```fortran
 module fm_quad_real


!  FM_quadreal 1.4            David M. Smith           Quadruple Precision Real and Complex Support

!  This module extends the definition of basic FM types (fm), (im), and (zm) so they can interact
!  with quadruple precision real and complex variables.

!  Warning:  This module is needed only when the user's program explicitly declares quadruple
!            precision variables.  If quad precision is obtained by using a compiler switch
!            to change the default real size for the entire program (such as with gfortran's
!            -fdefault-real-8 option), then compiling the basic FM package with the same option
!            means this module is not needed.

!  Not all compilers might support quadruple precision floating-point, but for those that do,
!  variables can be declared via the selected_real_kind function.

!  For example, when this module was first written, typical computer hardware supported 32-bit
!  floats as single precision and 64-bits as double precision.  Some compilers offered 128-bit
!  quadruple precision implemented in software.  This format used 113 bits for the fraction
!  part of a floating-point number, giving about 34 significant digits of precision.

!  So selected_real_kind(30) could be used to select this quad format.

!  The routines in this interface extend basic functions like to_fm, to_im, to_zm so they can be
!  used with quad real or complex arguments.  New conversion functions to_quad and to_quad_z
!  will take fm, im, or zm inputs and convert to quad real or complex.

!  Other mixed-mode operations, such as assignment ( a = b ), logical comparisons, and arithmetic
!  are also provided.  As with the basic fmzm module, assignments and arithmetic may also involve
!  1 or 2-dimensional arrays.

    use fmzm

    integer, parameter :: quad_fp = selected_real_kind(30)
    real (quad_fp), parameter :: q_zero = 0, q_one = 1

    interface to_fm
       module procedure fm_q
       module procedure fm_zq
       module procedure fm_q1
       module procedure fm_zq1
       module procedure fm_q2
       module procedure fm_zq2
    end interface

    interface to_im
       module procedure im_q
       module procedure im_zq
       module procedure im_q1
       module procedure im_zq1
       module procedure im_q2
       module procedure im_zq2
    end interface

    interface to_zm
       module procedure zm_q
```

```
    module procedure zm2_q
    module procedure zm_zq
    module procedure zm_q1
    module procedure zm_zq1
    module procedure zm_q2
    module procedure zm_zq2
 end interface

 interface to_quad
    module procedure fm_2quad
    module procedure im_2quad
    module procedure zm_2quad
    module procedure fm_2quad1
    module procedure im_2quad1
    module procedure zm_2quad1
    module procedure fm_2quad2
    module procedure im_2quad2
    module procedure zm_2quad2
 end interface

interface to_quad_z
   module procedure fm_2quadz
   module procedure im_2quadz
   module procedure zm_2quadz
   module procedure fm_2quadz1
   module procedure im_2quadz1
   module procedure zm_2quadz1
   module procedure fm_2quadz2
   module procedure im_2quadz2
   module procedure zm_2quadz2
end interface

 interface assignment (=)
    module procedure fmeq_qfm
    module procedure fmeq_qim
    module procedure fmeq_qzm
    module procedure fmeq_zqfm
    module procedure fmeq_zqim
    module procedure fmeq_zqzm
    module procedure fmeq_fmq
    module procedure fmeq_fmzq
    module procedure fmeq_imq
    module procedure fmeq_imzq
    module procedure fmeq_zmq
    module procedure fmeq_zmzq
    module procedure fmeq_fm1q
    module procedure fmeq_fm1zq
    module procedure fmeq_q1fm
    module procedure fmeq_zq1fm
    module procedure fmeq_fm1q1
    module procedure fmeq_fm1zq1
    module procedure fmeq_q1fm1
    module procedure fmeq_zq1fm1
    module procedure fmeq_im1q
    module procedure fmeq_im1zq
    module procedure fmeq_q1im
    module procedure fmeq_zq1im
    module procedure fmeq_im1q1
```

```fortran
      module procedure fmeq_im1zq1
      module procedure fmeq_q1im1
      module procedure fmeq_zq1im1
      module procedure fmeq_zm1q
      module procedure fmeq_zm1zq
      module procedure fmeq_q1zm
      module procedure fmeq_zq1zm
      module procedure fmeq_zm1q1
      module procedure fmeq_zm1zq1
      module procedure fmeq_q1zm1
      module procedure fmeq_zq1zm1
      module procedure fmeq_fm2q
      module procedure fmeq_fm2zq
      module procedure fmeq_q2fm
      module procedure fmeq_zq2fm
