! FMZM 1.4                        David M. Smith

! This module extends the definition of the basic Fortran arithmetic and function operations so
! they also apply to multiple precision numbers, using version 1.4 of FM.
! There are three multiple precision data types:
!   FM  (multiple precision real)
!   IM  (multiple precision integer)
!   ZM  (multiple precision complex)

! For some examples and general advice about using these multiple-precision data types, see the
! program SampleFM.f95.

! Most of the functions defined in this module are multiple precision versions of standard Fortran
! functions. In addition, there are functions for direct conversion, formatting, and some
! mathematical special functions.

! to_fm is a function for converting other types of numbers to type FM. Note that to_fm(3.12)
! converts the real constant to FM, but it is accurate only to single precision, since the number
! 3.12 cannot be represented exactly in binary and has already been rounded to single precision.
! Similarly, to_fm(3.12d0) agrees with 3.12 to double precision accuracy, and to_fm('3.12') or
! to_fm(312)/to_fm(100) agrees to full FM accuracy.

! to_im converts to type IM, and to_zm converts to type ZM.

! Functions are also supplied for converting the three multiple precision types to the other
! numeric data types:
!   to_int  converts to machine precision integer
!   to_sp   converts to single precision
!   to_dp   converts to double precision
!   to_spz  converts to single precision complex
!   to_dpz  converts to double precision complex

! WARNING: When multiple precision type declarations are inserted in an existing program, take
! care in converting functions like dble(x), where x has been declared as a multiple
! precision type. If x was single precision in the original program, then replacing
! the dble(x) by to_dp(x) in the new version could lose accuracy. For this reason, the
! Fortran type-conversion functions defined in this module assume that results should
! be multiple precision whenever inputs are. Examples:
! dble(to_fm('1.23e+123456')) is type FM
! real(to_fm('1.23e+123456')) is type FM
! real(to_zm('3.12+4.56i')) is type FM = to_fm('3.12')
! int(to_fm('1.23')) is type IM = to_im(1)
! int(to_im('1e+23')) is type IM
! cmplx(to_fm('1.23'),to_fm('4.56')) is type ZM

! is_overflow, is_underflow, and is_unknown are logical functions for checking whether a multiple
! precision number is in one of the exception categories. Testing to see if a type FM number is
! in the +overflow category by directly using an if can be tricky. When mafm is +overflow, the
! statement
!   if (mafm == to_fm(' +overflow ')) then
! will return false, since the comparison routine cannot be sure that two different overflowed
! results would have been equal if the overflow threshold had been higher. Instead, use
!   if (is_overflow(mafm)) then

which will be true if mafm is + or - overflow.

Programs using this module may sometimes need to call fm, im, or zm routines directly. This is normally the case when routines are needed that are not Fortran intrinsics, such as the formatting subroutine fm_form. In a program using this module, suppose mafm has been declared with type (fm) :: mafm. To convert the number to a character string with f65.60 format, use

```fortran
    call fm_form('f65.60',mafm,st1)
```

WARNING: To be safe, all multiple precision variables in a user's program should be declared as type (fm), (im), or (zm), and any direct calls to subroutines should be the kind with the underscore. To compute pi, use

```fortran
    call fm_pi(pi)
```

Calling the low-level routine in fm.f95 (call fmpi(pi%mfm)) is not recommended.

In subroutine or function subprograms all multiple precision variables that are local to that routine should be declared with the save attribute. It is not an error to omit save, but if the compiler creates new copies of the variables for each call to the routine, then the program could leak memory.

Type (fm), (im), or (zm) variables cannot have their multiple precision values initialized in the declaration statement, as can ordinary variables. If the original program had

```fortran
    double precision :: x = 2.3d0
```

then the corresponding FM version would have

```fortran
    type (fm), save :: x
    ... (other declarations) ...
    x = to_fm( '2.3' )
```

An attempt to use a multiple precision variable that has not been defined will be detected by the routines in this module and an error message printed.

For each of the operations =, ==, /=, <, <=, >, >=, +, -, *, /, and **, the interface module defines all mixed mode variations involving one of the three multiple precision derived types and another argument having one of the types: { integer, real, double, complex, complex double}. So mixed mode expressions such as

```fortran
    mafm = 12
    mafm = mafm + 1
    if (abs(mafm) > 1.0d-23) then
      ... (other code) ...
    end if
```

are handled correctly.

Not all the named functions are defined for all three multiple precision derived types, so the list below shows which can be used. The labels "real", "integer", and "complex" refer to types fm, im, and zm respectively, "string" means the function accepts character strings (e.g., to_fm('3.45')), and "other" means the function can accept any of the machine precision data types integer, real, double, complex, or complex double. For functions that accept two or more arguments, like atan2 or max, all the arguments must be of the same type.

