===   Contents   ===

==

==================================   Installing the FM package   ===================================

The files for version 1.4:

1.	fmsave.f95	Module for FM internal global variables
2.	fm.f95	Subroutine library for multiple-precision operations
3.	fmzm90.f95	Modules for interfaces and definitions of derived-types
4.	TestFM.f95	Test program for the FM routines

5. SampleFM.f95   Small sample program using FM

The first three files form the FM library routines that are used by an application program. They need to be compiled once, then any application program using FM can be compiled and linked to these library object files.

TestFM.f95 is a large program that does thousands of tests and calls all the library routines. It checks each operation and should be run once after compiling the FM library routines, to make sure FM is properly installed.

SampleFM.f95 is a small program with several examples showing how a typical user's program would use FM for some multiple precision real, integer, and complex calculations. It should be helpful as a model for getting started with FM.

Here are two example sets of compiler/linker commands for building the programs using the gfortran compiler (free -- click on the download link "Binaries for Windows, Linux, MacOS and much more" from this page: gcc.gnu.org/wiki/GFortran).

The first three files are compiled as object code libraries fmsave.o, fm.o, fmzm90.o, and then each program that uses FM is compiled and linked to those three libraries. Some compilers name these object files fmsave.obj, etc.

Most compilers also produce files fmvals.mod, fmzm.mod, etc., containing module information from the first three files.

1. For Windows, after installing the compiler, run in a PowerShell command prompt window:

```
    gfortran fmsave.f95  -c -O3

    gfortran fm.f95  -c -O3

    gfortran fmzm90.f95  -c -O3


    gfortran TestFM.f95  -c -O3

    gfortran  fmsave.o  fm.o  fmzm90.o  TestFM.o  -o TestFM.exe

    ./TestFM


    gfortran SampleFM.f95  -c -O3

    gfortran  fmsave.o  fm.o  fmzm90.o  SampleFM.o -o SampleFM.exe

    ./SampleFM
```

2. For a Mac, after installing the compiler, run in a Terminal window:

```
    gfortran fmsave.f95  -c -O3
    gfortran fm.f95  -c -O3

    gfortran fmzm90.f95  -c -O3


    gfortran TestFM.f95  -c -O3

    gfortran  fmsave.o  fm.o  fmzm90.o  TestFM.o  -o TestFM

    ./TestFM


    gfortran SampleFM.f95  -c -O3

    gfortran  fmsave.o  fm.o  fmzm90.o  SampleFM.o -o SampleFM

    ./SampleFM
```

```
The compiler options used in the examples were:
      -c       compile to object code -- don't make executable
      -O3      optimization level 3
      -o       output file name for the executable program
```

Most other compilers use options very similar to these. The nag compiler uses the same commands ("nagfor fmsave.f95 -c -O3"), and Lahey's compiler uses

```
                lf95 fmsave.f95  -ap -c -o1
                ...
                lf95  fmsave.obj  fm.obj  fmzm90.obj  TestFM.obj  -out TestFM.exe
```

Older versions of Lahey's compiler needed the -ap option (see below) for all files -- I have not tested the latest version.

Some compilers have options that might improve the speed of the program beyond the basic -o optimization. For example, "gfortran fm.f95 -c -O3 -funroll-loops". This runs slightly faster when precision is above 100 digits.

Some programs need 64-bit integers, and the easiest way to get them is often by using command-line compiler directives to change all integer constants and variables in a program.

Prior to June, 2015, the FM package was not compatible with 64-bit integers, since a few routines assumed that any integer value could be represented exactly when converted to double precision. But 64-bit integers can have more than 16 decimal digits, causing errors when converted to double precision. With the current version, FM can be used with either 32-bit or 64-bit integers.

With the gfortran compiler the command "gfortran prog.f95 -fdefault-integer-8 -o prog" will make 64-bit integers the default. When using 64-bit integers, all FM files and all files from the user's program should be compiled with the -fdefault-integer-8 option.

From the user's point of view, the only difference in results that come from the earlier versions and this one is that using 64-bit integers allows the range between fm's underflow and overflow thresholds to be greater.