      module procedure fmeq_fm2q2
      module procedure fmeq_fm2zq2
      module procedure fmeq_q2fm2
      module procedure fmeq_zq2fm2
      module procedure fmeq_im2q
      module procedure fmeq_im2zq
      module procedure fmeq_q2im
      module procedure fmeq_zq2im
      module procedure fmeq_im2q2
      module procedure fmeq_im2zq2
      module procedure fmeq_q2im2
      module procedure fmeq_zq2im2
      module procedure fmeq_zm2q
      module procedure fmeq_zm2zq
      module procedure fmeq_q2zm
      module procedure fmeq_zq2zm
      module procedure fmeq_zm2q2
      module procedure fmeq_zm2zq2
      module procedure fmeq_q2zm2
      module procedure fmeq_zq2zm2
   end interface

   interface operator (==)
      module procedure fmleq_qfm
      module procedure fmleq_qim
      module procedure fmleq_qzm
      module procedure fmleq_zqfm
      module procedure fmleq_zqim
      module procedure fmleq_zqzm
      module procedure fmleq_fmq
      module procedure fmleq_fmzq
      module procedure fmleq_imq
      module procedure fmleq_imzq
      module procedure fmleq_zmq
      module procedure fmleq_zmzq
   end interface

   interface operator (/=)
      module procedure fmlne_qfm
      module procedure fmlne_qim
      module procedure fmlne_qzm
      module procedure fmlne_zqfm
      module procedure fmlne_zqim
```

```
      module procedure fmlne_zqzm
      module procedure fmlne_fmq
      module procedure fmlne_fmzq
      module procedure fmlne_imq
      module procedure fmlne_imzq
      module procedure fmlne_zmq
      module procedure fmlne_zmzq
   end interface

   interface operator (>)
      module procedure fmlgt_qfm
      module procedure fmlgt_qim
      module procedure fmlgt_fmq
      module procedure fmlgt_imq
   end interface

   interface operator (>=)
      module procedure fmlge_qfm
      module procedure fmlge_qim
      module procedure fmlge_fmq
      module procedure fmlge_imq
   end interface

   interface operator (<)
      module procedure fmllt_qfm
      module procedure fmllt_qim
      module procedure fmllt_fmq
      module procedure fmllt_imq
   end interface

   interface operator (<=)
      module procedure fmlle_qfm
      module procedure fmlle_qim
      module procedure fmlle_fmq
      module procedure fmlle_imq
   end interface

   interface operator (+)
      module procedure fmadd_qfm
      module procedure fmadd_qim
      module procedure fmadd_qzm
      module procedure fmadd_zqfm
      module procedure fmadd_zqim
      module procedure fmadd_zqzm
      module procedure fmadd_fmq
      module procedure fmadd_fmzq
      module procedure fmadd_imq
      module procedure fmadd_imzq
      module procedure fmadd_zmq
      module procedure fmadd_zmzq
      module procedure fmadd_qfm1
      module procedure fmadd_zqfm1
      module procedure fmadd_fmq1
      module procedure fmadd_fmzq1
      module procedure fmadd_fm1q
      module procedure fmadd_fm1zq
      module procedure fmadd_q1fm
      module procedure fmadd_zq1fm
```

```
module procedure fmadd_q1fm1
module procedure fmadd_zq1fm1
module procedure fmadd_fm1q1
module procedure fmadd_fm1zq1
module procedure fmadd_qim1
module procedure fmadd_zqim1
module procedure fmadd_imq1
module procedure fmadd_imzq1
module procedure fmadd_im1q
module procedure fmadd_im1zq
module procedure fmadd_q1im
module procedure fmadd_zq1im
module procedure fmadd_q1im1
module procedure fmadd_zq1im1
module procedure fmadd_im1q1
module procedure fmadd_im1zq1
module procedure fmadd_qzm1
module procedure fmadd_zqzm1
module procedure fmadd_zmq1
module procedure fmadd_zmzq1
module procedure fmadd_zm1q
module procedure fmadd_zm1zq
module procedure fmadd_q1zm
module procedure fmadd_zq1zm
module procedure fmadd_q1zm1
module procedure fmadd_zq1zm1
module procedure fmadd_zm1q1
module procedure fmadd_zm1zq1
module procedure fmadd_qfm2
module procedure fmadd_zqfm2
module procedure fmadd_fmq2
module procedure fmadd_fmzq2
module procedure fmadd_fm2q
module procedure fmadd_fm2zq
module procedure fmadd_q2fm
module procedure fmadd_zq2fm
module procedure fmadd_q2fm2
module procedure fmadd_zq2fm2
module procedure fmadd_fm2q2
module procedure fmadd_fm2zq2
module procedure fmadd_qim2
module procedure fmadd_zqim2
module procedure fmadd_imq2
module procedure fmadd_imzq2
module procedure fmadd_im2q
module procedure fmadd_im2zq
module procedure fmadd_q2im
module procedure fmadd_zq2im
module procedure fmadd_q2im2
module procedure fmadd_zq2im2
module procedure fmadd_im2q2
module procedure fmadd_im2zq2
module procedure fmadd_qzm2
