MODULE FMZM_1

! FMZM 1.3                        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 1.3 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

    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. Subroutines that define one or more multiple precision output values, such as computing pi using

    CALL FM_PI(P1)

automatically cause PI to be put into the FM saved variable area of storage. Calling the low-level routine (CALL FMP (PI%MFM)) would cause PI to be treated as an FM temporary variable if PI had not been previously defined in the program. Then the value of PI could be discarded before the program is finished using it.

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 will 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

    DOUBLE PRECISION :: X = 2.3D0

then the corresponding FM version would have

    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, FM, IM, ZM }. So mixed mode expressions such as

    MAFM = 12
    MAFM = MAFM + 1
    IF (ABS(MAFM) > 1.0D-23) THEN

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 \( \text{atan}(x,y) \) is also provided. It is the same as the older \( \text{atan2} \).

**AVAILABLE FUNCTIONS:**

- `=`
- `+`
- `-`
- `*`
- `/`
- `**`
- `==`
- `=/`
- `<`
- `<=`
- `>`
- `>=`
- `ABS` real integer complex
- `ACOS` real complex
- `ACOSH` real complex
- `AIMAG` complex
- `AINT` real complex
- `ANINT` real complex
- `ASIN` real complex
- `ASINH` real complex
- `ATAN` real complex
- `ATAN2` real
- `ATANH` real complex
- `BTEST` integer
- `CEILING` real integer complex
- `CMPLX` real integer
- `CONJG` complex
- `COS` real complex
- `COSH` real complex
- `DBLE` real integer complex
- `DIGITS` real integer complex
- `DIM` real integer
- `DINT` real complex
- `EPSILON` real
- `EXP` real complex
- `EXPONENT` real
- `FLOOR` real integer complex
- `FRACTION` real complex
- `HUGE` real integer complex
- `HYPOT` real
- `INT` real integer complex
- `LOG` real complex
- `LOG10` real complex
- `MAX` real integer
- `MAXEXPONENT` real
- `MIN` real integer
- `MINEXPONENT` real
- `MOD` real integer
- `MODULO` real integer
- `NEAREST` real
- `NINT` real integer complex
! NORM2        real
! PRECISION    real               complex
! RADIX       real    integer    complex
! RANGE       real    integer    complex
! REAL        real    integer    complex
! RRSPOACING  real
! SCALE       real               complex
! SETEXPONENT real
! SIGN        real    integer
! SIN         real               complex
! SINH        real               complex
! 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_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.

! FM_COS_SIN(MA,MB,MC)      MB = COS(MA),  MC = SIN(MA)
! Faster than making two separate calls.

! FM_EULER(MA)             MA = Euler's constant ( 0.5772156649... )

! FM_FLAG(K)               K = KFLAG  get the value of the FM condition flag -- stored in
!                           the internal FM variable KFLAG in module FMVALS.

! FM_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 FMFORM('F60.40',MA,STRING)

! FM_FPRINT(FORM,MA)       Print MA on unit KW using FORM format.

! FM_PI(MA)                MA = pi

! FM_PRINT(MA)             Print MA on unit KW using current format.

! FM_RANDOM_NUMBER(X)      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 returns a single-precision result. A default initial seed is used if \(\text{FM_RANDOM_NUMBER}\) is called without calling \(\text{FM_RANDOM_SEED_PUT}\) first.

\[
\begin{align*}
\text{FM_RANDOM_SEED_GET(SEED)} & \quad \text{returns the seven integers SEED(1) through SEED(7) as the current seed for the \(\text{FM_RANDOM_NUMBER}\) generator.} \\
\text{FM_RANDOM_SEED_PUT(SEED)} & \quad \text{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.} \\
\text{FM_RANDOM_SEED_SIZE(SIZE)} & \quad \text{returns integer SIZE as the size of the SEED array used by the \(\text{FM_RANDOM_NUMBER}\) generator. Currently, SIZE = 7.} \\
\end{align*}
\]

