

! The routines in this package perform multiple precision arithmetic and functions
! on three kinds of numbers.
! FM routines handle floating-point real multiple precision numbers,
! IM routines handle integer multiple precision numbers, and
! ZM routines handle floating-point complex multiple precision numbers.
! References to FM numbers below mean the low-level array form of the number used by the routines
! in FM.f95, and not the derived type (fm) numbers handled by the FMZM module. Logically, both may
! refer to the same multiple precision number, but the syntax for dealing with the two types of
! objects is different. The same is true of references to IM numbers and ZM numbers below.

! These are the basic routines for the FM package, and the expectation is that the user will not
! call these routines directly. The typical usage is for a program to declare multiple precision
! variables with the three derived types defined in module FMZM in file FMZM90.f95. Then that
! module provides the interface between the user's program and the routines in this file. See the
! documentation in the FM_User_Manual.txt file for advice on using the FMZM module.
! The information below is intended as a technical reference on the inner workings of FM, and most
! FM users should not need to study it.

! 1. INITIALIZING THE PACKAGE

! The variables that contain values to be shared by the different routines are located in module
! FMVALS in file FMSAVE.f95. Variables that are described below for controlling various features
! of the FM package are found in this module. They are initialized to default values assuming
! 32-bit integers and 64-bit double precision representation of the arrays holding multiple
! precision numbers. The base and number of digits to be used are initialized to give slightly
! more than 50 decimal digits. Subroutine FMVARS can be used to get a list of these variables
! and their values.

! The intent of module FMVALS is to hide the FM internal variables from the user's program, so that
! no name conflicts can occur. Subroutine FMSETVAR can be used to change the variables listed
! below to new values. It is not always safe to try to change these variables directly by putting
! USE FMVALS into the calling program and then changing them by hand. Some of the saved constants
! depend upon others, so that changing one variable may cause errors if others depending on that
! one are not also changed. FMSETVAR automatically updates any others that depend upon the one
! being changed.

! Subroutine FMSET also initializes these variables. It tries to compute the best value for each,
! and it checks several of the default values set in FMVALS to see that they are reasonable for a
! given machine. FMSET can also be called to set or change the current precision level for the
! multiple precision numbers.

! Calling FMSET is optional starting in version 1.2 of the FM package. In previous versions one
! call was required before any other routine in the package could be used.

! The routine ZMSET from version 1.1 is no longer needed, and the complex operations are
! automatically initialized in FMVALS. It has been left in the package for compatibility with
! version 1.1.

! 2. REPRESENTATION OF FM NUMBERS

! MBASE is the base in which the arithmetic is done. MBASE must be bigger than one, and less than
! or equal to the square root of the largest representable integer. For best efficiency
! MBASE should be large, but no more than about 1/4 of the square root of the largest
! representable integer. Input and output conversions are much faster when MBASE is a
! power of ten.

! NDIG is the number of base MBASE digits that are carried in the multiple precision numbers.
! NDIG must be at least two. The upper limit for NDIG is restricted only by the amount
! of memory available.

! Sometimes it is useful to dynamically vary NDIG during the program. Routine FMEQU should be used
! to round numbers to lower precision or zero-pad them to higher precision when changing NDIG.

! The default value of MBASE is a large power of ten. FMSET also sets MBASE to a large power of
! ten. For an application where another base is used, such as simulating a given machine's base
! two arithmetic, use subroutine FMSETVAR to change MBASE, so that the other internal values
! depending on MBASE will be changed accordingly.

! There are two representations for a floating point multiple precision number. The unpacked
! representation used by the routines while doing the computations is base MBASE and is stored
! in NDIG+3 words. A packed representation is available to store the numbers in compressed form.
! In this format, the NDIG (base MBASE) digits of the mantissa are packed two per word to conserve
! storage. Thus the external, packed form of a number requires (NDIG+1)/2+3 words.

! This version uses double precision arrays to hold the numbers. Version 1.0 of FM used integer
! arrays, which are faster on some machines. The package can be changed to use integer arrays ---
! see section 10 on EFFICIENCY below.

! The unpacked format of a floating multiple precision number is as follows. A number MA refers
! to elements of an array with the multiple precision number stored as follows:

! 1 Sign of the number
! 2 Accuracy
! 3 Exponent of the number
! 4 First digit of the number
! ...
! NDIG+3 Last digit of the number.

! The accuracy is the approximate number of bits of precision of the number. This precision value
! is intended to be used by FM functions that need to monitor cancellation error in addition and
! subtraction. The cancellation monitor code is usually disabled for user calls, and FM functions
! only check for cancellation when they must. Tracking cancellation causes most routines to run
! slower, with addition and subtraction being affected the most.

! The exponent is a power of MBASE and the implied radix point is immediately before the first
! digit of the mantissa. The exponent is a signed integer. The overflow threshold is
! MBASE**(MXEXP+1), and the underflow threshold is MBASE**(-MXEXP-1). This means the valid
! exponents for an FM number can range from -MXEXP to MXEXP+1 (inclusive).
! Every nonzero number is normalized so that the first digit of the mantissa is nonzero.

! For MBASE = 10,000 and NDIG = 4, if MA is the number -pi, it would have these representations:

