***** This is only a draft. The algorithms listed in section B haven't been
***** tested thoroughly. Please don't redistribute it.
***** Authors: W. Kahan and K.C. Ng
***** Date: 5/6/86

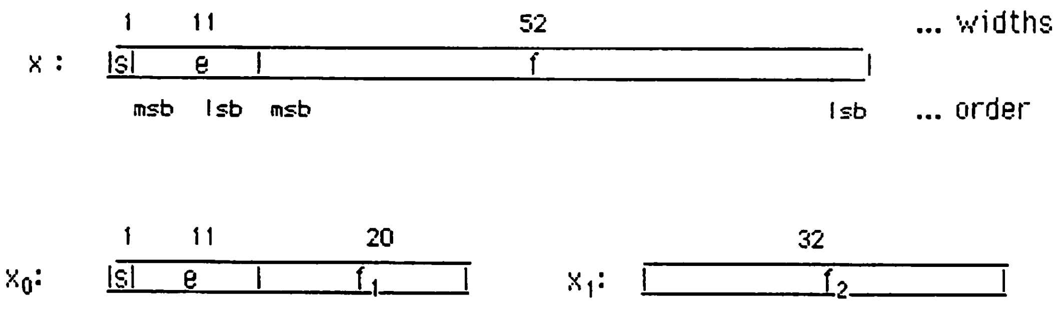
SQRT

Two algorithms are given in this document to implement $\sqrt{x}$ in software. Both supply $\sqrt{x}$ correctly rounded. The first alogirthm (in Section A) uses newton iterations and involves four divisions. The second one uses reciproot iterations to avoid division, but requires more multiplications. Both algorithms need the ability to chop results of arithmetic operations instead of round them, and the INEXACT flag to indicate when an arithmetic operation is executed exactly with no roundoff error, all part of the standard. The ability to perform shift, add, subtract and logical AND operations upon 32-bit words is needed too, though not part of the standard.

A. $\sqrt{X}$ by Newton Iteration

(1). Initial approximation

Let x_0 and x_1 be the leading and the trailing 32-bit words of a floating point number x (in IEEE double format) respectively (cf section B of REQUIRED SUPPORT FUNCTIONS):



By performing shifts and subtracts on x_0 and x_1 (both regarded as integers), we obtain an 8-bit approximation of $\sqrt{x}$ as follows.

```
    k    := (x0>>1)+0x1ff80000;
    y0   := k - T1[31&(k>>15)].       ...y ≐ √x to 8 bits
```

Here k is a 32-bit integer and T1[ ] is an integer array (see section C for its values) containing correction terms. Now magically the floating value of y (y's leading 32-bit word is y_0, the value of its trailing word y_1 is unimportant) approximates $\sqrt{x}$ to almost 8-bit.

(2) Iterative refinement

Apply Heron's rule three times to y, we have y approximates $\sqrt{x}$ to within 1 ulp (Unit in the Last Place):

$y := (y+x/y)/2$ ... almost 17 sig. bits
$y := (y+x/y)/2$ ... almost 35 sig. bits
$y := y-(y-x/y)/2$ ... within 1 ulp

Remark 1. Another way to improve y to within 1 ulp is:

$y := (y+x/y);$ ... almost 17 sig. bits to $2\sqrt{x}$
$y_0 := y_0$ - 0x00100006 ... almost 16 sig. bits to $\sqrt{x}$

$$y := y + 2*\frac{(x-y^2)*y}{3y^2+x}$$ ... within 1 ulp

This formula has one division fewer than the one above; however, it requires more multiplications and additions. Also x must be scaled in advance to avoid spurious overflow in evaluating the expression $3y^2+x$. Hence it is not recommended unless division is slow. If division is very slow, then one should use the reciproot algorithm given in section B.

(3) Final adjustment

By twiddling y's last bit it is possible to force y to be correctly rounded according to the prevailing rounding mode as follows. Let r and i be copies of the rounding mode and inexact flag before entering the square root program. Also we use the expression y±ulp for the next representable floating numbers (up and down) of y. Note that y±ulp = either fixed point y ±1 , or multiply y by nextafter(1,±∞) in chopped mode.

```
I := FALSE;       ... reset INEXACT flag I.
R := RZ ;         ... set rounding mode to *round-toward-zero*
z := x/y ;        ... *chopped* quotient, possibly inexact
If (not I) then {                 ...if the quotient is exact
        if (z=y) goto Label;      ...$\sqrt{x}$ is exact
        else z = z - ulp;         ...special rounding
        }
i := TRUE;                        ... $\sqrt{x}$ is inexact
If (r=RN) then z= z + ulp;        ...round-to-nearest
If (r=RP) then {                  ...round-toward- $+\infty$
        y= z + ulp; z= z + ulp;
        }
y  := y+z;                        ... *chopped* sum
$y_0$ := $y_0$-0x00100000;        ... y := y/2 is correctly rounded.
Label:
I := i;  ... restore inexact flag
R := r;  ... restore rounded mode
return $\sqrt{x}$ := y.
```

(4) Special cases

Square root of $+\infty$, ± 0, or NaN is itselt;
Square root of a negative number is NaN with invalid signal.