\[
\begin{align*}
\text{FM_RATIONAL_POWER(MA,K,J,MB)} & \quad \text{MB = MA}^{(K/J)} \quad \text{Rational power.} \\
& \quad \text{Faster than MB = MA}^{(\text{TO_FM(K)} / J)} \text{ for functions like the cube root.} \\
\text{FM_READ(KREAD,MA)} & \quad \text{MA is returned after reading one (possibly multi-line) FM number on unit KREAD. This routine reads numbers written by \(\text{FM_WRITE}\).} \\
\text{FM_SET(NPREC)} & \quad \text{Set the internal FM variables so that the precision is at least NPREC base 10 digits plus three base 10 guard digits.} \\
\text{FM_SETVAR(STRING)} & \quad \text{Define a new value for one of the internal FM variables in module FMVALS that controls one of the FM options. STRING has the form variable = value.} \\
& \quad \text{Example: To change the screen width for FM output:} \\
& \quad \quad \text{CALL FM_SETVAR(' KSWIDE = 120 ')} \\
& \quad \text{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:} \\
& \quad \quad \text{CALL FM_SETVAR(' CMCHAR = E ')} \\
\text{FM_ULP(MA,MB)} & \quad \text{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.} \\
& \quad \text{Examples: If MBASE = 10 and NDIG = 30, then ulp(1.0) = 1.0E-29, ulp(-4.5E+67) = -1.0E+38.} \\
\text{FM_VARS} & \quad \text{Write the current values of the internal FM variables on unit KW.} \\
\text{FM_WRITE(KWRITE,MA)} & \quad \text{Write MA on unit KWRITE.} \\
& \quad \text{Multi-line numbers will have '&' as the last nonblank character on all but the last line. These numbers can then be read easily} \\
\end{align*}
\]
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 * J2 mod J3
POWER_MOD(J1,J2,J3)     J1 ** J2 mod J3

Special functions:
BERNOULLI(N)            Nth 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(N,X)          Fortran-08 name for Y_n(x)
BESSEL_YN(N1,N2,X)      Returns array (/ Y_n1(x) , ... , Y_n2(x) /)
BETA(A,B)               Integral (0 to 1)  t**(a-1) * (1-t)**(b-1)  dt
BINOMIAL(A,B)           Binomial Coefficient  a! / ( b! (a-b)! )
COS_INTEGRAL(X)         Cosine Integral Ci(x)
COSH_INTEGRAL(X)        Hyperbolic Cosine Integral Chi(x)
ERF(X)                  Error function Erf(x)
ERFC(X)                 Complimentary error function Erfc(x)
ERFC_SCALED(X)          Exp(x^2) * Erfc(x)
EXP_INTEGRAL_EI(X)      Exponential Integral 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) * exp(-t)  dt
INCOMPLETE_BETA(A,B)    Integral (0 to x)  t**(a-1) * (1-t)**(b-1)  dt
INCOMPLETE_GAMMA1(A,X)  Integral (0 to x)  t**(a-1) * exp(-t)  dt
INCOMPLETE_GAMMA2(A,X)  Integral (x to infinity)  t**(a-1) * exp(-t)  dt
LOG_ERFC(X)             Ln( Erfc(x) )
LOG_GAMMA(X)            Ln( Gamma(x) )
LOG_INTEGRAL(X)         Logarithmic Integral Li(x)
POLYGAMMA(N,X)          Nth derivative of Psi(x)
POCHHAMMER(X,N)         x*(x+1)*(x+2)*...*(x+n-1)
PSI(X)                  Derivative of Ln(Gamma(x))
SIN_INTEGRAL(X)         Sine Integral 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:

! 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.