In earlier versions of FM, exp(1.0e+8) = 1.5500e+43429448 did not overflow, while exp(1.0e+9) overflowed. Using 64-bit integers with this version, exp(1.0e+15) = 6.7244e+434294481903251 does not overflow, while exp(1.0e+16) overflows.

FM has been run using many different compilers, both free and commercial. Most are used in a similar way to gfortran, although many can also be used with development environments that can make the process of compiling, linking, and executing the files even easier.

Two other files define optional multiprecision operations for exact rational arithmetic (fm_rational.f95) and interval arithmetic (fm_interval.f95). To use these, compile and link as with the examples above. See the FM web page for testing and sample programs for each of these, analogous to TestFM.f95 and SampleFM.f95.

There are many other files at the FM website at http://dmsmith.lmu.build/fmlib.html including more sample programs using the basic FM package, and some for FM's interval or exact rational arithmetic.

There is also a unix makefile (thanks, Tran Quoc Viet) that can compile and run all the programs from the website.

1. Compile and build all the programs:

```
make
```

2. Run all the programs:

```
make run
```

3. Clean up by deleting all the object files, executables, and output files that get produced by steps 1 and 2:

```
make clean
```

```
----------------------------------------------------------------------------------------------------------
---------------------------------------   Troubleshooting   ----------------------------------------
```

Troubleshooting

After compiling and running the programs TestFM and Samplefm, each should say "no errors were found" at the end. If there were problems compiling the programs or some errors were found when they ran, read the rest of this section for possible fixes.

1. If the SampleFM program fails in example 10 when using the gfortran compiler, make sure you have the latest version of FM. There was a bug in gfortran that showed up in the August, 2021 release of FM 1.4.

    Starting with the September, 2021 release, the code in the package has been changed to work around this bug, so gfortran should be ok.

2. If your program has a function subprogram that returns an array of any of the multiple precision types, it might also fail with gfortran due to the same bug that caused the August, 2021 version of SampleFM to fail.

    To avoid the gfortran problem, make sure the function statement includes a result variable and all references inside the function are to that variable and not the function name.

An example is function harmonic_sum in the TestFM program. Instead of writing

```
    function harmonic_sum(n)
    ...
    type(fm) :: harmonic_sum
    ...
    harmonic_sum = 0
    ...
```

write something like this:

```
    function harmonic_sum(n)     result (return_value)
    ...
    type(fm) :: return_value
    ...
    return_value = 0
    ...
```

Both versions are legal Fortran, but the first one seems to trigger the gfortran bug.

3. gfortran also seems to have a problem with the reshape intrinsic function when it is applied to an array of one of the multiple precision types.

   Look at ncase = 1173 in TestFM for an example showing one way to define a matrix of multiple precision numbers without using reshape.

4. The compiler gives an "out of memory" error message or crashes during compile of one or more of the files.

   It might be necessary to break the file into smaller pieces or split it into separate files for each routine or module. This could be caused by lack of system memory, lack of virtual memory, or a bug (memory leak) in the compiler.

   Some compilers have an option (e.g., -split) to do this automatically.

   This problem is much less likely on more recent compilers and computers. These days, almost all computers have more than 1Gb of memory, and also many of the memory leaks present in old versions of the compilers and operating systems have been fixed.

5. Most of the routines compile, but a few fail with error messages like "symbol 120 is not the label of a branch target statement".
   However, looking at the code shows there is a label 120 in that routine.

   This is also very rare now. Some older compilers may require additional options to be enabled (e.g., to force 32-bit branches or addresses to be used). Check in the compiler manual and try turning on any options that mention "long branches", "32-bit addresses", etc.

6. All files compile, but the TestFM program reports a few errors when it runs. There are other possibilities, but one thing to check is whether the compiler has any options controlling arithmetic precision of intermediate results.

   Because the FM numbers are stored as integer values in double precision arrays, any sloppy rounding can cause problems. In one case, a compiler optimized an expression by leaving the result of a division in an 80-bit register and then used that result later in the calculation. Rounding the division back to double precision would have fixed the error, but using the inaccurate extended precision value caused the final result to be off by one when it was returned to an integer value.

