctrig(y,x) nearly rounded ,
where pi = 2 ** 2 * .c90f daa2 2168 c234 c4c6 628b ...
[pi]= 2 ** 2 * .c90f daa2 2168 c235 .

In a test run with ??? random arguments on [-1,1] x [-1,1] ,
the maximum observed error was ??? ulps .

References:
ATAN (for computers that conform to IEEE standard 754)
by Dr. K.C. Ng, U.C. Berkeley.

```
Implementation:
     A1  =  2 **  0001 * 1.8000 0000 0000 021c =~  3           =~  3.00
     A2  =  2 **  0000 * 1.cccc cccc ccca 81f6 =~  9/5         =~  1.80
     A3  = -2 ** -0001 * 1.3bfa 2608 c357 b5f0 =~ -108/175     =~ - .617
     A4  =  2 **  0000 * 1.8888 8887 4061 c1d0 =~  23/15       =~  1.53
     A5  = -2 ** -0001 * 1.2786 d4d4 f5e8 7498 =~ -400/693     =~ - .577
     A6  =  2 **  0000 * 1.8348 1a77 d068 6434 =~  59/39       =~  1.51
     A7  = -2 ** -0001 * 1.237a 8123 9828 9f48 =~ -5292/9295   =~ - .569
     A8  =  2 **  0000 * 1.815e 503c 6b5b 4810 =~  333/221     =~  1.51
     A9  = -2 ** -0001 * 1.1982 5c9a 5461 7902 =~ -15552/27455 =~ - .566
     A10 =  2 **  0000 * 1.4ed9 f09c 4ceb 3d8e =~  179/119     =~  1.50
     ATAN12HI =  2 ** -0002 * 1.dac6 7056 1bb4 f68c = [pi]/pi atan(1/2) hi part
     ATAN12LO = -2 ** -0043 * 1.28bb 83f3 597a 57ec = [pi]/pi atan(1/2) lo part
     PIOVER4  =  2 ** -0001 * 1.921f b544 42d1 846a = [pi]/4
     ATAN32HI =  2 ** -0001 * 1.f730 bd28 1f69 b202 = [pi]/pi atan(3/2) hi part
     ATAN32LO = -2 ** -0043 * 1.eae0 d654 3812 74c0 = [pi]/pi atan(3/2) lo part
     PIOVER2  =  2 **  0000 * 1.921f b544 42d1 846a = [pi]/2
     PI       =  2 **  0001 * 1.921f b544 42d1 846a = [pi]
                     <hex>   <------- hex ------->

     After the input argument has been referenced,
     save the sticky flags; save the trap enables;
     leave the inexact trap as is; disable all other traps.

        signy := copysign(1, Y) ;
        signx := copysign(1, X) ;
        x := fabs(X) ;
        y := fabs(Y) ;
        t := y / x ;

     Re-save the sticky inexact flag and lower it.

        if  t != t  then  ... x & y are both infinite (or 0) or one is NaN
            if  x = y  then  ... neither is NaN
                if  x != 0  then  ... both are infinite
                    atan2(Y,X) := signy * PIOVER4  if  signx > 0 ,  else
                                := signy * 3 * PIOVER4 ;
                else  ... both are 0
                    t := 0 ;
            else  ... x or y is NaN
                atan2(Y,X) := t ;

     Rescale y/x to prevent loss of precision near under/overflow threshold.
     We assume the integer  k  can never represent INF or NaN
     in the scalb call.  Other implementations beware!
        k := logb(y) ;
        y := scalb(y, -k) ;
        x := scalb(x, -k) ;

        (head,tail,t) := (PIOVER2 ,    0   ,     -x/y      )  if  t>=39/16,else
                      := (   0    ,    0   ,       t       )  if  n = 0,1, else
                      := (ATAN12HI,ATAN12LO,(2*y-x)/(2*x+y))  if  n = 2 ,  else
                      := (PIOVER4 ,    0   ,(y - x)/(x + y))  if  n = 3,4, else
                      := (ATAN32HI,ATAN32LO,(2*y-3*x)/(2*x+3*y)) ,
        where  n := 4 * (t + 1/16) truncated to an integer ;

        atan2(Y,X)
           := signy * (head + (t + (tail - t / cf(A1/t^2)))) if  signx>0 , else
           := signy * (PI - (head + (t + (tail - t / cf(A1/t^2))))) ,
        where
        cf(z) := A2+A3/(A4+A5/(A6+A7/(A8+A9/(A10+z)+z)+z)+z)+z ;
```

Restore the saved flags or'ed with the sticky
inexact flag upon return; restore the trap enables.

Note: If truncation to an integer can signal inexact on your system,
disable the inexact trap just prior to the conversion;
immediately afterwards, clear the sticky inexact flag and
restore the inexact trap to its previous setting.