Note that to_zm also has a 2-argument form: to_zm(2,3) for getting 2 + 3*i. cmplx can be used for that, as in cmplx(to_fm(2),to_fm(3)), but the 2-argument form is more concise. The 2-argument form is available for machine precision integer, single and double precision real pairs. For others, such as x and y being type(fm), just use cmplx(x,y).

Fortran's 2-argument version of atan(x,y) is also provided. It is the same as the older atan2.

Functions in this list that are not provided by standard Fortran, such as special functions, have more information about their arguments farther down.

AVAILABLE FUNCTIONS:
! =
! +
! -
! *
! /
! **
! ==
! /=
! <
! <=
! >
! >=
! abs real integer complex
! acos real complex
! acosh real complex
! aimag complex
! aint real complex
! anint real complex
! arg complex
! asin real complex
! asinh real complex
! atan real complex
! atan2 real
! atanh real complex
! bernoulli real
! bessel_j real
! bessel_y real
! beta real
! binomial real integer complex
! btest integer
! ceiling real integer complex
! cmplx real integer
! conjg complex
! cos real complex
! cosh real complex
! cos_integral real
! cosh_integral real
! dble real integer complex
! digits real integer complex
! dim real integer
! dint real complex
! epsilon real
! erf real complex
! erfc real complex
! erfc_scaled real complex
! exp real complex
! exponent real
! exp_integral_ei real
! exp_integral_en real
! factorial real integer complex
! floor real integer complex
! fraction real complex
! fresnel_c real
! fresnel_s real
! gamma real complex
! gcd integer
! huge real integer complex
! hypot real
! incomplete_beta      real
! incomplete_gamma1    real
! incomplete_gamma2    real
! int                  real    integer    complex
! log                  real    complex
! log10                 real    complex
! log_erfc              real
! log_gamma             real    complex
! log_integral          real
! max                   real    integer
! maxexponent           real
! min                   real    integer
! minexponent           real
! mod                   real    integer
! modulo                real    integer
! multiply_mod          integer
! nearest               real
! nint                  real    integer    complex
! norm2                 real
! pochhammer            real
! polygamma             real    complex
! power_mod             integer
! precision             real    complex
! psi                   real    complex
! radix                 real    integer    complex
! range                 real    integer    complex
! real                   real    integer    complex
! rrspacing             real
! scale                 real    complex
! setexponent           real
! sign                  real    integer
! sin                   real    complex
! sinh                  real    complex
! sin_integral          real
! sinh_integral         real
! spacing               real
! sqrt                  real    complex
! tan                   real    complex
! tanh                  real    complex
! tiny                  real    integer    complex
! to_fm                 real    integer    complex    string    other
! to_im                 real    integer    complex    string    other
! to_zm                 real    integer    complex    string    other
! to_int                real    integer    complex
! to_sp                 real    integer    complex
! to_dp                 real    integer    complex
! to_spz                real    integer    complex
! to_dpz                real    integer    complex
! is_overflow           real    integer    complex
! is_underflow          real    integer    complex
! is_unknown            real    integer    complex

! SUBROUTINES THAT DO NOT CORRESPOND TO ANY FUNCTION ABOVE:

! 1. Type (fm). ma, mb, mc refer to type (fm) numbers.

!   fm_cosh_sinh(ma,mb,mc)     mb = cosh(ma), mc = sinh(ma)
Faster than making two separate calls.

\[ mb = \cos(ma), \ mc = \sin(ma) \]
Faster than making two separate calls.

\[ ma = \text{Euler's constant (0.5772156649...)} \]

\[ k = k\text{flag} \] get the value of the FM condition flag -- stored in the internal FM variable kflag in module fmvals.

\[ ma \text{ is converted to a character string using format form and returned in string. form can represent i, f, e, or es formats.} \]
\[ \text{Example:} \]
\[ \text{call fmform('f60.40',ma,string)} \]

Print \( ma \) on unit \( kw \) using form format.

\[ ma = \pi \]

Print \( ma \) on unit \( kw \) using current format.

\[ x \] is returned as a double precision random number, uniformly distributed on the open interval \((0,1)\). It is a high-quality, long-period generator based on 49-digit prime numbers.
Note that \( x \) is double precision, unlike the similar Fortran intrinsic random number routine, which can return a single or double precision result.
A default initial seed is used if \( \text{fm_random_number} \) is called without calling \( \text{fm_random_seed_put} \) first.

returns the seven integers \( seed(1) \) through \( seed(7) \) as the current seed for the \( \text{fm_random_number} \) generator.

initializes the \( \text{fm_random_number} \) generator using the seven integers \( seed(1) \) through \( seed(7) \). These get and put functions are slower than \( \text{fm_random_number} \), so \( \text{fm_random_number} \) should be called many times between \( \text{fm_random_seed_put} \) calls. Also, some generators that used a 9-digit modulus have failed randomness tests when used with only a few numbers being generated between calls to re-start with a new seed.

returns integer size as the size of the seed array used by the \( \text{fm_random_number} \) generator. Currently, size = 7.

\[ mb = ma^{\frac{k}{j}} \] Rational power.
Faster than \( mb = ma^{\frac{\text{to_fm}(k)}{j}} \) for functions like the cube root.

\[ ma \] is returned after reading one (possibly multi-line) FM number on unit \( k\text{read} \). This routine reads numbers written by \( \text{fm_write} \).