module procedure fmadd_zqzm2
module procedure fmadd_zmq2
module procedure fmadd_zmzq2
module procedure fmadd_zm2q
module procedure fmadd_zm2zq
```

```fortran
   module procedure fmadd_q2zm
   module procedure fmadd_zq2zm
   module procedure fmadd_q2zm2
   module procedure fmadd_zq2zm2
   module procedure fmadd_zm2q2
   module procedure fmadd_zm2zq2
end interface

interface operator (-)
   module procedure fmsub_qfm
   module procedure fmsub_qim
   module procedure fmsub_qzm
   module procedure fmsub_zqfm
   module procedure fmsub_zqim
   module procedure fmsub_zqzm
   module procedure fmsub_fmq
   module procedure fmsub_fmzq
   module procedure fmsub_imq
   module procedure fmsub_imzq
   module procedure fmsub_zmq
   module procedure fmsub_zmzq
   module procedure fmsub_qfm1
   module procedure fmsub_zqfm1
   module procedure fmsub_fmq1
   module procedure fmsub_fmzq1
   module procedure fmsub_fm1q
   module procedure fmsub_fm1zq
   module procedure fmsub_q1fm
   module procedure fmsub_zq1fm
   module procedure fmsub_q1fm1
   module procedure fmsub_zq1fm1
   module procedure fmsub_fm1q1
   module procedure fmsub_fm1zq1
   module procedure fmsub_qim1
   module procedure fmsub_zqim1
   module procedure fmsub_imq1
   module procedure fmsub_imzq1
   module procedure fmsub_im1q
   module procedure fmsub_im1zq
   module procedure fmsub_q1im
   module procedure fmsub_zq1im
   module procedure fmsub_q1im1
   module procedure fmsub_zq1im1
   module procedure fmsub_im1q1
   module procedure fmsub_im1zq1
   module procedure fmsub_qzm1
   module procedure fmsub_zqzm1
   module procedure fmsub_zmq1
   module procedure fmsub_zmzq1
   module procedure fmsub_zm1q
   module procedure fmsub_zm1zq
   module procedure fmsub_q1zm
   module procedure fmsub_zq1zm
   module procedure fmsub_q1zm1
   module procedure fmsub_zq1zm1
   module procedure fmsub_zm1q1
   module procedure fmsub_zm1zq1
   module procedure fmsub_qfm2
```

```
      module procedure fmsub_zqfm2
      module procedure fmsub_fmq2
      module procedure fmsub_fmzq2
      module procedure fmsub_fm2q
      module procedure fmsub_fm2zq
      module procedure fmsub_q2fm
      module procedure fmsub_zq2fm
      module procedure fmsub_q2fm2
      module procedure fmsub_zq2fm2
      module procedure fmsub_fm2q2
      module procedure fmsub_fm2zq2
      module procedure fmsub_qim2
      module procedure fmsub_zqim2
      module procedure fmsub_imq2
      module procedure fmsub_imzq2
      module procedure fmsub_im2q
      module procedure fmsub_im2zq
      module procedure fmsub_q2im
      module procedure fmsub_zq2im
      module procedure fmsub_q2im2
      module procedure fmsub_zq2im2
      module procedure fmsub_im2q2
      module procedure fmsub_im2zq2
      module procedure fmsub_qzm2
      module procedure fmsub_zqzm2
      module procedure fmsub_zmq2
      module procedure fmsub_zmzq2
      module procedure fmsub_zm2q
      module procedure fmsub_zm2zq
      module procedure fmsub_q2zm
      module procedure fmsub_zq2zm
      module procedure fmsub_q2zm2
      module procedure fmsub_zq2zm2
      module procedure fmsub_zm2q2
      module procedure fmsub_zm2zq2
   end interface

   interface operator (*)
      module procedure fmmpy_qfm
      module procedure fmmpy_qim
      module procedure fmmpy_qzm
      module procedure fmmpy_zqfm
      module procedure fmmpy_zqim
      module procedure fmmpy_zqzm
      module procedure fmmpy_fmq
      module procedure fmmpy_fmzq
      module procedure fmmpy_imq
      module procedure fmmpy_imzq
      module procedure fmmpy_zmq
      module procedure fmmpy_zmzq
      module procedure fmmpy_qfm1
      module procedure fmmpy_zqfm1
      module procedure fmmpy_fmq1
      module procedure fmmpy_fmzq1
      module procedure fmmpy_fm1q
      module procedure fmmpy_fm1zq
      module procedure fmmpy_q1fm
      module procedure fmmpy_zq1fm
```

```
module procedure fmmpy_q1fm1
module procedure fmmpy_zq1fm1
module procedure fmmpy_fm1q1
module procedure fmmpy_fm1zq1
module procedure fmmpy_qim1
module procedure fmmpy_zqim1
module procedure fmmpy_imq1
module procedure fmmpy_imzq1
module procedure fmmpy_im1q
module procedure fmmpy_im1zq
module procedure fmmpy_q1im
module procedure fmmpy_zq1im
module procedure fmmpy_q1im1
module procedure fmmpy_zq1im1
module procedure fmmpy_im1q1
module procedure fmmpy_im1zq1
module procedure fmmpy_qzm1
module procedure fmmpy_zqzm1
module procedure fmmpy_zmq1
module procedure fmmpy_zmzq1
module procedure fmmpy_zm1q
module procedure fmmpy_zm1zq
module procedure fmmpy_q1zm
module procedure fmmpy_zq1zm