P O W (X , Y) := X R A I S E D T O T H E Y P O W E R
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions:

```
scalb(x,n)
logb(x)
copysign(x,y)
finite(x)
frem(x,y)
fabs(x)          ... floating absolute value
fint(x)          ... round to nearest floating integer
fscalb(x,an)     ... scalb for floating integers  an
```

Required kernel functions:

```
exp__E(a,c)      ... return  exp(a+c) - 1 - a*a/2
log__L(x)        ... return  (log(1+x) - 2s)/s, s=x/(2+x)
pow_p(x,y)       ... return  +(anything)^(finite non zero)
```

Method (Due to Dr. K.C. Ng, UCB) :

1. Compute and return log(x) in three pieces:
   log x = n log 2 + hi + lo ,
   where n is an integer.
2. Perform y log(x) by simulating multi-precision arithmetic;
   return the answer in three pieces:
   y log x = m log 2 + hi + lo ,
   where m is an integer.
3. Return x^y = exp(y log x)
   = 2^m * exp(hi + lo) .

Special cases (in decreasing order of precedence):

```
x^0  is  1 ;
x^1  is  x ;
x^y  is  NaN  for  x  or  y  NaN ;
INF  is an even integer;
-0   is a negative integer;
x^y  is  NaN  with invalid exception for
     | x | = 1  and  y  infinite ,  OR
     x  infinite or negative and  y  not an integer;
x^-y has  1 / x^y 's exceptions.
```

Accuracy:

pow(x,y) returns x^y nearly rounded. In particular,
pow(integer,integer)
always returns the correct integer provided it is representable.
In a test run with ??? random arguments from 0 < x,y < 20.0 ,
the maximum observed error was ???1.79 ulps .

Implementation: <---------- hex --------->

```
LOG2HI  =  2 ** -0001 * 1.62e4 2fef a3a0 0000  =  hi  part log 2
LOG2LO  = -2 ** -0031 * 1.0ca8 6c38 98cf f81a  =  low part log 2
SQRT2   =  2 **  0000 * 1.6a09 e667 f3bc c908  =  sqrt 2
```

```
   pow(x, y)

x^0  is  1 .
            := 1  if  y = 0 ,  else

x^y  is  x  for  y = 1  or  x = NaN .
            := x  if  y = 1  or  x != x ,  else

x^NaN  is  NaN .
            := y  if  y != y ,  else

x^y    is  NaN  with invalid exception for  | x | = 1  and  y  infinite;
            := 0/0  if not finite(y) and  fabs(x) = 1 ,  else

x^INF  is  +INF  or  +0  for positive or negative  INF and  | x |  >  1 ;
            := y    if not finite(y) and  fabs(x) > 1  and  y > 0 ,  else
            := 0    if not finite(y) and  fabs(x) > 1  and  y < 0 ,  else

x^INF  is  +0  or  +INF  for positive or negative  INF and  | x |  <  1 .
            := 0    if not finite(y) and  fabs(x) < 1  and  y > 0 ,  else
            := -y   if not finite(y) and  fabs(x) < 1  and  y < 0 ,  else

x^2 = x * x .
            := x * x  if  y = 2 ,  else

x^-1 = 1 / x .
            := 1 / x  if  y = -1 ,  else

x^y = pow_p(x, y)  , if the sign of  x  is `+'.
            := pow_p(x, y)  if  copysign(1, x) > 0 ,  else

x^y = pow_p(-x, y) , if the sign of  x  is `-' and  y  is an even integer.
            := pow_p(-x, y)  if  frem(y, 2) = 0 , else

x^y = -pow_p(-x, y), if the sign of  x  is `-' and  y  is an odd integer.
            := -pow_p(-x, y)  if  fabs(frem(y, 2)) = 1 , else

(-0)^y = +0 or +INF , if finite  y  isn't an integer.
            := -x    if  x = 0  and  y > 0 , else
            := 1/-x  if  x = 0  and  y < 0 , else

x^y = NaN with invalid exception, if the sign of non-zero  x  is `-' and
finite  y  isn't an integer.
            := 0/0 ;


pow_p(x,y) returns  x^y  where the sign of  x  is pos. and  y  is finite.


x^y = +0 or +INF  if  x  is  +INF or +0  and  y  is finite.

   if x = 0 or not finite(x) then
   {
       pow_p(x, y) := x  if  y > 0 ,  else
                   := 1 / x ;
   }

Reduce  x  to  z  in  [sqrt(1/2)-1, sqrt(2)-1] .

   n := logb(x) ;  ... where  n  is a 32-bit integer
   z := scalb(x, -n) ;
```

Handle subnormal numbers.

```
   if n <= -16383 then
   {
       m := logb(z) ;  ... where  m  is a 32-bit integer
       n := n + m ;
       z := scalb(z,-m) ;
   }
```

Finish reducing to the desired range.

```
   if z >= SQRT2 then
   {
       n := n + 1 ;
       z := z * 0.5 ;
   }
   z := z - 1 ;
```

Log x = n log 2 + log(1+z) ~= n log 2 + t + tx .

```
   t := z / (TWO + z) ;
   c := z * z * 0.5 ;
   tx := t * (c + log__L(t)) ;
   t := z - (c - tx) ;
   tx := tx + ((z - t) - c) ;
```

If y log x is neither too big nor too small, do the usual processing.