Set the internal FM variables so that the precision is at least \( n\text{prec} \) base 10 digits plus three base 10 guard digits.

Define a new value for one of the internal FM variables in module \( \text{fmvals} \) that controls one of the FM options. string has the form
\[ \text{variable} = \text{value} \]
\[ \text{Example: To change the screen width for FM output:} \]
call fm_setvar(' kswide = 120 ')

The variables that can be changed and the options they control are listed in sections 2 through 6 of the comments at the top of the fm.f95 file. Only one variable can be set per call. The variable name in string must have no embedded blanks. The value part of string can be in any numerical format, except in the case of variable cmchar, which is character type. To set cmchar to 'e', don't use any quotes in string:
call fm_setvar(' cmchar = e ')

fm_ulp(ma,mb) mb = One Unit in the Last Place of ma. For positive ma this is the same as the Fortran function spacing, but mb < 0 if ma < 0.
Examples: If mbase = 10 and ndig = 30, then ulp(1.0) = 1.0e-29, ulp(-4.5e+67) = -1.0e+38.

fm_vars Write the current values of the internal FM variables on unit kw.

fm_write(kwrite,ma) Write ma on unit kwrite.
Multi-line numbers will have '&' as the last nonblank character on all but the last line. These numbers can then be read easily using fm_read.

! 2. Type (im). ma, mb, mc refer to type (im) numbers.

im_divr(ma,mb,mc,md) mc = int(ma/mb), md = ma mod mb
When both the quotient and remainder are needed, this routine is twice as fast as doing mc = ma/mb and md = mod(ma,mb) separately.

im_dvir(ma,ival,mb,irem) mb = int(ma/ival), irem = ma mod ival
ival and irem are one word integers. Faster than doing separately.

im_form(form,ma,string) ma is converted to a character string using format form and returned in string. form can represent i, f, e, or es formats.
Example: call imform('i70',ma,string)

im_fprint(form,ma) Print ma on unit kw using form format.

im_print(ma) Print ma on unit kw.

im_read(kread,ma) ma is returned after reading one (possibly multi-line) IM number on unit kread. This routine reads numbers written by im_write.

im_write(kwrite,ma) Write ma on unit kwrite. Multi-line numbers will have '&' as the last nonblank character on all but the last line. These numbers can then be read easily using im_read.

! 3. Type (zm). ma, mb, mc refer to type (zm) numbers. mbfm is type (fm).

zm_arg(ma,mbfm) mbfm = complex argument of ma. mbfm is the (real) angle in the interval ( -pi , pi ] from the positive real axis to the point (x,y) when ma = x + y*i.

zm_cosh_sinh(ma,mb,mc) mb = cosh(ma), mc = sinh(ma).
Faster than 2 calls.
zm_cos_sin(ma,mb,mc)       mb = cos(ma),  mc = sin(ma).
           Faster than 2 calls.

zm_form(form1,form2,ma,string)
    string = ma
    ma is converted to a character string using format form1 for the
    real part and form2 for the imaginary part. The result is returned
    in string.  form1 and form2 can represent i, f, e, or es formats.
    Example:
    call zmform('f20.10', 'f15.10',ma,string)

zm_fprint(form1,form2,ma)  Print ma on unit kw using formats form1 and form2.

zm_print(ma)               Print ma on unit kw using current format.

zm_read(kread,ma)          ma is returned after reading one (possibly multi-line) ZM number
                           on unit kread. This routine reads numbers written by zmwrite.

zm_rational_power(ma,ival,jval,mb)
    mb = ma ** (ival/jval)
    Faster than mb = ma**(to_fm(k)/j) for functions like the cube root.

zm_write(kwrite,ma)        Write ma on unit kwrite. Multi-line numbers are formatted for
                           automatic reading with zmread.

Some other functions are defined that do not correspond to machine precision intrinsic
functions. These include formatting functions, integer modular functions and gcd, and some
mathematical special functions.
    n, k below are machine precision integers, j1, j2, j3 are type (im), fmt, fmtr, fmti are
    character strings, a, b, x are type (fm), and z is type (zm).
    The three formatting functions return a character string containing the formatted number, the
    three type (im) functions return a type (im) result, and the 12 special functions return
    type (fm) results.