module procedure fmmpy_q1zm1
module procedure fmmpy_zq1zm1
module procedure fmmpy_zm1q1
module procedure fmmpy_zm1zq1
module procedure fmmpy_qfm2
module procedure fmmpy_zqfm2
module procedure fmmpy_fmq2
module procedure fmmpy_fmzq2
module procedure fmmpy_fm2q
module procedure fmmpy_fm2zq
module procedure fmmpy_q2fm
module procedure fmmpy_zq2fm
module procedure fmmpy_q2fm2
module procedure fmmpy_zq2fm2
module procedure fmmpy_fm2q2
module procedure fmmpy_fm2zq2
module procedure fmmpy_qim2
module procedure fmmpy_zqim2
module procedure fmmpy_imq2
module procedure fmmpy_imzq2
module procedure fmmpy_im2q
module procedure fmmpy_im2zq
module procedure fmmpy_q2im
module procedure fmmpy_zq2im
module procedure fmmpy_q2im2
module procedure fmmpy_zq2im2
module procedure fmmpy_im2q2
module procedure fmmpy_im2zq2
module procedure fmmpy_qzm2
module procedure fmmpy_zqzm2
module procedure fmmpy_zmq2
module procedure fmmpy_zmzq2
module procedure fmmpy_zm2q
module procedure fmmpy_zm2zq
```

```
   module procedure fmmpy_q2zm
   module procedure fmmpy_zq2zm
   module procedure fmmpy_q2zm2
   module procedure fmmpy_zq2zm2
   module procedure fmmpy_zm2q2
   module procedure fmmpy_zm2zq2
end interface

interface operator (/)
   module procedure fmdiv_qfm
   module procedure fmdiv_qim
   module procedure fmdiv_qzm
   module procedure fmdiv_zqfm
   module procedure fmdiv_zqim
   module procedure fmdiv_zqzm
   module procedure fmdiv_fmq
   module procedure fmdiv_fmzq
   module procedure fmdiv_imq
   module procedure fmdiv_imzq
   module procedure fmdiv_zmq
   module procedure fmdiv_zmzq
   module procedure fmdiv_qfm1
   module procedure fmdiv_zqfm1
   module procedure fmdiv_fmq1
   module procedure fmdiv_fmzq1
   module procedure fmdiv_fm1q
   module procedure fmdiv_fm1zq
   module procedure fmdiv_q1fm
   module procedure fmdiv_zq1fm
   module procedure fmdiv_q1fm1
   module procedure fmdiv_zq1fm1
   module procedure fmdiv_fm1q1
   module procedure fmdiv_fm1zq1
   module procedure fmdiv_qim1
   module procedure fmdiv_zqim1
   module procedure fmdiv_imq1
   module procedure fmdiv_imzq1
   module procedure fmdiv_im1q
   module procedure fmdiv_im1zq
   module procedure fmdiv_q1im
   module procedure fmdiv_zq1im
   module procedure fmdiv_q1im1
   module procedure fmdiv_zq1im1
   module procedure fmdiv_im1q1
   module procedure fmdiv_im1zq1
   module procedure fmdiv_qzm1
   module procedure fmdiv_zqzm1
   module procedure fmdiv_zmq1
   module procedure fmdiv_zmzq1
   module procedure fmdiv_zm1q
   module procedure fmdiv_zm1zq
   module procedure fmdiv_q1zm
   module procedure fmdiv_zq1zm
   module procedure fmdiv_q1zm1
   module procedure fmdiv_zq1zm1
   module procedure fmdiv_zm1q1
   module procedure fmdiv_zm1zq1
   module procedure fmdiv_qfm2
```

```
      module procedure fmdiv_zqfm2
      module procedure fmdiv_fmq2
      module procedure fmdiv_fmzq2
      module procedure fmdiv_fm2q
      module procedure fmdiv_fm2zq
      module procedure fmdiv_q2fm
      module procedure fmdiv_zq2fm
      module procedure fmdiv_q2fm2
      module procedure fmdiv_zq2fm2
      module procedure fmdiv_fm2q2
      module procedure fmdiv_fm2zq2
      module procedure fmdiv_qim2
      module procedure fmdiv_zqim2
      module procedure fmdiv_imq2
      module procedure fmdiv_imzq2
      module procedure fmdiv_im2q
      module procedure fmdiv_im2zq
      module procedure fmdiv_q2im
      module procedure fmdiv_zq2im
      module procedure fmdiv_q2im2
      module procedure fmdiv_zq2im2
      module procedure fmdiv_im2q2
      module procedure fmdiv_im2zq2
      module procedure fmdiv_qzm2
      module procedure fmdiv_zqzm2
      module procedure fmdiv_zmq2
      module procedure fmdiv_zmzq2
      module procedure fmdiv_zm2q
      module procedure fmdiv_zm2zq
      module procedure fmdiv_q2zm
      module procedure fmdiv_zq2zm
      module procedure fmdiv_q2zm2
      module procedure fmdiv_zq2zm2
      module procedure fmdiv_zm2q2
      module procedure fmdiv_zm2zq2
   end interface

   interface operator (**)
      module procedure fmpwr_qfm
      module procedure fmpwr_qim
      module procedure fmpwr_qzm
      module procedure fmpwr_zqfm
      module procedure fmpwr_zqim
      module procedure fmpwr_zqzm
      module procedure fmpwr_fmq
      module procedure fmpwr_fmzq
      module procedure fmpwr_imq
      module procedure fmpwr_imzq
      module procedure fmpwr_zmq
      module procedure fmpwr_zmzq
   end interface

 contains

      subroutine fmq2m(x, ma)

!  Convert quadruple precision x to multiple precision ma.