	Word 1	2	3	4	5	6	7
! Unpacked:	-1	42	1	3	1415	9265	3590
! Packed:	-1	42	1	31415	92653590		

```

! The mantissa has about 42 bits of precision, and the number represented is
!  $(-1)^{(10000**1)}*(.0003141592653590)$ .

! Because of the normalization of the digits with a large base, the equivalent number of base 10
! significant digits for an FM number may be as small as  $\text{LOG}_{10}(\text{MBASE})*(\text{NDIG}-1) + 1$ . In the -pi
! example above, this is  $4*3 + 1 = 13$ .

! In version 1.2 and before, each variable like MA was a fixed-size array. Now the array values
! for all multiple precision numbers are stored together in one dynamic array, MWK. A variable
! name like MA is just an integer index giving the location within MWK where the digits are put.
! This allows the memory management of the package to be more flexible than before, to handle
! large arrays of multiple precision numbers at low precision (30 to 50 s.d.) as well as some
! at high precision (millions of s.d.).

! The integer routines use the FM format to represent numbers, without the number of digits (NDIG)
! being fixed. Integers in IM format are essentially variable precision, using the minimum number
! of words to represent each value.

! The unpacked format is the default. As machines' memories have gotten bigger, few applications
! need the packed format. A program that uses packed format numbers should not use the FMZM module
! or the multiple precision derived types defined in FMZM. Packed numbers are treated as temporary
! values by the routines in FMZM, so mixing packed numbers with derived type operations from FMZM
! means the packed numbers could be deleted before the main program is finished with them.

! For programs using both FM and IM numbers, FM routines should not be called with IM numbers, and
! IM routines should not be called with FM numbers, since the implied value of NDIG used for an IM
! number may not match the explicit NDIG expected by an FM routine. Use the conversion routines
! IMFM2I and IMI2FM to change between the FM and IM formats.

! The format for complex FM numbers (called ZM numbers below) is very similar to that for real FM
! numbers. Each ZM number consists of two FM numbers representing the real and imaginary parts of
! a complex number. If MA is a ZM number, then the real part is MA(1) and the imaginary part is
! MA(2). As with FM, there are packed and unpacked formats for the numbers.

! 3. INPUT/OUTPUT ROUTINES

! All versions of the input routines perform free-format conversion from characters to FM numbers.

! a. Conversion to or from a character array

!     FMINP converts from a character(1) array to an FM number.

!     FMOUT converts an FM number to base 10 and formats it for output as an array of type
!     character(1). The output is left justified in the array, and the format is defined
!     by two variables in module FMVALS, so that a separate format definition does not have
!     to be provided for each output call.

!     JFORM1 and JFORM2 define a default output format.

!     JFORM1 = 0      E   format      ( .314159M+6 )
!               = 1    ES  format      ( 3.14159M+5 )
!               = 2    F   format      ( 314159.000 )

!     JFORM2 is the number of significant digits to display (if JFORM1 = 0 or 1).

```

```

!           If JFORM2 = 0 then a default number of digits is chosen. The default is roughly
!           the full precision of the number.
!   JFORM2 is the number of digits after the decimal point (if JFORM1 = 2).
!           See the FMOUT documentation for more details.

! b. Conversion to or from a character string

!   FMST2M converts from a character string to an FM number.

!   FMFORM converts an FM number to a character string according to a format provided in each
!   call. The format description is more like that of a Fortran FORMAT statement, and
!   integer or fixed-point output is right justified.

! c. Direct read or write

!   FMPRINT uses FMOUT to print one FM number.

!   FMFPRINT uses FMFORM to print one FM number.

!   FMWRITE writes FM numbers for later input using FMREAD.

!   FMREAD reads FM numbers written by FMWRITE.

! The values given to JFORM1 and JFORM2 can be used to define a default output format when FMOUT
! or FMPRINT are called. The explicit format used in a call to FMFORM or FMFPRINT overrides the
! settings of JFORM1 and JFORM2.

! KW is the unit number to be used for standard output from the package, including error and
! warning messages, and trace output.

! For multiple precision integers, the corresponding routines IMINP, IMOUT, IMST2M, IMFORM,
! IMPRINT, IMFPRINT, IMWRITE, and IMREAD provide similar input and output conversions. For output
! of IM numbers, JFORM1 and JFORM2 are ignored and integer format (JFORM1=2, JFORM2=0) is used.

! For ZM numbers, the corresponding routines ZMINP, ZMOUT, ZMST2M, ZMFORM, ZMPRINT, ZMFPRINT,
! ZMWRITE, and ZMREAD provide similar input and output conversions.

! For the output format of ZM numbers, JFORM1 and JFORM2 determine the default format for the
! individual parts of a complex number as with FM numbers.

!   JFORMZ determines the combined output format of the real and imaginary parts.

!   JFORMZ = 1 normal setting      :  1.23 - 4.56 i
!           = 2 use capital I      :  1.23 - 4.56 I
!           = 3 parenthesis format: ( 1.23 , -4.56 )

!   JPRNTZ controls whether to print real and imaginary parts on one line whenever possible.

!   JPRNTZ = 1 print both parts as a single string :
!           1.23456789M+321 - 9.87654321M-123 i
!           = 2 print on separate lines without the 'i' :
!           1.23456789M+321
!           -9.87654321M-123

! For further description of these routines, see section 9 below.

```

! 4. ARITHMETIC TRACING

! NTRACE and LVLTRC control trace printout from the package.

! NTRACE = 0 No output except warnings and errors. (Default)
! = 1 The result of each call to one of the routines is printed in base 10, using FMOUT.
! = -1 The result of each call to one of the routines is printed in internal base MBASE
! format.
! = 2 The input arguments and result of each call to one of the routines is printed in
! base 10, using FMOUT.
! = -2 The input arguments and result of each call to one of the routines is printed in
! base MBASE format.

! LVLTRC defines the call level to which the trace is done. LVLTRC = 1 means only FM routines
! called directly by the user are traced, LVLTRC = 2 also prints traces for FM routines
! called by other FM routines called directly by the user, etc. Default is 1.

! In the above description, internal MBASE format means the number is printed as it appears in the
! array --- the sign, accuracy, exponent, then the NDIG base MBASE digits.

! 5. ERROR CONDITIONS

! KFLAG is a condition value returned by the package after each call to one of the routines.
! Negative values indicate conditions for which a warning message will be printed unless
! KWARN = 0.
! Positive values indicate conditions that may be of interest but are not errors. No warning
! message is printed if KFLAG is nonnegative.