Formatting functions:

fm_format(fmt,a)        Put a into fmt (real) format
im_format(fmt,j1)       Put j1 into fmt (integer) format
zm_format(fmtr,fmti,z)  Put z into (complex) format, fmtr for the real
                        part and fmti for the imaginary part

Examples:
    st = fm_format('f65.60',a)
    write (*,*) ' a = ',trim(st)
    st = fm_format('e75.60',b)
    write (*,*) ' b = ',st(1:75)
    st = im_format('i50',j1)
    write (*,*) ' j1 = ',st(1:50)
    st = zm_format('f35.30', 'f30.25',z)
    write (*,*) ' z = ',st(1:70)

    These functions are used for one-line output. The returned character strings are of
    length 200.

For higher precision numbers, the output can be broken onto multiple lines automatically by
calling subroutines fm_print, im_print, zm_print, or the line breaks can be done by hand after
calling one of the subroutines fm_form, im_form, zm_form.

For zm_format the length of the output is 5 more than the sum of the two field widths.

Integer functions:

- **binomial(n,k)**: Binomial coefficient $n$ choose $k$. Returns the exact result as a type IM value.
- **binomial(j1,j2)**: Binomial coefficient $j1$ choose $j2$. Like factorial below, the result might be too large unless $\min(j2, j1-j2)$ is fairly small.
- **factorial(n)**: $n!$ Returns the exact result as a type IM value.
- **factorial(j1)**: $j1!$ Note that the factorial function grows so rapidly that if type IM variable $j1$ is larger than the largest machine precision integer, then $j1!$ has over 10 billion digits and the calculation would likely fail due to memory or time constraints. This version is provided for convenience, and will return unknown if $j1$ cannot be represented as a machine precision integer.
- **gcd(j1,j2)**: Greatest Common Divisor of $j1$ and $j2$.
- **multiply_mod(j1,j2,j3)**: $j1 \times j2 \mod j3$
- **power_mod(j1,j2,j3)**: $j1^{j2} \mod j3$

Special functions:

- **bernonulli(n)**: $n$th Bernoulli number
- **bessel_j(n,x)**: Bessel function of the first kind $J_n(x)$
- **bessel_j0(x)**: Fortran-08 name for $j_0(x)$
- **bessel_j1(x)**: Fortran-08 name for $j_1(x)$
- **bessel_jn(n,x)**: Fortran-08 name for $J_n(x)$
- **bessel_jn(n1,n2,x)**: Returns array (/ $J_{n1}(x)$, ... , $J_{n2}(x)$ /)
- **bessel_y(n,x)**: Bessel function of the second kind $Y_n(x)$
- **bessel_y0(x)**: Fortran-08 name for $y_0(x)$
- **bessel_y1(x)**: Fortran-08 name for $y_1(x)$
- **bessel_yn(n1,n2,x)**: Returns array (/ $Y_{n1}(x)$, ... , $Y_{n2}(x)$ /)
- **beta(a,b)**: Integral (0 to 1) $t^{(a-1)} \times (1-t)^{(b-1)} \; dt$
- **binomial(a,b)**: Binomial Coefficient $a! / (b! \cdot (a-b)!)$
- **cos_integral(x)**: Cosine Integral $\text{Ci}(x)$
- **cosh_integral(x)**: Hyperbolic Cosine Integral $\text{Chi}(x)$
- **erf(x)**: Error function $\text{Erf}(x)$
- **erfc(x)**: Complimentary error function $\text{Erfc}(x)$
- **erfc_scaled(x)**: $\exp(x^2) \times \text{Erfc}(x)$
- **exp_integral_ei(x)**: Exponential Integral $\text{Ei}(x)$
- **exp_integral_en(n,x)**: Exponential Integral $E_n(x)$
- **factorial(x)**: $x! = \Gamma(x+1)$
- **fresnel_c(x)**: Fresnel Cosine Integral $c(x)$
- **fresnel_s(x)**: Fresnel Sine Integral $s(x)$
- **gamma(x)**: Integral (0 to infinity) $t^{(x-1)} \times \exp(-t) \; dt$
- **incomplete_beta(x,a,b)**: Integral (0 to x) $t^{(a-1)} \times (1-t)^{(b-1)} \; dt$
- **incomplete_gamma1(a,x)**: Integral (0 to x) $t^{(a-1)} \times \exp(-t) \; dt$
- **incomplete_gamma2(a,x)**: Integral (x to infinity) $t^{(a-1)} \times \exp(-t) \; dt$
- **log_erfc(x)**: $\ln(\text{Erfc}(x))$
- **log_gamma(x)**: Analytic continuation of real $\ln(\Gamma(x))$. May differ from complex $\ln(\Gamma(x))$ by an integer multiple of $2\pi i$.
- **log_integral(x)**: Logarithmic Integral $\text{Li}(x)$
- **pochhammer(x,n)**: $x*(x+1)*(x+2)*...*(x+n-1)$
- **polygamma(n,x)**: $n$th derivative of $\psi(x)$
- **psi(x)**: Derivative of $\ln(\Gamma(x))$
- **sin_integral(x)**: Sine Integral $\text{Si}(x)$
! sinh_integral(x)   Hyperbolic Sine Integral Shi(x)

! Array operations:

! Arithmetic operations and functions on arrays of dimension (rank) one or two are supported for
! each of the three multiple-precision types. Binary operations (+-*/-) require both arguments to
! have the same rank and shape.