```

```fortran
      use fmvals
      implicit none

      real (quad_fp) :: x
      type(multi) :: ma
      real (quad_fp) :: f1, f2, y, y1, y2, two20
      integer :: j, j1, j2, jd, jexp, k, kexp, kl, l, ndsave
      intent (in) :: x
      intent (inout) :: ma
      type(multi) :: mxy(4)

      if (.not. allocated(ma%mp)) then
          allocate(ma%mp(ndig+2), stat=k_stat)
          if (k_stat /= 0) call fmdefine_error
      else if (size(ma%mp) < ndig+2) then
          deallocate(ma%mp)
          allocate(ma%mp(ndig+2), stat=k_stat)
          if (k_stat /= 0) call fmdefine_error
      endif

!             Increase the working precision.

      ndsave = ndig
      if (ncall == 1) then
          k = max(ngrd21, 1)
          ndig = max(ndig+k, 3)
      endif

      if (mblogs /= mbase) call fmcons
      kflag = 0

!             Special case for x = 0.

      if (x == 0) then
          do j = 1, ndsave+1
             ma%mp(j+1) = 0
          enddo
          ma%mp(1) = 1
          if (x < 0.0 .and. ma%mp(2) /= munkno .and. ma%mp(3) /= 0) ma%mp(1) = -1
          ndig = ndsave
          return
      endif

!             Check for x = + or - Infinity, or Nan.  Return unknown if so.

      if (x > huge(x) .or. x < -huge(x) .or. (.not.(x == x))) then
          do j = 2, ndsave
             ma%mp(j+2) = 0
          enddo
          kflag = -4
          ma%mp(2) = munkno
          ma%mp(3) = 1
          ma%mp(1) = 1
          call fmwarn
          ndig = ndsave
          return
      endif
```

```fortran
!              Special case for mbase = 2.

      if (mbase == 2 .and. radix(x) == 2) then
          ndig = max(ndig, digits(x))
          y = fraction(abs(x))
          call fmi2m(0, mxy(4))
          do j = 1, min(digits(x), ndig)
             y = y + y
             mxy(4)%mp(j+2) = int(y)
             y = y - int(y)
          enddo
          mxy(4)%mp(2) = exponent(x)
          call fmequ(mxy(4), ma, ndig, ndsave)
          ma%mp(1) = 1
          if (x < 0.0 .and. ma%mp(2) /= munkno .and. ma%mp(3) /= 0) ma%mp(1) = -1
          ndig = ndsave
          return
      endif

!              Special case for mbase = 10**l.

      k = mbase
      l = 0
      do
         if (mod(k, 10) == 0) then
             l = l + 1
             k = k/10
             if (k == 1) exit
         else
             l = 0
             exit
         endif
      enddo
      if (l > 0) then
          ndig = max(ndig, int(digits(x)*0.30103/l)+1)
          y = fraction(abs(x))
          call fmi2m(0, mxy(4))
          do j = 1, ndig

!              Multiply by 10**l to get the next digit in base mbase.
!              To avoid any rounding errors in quad precision, do each multiply by 10 as
!              one multiply by 8 and one by 2, and keep two integer and two fraction results.
!              So 10*y is broken into 8*y + 2*y, since there will be no rounding with either
!              term in quad precision on a binary machine.