! Subroutine FMFLAG is provided to give the user access to the current condition code. For
! example, to set the user's local variable LFLAG to FM's internal KFLAG value:
! CALL FMFLAG(LFLAG)

! KFLAG = 0 Normal operation.
!
! = 1 One of the operands in FMADD or FMSUB was insignificant with respect to the
! other. This means that in the default (symmetric) rounding mode the result
! is equal to the argument of larger magnitude. KFLAG = 1 is still returned
! with the other three rounding modes (see KROUND below), but the result may
! not be equal to either input argument.
! = 2 In converting an FM number to a one word integer in FMM2I, the FM number was
! not exactly an integer. The next integer toward zero was returned.
!
! = -1 NDIG was less than 2.
! = -2 MBASE was less than 2 or more than MXBASE.
! = -3 An exponent was out of range.
! = -4 Invalid input argument(s) to an FM routine. UNKNOWN was returned.
! = -5 + or - OVERFLOW was generated as a result from an FM routine.
! = -6 + or - UNDERFLOW was generated as a result from an FM routine.
! = -7 The input string (array) to FMINP was not legal.
! = -8 The character array was not large enough in an input or output routine.
! = -9 Precision could not be raised enough to provide all requested guard digits.
! This means the program has run out of memory.
! UNKNOWN was returned.
! = -10 An FM input argument was too small in magnitude to convert to the machine's

! single or double precision in FMM2SP or FMM2DP. Check that the definitions
! of SPMAX and DPMAX in file FMSAVE.f95 are correct for the current machine.
! Zero was returned.
! = -11 Array MBERN is not dimensioned large enough for the requested number of
! Bernoulli numbers.
! = -12 Array MJSUMS is not dimensioned large enough for the number of coefficients
! needed in the reflection formula in FMPGAM.

! When a negative KFLAG condition is encountered, the value of KWARN determines the action to
! be taken.

! KWARN = 0 Execution continues and no message is printed.
! = 1 A warning message is printed and execution continues.
! = 2 A warning message is printed and execution stops.

! The default setting is KWARN = 1.

! When an overflow or underflow is generated for an operation in which an input argument was
! already an overflow or underflow, no additional message is printed. When an unknown result
! is generated and an input argument was already unknown, no additional message is printed.
! In these cases the negative KFLAG value is still returned.

! IM routines handle exceptions like OVERFLOW or UNKNOWN in the same way as FM routines, but there
! are some differences because the number of digits carried for IM numbers is not fixed. For
! example, in computing the product of two large integers FM will try to allocate more space rather
! than returning +OVERFLOW. If this allocation fails, FM will write an error message indicating it
! could not get more memory, and the program will stop. The routines IMMPY_MOD and IMPower_MOD can
! be used to obtain modular products and powers without as much chance of running out of memory.

! 6. OTHER OPTIONS

! KRAD = 0 All angles in the real trigonometric functions and inverse functions are measured
! in degrees.
! = 1 All angles are measured in radians. (Default)

! KROUND = -1 All results are rounded toward minus infinity.
! = 0 All results are rounded toward zero (chopped).
! = 1 All results are rounded to the nearest FM number, or to the value with an even last
! digit if the result is exactly halfway between two FM numbers. (Default)
! = 2 All results are rounded toward plus infinity.

! KSWIDE defines the maximum screen width to be used for all unit KW output. Default is 80.

! KESWCH controls the action taken in FMINP and other input routines for strings like 'E7' that
! have no digits before the exponent field. This is sometimes a convenient abbreviation
! when doing interactive keyboard input.
! KESWCH = 1 causes 'E7' to translate like '1.0E+7'. (Default)
! KESWCH = 0 causes 'E7' to translate like '0.0E+7' and give 0.

! CMCHAR defines the exponent letter to be used for FM variable output.
! Default is 'M', as in 1.2345M+678.
! Change it to 'E' for output to be read by a non-FM program.

! KDEBUG = 0 No error checking is done to see if input arguments are valid and parameters like
! NDIG and MBASE are correct upon entry to each routine. (Default)

! = 1 Some error checking is done. (Slower speed)

! See module FMVALS in file FMSAVE.f95 for additional description of these and other variables
! defining various FM conditions.

! 7. ARRAY DIMENSIONS

! Before version 1.3 the multiple-precision numbers were each stored in a separate array. Now all
! these arrays have been combined into one, MWK. Each multiple-precision number is represented by
! a single integer that is used to access the number in the MWK array. This makes the memory usage
! within the package much more flexible.

! 8. PORTABILITY

! In FMSET several variables are set to machine-dependent values, and many of the variables
! initialized in module FMVALS in file FMSAVE.f95 are checked to see that they have reasonable
! values. FMSET will print warning messages on unit KW for any of the FMVALS variables that
! seem to be poorly initialized.

! If an FM run fails, call FMVARS to get a list of all the FMVALS variables printed on unit KW.
! Setting KDEBUG = 1 at the start may also identify some errors.

! In the routines for special functions, several constants are used that require the machine's
! integer word size to be at least 32 bits.

! 9. LIST OF ROUTINES

! First are the routines that deal with multiple precision real numbers. All of these are
! subroutines except logical function FMCOMPARE.

! MA, MB, MC refer to FM format numbers (i.e., integers as opposed to the derived types with
! integer components that are defined in file FMZM90.f95)

! In Fortran-90 and later versions of the Fortran standard, it is potentially unsafe to use the
! same variable more than once in the calling sequence. The operation MA = MA + MB should not be
! written as

```
!       CALL FMADD(MA,MB,MA)
```

! since the code for the subroutine will not know that the first and third arguments are the same,
! and some code optimizations under the assumption that all three arguments are different could
! cause errors.

! One solution is to use a third array and then put the result back in MA:

```
!       CALL FMADD(MA,MB,MC)
```

```
!       CALL FMEQ(MC,MA)
```

! When the first call is doing one of the "fast" operations like addition, the extra call to move
! the result back to MA can cause a noticeable loss in efficiency. To avoid this, separate
! routines are provided for the basic arithmetic operations when the result is to be returned in
! the same array as one of the inputs.