B. $\sqrt{X}$ by Reciproot Iteration

(1). Initial approximation

Let x_0 and x_1 be the leading and the trailing 32-bit words of a floating point number x (in IEEE double format) respectively (see section A). By performing shifts and subtracts on x_0 and x_1, we obtain a 7.8-bit approximation of $1/\sqrt{x}$ as follows.

```
k    := 0x5fe80000 - ($x_0$>>1);
$y_0$   := k - T2[63&(k>>14)].       ...$y \doteq 1/\sqrt{x}$ to 7.8 bits
```

Here k is a 32-bit integer and T2[ ] is an integer array (see section D below for its values) containing correction terms. Now magically the floating value of y (y's leading 32-bit word is y_0, the value of its trailing word y_1 is unimportant) approximates $1/\sqrt{x}$ to almost 7.8-bits.

(2) Iterative refinement

Apply Reciproot iteration three times to y and multiply the result by x to get an approximation z that matches $\sqrt{x}$ to about 1 ulp. To be exact, we have -1 ulp $< \sqrt{x} - z <$ 1.0625 ulp:

```
...set rounding mode to Round-to-nearest
        y  :=  y*(1.5-0.5*x*y*y)          ... almost 15 sig. bits to 1/√x
        y  :=  y*((1.5-2^-30)+0.5*x*y*y) ... about 29 sig. bits to 1/√x
...special arrangement for better accuracy
        z  :=  x*y                        ... 29 bits to √x, and z*y<1
        z  :=  z+0.5*z*(1-z*y)            ... about 1 ulp to √x
```

Remark 2. The constant $1.5-2^{-30}$ is chosen to bias the error so that

(a) the term z*y in the final iteration is always less than 1;

(b) the error in the final result is biased upward so that

-1 ulp $< \sqrt{x} - z <$ 1.0625 ulp

instead of $|\sqrt{x} - z| <$ 1.03125 ulp.

(3) Final adjustment

By twiddling z's last bit it is possible to force z to be correctly rounded according to the prevailing rounding mode as follows. Let r and i be copies of the rounding mode and inexact flag before entering the square root program. Also we use the expression y±ulp for the next representable floating numbers (up and down) of y.

```
R := RZ                         ... set rounding mode to round-toward-zero
switch (r) {
        case RN:                      ... round-to-nearest
                if ( x ≤ z*(z-ulp) ... chopped ) z = z - ulp; else
                if ( x ≤ z*(z+ulp) ... chopped ) z = z ; else z=z+ulp;
                break;
        case RZ: case RM:             ... round-to-zero  or round-to- -∞
                R := RP;              ... reset rounding mode to round-to- +∞
                if ( x < z*z ... rounded up) z = z - ulp; else
                if ( x ≥ (z+ulp)*(z+ulp) ... rounded up) z = z + ulp;
                break;
        case RP:                      ... round-to- +∞
                if ( x > (z+ulp)*(z+ulp) ... chopped ) z = z+ 2*ulp; else
                if ( x > z*z ... chopped ) z = z+ulp;
                break;
}
```

Remark 3. The above comparisons can be done in fixed point. For example, to compare x and w=z*z *chopped*, it suffices to compare x_1 and w_1 (the trailing parts of x and w), regarding them as **two's complement** integers.

...Is z an exact square root?
To determine whether z is an exact square root of x, let z_1 be the trailing part of z, and also let x_0 and x_1 be the leading and trailing parts of x.

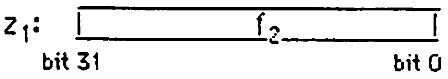
```
If ( ($z_1$&0x3ffffff) != 0 )     ...not exact if trailing 26 bits of z != 0
        I := 1;                   ...Raise Inexact flag: z is not exact
else {
        j := 1 - [($x_0$>>20)&1];  ...j = logb(x) mod 2
        k := $z_1$ >> 26;          ...get z's 25-th and 26-th fraction bits
        I := i or (k&j) or [ (k&(j+j+1))!=($x_1$&3) ];
}
R := r              ... restore rounded mode
return $\sqrt{x}$ := z.
```

If multiplication is cheaper then the foregoing red tape, the Inexact flag can be evaluated by

```
                I := i;
                I := (z*z != x) or I.
```

Note that z*z can overwrite I; this value **must** be sensed if it is TRUE.