             jd = 0
             do k = 1, l
                y1 = 8*y
                y2 = 2*y
                j1 = y1
                j2 = y2
                f1 = y1 - j1
                f2 = y2 - j2
                jd = 10*jd + j1 + j2
                y = f1 + f2
                if (y >= 1) then
                    jd = jd + 1
                    y = y - 1
```

```
               endif
            enddo
            mxy(4)%mp(j+2) = jd
            if (y == 0) exit
         enddo
         k = intmax
         if (maxint/mbase < k) k = maxint/mbase
         k = k/2
         j2 = 1
         jexp = exponent(x)
         do j = 1, abs(jexp)
            j2 = 2*j2
            if (j2 >= k .or. j == abs(jexp)) then
                if (jexp > 0) then
                    call fmmpyi_r1(mxy(4), j2)
                else
                    call fmdivi_r1(mxy(4), j2)
                endif
                j2 = 1
            endif
         enddo
         call fmequ(mxy(4), ma, ndig, ndsave)
         ma%mp(1) = 1
         if (x < 0.0 .and. ma%mp(2) /= munkno .and. ma%mp(3) /= 0) ma%mp(1) = -1
         ndig = ndsave
         return
     endif

     y = abs(x)
     two20 = 1048576.0d0

!             If this power of two is not representable at the current base and precision, use a
!             smaller one.

     if (int(ndig*alogm2) < 20) then
         k = int(ndig*alogm2)
         two20 = 1
         do j = 1, k
            two20 = two20*2.0d0
         enddo
     endif

     kexp = 0
     if (y > two20) then
         do while (y > two20)
            y = y/two20
            kexp = kexp + 1
         enddo
     else if (y < 1) then
         do while (y < 1)
            y = y*two20
            kexp = kexp - 1
         enddo
     endif

     k = int(two20)
     call fmi2m(k, mxy(3))
     k = int(y)
```

```
      call fmi2m(k, mxy(1))
      y = (y-dble(k))*two20
      jexp = 0

      kl = 1
      do while (kl == 1)
         kl = 0
         k = int(y)
         call fmi2m(k, mxy(2))
         call fmmpy_r1(mxy(1), mxy(3))
         jexp = jexp + 1
         call fmadd_r1(mxy(1), mxy(2))
         y = (y-dble(k))*two20
         if (jexp <= 1000 .and. y /= 0) kl = 1
      enddo

      k = kexp - jexp
      if (k >= 0) then
          if (k == 0) then
              call fmeq(mxy(1), mxy(4))
          else if (k == 1) then
              call fmmpy(mxy(1), mxy(3), mxy(4))
          else if (k == 2) then
              call fmsqr(mxy(3), mxy(2))
              call fmmpy(mxy(1), mxy(2), mxy(4))
          else
              call fmipwr(mxy(3), k, mxy(2))
              call fmmpy(mxy(1), mxy(2), mxy(4))
          endif
      else
          if (k == -1) then
              call fmdiv(mxy(1), mxy(3), mxy(4))
          else if (k == -2) then
              call fmsqr(mxy(3), mxy(2))
              call fmdiv(mxy(1), mxy(2), mxy(4))
          else
              call fmipwr(mxy(3), -k, mxy(2))
              call fmdiv(mxy(1), mxy(2), mxy(4))
          endif
      endif
      call fmequ(mxy(4), ma, ndig, ndsave)

      ma%mp(1) = 1
      if (x < 0.0 .and. ma%mp(2) /= munkno .and. ma%mp(3) /= 0) ma%mp(1) = -1
      ndig = ndsave
      return
      end subroutine fmq2m


      subroutine fmm2q(ma, x)

!  Convert multiple precision ma to quadruple precision x.

      use fmvals
      implicit none

      type(multi) :: ma
      real (quad_fp) :: x
```

```fortran
      real (quad_fp) :: aq(2), xq(2), yq(2), y1(2), y2(2), xbase, pmax, dlogdp,  &
                        a1, a2, c, c1, c2, c21, c22, q1, q2, t, z1, z2
      real (kind(1.0d0)) :: ma1, mas
      integer :: j, k, kl, kwrnsv, ncase
      intent (in) :: ma
      intent (inout) :: x

!             Check to see if ma is in range for quadruple precision.

      if (mblogs /= mbase) call fmcons
      pmax = huge(x) / 5
      dlogdp = log(pmax)
      ma1 = ma%mp(2)
      ncase = 0
      if (dble(ma%mp(2)-1)*dlogmb > dlogdp) then
          kflag = -4
          x = -1.01*(huge(x)/3.0)
          call fmwarn
          return
      else if (dble(ma%mp(2)+1)*dlogmb > dlogdp) then
          ma1 = ma1 - 2
          ncase = 1
      else if (dble(ma%mp(2)+1)*dlogmb < -dlogdp) then
          kflag = -10
          x = 0
          call fmwarn
          return
      else if (dble(ma%mp(2)-1)*dlogmb < -dlogdp) then
          ma1 = ma1 + 2
          ncase = 2
      endif

!             Try fmm2i2 first so that small integers will be converted quickly.