! A routine name with a suffix of "_R1" returns the result in the first input array, and a suffix
! of "_R2" returns the result in the second input array. The example above would then be:

```
!       CALL FMADD_R1(MA,MB)
```

```

! These routines each have one less argument than the original version, since the output is
! re-directed to one of the inputs. The result array should not be the same as any input array
! when the original version of the routine is used.

! The routines that can be used this way are listed below. For others, like
!   CALL FMEXP(MA,MA)
! the relative cost of doing an extra copy is small. This one should become
!   CALL FMEXP(MA,MB)
!   CALL FMEQ(MB,MA)

! When the derived-type interface is used, as in
!   TYPE (FM), SAVE :: A, B
!   ...
!   A = A + B
! there is no problem putting the result back into A, since the interface routine creates a
! temporary scratch array for the result of A + B.

! For each of these routines there is also a version available for which the argument list is
! the same but all FM numbers are in packed format. The routines using packed numbers have the
! same names except 'FM' is replaced by 'FP' at the start of each name.

! Some of the routine names were restricted to 6 characters in earlier versions of FM. The old
! names have been retained for compatibility, but new names that are longer and more readable
! have been added. For example, the old routine FMCSSN can now also be called as FMCOS_SIN.
! Both old and new names are listed below.

! FMABS(MA,MB)           MB = ABS(MA)
! FMACOS(MA,MB)         MB = ACOS(MA)
! FMACOSH(MA,MB)        MB = ACOSH(MA)
! FMADD(MA,MB,MC)       MC = MA + MB
! FMADD_R1(MA,MB)       MA = MA + MB
! FMADD_R2(MA,MB)       MB = MA + MB
! FMADDI(MA,IVAL)       MA = MA + IVAL   Increment an FM number by a one word integer.
!                                     Note this call does not have an "MB" result
!                                     like FMDIVI and FMMPYI.
! FMASIN(MA,MB)         MB = ASIN(MA)
! FMASINH(MA,MB)        MB = ASINH(MA)
! FMATAN(MA,MB)         MB = ATAN(MA)
! FMATANH(MA,MB)        MB = ATANH(MA)
! FMATAN2(MA,MB,MC)     MC = ATAN2(MA,MB)   < old name: FMATN2 >
! FMBIG(MA)             MA = Biggest FM number less than overflow.

```



```

! FMCHANGEBASE(MA,MB,NEW_MBASE,NEW_NDIG)
!
!           MB is returned with the base NEW_MBASE and precision NEW_NDIG
!           representation of MA, where MA is given in the current base (MBASE)
!           and precision (NDIG).  This routine is primarily meant to be used
!           for input and output conversion when a base is being used that is
!           not a power of ten.

! FMCOMPARE(MA,LREL,MB)   Logical comparison of MA and MB.      < old name: FMCOMP >
!
!           LREL is a character(2) value identifying which of the six comparisons
!           is to be made.
!           Example: IF (FMCOMPARE(MA,'>=',MB)) ...
!           Also can be: IF (FMCOMPARE(MA,'GE',MB)) ...
!           character(1) is ok: IF (FMCOMPARE(MA,'>',MB)) ...

! FMCONS               Set several saved constants that depend on MBASE, the base being used.
!
!           FMCONS should be called immediately after changing MBASE.

! FMCOS(MA,MB)         MB = COS(MA)

! FMCOS_SIN(MA,MB,MC)  MB = COS(MA),  MC = SIN(MA).      < old name: FMCSSN >
!
!           Faster than making two separate calls.

! FMCOSH(MA,MB)        MB = COSH(MA)

! FMCOSH_SINH(MA,MB,MC) MB = COSH(MA),  MC = SINH(MA).    < old name: FMCHSH >
!
!           Faster than making two separate calls.

! FMDIG(NSTACK,KST)    Find a set of precisions to use during Newton iteration for finding a
!
!           simple root starting with about double precision accuracy.

! FMDIM(MA,MB,MC)      MC = DIM(MA,MB)

! FMDIV(MA,MB,MC)      MC = MA / MB

! FMDIV_R1(MA,MB)      MA = MA / MB

! FMDIV_R2(MA,MB)      MB = MA / MB

! FMDIVI(MA,IVAL,MB)   MB = MA/IVAL   IVAL is a one word integer.

! FMDIVI_R1(MA,IVAL)   MA = MA/IVAL

! FMDP2M(X,MA)         MA = X   Convert from double precision to FM.

! FMDPM(X,MA)          MA = X   Convert from double precision to FM.
!
!           Faster than FMDP2M, but MA agrees with X only to D.P.
!           accuracy.  See the comments in the two routines.

! FMEQ(MA,MB)          MB = MA   Both have precision NDIG.
!
!           This is the version to use for standard B = A statements.

! FMEQU(MA,MB,NA,NB)   MB = MA   Version for changing precision.
!
!           MA has NA digits (i.e., MA was computed using NDIG = NA), and
!           MB will be defined having NB digits.
!           MB is rounded if NB < NA
!           MB is zero-padded if NB > NA

```

```

! FMEXP(MA,MB)           MB = EXP(MA)

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

! FMFORM(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)

! FMFPRINT(FORM,MA)     Print MA on unit KW using FORM format.      < old name: FMFPRT >

! FMHYPOT(MA,MB,MC)     MA = SQRT(MA**2 + MB**2)

! FMI2M(IVAL,MA)       MA = IVAL   Convert from one word integer to FM.

! FMINP(LINE,MA,LA,LB)  MA = LINE   Input conversion.
!                       Convert LINE(LA) through LINE(LB) from characters to FM.

! FMINT(MA,MB)         MB = INT(MA)  Integer part of MA.

! FMIPOWER(MA,IVAL,MB)  MB = MA**IVAL  Raise an FM number to a one word integer power.
!                       < old name: FMIPWR >

! FMLOG10(MA,MB)       MB = LOG10(MA)  < old name: FMLG10 >

! FMLN(MA,MB)          MB = LOG(MA)

! FMLNI(IVAL,MA)       MA = LOG(IVAL)  Natural log of a one word integer.

! FMM2DP(MA,X)         X = MA       Convert from FM to double precision.

! FMM2I(MA,IVAL)       IVAL = MA     Convert from FM to integer.

! FMM2SP(MA,X)         X = MA       Convert from FM to single precision.

! FMMAX(MA,MB,MC)      MC = MAX(MA,MB)

! FMMIN(MA,MB,MC)      MC = MIN(MA,MB)

! FMMOD(MA,MB,MC)      MC = MA mod MB

! FMMPY(MA,MB,MC)      MC = MA * MB

! FMMPY_R1(MA,MB)      MA = MA * MB

! FMMPY_R2(MA,MB)      MB = MA * MB

! FMMPYI(MA,IVAL,MB)   MB = MA*IVAL  Multiply by a one word integer.

! FMMPYI_R1(MA,IVAL)   MA = MA*IVAL

! FMNINT(MA,MB)        MB = NINT(MA)  Nearest FM integer.

! FMNORM2(MA,N,MB)     MB = SQRT( MA(1)**2 + MA(2)**2 + ... + MA(N)**2 )