Remark 5. If z*z=x exactly, then bit 25 to bit 0 of z_1 must be zero.



Further more, bit 27 and 26 of z_1, bit 0 and 1 of x_1, and the odd of even of logb(x) have the following relations:

bit 27,26 of z_1	bit 1,0 of x_1	logb(x)
00	00	odd and even
01	01	even
10	10	odd
10	00	even
11	01	even

(4) Special cases

See (4) of Section A.

C. IEEE double $\sqrt{x}$ using Newton iterations in pseudo C

```
/* Constants: */
    static long T1[ ]= {            /*  table lookup constants */
       0, 1024, 3062, 5746, 9193, 13348, 18162, 23592, 29598,
       36145, 43202, 50740, 58733, 67158, 75992, 85215, 83599,
       71378, 60428, 50647, 41945, 34246, 27478, 21581, 16499,
       12183, 8588, 5674, 3403, 1742, 661, 130, };
    static int
       RZ =        ... round-toward-zero
       RN =        ... round-to-nearest
       RP =        ... round-toward- +∞
       RM =        ... round-toward- -∞;
       j0  =       ... position of leading word
       j1  =       ... position of trailing word (see section B of REQUIRED
                   ... SUPPORT FUNCTIONS)

/* Main program */
    double sqrt(x)  double x;
    double scalb();
    int finite(), isnan();
    int nx, i, r, e;
    double y;
    unsigned long  k,
       *px = (unsigned long *) &x,          /* pointer to x */
       *py = (unsigned long *) &y ;         /* pointer to y */
/*
 * filter out exceptions
 */
    if ( ! finite(x) || x <= 0.0 )
       if ( isnan(x) || x >= 0.0 )
           return x;       /* sqrt of NaN, +∞, or ±0 is itself */
       else
           return (x-x)/(x-x);     /* sqrt(x<0) is NaN */
    else {
/*
 * Save, reset and initialize:
 */
       i := I                  ... save INEXACT flag I.
       r := R; R := RZ         ... save rounding mode and
```

```
                          ... reset to round-toward-zero.
        k = px[j0];             /* k = the higher 32 bits of x */
        nx = 0;
        if ( k < 0x00100000 )
/*
 * Subnormal number: scale up x by 2^54  and recompute X.
 */
           { nx = 27;
             x  = scalb(x,54);     ... perform x = x*2^54
             k  = px[j0];
            }


/*
 * Magic initial approximation to almost 8 significant bits
 */
        k = ( k >>1) + 0x1ff80000;
        py[j0] = k - T1[ ( k >>15)&31 ];
/*
 * Heron's rule thrice:
 */
        y =  0.5*(y + x/y);
        y =  0.5*(y + x/y);
        y =  y - 0.5*(y - x/y);       /* y ≈ sqrt(x) to within 1 ulp */
/*
 * Twiddle last bit for correctly rounded sqrt(x)
 */
        I := FALSE ... reset INEXACT flag I.
        z = x/y;    /* chopped quotient, possibly inexact */
        if (not I)  ... if quotient is exact then goto final if z=y
           if (z == y) goto final;          /* z is the exact sqrt(x) */
           else z =  z - ulp;       ... z = z - ulp
        i := TRUE   ... sqrt(x) is inexact.

        switch (r) {
           case RN:
              z =   z + ulp;        ... z = z + ulp
              break ;

           case  RP:
              z =   z + ulp;        ... z = z + ulp
              y =   y + ulp;        ... y = y + ulp
              break ;
           }
        y = 0.5*(y+z);        /* chopped sum */
/*
 * final step: restore flags and offset scaling
 */
final:  R := r;  I := i;        ... restore rounding mode and INEXACT flag.
        if ( nx != 0 )
            y = scalb(y,-nx); /* offset scaling for subnormal number */
        return  y;
    }
}
```