Save the sticky flags and trap enables before the second logb() call;
disable int overflow and /0 traps; restore everything after the convert.

x^y overflows for the first time (with no possibility
of exponent wrap-around) when

$$\frac{y}{x} \geq 2^{1.25 \cdot 2^{\text{(exponent field width)}}} .$$

Since m = logb(y) + logb(n+t) approximates
log2(y log x) , the test m < (exponent field width) + 1 + 1
is used, where an extra one is added for good measure.

```
   m := logb(y) + logb(n + t) ;
   if m < 17 then
```

x^y rounds to one if y log x < 2**(-precision) ; therefore, the test
m > -(precision + 4) is used, with 4 being added for good measure.

```
       if m > -68 then
       {
```

Compute y log x ~= m log 2 + t + c .

```
           m := fint(y * (n + t / LOG2)) ;

           if y = fint(y) then  ...  y  is exactly an integer
           {
               sx := t ;  ...  sx  is single precision
               tx := tx + (t - sx) ;
               k := m - y * n ;
           }
           else  ...  y  isn't an integer
           {
               tx := tx + n * LOG2LO ;
               c := n * LOG2HI ;
```

```
                sx := c + t ;
                tx := tx + ((c - sx) + t) ;
                k := m ;
            }

Represent  y  as   sy + ty .

            sy := y ;  ...  sy  is single precision
            ty := y - sy ;

Compute  t = (sy + ty) * (sx + tx) - k log 2  carefully.

The product  sx * sy  mustn't be computed in single precision;
instead, compute as  single x single = double  (or extended) .

            s := sx * sy - k * LOG2HI ;  ... compute  sx * sy  exactly
            z := tx * ty - k * LOG2LO ;
            tx := tx * sy ;
            ty := ty * sx ;
            t := ((ty + z) + tx) + s ;

Finally, return  exp(y log x) .

            pow_p(x, y) :=
             fscalb(1 + (t + exp__E(t, -((((t - s) - tx) - ty) - z))), m) ;

        }

        else  ... log2(y log x) =< -68; hence return  x^y = 1  inexactly.
        {
            1 + LOG2LO ;  ... set inexact
            pow_p(x, y) := 1 ;  ... and return
        }
    else ... log2(y log x) >= 17; hence x^y under or overflows to 0 or INF.
        pow_p(x, y)
         := fscalb(1,-50000)  if  copysign(1,y) * (n + t / LOG2) < 0 , else
         := fscalb(1, 50000) ;
```

H Y P O T (R E A L , I M A G) := sqrt(real ^ 2 + imag ^ 2) .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required system supported functions :
fabs(x)
finite(x)
scalb(x,N)
sqrt(x)

Method (Due to Prof. Kahan and Dr. K.C. Ng, UCB):
1. Replace real by | real | and imag by | imag | , and swap real and imag if imag > real (hence real is never smaller than imag).
2. Let X = real and Y = imag ; hypot(X,Y) is computed by:

Case I , X / Y > 2

$$\text{hypot} = X + \frac{Y}{\sqrt{1 + [X/Y]^2} + X/Y}$$

Case II, X / Y =< 2

$$\text{hypot} = X + \frac{Y}{(\sqrt{2}+1) + (X-Y)/Y + \dfrac{[X/Y]^2 - 2}{\sqrt{1 + [X/Y]^2} + \sqrt{2}}}$$

Special cases:
hypot(x,y) is INF if x or y is +INF or -INF; else
hypot(x,y) is NAN if x or y is NAN.

Accuracy:
Hypot(x,y) returns sqrt(x^2+y^2) with error less than 1 ulp , see Kahan's "Interval Arithmetic Options in the Proposed IEEE Floating Point Arithmetic Standard", Interval Mathematics 1980, Edited by Karl L.E. Nickel, pp 99-128. In a test run with ??? random arguments, the maximum observed error was ???.959 ulps.