```

```

! FMOUT(MA,LINE,LB)      LINE = MA   Convert from FM to character.
!                          LINE is a character array of length LB.

! FMPI(MA)               MA = pi

! FMPRINT(MA)            Print MA on unit KW using current format.    < old name: FMPRNT >

! FMPOWER(MA,MB,MC)     MC = MA**MB    < old name: FMPWR >

! 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 FM_RANDOM_NUMBER is called without
!                       calling FM_RANDOM_SEED_PUT first. See the comments in section 11 below
!                       and also those in the routine for more details.

! FM_RANDOM_SEED_GET(SEED) returns the seven integers SEED(1) through SEED(7) as the current seed
!                       for the FM_RANDOM_NUMBER generator.

! FM_RANDOM_SEED_PUT(SEED) initializes the FM_RANDOM_NUMBER generator using the seven integers
!                       SEED(1) through SEED(7). These get and put functions are slower than
!                       FM_RANDOM_NUMBER, so FM_RANDOM_NUMBER should be called many times
!                       between 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.

! FM_RANDOM_SEED_SIZE(SIZE) returns integer SIZE as the size of the SEED array used by the
!                       FM_RANDOM_NUMBER generator. Currently, SIZE = 7.

! FMRATIONAL_POWER(MA,K,J,MB)
!                       MB = MA**(K/J) Rational power.    < old name: FMRPWR >
!                       Faster than FMPOWER for functions like the cube root.

! FMREAD(KREAD,MA)      MA   is returned after reading one (possibly multi-line) FM number
!                       on unit KREAD. This routine reads numbers written by FMWRITE.

! FMSET(NPREC)          Set the internal FM variables so that the precision is at least NPREC
!                       base 10 digits plus three base 10 guard digits.

! FMSETVAR(STRING)     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.
!                       Example: To change the screen width for FM output:
!                               CALL FMSETVAR(' KSWIDE = 120 ')
!                       The variables that can be changed and the options they control are
!                       listed in sections 2 through 6 above. 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 FMSETVAR(' CMCHAR = E ')

! FMSIGN(MA,MB,MC)     MC = SIGN(MA,MB) Returns the absolute value of MA times the sign
!                       of MB.

```

```

! FMSIN(MA,MB)           MB = SIN(MA)
! FMSINH(MA,MB)         MB = SINH(MA)
! FMSP2M(X,MA)          MA = X   Convert from single precision to FM.
! FMSQR(MA,MB)          MB = MA * MA   Faster than FMMPY.
! FMSQR_R1(MA)          MA = MA * MA
! FMSQRT(MA,MB)         MB = SQRT(MA)
! FMSQRT_R1(MA)         MA = SQRT(MA)
! FMST2M(String,MA)     MA = String
!                       Convert from character string to FM.  String may be in any
!                       numerical format.  FMST2M is often more convenient than FMINP,
!                       which converts an array of character(1) values.  Example:
!                       CALL FMST2M('123.4',MA)
! FMSUB(MA,MB,MC)       MC = MA - MB
! FMSUB_R1(MA,MB)       MA = MA - MB
! FMSUB_R2(MA,MB)       MB = MA - MB
! FMTAN(MA,MB)          MB = TAN(MA)
! FMTANH(MA,MB)         MB = TANH(MA)
! FMTINY(MA)            MA = Smallest positive FM number greater than underflow.
! FMULP(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.
! FMVARS                Write the current values of the internal FM variables on unit KW.
! FMWRITE(KWRITE,MA)    Write MA on unit KWRITE.      < old name: FMWRIT >
!                       Multi-line numbers will have '&' as the last nonblank character on all
!                       but the last line.  These numbers can then be read easily using FMREAD.
!
! These are the available mathematical special functions.
! FMBERNOULLI(N,MA)     MA = B(N)      Nth Bernoulli number
! FMBESJ(N,MA,MB)      MB = J(N,MA)   Bessel function of the first kind
! FMBESJ2(N1,N2,MA,MB) MB = (/ J(N1,MA) , ..., J(N2,MA) /) returns an array
! FMBESY(N,MA,MB)      MB = Y(N,MA)   Bessel function of the second kind
! FMBESY2(N1,N2,MA,MB) MB = (/ Y(N1,MA) , ..., Y(N2,MA) /) returns an array

```

! FMBETA(MA,MB,MC) MC = Beta(MA,MB)

! FMC(MA,MB) MB = C(MA) Fresnel Cosine Integral

! FMCHI(MA,MB) MB = Chi(MA) Hyperbolic Cosine Integral

! FMCI(MA,MB) MB = Ci(MA) Cosine Integral

! FMCOMB(MA,MB,MC) MC = Combination MA choose MB (Binomial coefficient)

! FMEI(MA,MB) MB = Ei(MA) Exponential Integral

! FMEN(N,MA,MB) MB = E(N,MA) Exponential Integral E_n

! FMERF(MA,MB) MB = Erf(MA) Error function

! FMERFC(MA,MB) MB = Erfc(MA) Complimentary Error function

! FMERFC_SCALED(MA,MB) MB = Exp(x**2) * Erfc(MA)

! FMEULER(MA) MA = Euler's constant (0.5772156649...) < old name: FMEULR >

! FMFACT(MA,MB) MB = MA Factorial (Gamma(MA+1))

! FMGAM(MA,MB) MB = Gamma(MA)

! FMIBTA(MX,MA,MB,MC) MC = Incomplete Beta(MX,MA,MB)

! FMIGM1(MA,MB,MC) MC = Incomplete Gamma(MA,MB). Lower case Gamma(a,x)

! FMIGM2(MA,MB,MC) MC = Incomplete Gamma(MA,MB). Upper case Gamma(a,x)

! FMLERC(MA,MB) MB = Ln(Erfc(MA)) Log Erfc

! FMLI(MA,MB) MB = Li(MA) Logarithmic Integral

! FMLNGM(MA,MB) MB = Ln(Gamma(MA))