      kwrnsv = kwarn
      kwarn = 0
      call fmm2i2(ma, j)
      kwarn = kwrnsv
      if (kflag == 0) then
          x = j
          return
      endif
      kflag = 0

!             General case.
!             In order to get the correctly rounded x, the arithmetic for computing x is done
!             with twice quadruple precision using the arrays of length 2.

      mas = ma%mp(1)
      xbase = mbase
      xq = (/ 0 , 0 /)
      yq = (/ 1 , 0 /)
      c = radix(x)
      k = digits(x) - digits(x)/2
      c = c ** k
      k = (log(dble(radix(x)))/dlogmb)*digits(x) + ngrd52
      do j = 2, min(k+1, ndig+1)
```

```
   z1 = yq(1) / xbase
   t = xbase*c
   a1 = (xbase - t) + t
   a2 = xbase - a1
   t = z1*c
   c1 = (z1 - t) + t
   c2 = z1 - c1
   t = c2*c
   c21 = (c2 - t) + t
   c22 = c2 - c21
   q1 = xbase*z1
   q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
   z2 = ((((yq(1)-q1) - q2) + yq(2))) / xbase
   yq(1) = z1 + z2
   yq(2) = (z1-yq(1)) + z2
   t = yq(1)*c
   a1 = (yq(1) - t) + t
   a2 = yq(1) - a1
   t = dble(ma%mp(j+1))*c
   c1 = (dble(ma%mp(j+1)) - t) + t
   c2 = dble(ma%mp(j+1)) - c1
   t = c2*c
   c21 = (c2 - t) + t
   c22 = c2 - c21
   q1 = yq(1)*dble(ma%mp(j+1))
   q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
   z2 = yq(2)*dble(ma%mp(j+1)) + q2
   aq(1) = q1 + z2
   aq(2) = (q1-aq(1)) + z2
   z1 = xq(1) + aq(1)
   q1 = xq(1) - z1
   z2 = (((q1+aq(1)) + (xq(1)-(q1+z1))) + xq(2)) + aq(2)
   xq(1) = z1 + z2
   xq(2) = (z1-xq(1)) + z2
enddo

y1 = (/ xbase , q_zero /)
k = abs(ma1)
if (mod(k, 2) == 0) then
    y2 = (/ 1 , 0 /)
else
    y2 = (/ xbase , q_zero /)
endif

kl = 1
do while (kl == 1)
   kl = 0
   k = k/2
   t = y1(1)*c
   a1 = (y1(1) - t) + t
   a2 = y1(1) - a1
   t = y1(1)*c
   c1 = (y1(1) - t) + t
   c2 = y1(1) - c1
   t = c2*c
   c21 = (c2 - t) + t
   c22 = c2 - c21
   q1 = y1(1)*y1(1)
```

```
   q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
   z2 = ((y1(1) + y1(2))*y1(2) + y1(2)*y1(1)) + q2
   y1(1) = q1 + z2
   y1(2) = (q1-y1(1)) + z2
   if (mod(k, 2) == 1) then
       t = y1(1)*c
       a1 = (y1(1) - t) + t
       a2 = y1(1) - a1
       t = y2(1)*c
       c1 = (y2(1) - t) + t
       c2 = y2(1) - c1
       t = c2*c
       c21 = (c2 - t) + t
       c22 = c2 - c21
       q1 = y1(1)*y2(1)
       q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
       z2 = ((y1(1) + y1(2))*y2(2) + y1(2)*y2(1)) + q2
       y2(1) = q1 + z2
       y2(2) = (q1-y2(1)) + z2
   endif
   if (k > 1) kl = 1
enddo

if (ma1 < 0) then
    z1 = xq(1) / y2(1)
    t = y2(1)*c
    a1 = (y2(1) - t) + t
    a2 = y2(1) - a1
    t = z1*c
    c1 = (z1 - t) + t
    c2 = z1 - c1
    t = c2*c
    c21 = (c2 - t) + t
    c22 = c2 - c21
    q1 = y2(1)*z1
    q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
    z2 = ((((xq(1)-q1) - q2) + xq(2)) - z1*y2(2)) / (y2(1) + y2(2))
    aq(1) = z1 + z2
    aq(2) = (z1-aq(1)) + z2
else
    t = xq(1)*c
    a1 = (xq(1) - t) + t
    a2 = xq(1) - a1
    t = y2(1)*c
    c1 = (y2(1) - t) + t
    c2 = y2(1) - c1
    t = c2*c
    c21 = (c2 - t) + t
    c22 = c2 - c21
    q1 = xq(1)*y2(1)
    q2 = ((((a1*c1 - q1) + a1*c2) + c1*a2) + c21*a2) + c22*a2
    z2 = ((xq(1) + xq(2))*y2(2) + xq(2)*y2(1)) + q2
    aq(1) = q1 + z2
    aq(2) = (q1-aq(1)) + z2
endif

x = aq(1) + aq(2)
```

```
      if (mas < 0) x = -x

!               Check the result if it is near overflow or underflow.