! Examples:
! type (fm), save, dimension(10) :: a, b
! type (fm), save, dimension(3,3) :: c
! type (im), save, dimension(10) :: j, k
! type (im), save, dimension(3,3) :: l
! ...
! a = 0                       ! Set the whole array to zero
! j = j * k                   ! Set j(i) = j(i) * k(i) for i = 1, ..., 10
! b = a - k                   ! Mixed-mode operations are ok
! c = 7.3d0 * c - (c + 2*l)/3

! Array functions:

! dot_product(x,y)     Dot product of rank 1 vectors of the same type.
! Note that when x and y are complex, the result is not just the sum
! of the products of the corresponding array elements, as it is for
! types FM and IM. For ZM the formula is the sum of
! conjg(x(j)) * y(j).
! is_overflow(x)        Returns true if any element is + or - overflow.
! is_underflow(x)       Returns true if any element is + or - underflow.
! is_unknown(x)         Returns true if any element is unknown.
! matmul(x,y)           Matrix multiplication of arrays of the same type
! Cases for valid argument shapes:
! (1) (n,m) * (m,k) --> (n,k)
! (2) (m) * (m,k) --> (k)
! (3) (n,m) * (m) --> (n)
! maxloc(x)             Location of the maximum value in the array
! maxval(x)             Maximum value in the array
! minloc(x)             Location of the minimum value in the array
! minval(x)             Minimum value in the array
! product(x)            Product of all values in the array
! sum(x)                Sum of all values in the array
! transpose(x)          Matrix transposition. If x is a rank 2 array with shape (n,m), then
!                       y = transpose(x) has shape (m,n) with y(i,j) = x(j,i).
! to_fm(x)              Rank 1 or 2 arrays are converted to similar type (fm) arrays.
! to_im(x)              Rank 1 or 2 arrays are converted to similar type (im) arrays.
! to_zm(x)              Rank 1 or 2 arrays are converted to similar type (zm) arrays.
! to_int(x)             Rank 1 or 2 arrays are converted to similar integer arrays.
! to_sp(x)              Rank 1 or 2 arrays are converted to similar single precision arrays.
! to_dp(x)              Rank 1 or 2 arrays are converted to similar double precision arrays.
! to_spz(x)             Rank 1 or 2 arrays are converted to similar single complex arrays.
! to_dpz(x)             Rank 1 or 2 arrays are converted to similar double complex arrays.

! The arithmetic array functions dot_product, matmul, product, and sum work like the other
! functions in the FM package in that they raise precision and compute the sums and/or products
! at the higher precision, then round the final result back to the user's precision to provide
! a more accurate result.

! Fortran's optional [,mask] argument for these functions is not provided.
Many of the 1-argument functions can be used with array arguments, with the result being an array of the same size and shape where the function has been applied to each element.

Examples:
```fortran
  type (fm), save, dimension(10) :: a, b, c
  ...
  a = abs(b)               ! Set a(i) = abs(b(i)) for i = 1, ..., 10
  c = sqrt(a+4+b*b)         ! Set c(i) = sqrt(a(i)+4+b(i)*b(i)) for i = 1, ..., 10
```

Functions that can have array arguments. As above, "real", "integer", and "complex" refer to types FM, IM, and ZM respectively.

<table>
<thead>
<tr>
<th>Function</th>
<th>Real</th>
<th>Integer</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>acos, acosh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>aimag</td>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aint, anint</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>arg</td>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asin, asinh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>atan, atanh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>ceiling</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>conjg</td>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cos, cosh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>fraction</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>log, log10</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>nint</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>sin, sinh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>tan, tanh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>ceiling</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>conjg</td>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cos, cosh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>fraction</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>log, log10</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>nint</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>sin, sinh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>tan, tanh</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>factorial</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>erf</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>erfc, erfc_scaled</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>exp_integral_ei</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>factorial</td>
<td>real</td>
<td>integer</td>
<td>complex</td>
</tr>
<tr>
<td>fresnel_c</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fresnel_s</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gamma</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>log_erfc</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log_gamma</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>log_integral</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>psi</td>
<td>real</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>sin_integral</td>
<td>real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sinh_integral</td>
<td>real</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
interface to_fm
  module procedure fm_i
  module procedure fm_r
  module procedure fm_d
  module procedure fm_z
  module procedure fm_zd
  module procedure fm_fm
  module procedure fm_im
  module procedure fm_st
  module procedure fm_i1
  module procedure fm_r1
  module procedure fm_d1
  module procedure fm_z1
  module procedure fm_zd1
  module procedure fm_fm1
  module procedure fm_im1
  module procedure fm_zm1
  module procedure fm_st1
  module procedure fm_i2
  module procedure fm_r2
  module procedure fm_d2
  module procedure fm_z2
  module procedure fm_zd2
  module procedure fm_fm2
  module procedure fm_im2
  module procedure fm_zm2
  module procedure fm_st2
end interface