! FMPGAM(N,MA,MB) MB = Polygamma(N,MA) (Nth derivative of Psi)

! FMPOCH(MA,N,MB) MB = MA*(MA+1)*(MA+2)*...*(MA+N-1) (Pochhammer)

! FMPSI(MA,MB) MB = Psi(MA) (Derivative of Ln(Gamma(MA)))

! FMS(MA,MB) MB = S(MA) Fresnel Sine Integral

! FMShI(MA,MB) MB = Shi(MA) Hyperbolic Sine Integral

! FMShI(MA,MB) MB = Si(MA) Sine Integral

! These are the routines that deal with multiple precision integer numbers.
! All are subroutines except logical function IMCOMPARE. MA, MB, MC refer to IM format numbers.
! In each case the version of the routine to handle packed IM numbers has the same name, with
! 'IM' replaced by 'IP'.

```

!  IMABS(MA,MB)           MB = ABS(MA)

!  IMADD(MA,MB,MC)       MC = MA + MB

!  IMBIG(MA)             MA = 10**(10**6).
!                          Larger IM numbers can be obtained, but setting MA to the largest
!                          possible value would leave no room for any other numbers.

!  IMCOMPARE(MA,LREL,MB) Logical comparison of MA and MB.      < old name: IMCOMP >
!                          LREL is a character(2) value identifying which of the six comparisons
!                          is to be made.
!                          Example: IF (IMCOMPARE(MA,'GE',MB)) ...
!                          Also can be: IF (IMCOMPARE(MA,'>=',MB))
!                          character(1) is ok: IF (IMCOMPARE(MA,'>',MB)) ...

!  IMDIM(MA,MB,MC)       MC = DIM(MA,MB)

!  IMDIV(MA,MB,MC)       MC = int(MA/MB)
!                          Use IMDIVR if the remainder is also needed.

!  IMDIVI(MA,IVAL,MB)    MB = int(MA/IVAL)
!                          IVAL is a one word integer. Use IMDVIR to get the remainder also.

!  IMDIVR(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 calling both IMDIV and IMMOD.

!  IMDVIR(MA,IVAL,MB,IREM) MB = int(MA/IVAL),   IREM = MA mod IVAL
!                          IVAL and IREM are one word integers.

!  IMEQ(MA,MB)           MB = MA

!  IMFM2I(MAFM,MB)       MB = MAFM Convert from real (FM) format to integer (IM) format.

!  IMFORM(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)

!  IMFPRINT(FORM,MA)     Print MA on unit KW using FORM format.      < old name: IMFPR >

!  IMGCD(MA,MB,MC)       MC = greatest common divisor of MA and MB.

!  IMI2FM(MA,MBFM)       MBFM = MA Convert from integer (IM) format to real (FM) format.

!  IMI2M(IVAL,MA)        MA = IVAL Convert from one word integer to IM.

!  IMINP(LINE,MA,LA,LB)  MA = LINE Input conversion.
!                          Convert LINE(LA) through LINE(LB) from characters to IM.

!  IMM2DP(MA,X)          X = MA Convert from IM to double precision.

!  IMM2I(MA,IVAL)        IVAL = MA Convert from IM to one word integer.

!  IMM2SP(MA,X)          X = MA Convert from IM to single precision.

```

```

! IMMAX(MA,MB,MC)          MC = MAX(MA,MB)
! IMMIN(MA,MB,MC)         MC = MIN(MA,MB)
! IMMOD(MA,MB,MC)         MC = MA mod MB
! IMMPY(MA,MB,MC)         MC = MA*MB
! IMMPYI(MA,IVAL,MB)      MB = MA*IVAL   Multiply by a one word integer.
! IMMPY_MOD(MA,MB,MC,MD)  MD = MA*MB mod MC   < old name: IMMPYM >
!                          Slightly faster than calling IMMPY and IMMOD separately.
! IMOUT(MA,LINE,LB)       LINE = MA   Convert from IM to character.
!                          LINE is a character array of length LB.
! IMPOWER(MA,MB,MC)       MC = MA**MB   < old name: IMPWR >
! IMPOWER_MOD(MA,MB,MC,MD) MD = MA**MB mod MC   < old name: IMPMOD >
! IMPRINT(MA)              Print MA on unit KW.   < old name: IMPRNT >
! IMREAD(KREAD,MA)        MA   is returned after reading one (possibly multi-line)
!                          IM number on unit KREAD.
!                          This routine reads numbers written by IMWRITE.
! IMSIGN(MA,MB,MC)        MC = SIGN(MA,MB)   Returns the absolute value of MA times the
!                          sign of MB.
! IMSQR(MA,MB)            MB = MA*MA   Faster than IMMPY.
! IMST2M(String,MA)       MA = STRING
!                          Convert from character string to IM.
!                          IMST2M is often more convenient than IMINP, which converts
!                          an array of character(1) values.  Example:
!                          CALL IMST2M('12345678901',MA)
! IMSUB(MA,MB,MC)         MC = MA - MB
! IMWRITE(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 IMREAD.

! These are the routines that deal with multiple precision complex numbers.
! All are subroutines, and in each case the version of the routine to handle packed ZM numbers has
! the same name, with 'ZM' replaced by 'ZP'.

! MA, MB, MC refer to ZM format complex numbers.
! MAFM, MBFM, MCFM refer to FM format real numbers.
! INTEG is a Fortran INTEGER variable.
! ZVAL is a Fortran COMPLEX variable.

! ZMABS(MA,MBFM)          MBFM = ABS(MA)   Result is real.
! ZMACOS(MA,MB)          MB = ACOS(MA)