ABS    real    integer    complex
! ACOS          real       complex
! ACOSH        real       complex
! AIMAG             complex
! AINT          real       complex
! ANINT         real       complex
! ASIN          real       complex
! ASINH        real       complex
! ATAN          real       complex
! ATANH       real       complex
! CEILING     real   integer    complex
! CONJG                      complex
! COS           real       complex
! COSH         real       complex
! EXP           real       complex
! FLOOR       real   integer    complex
! FRACTION   real       complex
! INT          real   integer    complex
! LOG          real       complex
! LOG10      real       complex
! NINT         real   integer    complex
! SIN            real       complex
! SINH        real       complex
! SQRT      real       complex
! TAN          real       complex
! TANH        real       complex
! COS_INTEGRAL real
! COSH_INTEGRAL real
! ERF                      real
! ERFC                     real
! ERFC_SCALED     real
! EXP_INTEGRAL_EI     real
! FACTORIAL     real   integer    complex
! GAMMA     real
! LOG_ERFC     real
! LOG_GAMMA     real
! LOG_INTEGRAL   real
! PSI      real
! SIN_INTEGRAL  real
! SINH_INTEGRAL  real
! Work variables for derived type operations.

TYPE FM
  INTEGER :: MFM = -1
END TYPE

TYPE IM
  INTEGER :: MIM = -1
END TYPE

TYPE ZM
  INTEGER :: MZM(2) = -1
END TYPE
INTEGER, SAVE :: MTFM = -3
INTEGER, SAVE :: MUJM = -3
INTEGER, SAVE :: MVFM = -3
INTEGER, SAVE :: M1FM = -3
INTEGER, SAVE :: M2FM = -3
INTEGER, SAVE :: M3FM = -3
INTEGER, SAVE :: MTIM = -3
INTEGER, SAVE :: MUIM = -3
INTEGER, SAVE :: MVIM = -3
INTEGER, SAVE :: M1IM = -3
INTEGER, SAVE :: M2IM = -3
INTEGER, SAVE :: M3IM = -3
INTEGER, SAVE :: M01 = -3
INTEGER, SAVE :: MTZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: MUZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: MVZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: M1ZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: M2ZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: M3ZM(2) = (/ -3, -3 /)
INTEGER, SAVE :: MZ01(2) = (/ -3, -3 /)
TYPE(FM), SAVE :: MT_FM = FM(-3)
TYPE(IM), SAVE :: MT_IM = IM(-3)
TYPE(ZM), SAVE :: MT_ZM = ZM( (/ -3, -3 / ) )

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_ZM
  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

INTERFACE
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

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 FMEQ_INDEX
  MODULE PROCEDURE FMEQ_INDEX_FM0
  MODULE PROCEDURE FMEQ_INDEX_FM1
  MODULE PROCEDURE FMEQ_INDEX_FM2
  MODULE PROCEDURE FMEQ_INDEX_IM0
  MODULE PROCEDURE FMEQ_INDEX_IM1
  MODULE PROCEDURE FMEQ_INDEX_IM2
  MODULE PROCEDURE FMEQ_INDEX_ZM0
  MODULE PROCEDURE FMEQ_INDEX_ZM1
MODULE PROCEDURE FMEQ_INDEX_ZM2
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

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)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_I
  INTEGER :: IVAL
  INTENT (IN) :: IVAL
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FMI2M(IVAL,FM_I%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_I

FUNCTION FM_R(R)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_R
  REAL :: R
  INTENT (IN) :: R
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FMSP2M(R,FM_R%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_R

FUNCTION FM_D(D)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_D
  DOUBLE PRECISION :: D
  INTENT (IN) :: D

TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
CALL FMDP2M(D,FM_D%MFM)
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_D

FUNCTION FM_Z(Z)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_Z
  COMPLEX :: Z
  INTENT (IN) :: Z
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FMSP2M(REAL(Z),FM_Z%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_Z

FUNCTION FM_ZD(C)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_ZD
  COMPLEX(KIND(0.0D0)) :: C
  INTENT (IN) :: C
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FMDP2M(REAL(C,KIND(0.0D0)),FM_ZD%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ZD

FUNCTION FM_FM(MA)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_FM,MA
  INTENT (IN) :: MA
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FM_UNDEF_INP(MA)
  CALL FMEQ(MA%MFM,FM_FM%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_FM

FUNCTION FM_IM(MA)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_IM
  TYPE (IM) :: MA
  INTENT (IN) :: MA
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FM_UNDEF_INP(MA)
  CALL IMI2FM(MA%MIM,FM_IM%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_IM