D. IEEE double $\sqrt{x}$ using Reciproot iterations in pseudo C

```
/* Constants: */
     static long T2[ ]= {          /*  table lookup constants */
     0x1500, 0x2ef8, 0x4d67, 0x6b02, 0x87be, 0xa395, 0xbe7a, 0xd866,
     0xf14a, 0x1091b, 0x11fcd, 0x13552, 0x14999, 0x15c98, 0x16e34,
     0x17e5f, 0x18d03, 0x19a01, 0x1a545, 0x1ae8a, 0x1b5c4, 0x1bb01,
     0x1bfde, 0x1c28d, 0x1c2de, 0x1c0db, 0x1ba7e, 0x1b11c, 0x1a4b5,
     0x1953d, 0x18266, 0x16be0, 0x1683e, 0x179d8, 0x18a4d, 0x19992,
     0x1a789, 0x1b445, 0x1bf61, 0x1c989, 0x1d16d, 0x1d77b, 0x1dddf,
     0x1e2ad, 0x1e5bf, 0x1e6e8, 0x1e654, 0x1e3cd, 0x1df2a, 0x1d635,
     0x1cb16, 0x1be2c, 0x1ae4e, 0x19bde, 0x1868e, 0x16e2e, 0x1527f,
     0x1334a, 0x11051, 0xe951, 0xbe01, 0x8e0d, 0x5924, 0x1edd};
     static int
        RZ =        ... round-toward-zero
        RN =        ... round-to-nearest
        RP =        ... round-toward- +∞
        RM =        ... round-toward- -∞;
        j0  =       ... position of leading word
        j1  =       ... position of trailing word (see REQUIRED SUPPORT
                    ... FUNCTIONS)

/* Main program */
     double sqrt(x)  double x;
```

```
    double scalb();
    int finite(), isnan();
    int nx, i, r, e;
    double y,z;
    unsigned long  k,
       *px = (unsigned long *) &x,          /* pointer to x */
       *pz = (unsigned long *) &z ;         /* pointer to z */
/*
 * filter out exceptions
 */
    if ( ! finite(x) || x <= 0.0 )
       if (  isnan(x) || x >= 0.0 )
          return x;       /* sqrt of NaN, +∞, or ±0 is itself */
       else
          return (x-x)/(x-x);     /* sqrt(x<0) is NaN */
    else {
/*
 * Save, reset and initialize:
 */
       i := I                  ... save INEXACT flag I.
       r := R; R := RN         ... save rounding mode and
                               ... reset to round-to-nearest.
       k = px[j0];             /* k = the higher 32 bits of x */
       nx = 0;
       if ( k < 0x00200000 )
/*
 * Subnormal number: scale up x by 2^54  and recompute x.
 */
          { nx = 27;
            x  =  scalb(x,54);     ... perform x = x*2^54
            k  = px[j0];
           }

/*
 * Magic initial approximation to almost 7.8 significant bits
 */
       k = 0x5fe80000 - ( k >>1);
       py[j0] = k - T2[ ( k >>14)&63 ];
/*
 * Reciproot iteration:
 */
       y =  y*(1.5 - 0.5*y*y*x);
       y =  y*(1.4999999990686774 25 - 0.5*y*y*x);
```

```
        z =  y*x;
        z =  z + 0.5*z*(1 - z*y);    /* y ≈ sqrt(x) to about 1 ulp */
/*
 * Twiddle last bit for correctly rounded sqrt(x)
 */
    R := RZ          ... set rounding mode to round-toward-zero
    switch (r) {
        case RN:              ... round-to-nearest
            if ( x ≤ z*(z-ulp) ) z = z - ulp; else
            if ( x ≤ z*(z+ulp) ) z = z ; else z=z+ulp;
            break;
        case RZ: case RM: ... round-to-zero and  round-to- -∞
            R := RP           ... reset rounding mode to  round-to- +∞
            if ( x < z*z ) z = z - ulp; else
            if ( x ≥ (z+ulp)*(z+ulp) ) z = z + ulp;
            break;
        case RP:              ... round-to- +∞
            if ( x > (z+ulp)*(z+ulp) ) z = z+ 2*ulp; else
            if ( x > z*z ) z = z+ulp;
            break;
    }

    If ( ($z_1$&0x3ffffff) != 0 )      ...not exact if trailing 26 bits of z != 0
        I := 1;               ...Raise Inexact flag: z is not exact
    else {
        j := 1 - [($x_0$>>20)&1];       ...j = logb(x) mod 2
        k := $z_1$ >> 26;               ...get z's 25-th and 26-th fraction bits
        I := i or (k&j) or [ (k&(j+j+1))!=($x_1$&3) ];
    }

/*
 * final step: restore flags and offset scaling
 */
        R := r;               ... restore rounding mode,
        if ( nx != 0 )
            z = scalb(z,-nx);  /* offset scaling for subnormal number */
        return  z;
    }
}
```