interface to_im
  module procedure im_i
  module procedure im_r
  module procedure im_d
  module procedure im_z
  module procedure im_c
  module procedure im_fm
  module procedure im_im
  module procedure im_zm
  module procedure im_st
  module procedure im_i1
  module procedure im_r1
  module procedure im_d1
  module procedure im_z1
  module procedure im_c1
  module procedure im_fm1
  module procedure im_im1
  module procedure im_zm1
  module procedure im_st1
  module procedure im_i2
  module procedure im_r2
  module procedure im_d2
  module procedure im_z2
  module procedure im_c2
  module procedure im_fm2
  module procedure im_im2
  module procedure im_zm2
  module procedure im_st2
end interface

interface to_zm
    module procedure zm_i
    module procedure zm2_i
    module procedure zm_r
    module procedure zm2_r
    module procedure zm_d
    module procedure zm2_d
    module procedure zm_z
    module procedure zm_c
    module procedure zm_fm
    module procedure zm_im
    module procedure zm_zm
    module procedure zm_st
    module procedure zm_i1
    module procedure zm_r1
    module procedure zm_d1
    module procedure zm_z1
    module procedure zm_c1
    module procedure zm_fm1
    module procedure zm_im1
    module procedure zm_zm1
    module procedure zm_st1
    module procedure zm_i2
    module procedure zm_r2
    module procedure zm_d2
    module procedure zm_z2
    module procedure zm_c2
    module procedure zm_fm2
    module procedure zm_im2
    module procedure zm_zm2
    module procedure zm_st2
end interface

interface to_int
    module procedure fm_2int
    module procedure im_2int
    module procedure zm_2int
    module procedure fm_2int1
    module procedure im_2int1
    module procedure zm_2int1
    module procedure fm_2int2
    module procedure im_2int2
    module procedure zm_2int2
end interface

interface to_sp
    module procedure fm_2sp
    module procedure im_2sp
    module procedure zm_2sp
    module procedure fm_2sp1
    module procedure im_2sp1
    module procedure zm_2sp1
    module procedure fm_2sp2
    module procedure im_2sp2
    module procedure zm_2sp2
end interface
interface to_dp
  module procedure fm_2dp
  module procedure im_2dp
  module procedure zm_2dp
  module procedure fm_2dp1
  module procedure im_2dp1
  module procedure zm_2dp1
  module procedure fm_2dp2
  module procedure im_2dp2
  module procedure zm_2dp2
end interface

interface to_spz
  module procedure fm_2spz
  module procedure im_2spz
  module procedure zm_2spz
  module procedure fm_2spz1
  module procedure im_2spz1
  module procedure zm_2spz1
  module procedure fm_2spz2
  module procedure im_2spz2
  module procedure zm_2spz2
end interface

interface to_dpz
  module procedure fm_2dpz
  module procedure im_2dpz
  module procedure zm_2dpz
  module procedure fm_2dpz1
  module procedure im_2dpz1
  module procedure zm_2dpz1
  module procedure fm_2dpz2
  module procedure im_2dpz2
  module procedure zm_2dpz2
end interface

interface is_overflow
  module procedure fm_is_overflow
  module procedure im_is_overflow
  module procedure zm_is_overflow
  module procedure fm_is_overflow1
  module procedure im_is_overflow1
  module procedure zm_is_overflow1
  module procedure fm_is_overflow2
  module procedure im_is_overflow2
  module procedure zm_is_overflow2
end interface

interface is_underflow
  module procedure fm_is_underflow
  module procedure im_is_underflow
  module procedure zm_is_underflow
  module procedure fm_is_underflow1
  module procedure im_is_underflow1
  module procedure zm_is_underflow1
  module procedure fm_is_underflow2
  module procedure im_is_underflow2
  module procedure zm_is_underflow2
end interface
module procedure zm_is_underflow2
end interface

interface is_unknown
module procedure fm_is_unknown
module procedure im_is_unknown
module procedure zm_is_unknown
module procedure fm_is_unknown1
module procedure im_is_unknown1
module procedure zm_is_unknown1
module procedure fm_is_unknown2
module procedure im_is_unknown2
module procedure zm_is_unknown2
end interface

interface fm_undef_inp
module procedure fm_undef_inp_fm0
module procedure fm_undef_inp_im0
module procedure fm_undef_inp_zm0
module procedure fm_undef_inp_fm1
module procedure fm_undef_inp_im1
module procedure fm_undef_inp_zm1
module procedure fm_undef_inp_fm2
module procedure fm_undef_inp_im2
module procedure fm_undef_inp_zm2
end interface