FUNCTION FM_ZM(MA)
  USE FMVALS
  IMPLICIT NONE
  TYPE (FM) :: FM_ZM
  TYPE (ZM) :: MA
  INTENT (IN) :: MA
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  CALL FM_UNDEF_INP(MA)
  CALL IMZFM(MA%MIM,FM_IM%MFM)
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ZM
TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
CALL FM_UNDEF_INP(MA)
CALL ZMREAL(MA%MZM,FM_ZM%MFM)
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ZM

FUNCTION FM_ST(ST)
    USE FMVALS
    IMPLICIT NONE
    TYPE (FM) :: FM_ST
    CHARACTER(*) :: ST
    INTENT (IN) :: ST
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    CALL FMST2M(ST,FM_ST%MFM)
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ST

FUNCTION FM_I1(IVAL)
    USE FMVALS
    IMPLICIT NONE
    INTEGER, DIMENSION(:) :: IVAL
    TYPE (FM), DIMENSION(SIZE(IVAL)) :: FM_I1
    INTEGER :: J,N
    INTENT (IN) :: IVAL
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    N = SIZE(IVAL)
    DO J = 1, N
        CALL FMI2M(IVAL(J),FM_I1(J)%MFM)
    ENDDO
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_I1

FUNCTION FM_R1(R)
    USE FMVALS
    IMPLICIT NONE
    REAL, DIMENSION(:) :: R
    TYPE (FM), DIMENSION(SIZE(R)) :: FM_R1
    INTEGER :: J,N
    INTENT (IN) :: R
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    N = SIZE(R)
    DO J = 1, N
        CALL FMSP2M(R(J),FM_R1(J)%MFM)
    ENDDO
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_R1

FUNCTION FM_D1(D)
    USE FMVALS
    IMPLICIT NONE
    DOUBLE PRECISION, DIMENSION(:) :: D
    TYPE (FM), DIMENSION(SIZE(D)) :: FM_D1
    INTEGER :: J,N
    INTENT (IN) :: D
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    N = SIZE(D)
DO J = 1, N
    CALL FMDP2M(D(J),FM_D1(J)%MFM)
ENDDO
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_D1

FUNCTION FM_Z1(Z)
    USE FMVALS
    IMPLICIT NONE
    COMPLEX, DIMENSION(:) :: Z
    TYPE (FM), DIMENSION(SIZE(Z)) :: FM_Z1
    INTEGER :: J,N
    INTENT (IN) :: Z
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    N = SIZE(Z)
    DO J = 1, N
        CALL FMSP2M(REAL(Z(J)),FM_Z1(J)%MFM)
    ENDDO
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_Z1

FUNCTION FM_ZD1(C)
    USE FMVALS
    IMPLICIT NONE
    COMPLEX (KIND(0.0D0)), DIMENSION(:) :: C
    TYPE (FM), DIMENSION(SIZE(C)) :: FM_ZD1
    INTEGER :: J,N
    INTENT (IN) :: C
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    N = SIZE(C)
    DO J = 1, N
        CALL FMDP2M(REAL(C(J),KIND(0.0D0)),FM_ZD1(J)%MFM)
    ENDDO
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ZD1

FUNCTION FM_FM1(MA)
    USE FMVALS
    IMPLICIT NONE
    TYPE (FM), DIMENSION(:) :: MA
    TYPE (FM), DIMENSION(SIZE(MA)) :: FM_FM1
    INTEGER :: J,N
    INTENT (IN) :: MA
    TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
    CALL FM_UNDEF_INP(MA)
    N = SIZE(MA)
    DO J = 1, N
        CALL FMEQ(MA(J)%MFM,FM_FM1(J)%MFM)
    ENDDO
    TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_FM1

FUNCTION FM_IM1(MA)
    USE FMVALS
    IMPLICIT NONE
    TYPE (IM), DIMENSION(:) :: MA
TYPE (FM), DIMENSION (SIZE(MA)) :: FM_IM1
INTEGER :: J,N
INTENT (IN) :: MA
TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
CALL FM_UNDEF_INP(MA)
N = SIZE(MA)
DO J = 1, N
   CALL IMI2FM(MA(J)%MIM,FM_IM1(J)%MFM)
ENDDO
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_IM1

