```
!     FM 1.4                                   David M. Smith                              2023-12-20



!  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 three.  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.

!  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  Exponent of the number
!      3  First digit of the number
!    ...
!  ndig+2  Last digit of the number.

!  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

!        Unpacked:     -1         1         3      1415      9265      3590
!        Packed:       -1         1     31415  92653590

!  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 log10(mbase)*(ndig-1) + 1.  In the -pi
!  example above, this is 4*3 + 1 = 13.

!  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.
!  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.

!  Besides these representable FM numbers there are three exception categories, called
!  overflow, underflow, and unknown.  They have special exponents that are outside the range of
!  exponents for representable numbers, with words 1,2,3 defining which type of category the
!  number is in.

!  FM numbers in these categories do not have full precision, but use words 4,5 to preserve some
!  information about where in each category the values lie.  This allows more robust handling of
!  exceptions.


!  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 8 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, 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 3.
!          = -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,
!                        or allocation of memory for a multiple-precision number failed.
!                        This means the program has run out of memory.
!                        The current version of FM stops the program at that point, instead of
!                        returning kflag.
!
!          = -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.

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


!  7. 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.

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


!  8.  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., low-level type(multi) as opposed to the type(fm),
!  (im), or (zm) 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 both as input and output arguments 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 may 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 from fmzm 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.
!                            A default initial seed is used if fm_random_number is called without
!                            calling fm_random_seed_put first.

!  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

!  fmerfcs(ma,mb)            mb = Erfc_Scaled(ma)  Scaled Complimentary Error function

!  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

!  fmsi(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: imfprt >

!  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)

!  zmerf(ma,mb)              mb = erf(ma)    Error function

!  zmerfc(ma,mb)             mb = erfc(ma)   Complimentary error function

!  zmerfcs(ma,mb)            mb = erfc_scaled(ma)   Scaled complimentary error function

!  zmexp(ma,mb)              mb = exp(ma)

!  zmfact(ma,mb)             mb = ma!              Factorial function

!  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)

!  zmfprint(form1,form2,ma)  Print ma on unit kw using formats form1 and form2.
!                            < old name: zmfprt >

!  zmgam(ma,mb)              mb = Gamma(ma)        Gamma function

!  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)

!  zmlngm(ma,mb)             mb = Log_Gamma(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: zmwrit >

!  zmz2m(zval,ma)            ma = zval


!  9. NEW FOR VERSION 1.3

!  The first edition of version 1.3 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) * 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 + ... + a(n)**2 )
!         This could previously have been done with array operations as sqrt(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 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.


!  10. NEW FOR VERSION 1.4

!  The changes in version 1.4 were made to enable a thread-safe special version of FM to be created.
!  See file FM_parallel.f95 for the thread-safe version.

!  The memory model for multi-precision variables has been changed from having one global database
!  kept in module fmvals that holds all the numbers to making the multi-precision variables local
!  to the routines using them.

!  The way in which the user declares and uses type(fm), etc., variables is the same in this
!  version as before.
!  Improvements from the user's point of view are:
!     a.  No longer needing to insert calls into the user's routines to fm_enter_function, etc.
!     b.  No need to call fm_deallocate before deallocating a multi-precision variable.

!  Starting with the 2023 version of FM, numbers in the overflow, underflow, and unknown categories
!  have been enhanced with some "tracking" information.  This allows better handling of exceptions.
!  For example, in the expression  (2 + 3*exp(-x*x)),  when the exp underflowed in previous versions
!  of FM, unknown would be returned for the value of the expression.  When multiplying an underflow
!  by 3 we would not know whether the true result was still in the underflow region or whether it
!  should be a representable FM number, so unknown was returned.

!  Now the underflow retains some extra information, so the 3*exp(-x*x) can be recognized as being
!  insignificant compared to 2 and the expression evaluates to 2.

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



      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
      intent (in) :: nprec

      real (kind(1.0d0)) :: maxint_chk, mxexp2_chk, mexpov_chk, mexpun_chk, munkno_chk
      double precision :: dpeps_chk, dpmax_chk, spmax_chk, temp
```

```fortran
      integer :: intmax_chk, k, npsave

!             maxint should be set to a very large integer, possibly the largest representable
!                    integer for the current machine.  For most 32-bit machines, maxint is set
!                    to  2**53 - 1 = 9.007d+15  when double precision arithmetic is used for
!                    m-variables.  Using integer m-variables usually gives
!                    maxint = 2**31 - 1 = 2147483647.

!                    Setting maxint to a smaller number is ok, but this unnecessarily restricts
!                    the permissible range of mbase and mxexp.

      maxint_chk = max_representable_m_var
      if (maxint > maxint_chk) then
          write (kw,*) ' '
          write (kw,*) ' In routine FMSET it appears that FM internal variable'
          write (kw,*) ' MAXINT was set to ', maxint, ' in file FMSAVE.f95'
          write (kw,*) ' For this machine it should be no more than ', maxint_chk
          write (kw,*) ' Change the initialization in FMSAVE.f95 to this value.'
          write (kw,*) ' For this run, MAXINT has been changed to ', maxint_chk
          write (kw,*) ' '
          maxint = maxint_chk
      else if (maxint < maxint_chk/2) then
          write (kw,*) ' '
          write (kw,*) ' In routine FMSET it appears that FM internal variable'
          write (kw,*) ' MAXINT was set to ', maxint, ' in file FMSAVE.f95'
          write (kw,*) ' For better performance set it to ', maxint_chk
          write (kw,*) ' Change the initialization in FMSAVE.f95 to this value.'
          write (kw,*) ' For this run, MAXINT has been changed to ', maxint_chk
          write (kw,*) ' '
          maxint = maxint_chk
      endif

!             intmax is a large value close to the overflow threshold for integer variables.
!                    It is usually 2**31 - 1 for machines with 32-bit integer arithmetic.

!                    The following code sets intmax_chk to the largest representable integer.
!                    Then intmax is checked against this value.

      intmax_chk = huge(i_two)
      if (intmax > intmax_chk) then
          write (kw,*) ' '
          write (kw,*) ' In routine FMSET it appears that FM internal variable'
          write (kw,*) ' INTMAX was set to ', intmax, ' in file FMSAVE.f95'
          write (kw,*) ' For this machine it should be no more than ', intmax_chk
          write (kw,*) ' Change the initialization in FMSAVE.f95 to this value.'
          write (kw,*) ' For this run, INTMAX has been changed to ', intmax_chk
          write (kw,*) ' '
          intmax = intmax_chk
      else if (intmax < intmax_chk/2) then
          write (kw,*) ' '
          write (kw,*) ' In routine FMSET it appears that FM internal variable'
          write (kw,*) ' INTMAX was set to ', intmax, ' in file FMSAVE.f95'
          write (kw,*) ' For better performance set it to ', intmax_chk
          write (kw,*) ' Change the initialization in FMSAVE.f95 to this value.'
          write (kw,*) ' For this run, INTMAX has been changed to ', intmax_chk
          write (kw,*) ' '
          intmax = intmax_chk
      endif
```