```

```

! ZMACOSH(MA,MB)          MB = ACOSH(MA)

! ZMADD(MA,MB,MC)        MC = MA + MB

! ZMADDI(MA,INTEG)      MA = MA + INTEG  Increment an ZM number by a one word integer.
!                               Note this call does not have an "MB" result
!                               like ZMDIVI and ZMMPYI.

! ZMARG(MA,MBFM)        MBFM = Argument(MA)    Result is real.

! ZMASIN(MA,MB)         MB = ASIN(MA)

! ZMASINH(MA,MB)        MB = ASINH(MA)

! ZMATAN(MA,MB)         MB = ATAN(MA)

! ZMATANH(MA,MB)        MB = ATANH(MA)

! ZMCOMPLEX(MAFM,MBFM,MC)  MC = CMPLX(MAFM,MBFM)    < old name: ZMCMPX >

! ZMCONJUGATE(MA,MB)     MB = CONJG(MA)    < old name: ZMCONJ >

! ZMCOS(MA,MB)          MB = COS(MA)

! ZMCOS_SIN(MA,MB,MC)   MB = COS(MA), MC = SIN(MA).    < old name: ZMCSSN >
!                               Faster than 2 calls.

! ZMCOSH(MA,MB)         MB = COSH(MA)

! ZMCOSH_SINH(MA,MB,MC)  MB = COSH(MA), MC = SINH(MA).    < old name: ZMCHSH >
!                               Faster than 2 calls.

! ZMDIV(MA,MB,MC)       MC = MA / MB

! ZMDIVI(MA,INTEG,MB)   MB = MA / INTEG

! ZMEQ(MA,MB)           MB = MA

! ZMEQU(MA,MB,NDA,NDB)  MB = MA    Version for changing precision.
!                               (NDA and NDB are as in FMEQU)

! ZMEXP(MA,MB)          MB = EXP(MA)

! ZMFORM(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)

! ZMFPRT(FORM1,FORM2,MA) Print MA on unit KW using formats FORM1 and FORM2.
!                               < old name: ZMFPRT >

! ZMI2M(INTEG,MA)       MA = CMPLX(INTEG,0)

```



```

! ZM2I2M(INTEG1,INTEG2,MA) MA = CMPLX(INTEG1,INTEG2)

! ZMIMAG(MA,MBFM) MBFM = IMAG(MA) Imaginary part.

! ZMINP(LINE,MA,LA,LB) MA = LINE Input conversion.
! Convert LINE(LA) through LINE(LB) from characters to ZM.
! LINE is a character array of length at least LB.

! ZMINT(MA,MB) MB = INT(MA) Integer part of both Real and Imaginary parts of MA.

! ZMIPOWER(MA,INTEG,MB) MB = MA ** INTEG Integer power function. < old name: ZMIPWR >

! ZMLOG10(MA,MB) MB = LOG10(MA) < old name: ZMLG10 >

! ZMLN(MA,MB) MB = LOG(MA)

! ZMM2I(MA,INTEG) INTEG = INT(REAL(MA))

! ZMM2Z(MA,ZVAL) ZVAL = MA

! ZMMPY(MA,MB,MC) MC = MA * MB

! ZMMPYI(MA,INTEG,MB) MB = MA * INTEG

! ZMNINT(MA,MB) MB = NINT(MA) Nearest integer of both Real and Imaginary.

! ZMOUT(MA,LINE,LB,LAST1,LAST2)
! LINE = MA
! Convert from FM to character.
! LINE is the returned character(1) array.
! LB is the dimensioned size of LINE.
! LAST1 is returned as the position in LINE of the last character
! of REAL(MA)
! LAST2 is returned as the position in LINE of the last character
! of AIMAG(MA)

! ZMPOWER(MA,MB,MC) MC = MA ** MB < old name: ZMPWR >

! ZMPRINT(MA) Print MA on unit KW using current format. < old name: ZMPRNT >

! ZMRATIONAL_POWER(MA,IVAL,JVAL,MB)
! MB = MA ** (IVAL/JVAL) < old name: ZMRPWR >

! ZMREAD(KREAD,MA) MA is returned after reading one (possibly multi-line) ZM number on
! unit KREAD. This routine reads numbers written by ZMWRITE.

! ZMREAL(MA,MBFM) MBFM = REAL(MA) Real part.

! ZMSET(NPREC) Set precision to the equivalent of a few more than NPREC base 10
! digits. This is now the same as FMSET, but is retained for
! compatibility with earlier versions of the package.

! ZMSIN(MA,MB) MB = SIN(MA)

! ZMSINH(MA,MB) MB = SINH(MA)

```

```

! ZMSQR(MA,MB)          MB = MA*MA    Faster than ZMMPY.
! ZMSQRT(MA,MB)        MB = SQRT(MA)
! ZMST2M(String,MA)    MA = STRING
!                      Convert from character string to ZM. ZMST2M is often more
!                      convenient than ZMINP, which converts an array of character(1)
!                      values. Example:
!                      CALL ZMST2M('123.4+5.67i',MA).
! ZMSUB(MA,MB,MC)      MC = MA - MB
! ZMTAN(MA,MB)         MB = TAN(MA)
! ZMTANH(MA,MB)       MB = TANH(MA)
! ZMWRITE(KWRITE,MA)   Write MA on unit KWRITE. Multi-line numbers are formatted for
!                      automatic reading with ZMREAD.    < old name: ZMWRT >
! ZMZ2M(ZVAL,MA)      MA = ZVAL

```

! 10. EFFICIENCY

! To take advantage of hardware architecture on different machines, the package has been designed so that the arithmetic used to perform the multiple precision operations can easily be changed. All variables that must be changed to get a different arithmetic have names beginning with 'M' and are declared using REAL (KIND(1.0D0)) ...

! For example, to change the package to use integer arithmetic internally, make these two changes everywhere in the FM.f95 file.

! Change 'REAL (KIND(1.0D0))' to 'INTEGER'.

! Change 'AINT (' to '('. Note the blank between AINT and (.

! In many places in FM, an AINT function is not supposed to be changed. These are written 'AINT(', with no embedded blank, so they will not be changed by the global change above.

! The first of these changes must also be made throughout the file FMsave.f95.

! Change 'REAL (KIND(1.0D0))' to 'INTEGER'.

! This version of FM restricts the base used to be also representable in integer variables, so using precision above double usually does not save much time unless integers can also be declared at a higher precision. Using IEEE Extended would allow a base of around 10^{**9} to be chosen, but the delayed digit-normalization method used for multiplication and division means that a slightly smaller base like 10^{**8} would probably run faster. This would usually not be much faster than using the usual base 10^{**7} with double precision.

! For cases where special compiler directives or minor re-writing of the code may improve speed, several of the most important loops in FM are identified by comments containing the string '(Inner Loop)'.