      if (ncase == 1) then
          if (x <= pmax/(xbase*xbase)) then
              x = x*xbase*xbase
          else
              kflag = -4
              x = -1.01*(huge(x)/3.0)
              call fmwarn
          endif
      else if (ncase == 2) then
          if (x >= (1/pmax)*xbase*xbase) then
              x = x/(xbase*xbase)
          else
              kflag = -10
              x = 0
              call fmwarn
          endif
      endif
      return
      end subroutine fmm2q


      subroutine imm2q(ma, x)

!   x = ma

!   Convert an IM number to quadruple precision.

      use fmvals
      implicit none

      type(multi) :: ma
      real (quad_fp) :: x

      integer :: nd2, ndsave
      intent (in) :: ma
      intent (inout) :: x

      ndsave = ndig
      ndig = max(3, int(ma%mp(2)))
      nd2 = 2 - log(epsilon(q_one))/dlogmb
      if (ndig >= nd2) ndig = nd2

      call fmm2q(ma, x)

      ndig = ndsave
      return
      end subroutine imm2q


   function fm_q(d)      result (return_value)
      use fmvals
      implicit none
      type (fm) :: return_value
      real (quad_fp) :: d
```

```
   intent (in) :: d
   call fmq2m(d, return_value%mfm)
end function fm_q

function fm_zq(c)     result (return_value)
   use fmvals
   implicit none
   type (fm) :: return_value
   complex (quad_fp) :: c
   intent (in) :: c
   call fmq2m(real(c, quad_fp), return_value%mfm)
end function fm_zq

function fm_q1(d)     result (return_value)
   use fmvals
   implicit none
   real (quad_fp), dimension(:) :: d
   type (fm), dimension(size(d)) :: return_value
   integer :: j, n
   intent (in) :: d
   n = size(d)
   do j = 1, n
      call fmq2m(d(j), return_value(j)%mfm)
   enddo
end function fm_q1

function fm_zq1(c)     result (return_value)
   use fmvals
   implicit none
   complex (quad_fp), dimension(:) :: c
   type (fm), dimension(size(c)) :: return_value
   integer :: j, n
   intent (in) :: c
   n = size(c)
   do j = 1, n
      call fmq2m(real(c(j), quad_fp), return_value(j)%mfm)
   enddo
end function fm_zq1

function fm_q2(d)     result (return_value)
   use fmvals
   implicit none
   real (quad_fp), dimension(:,:) :: d
   type (fm), dimension(size(d, dim=1), size(d, dim=2)) :: return_value
   integer :: j, k
   intent (in) :: d
   do j = 1, size(d, dim=1)
      do k = 1, size(d, dim=2)
         call fmq2m(d(j, k), return_value(j, k)%mfm)
      enddo
   enddo
end function fm_q2

function fm_zq2(c)     result (return_value)
   use fmvals
   implicit none
   complex (quad_fp), dimension(:,:) :: c
   type (fm), dimension(size(c, dim=1), size(c, dim=2)) :: return_value
```

```fortran
   integer :: j, k
   intent (in) :: c
   do j = 1, size(c, dim=1)
      do k = 1, size(c, dim=2)
         call fmq2m(real(c(j, k), quad_fp), return_value(j, k)%mfm)
      enddo
   enddo
end function fm_zq2

function im_q(d)     result (return_value)
   use fmvals
   implicit none
   type (im) :: return_value
   real (quad_fp) :: d
   character(50) :: st
   integer :: ival
   intent (in) :: d
   if (abs(d) < huge(1)) then
       ival = int(d)
       call imi2m(ival, return_value%mim)
   else
       write (st, '(e50.39)') d
       call imst2m(st, return_value%mim)
   endif
end function im_q

function im_zq(c)     result (return_value)
   use fmvals
   implicit none
   type (im) :: return_value
   complex (quad_fp) :: c
   real (quad_fp) :: d
   character(50) :: st
   integer :: ival
   intent (in) :: c
   d = real(c, quad_fp)
   if (abs(d) < huge(1)) then
       ival = int(d)
       call imi2m(ival, return_value%mim)
   else
       write (st, '(e50.39)') d
       call imst2m(st, return_value%mim)
   endif
end function im_zq

function im_q1(d)     result (return_value)
   use fmvals
   implicit none
   real (quad_fp), dimension(:) :: d
   type (im), dimension(size(d)) :: return_value
   character(50) :: st
   integer :: ival, j, n
   intent (in) :: d
   n = size(d)
   do j = 1, n
      if (abs(d(j)) < huge(1)) then
          ival = int(d(j))
          call imi2m(ival, return_value(j)%mim)
```