! The next function is no longer needed in version 1.4.
! Dummy versions of the individual procedures are included for compatibility with version 1.3.

interface fm_deallocate
module procedure fm_deallocate_fm1
module procedure fm_deallocate_im1
module procedure fm_deallocate_zm1
module procedure fm_deallocate_fm2
module procedure fm_deallocate_im2
module procedure fm_deallocate_zm2
end interface

contains

! to_fm

function fm_i(ival) result (return_value)
use fmvals
implicit none
type (fm) :: return_value
integer :: ival
intent (in) :: ival
call fmi2m(ival, return_value%mfm)
end function fm_i

function fm_r(r) result (return_value)
use fmvals
implicit none
type (fm) :: return_value
real :: r
intent (in) :: r
call fmsp2m(r, return_value%mfm)
end function fm_r

function fm_d(d)  result (return_value)
  use fmvals
  implicit none
  type (fm) :: return_value
double precision :: d
  intent (in) :: d
  call fmdp2m(d, return_value%mfm)
end function fm_d

function fm_z(z)  result (return_value)
  use fmvals
  implicit none
  type (fm) :: return_value
  complex :: z
  intent (in) :: z
  call fmsp2m(real(z), return_value%mfm)
end function fm_z

function fm_zd(c)  result (return_value)
  use fmvals
  implicit none
  type (fm) :: return_value
complex (kind(0.0d0)) :: c
  intent (in) :: c
  call fmdp2m(real(c, kind(0.0d0)), return_value%mfm)
end function fm_zd

function fm_fm(ma)  result (return_value)
  use fmvals
  implicit none
type (fm) :: return_value
  type (im) :: ma
  intent (in) :: ma
  call fm_undef_inp(ma)
call fmeq(ma%mfm, return_value%mfm)
end function fm_fm

function fm_im(ma)  result (return_value)
  use fmvals
  implicit none
type (fm) :: return_value
  type (im) :: ma
  intent (in) :: ma
  call fm_undef_inp(ma)
call im2fm(ma%mim, return_value%mfm)
end function fm_im

function fm_zm(ma)  result (return_value)
  use fmvals
  implicit none
type (fm) :: return_value
  type (zm) :: ma
  intent (in) :: ma
  call fm_undef_inp(ma)
call zmrreal(ma%mzm, return_value%mfm)
end function fm_zm

function fm_st(st) result (return_value)
use fmvals
implicit none
type (fm) :: return_value
character(*) :: st
intent (in) :: st
call fmst2m(st, return_value%mfm)
end function fm_st

function fm_i1(ival) result (return_value)
use fmvals
implicit none
integer, dimension(:) :: ival
type (fm), dimension(size(ival)) :: return_value
integer :: j, n
intent (in) :: ival
n = size(ival)
do j = 1, n
call fmi2m(ival(j), return_value(j)%mfm)
enddo
end function fm_i1

function fm_r1(r) result (return_value)
use fmvals
implicit none
real, dimension(:) :: r
type (fm), dimension(size(r)) :: return_value
integer :: j, n
intent (in) :: r
n = size(r)
do j = 1, n
call fmsp2m(r(j), return_value(j)%mfm)
enddo
end function fm_r1

function fm_d1(d) result (return_value)
use fmvals
implicit none
double precision, dimension(:) :: d
type (fm), dimension(size(d)) :: return_value
integer :: j, n
intent (in) :: d
n = size(d)
do j = 1, n
call fmdp2m(d(j), return_value(j)%mfm)
enddo
end function fm_d1

function fm_z1(z) result (return_value)
use fmvals
implicit none
complex, dimension(:) :: z
type (fm), dimension(size(z)) :: return_value
integer :: j, n
intent (in) :: z
n = size(z)
do j = 1, n
    call fmsp2m(real(z(j)), return_value(j)%mfm)
endo
end function fm_z1

function fm_zd1(c) result (return_value)
  use fmvals
  implicit none
  complex (kind(0.0d0)), dimension(:) :: c
  type (fm), dimension(size(c)) :: return_value
  integer :: j, n
  intent (in) :: c
  n = size(c)
  do j = 1, n
    call fmdp2m(real(c(j), kind(0.0d0)), return_value(j)%mfm)
  enddo
end function fm_zd1

function fm_fm1(ma) result (return_value)
  use fmvals
  implicit none
  type (fm), dimension(:) :: ma
  type (fm), dimension(size(ma)) :: return_value
  integer :: j, n
  intent (in) :: ma
  call fm_undef_inp(ma)
  n = size(ma)
  do j = 1, n
    call fmeq(ma(j)%mfm, return_value(j)%mfm)
  enddo
end function fm_fm1