! 11. NEW FOR VERSION 1.3

! This is the second edition of version 1.3. The first edition appeared in ACM Transactions on

! Mathematical Software (2-2011). Since then several additions have been made.

! (a) New Fortran-08 functions are available in FMZM

! ACOSH(X), ASINH(X), ATANH(X) for real and complex X

! ATAN(X,Y) can be used in place of ATAN2(X,Y)

! BESSEL_J0(X), BESSEL_J1(X), BESSEL_JN(N,X), BESSEL_JN(N1,N2,X)

! BESSEL_Y0(X), BESSEL_Y1(X), BESSEL_YN(N,X), BESSEL_YN(N1,N2,X)

! The older FM names, BESSEL_J(N,X) and BESSEL_Y(N,X) are still available.

! ERFC_SCALED(X) for $\exp(x**2) * \operatorname{erfc}(x)$

! The older FM function LOG_ERFC(X) is also still available for avoiding underflow in erfc .

! HYPOT(X,Y) for $\sqrt{x**2 + y**2}$

! NORM2(A) for $\sqrt{a(1)**2 + a(2)**2 + \dots + a(n)**2}$

! This could previously have been done with array operations as $\operatorname{SQRT}(\operatorname{DOT_PRODUCT}(A,A))$.

! (b) Many of the elementary and special functions are now faster, after some code-tuning was done and a few new methods were added.

! The routines for the exponential integral function and related mathematical special functions are new in version 1.3. These routines are:

! FMBESJ, FMBESY, FMC, FMCHI, FMCI, FMEI, FMEN, FMERF, FMERFC, FMLERC, FMLI, FMS, FMSHI, FMSI.

! Some of the routines were moved between files FM.f95 and FMZM90.f95 so that now all routines using the module FMZM (in file FMZM90.f95) for multiple precision derived types and operator overloading are located in FMZM90.f95. This means that programs not using derived types can skip compiling and/or linking FMZM90.f95.

! The array function DOTPRODUCT in FMZM has been re-named DOT_PRODUCT to agree with the Fortran standard. For type ZM complex arguments its definition has been changed to agree with the Fortran intrinsic function. When X and Y are complex, DOT_PRODUCT(X,Y) is not just the sum of the products of the corresponding array elements, as it is for types FM and IM. For type ZM, the formula is the sum of $\operatorname{conjg}(X(j)) * Y(j)$. This definition is used so that the complex dot product will be an inner product in the mathematical sense.

! New routines have been added to module FMZM to provide array syntax for the three multiple precision derived types. This means statements like $V = 1$ and $A = B + C$ now work when these variables are vectors or matrices of multiple precision numbers.

! One routine from FM 1.2 has been split into three routines in version 1.3. The routine FM_RANDOM_SEED from FM 1.2 has become three subroutines, so that the optional arguments and the need for an explicit interface can be avoided. See the three routines starting with FM_RANDOM_SEED in the list above. The same multiplicative congruential generator as before is used, but the shuffling of those values has been removed, so that saving seeds and re-starting the generator now works more like the standard Fortran random function.

! Multiple precision variables were separate fixed-size arrays in previous versions. Now they are single integers that serve as index values to a single large array (MWK, defined in file FMSAVE.f95) where the actual values are stored. This often improves both efficiency and memory utilization, since many compilers implemented the derived type operations using copy in and copy out of the arguments for a given operation. Copying entire arrays was slower, and there were often memory leaks when the compiler automatically created temporary derived type objects while evaluating derived type expressions. The static arrays in previous versions also meant that memory was wasted when only a few kinds of operations were used at high precision. Now the space needed by any unused operations never gets allocated.

! Some new error checking is now done for the derived type multiple precision variables. Attempting to use an undefined variable will cause an error message to be printed.

! Much higher precision can be attained in version 1.3, since machines are faster and have more

! memory. To support higher precision, a routine for FFT-based multiplication has been included,
! and when precision gets high enough, the algorithms for multiplication, division, squares, square
! roots, etc., will switch to the FFT routine.

! Binary splitting algorithms are used for the mathematical constants at high precision. At the
! time version 1.3 was released, computing a million digits of e, pi, or the logarithm of a small
! integer took a few seconds, while a million digits of Euler's constant took a few minutes.

! Perfect rounding is now done all the time. In version 1.2 perfect rounding was an option, but
! the default rounding could round the wrong direction once every few million operations, when the
! exact result was very close to halfway between two adjacent representable numbers.

! -----
! -----

SUBROUTINE FMSET(NPREC)

! Initialize the global FM variables that must be set before calling other FM routines.
! These variables are initialized to fairly standard values in the FMSAVE.f95 file (MODULE FMVALS),
! so calling FMSET at the beginning of a program is now optional. FMSET is a convenient way to set
! or change the precision being used, and it also checks to see that the generic values chosen for
! several machine-dependent variables are valid.

! Base and precision will be set to give at least NPREC+3 decimal digits of precision (giving the
! user at least three base ten guard digits). When the base is large, each extra word contains
! several extra digits when viewed in base ten. This means that some choices of NPREC will give
! a few more than three base ten guard digits.

! MBASE (base for FM arithmetic) is set to a large power of ten.
! JFORM1 and JFORM2 (default output format controls) are set to ES format displaying NPREC
! significant digits.

! Several FM options were set here in previous versions of the package, and are now initialized to
! their default values in module FMVALS.
! Here are the initial settings:

! The trace option is set off.
! The mode for angles in trig functions is set to radians.
! The rounding mode is set to symmetric rounding.
! Warning error message level is set to 1.
! Cancellation error monitor is set off.
! Screen width for output is set to 80 columns.
! The exponent character for FM output is set to 'M'.
! Debug error checking is set off.

USE FMVALS
IMPLICIT NONE

INTEGER :: NPREC

REAL (KIND(1.0D0)) :: MAXINT_CHK, MXEXP2_CHK, MEXPOV_CHK, MEXPUN_CHK, MUNKNO_CHK
DOUBLE PRECISION :: DPEPS_CHK, DPMAX_CHK, SPMAX_CHK, TEMP
INTEGER :: INTMAX_CHK, K, NPSAVE
INTENT (IN) :: NPREC