Implementation: <--------- hex ------------>
R2P1HI = 2 ** 0001 * 1.3504 f333 f9de 6484 = hi part 1+sqrt2
R2P1LO = 2 ** -0041 * 1.65f6 26cd d52a fa7c = low part 1+sqrt2
SQRT2 = 2 ** 0000 * 1.6a09 e667 f3bc c908 = sqrt 2
SMALL = 2 ** -40 * 1.00 00 00 = 2^-64 ... fl(1 + SMALL) = 1
IBIG = 32 ... fl(1 + 2 ^ -(2 IBIG)) = 1

```
if finite(Real) then
   if finite(Imag) then
```

```
        {
           (real, imag) := (fabs(Real), fabs(Imag)) ;
           if (imag > real)
              (real, imag) := (imag, real) ;
           hypot(Real, Imag)
              := 0  if  real = 0 ,  else
              := real  if  imag = 0 ,  else
              := real & "raise inexact" if logb(real)-logb(imag) > IBIG, else
              := real + imag / r ,  where  r  is given below;
        }
        else  ... Imag is NaN or INF
           hypot(Real, Imag)
              := fabs(Imag)  if Imag = Imag ,  else
              := Imag ;  ... Imag is NaN
     else  ... Real is NaN or INF
        hypot(Real, Imag)
           := fabs(Real)  if Real = Real ,  else
           := Real  if finite(Imag) ,  else
           := Imag  if Image != Imag ,  else
           := fabs(Imag) ;

  Compute  r  as follows:

     r := real - imag ;
     if r > imag then  ... real/imag > 2
     {
         r := real / imag ;
         r := r + sqrt(1 + r * r) ;
     }
     else  ... 1 =< real/imag =< 2
     {
         r := r / imag ;
         t := r * (r + 2) ;
         r := ((r + t / (SQRT2 + sqrt(2 + t))) + R2P1LO) + R2P1HI ;
     }
```

F S C A L B (X , FN) := x * 2 ^ fn for floating integers fn .
IEEE double extended precision (64 bits)

WORK IN PROGRESS

Written by Stuart Ian McDonald under direction of Professor William Kahan.
The author's current electronic mail address as of December 1985:
Domain: "mcdonald@renoir.Berkeley.EDU" Path: "ucbvax!renoir!mcdonald"

Required functions:

```
fabs(x)
scalb(x,n)     ... for 16-bit integers  n
copysign(x,y)
finite(x)
```

Method:

1. If the floating point integer fn can be represented as a sixteen bit integer, then an integer scalb is used; otherwise, a flush to tiny * tiny or huge / tiny is performed, respectively, for underflow or overflow and the sign of x is affixed.

Special cases:

```
fscalb(x,NaN)  is  NaN;
fscalb(x,+INF) is  x * +INF
fscalb(x,-INF) is  x * +0
```

Comments:

Ideally, if x * 2 ^ fn can be delivered to the under/overflow trap handler without more than one re-biasing of its exponent range, you should deliver the result; otherwise, you should deliver infinity or zero to the trap handler, as appropriate, with the correct sign.

Since the delivery of non-standard (i.e. user supplied) values to the floating point trap handlers is implementation dependent, flushing to tiny * tiny or huge / tiny is used instead. This has two defects, as discussed below.

First, values of x * 2 ^ fn deliverable with a single exponent re-biasing but not generatable with a multiply or divide instruction are prematurely flushed to tiny * tiny or huge / tiny .

Second, values of x * 2 ^ fn not deliverable with a single exponent re-biasing are indistinguishable from the values delivered for tiny * tiny and huge / tiny . Hence the suggestion to deliver zero and infinity instead.

On Zilog's Z8070 floating point processor, for example, the systems people shall provide a system call for delivery of non-standard values thus:

> First, disable master interrupts by writing to the privileged MIE bit in the Z8070's system configuration register. Second, cause the user requested exception to occur. Third, replace FOP1 with the user's supplied value. Fourth, re-enable master interrupts, causing the CPU to service the interrupt.

Every implementation shall provide a similar mechanism since the IEEE floating point standard 754 requires the delivery of a non-standard value, a NaN, to the under/overflow trap handler when one bias adjustment is not enough during decimal-to-binary conversion; the proposed radix- and word-length-independent standard IEEE P854, furthermore, allows zero or infinity to be delivered instead of NaN.

In short, treat trapped under/overflow during scaling just like trapped under/overflow during decimal-to-binary conversion.

Implementation:

```
TINY =  2 ** -3fff * 1.0000 0000 0000 0000 = smallest positive normal
HUGE =  2 **  3fff * 1.ffff ffff ffff fffe = largest finite

   fscalb(x, fn)
     := scalb(x, (int)fn)  if  fabs(fn) NOT(?>=) 2^15 ,  else
     := TINY * copysign(TINY,x)  if finite(x) & finite(fn) & fn < 0, else
     := copysign(HUGE,x) / TINY  if finite(x) & finite(fn) & fn >= 0, else
     := x * 0  if  fn = -INFINITY , else
     := x * fabs(fn) ;
```