FUNCTION FM_ZM1(MA)
USE FMVALS
IMPLICIT NONE
TYPE (ZM), DIMENSION(:) :: MA
TYPE (FM), DIMENSION (SIZE(MA)) :: FM_ZM1
INTEGER :: J,N
INTENT (IN) :: MA
TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
CALL FM_UNDEF_INP(MA)
N = SIZE(MA)
DO J = 1, N
   CALL ZMREAL(MA(J)%MZM,FM_ZM1(J)%MFM)
ENDDO
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ZM1

FUNCTION FM_ST1(ST)
USE FMVALS
IMPLICIT NONE
CHARACTER(*) , DIMENSION(:) :: ST
TYPE (FM), DIMENSION (SIZE(ST)) :: FM_ST1
INTEGER :: J,N
INTENT (IN) :: ST
TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
N = SIZE(ST)
DO J = 1, N
   CALL FMST2M(ST(J),FM_ST1(J)%MFM)
ENDDO
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_ST1

FUNCTION FM_I2(IVAL)
USE FMVALS
IMPLICIT NONE
INTEGER, DIMENSION(:, :) :: IVAL
TYPE (FM), DIMENSION (SIZE(IVAL,DIM=1),SIZE(IVAL,DIM=2)) :: FM_I2
INTEGER :: J,K
INTENT (IN) :: IVAL
TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
DO J = 1, SIZE(IVAL,DIM=1)
   DO K = 1, SIZE(IVAL,DIM=2)
      CALL FMI2M(IVAL(J,K),FM_I2(J,K)%MFM)
   ENDDO
ENDDO
END FUNCTION FM_I2
TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_I2

FUNCTION FM_R2(R)
  USE FMVALS
  IMPLICIT NONE
  REAL, DIMENSION(:, :) :: R
  TYPE (FM), DIMENSION(SIZE(R, DIM=1), SIZE(R, DIM=2)) :: FM_R2
  INTEGER :: J, K
  INTENT (IN) :: R
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  DO J = 1, SIZE(R, DIM=1)
    DO K = 1, SIZE(R, DIM=2)
      CALL FMSP2M(R(J, K), FM_R2(J, K)%MFM)
    ENDDO
  ENDDO
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_R2

FUNCTION FM_D2(D)
  USE FMVALS
  IMPLICIT NONE
  DOUBLE PRECISION, DIMENSION(:, :) :: D
  TYPE (FM), DIMENSION(SIZE(D, DIM=1), SIZE(D, DIM=2)) :: FM_D2
  INTEGER :: J, K
  INTENT (IN) :: D
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  DO J = 1, SIZE(D, DIM=1)
    DO K = 1, SIZE(D, DIM=2)
      CALL FMDP2M(D(J, K), FM_D2(J, K)%MFM)
    ENDDO
  ENDDO
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_D2

FUNCTION FM_Z2(Z)
  USE FMVALS
  IMPLICIT NONE
  COMPLEX, DIMENSION(:, :) :: Z
  TYPE (FM), DIMENSION(SIZE(Z, DIM=1), SIZE(Z, DIM=2)) :: FM_Z2
  INTEGER :: J, K
  INTENT (IN) :: Z
  TEMPV_CALL_STACK = TEMPV_CALL_STACK + 1
  DO J = 1, SIZE(Z, DIM=1)
    DO K = 1, SIZE(Z, DIM=2)
      CALL FMSP2M(REAL(Z(J, K)), FM_Z2(J, K)%MFM)
    ENDDO
  ENDDO
  TEMPV_CALL_STACK = TEMPV_CALL_STACK - 1
END FUNCTION FM_Z2

FUNCTION FM_ZD2(C)
  USE FMVALS
  IMPLICIT NONE
  COMPLEX (KIND(0.0D0)), DIMENSION(:, :) :: C
  TYPE (FM), DIMENSION(SIZE(C, DIM=1), SIZE(C, DIM=2)) :: FM_ZD2