function fm_im1(ma) result (return_value)
  use fmvals
  implicit none
  type (im), dimension(:) :: ma
  type (fm), dimension(size(ma)) :: return_value
  integer :: j, n
  intent (in) :: ma
  call fm_undef_inp(ma)
  n = size(ma)
  do j = 1, n
    call imi2fm(ma(j)%mim, return_value(j)%mfm)
  enddo
end function fm_im1

function fm_zm1(ma) result (return_value)
  use fmvals
  implicit none
  type (zm), dimension(:) :: ma
  type (fm), dimension(size(ma)) :: return_value
  integer :: j, n
  intent (in) :: ma
  call fm_undef_inp(ma)
  n = size(ma)
  do j = 1, n
    call zmreal(ma(j)%mzm, return_value(j)%mfm)
  enddo
end function fm_zm1
end function fm_zm1

function fm_st1(st) result (return_value)
  use fmvals
  implicit none
  character(*) :: st
  type (fm), dimension(size(st)) :: return_value
  integer :: j, n
  intent (in) :: st
  n = size(st)
  do j = 1, n
    call fmst2m(st(j), return_value(j)%mfm)
  enddo
end function fm_st1

function fm_i2(ival) result (return_value)
  use fmvals
  implicit none
  integer, dimension(:,:) :: ival
  type (fm), dimension(size(ival, dim=1), size(ival, dim=2)) :: return_value
  integer :: j, k
  intent (in) :: ival
  do j = 1, size(ival, dim=1)
    do k = 1, size(ival, dim=2)
      call fmi2m(ival(j, k), return_value(j, k)%mfm)
    enddo
  enddo
end function fm_i2

function fm_r2(r) result (return_value)
  use fmvals
  implicit none
  real, dimension(:,:) :: r
  type (fm), dimension(size(r, dim=1), size(r, dim=2)) :: return_value
  integer :: j, k
  intent (in) :: r
  do j = 1, size(r, dim=1)
    do k = 1, size(r, dim=2)
      call fmsp2m(r(j, k), return_value(j, k)%mfm)
    enddo
  enddo
end function fm_r2

function fm_d2(d) result (return_value)
  use fmvals
  implicit none
  double precision, 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 fmdp2m(d(j, k), return_value(j, k)%mfm)
    enddo
  enddo
end function fm_d2

function fm_z2(z) result (return_value)
use fmvals
implicit none
complex, dimension(:,::) :: z

! The rest of the code is similar to the first function, with slight variations.

end function fm_z2

function fm_zd2(c) result (return_value)
use fmvals
implicit none
complex(kind(0.0d0)), dimension(:,::) :: c

end function fm_zd2

function fm_fm2(ma) result (return_value)
use fmvals
implicit none
type (fm), dimension(:,::) :: ma
type (fm), dimension(size(ma, dim=1), size(ma, dim=2)) :: return_value

end function fm_fm2

function fm_im2(ma) result (return_value)
use fmvals
implicit none
type (im), dimension(:,::) :: ma
type (fm), dimension(size(ma, dim=1), size(ma, dim=2)) :: return_value

end function fm_im2

function fm_zm2(ma) result (return_value)
use fmvals
implicit none
type (fm), dimension(:,::) :: ma

end function fm_zm2
use fmvals
implicit none
type (zm), dimension(::,::) :: ma
type (fm), dimension(size(ma, dim=1), size(ma, dim=2)) :: return_value
integer :: j, k
intent (in) :: ma
call fm_undef_inp(ma)
do j = 1, size(ma, dim=1)
do k = 1, size(ma, dim=2)
call zmreal(ma(j, k)%mzm, return_value(j, k)%mfm)
enddo
d PDO
end function fm_zm2

function fm_st2(st) result (return_value)
use fmvals
implicit none
character(*), dimension(::,::) :: st
type (fm), dimension(size(st, dim=1), size(st, dim=2)) :: return_value
integer :: j, k
intent (in) :: st
do j = 1, size(st, dim=1)
do k = 1, size(st, dim=2)
call fmst2m(st(j, k), return_value(j, k)%mfm)
enddo
end function fm_st2

! to_im

function im_i(ival) result (return_value)
use fmvals
implicit none
type (im) :: return_value
integer :: ival
intent (in) :: ival
call imi2m(ival, return_value%mim)
end function im_i

function im_r(r) result (return_value)
use fmvals
implicit none
type (im) :: return_value
real :: r
character(25) :: st
integer :: ival
intent (in) :: r
if (abs(r) < huge(1)) then
  ival = int(r)
call imi2m(ival, return_value%mim)
else
  write (st, '(E25.16)') r
call imst2m(st, return_value%mim)
endif
end function im_r

function im_d(d) result (return_value)
use fmvals