   This compiler had an option (-ap) to force intermediate results to not be left in registers, and that fixed the problem.

7. TestFM or SampleFM gives an error message beginning something like this:

```
    Element ( 1 ) of a multiple precision one-dimensional array is undefined in an expression.
```

This could happen with older compilers that support earlier versions of Fortran. Version 1.4 of FM needs Fortran-2008 or later. Try upgrading to the latest version of your compiler, or try the latest version of the free gfortran compiler to check to see if this is causing that error message.

8. Running a program ends with an error message from FM, like

```
    ***  Error in a program using the FM package  ***

    a multiple precision number is undefined in an expression or as an input
    argument to a subprogram.
    ...
```

Usually this message is caused by an fm variable appearing in an expression in the user's program before it has been given a value.

```
====================================================================================================
===============================   User's guide for the FM package   ================================
```

User's guide for the FM package

The various lists of available multiple precision operations and routines have been collected here, along with some general advice on using the package.

See the program SampleFM.f95 for some examples of initializing and using the package.

```
----------------------------------------------------------------------------------------------------
------------------------------------   Functions available   ---------------------------------------
```

Functions available

The FM package provides 5 types of multiple precision operations:

1. type(fm) - floating-point

2. type(im) - integer

3. type(zm) - complex

4. type(fm_rational) - rational

5. type(fm_interval) - interval

The type definitions and interfaces for the first three are in file fmzm90.f95, and the other two are in files fm_rational.f95 and fm_interval.f95.

Some multiple precision functions take input arguments that are intrinsic types. In the table below, the types of arguments allowed for each function are abbreviated as:
fm, im, zm, rat, ivl, for the 5 types above, and int, sp, dp, spz, dpz, str for the intrinsic Fortran types integer, single precision real, double precision real, single precision complex, double precision complex, character string.

Further description of many of these functions can be found later in this file.

abs	fm	im	zm	rat	ivl	
acos	fm		zm		ivl	
acosh	fm		zm		ivl	
aimag			zm			
aint	fm		zm		ivl	
anint	fm		zm		ivl	
arg			zm			
asin	fm		zm		ivl	
asinh	fm		zm		ivl	
atan	fm		zm		ivl	
atanh	fm		zm		ivl	
atan2	fm				ivl	
bernoulli	fm					int
bessel_j0	fm				ivl	
bessel_j1	fm				ivl	
bessel_jn	fm				ivl	
bessel_y0	fm				ivl	
bessel_y1	fm				ivl	
bessel_yn	fm				ivl	
beta	fm				ivl	
binomial	fm	im	zm		ivl	int
btest		im				
ceiling	fm	im	zm	rat	ivl	
cmplx	fm	im				
conjg			zm			
cos	fm		zm		ivl	
cosh	fm		zm		ivl	
cos_integral	fm				ivl	
cosh_integral	fm				ivl	
dble	fm	im	zm		ivl	
digits	fm	im	zm		ivl	
dim	fm	im		rat	ivl	
dint	fm		zm		ivl	
dot_product	fm	im	zm	rat	ivl	
epsilon	fm				ivl	
erf	fm		zm		ivl	
erfc	fm		zm		ivl	
erfc_scaled	fm		zm			
exp	fm		zm		ivl	
exponent	fm				ivl	
exp_integral_ei	fm				ivl	
exp_integral_en	fm				ivl	
factorial	fm	im	zm		ivl	int
floor	fm	im	zm	rat	ivl	
fraction	fm		zm		ivl	
fresnel_c	fm				ivl	
fresnel_s	fm				ivl	
gamma	fm		zm		ivl	
gcd		im				

huge	fm	im	zm		ivl						
hypot	fm										
incomplete_beta	fm				ivl						
incomplete_gamma1	fm				ivl						
incomplete_gamma2	fm				ivl						
int	fm	im	zm	rat	ivl						
is_overflow	fm	im	zm		ivl						
is_underflow	fm	im	zm		ivl						
is_unknown	fm	im	zm	rat	ivl						
left_endpoint					ivl						
log	fm		zm		ivl						
log10	fm		zm		ivl						
log_erfc	fm				ivl						
log_gamma	fm		zm		ivl						
log_integral	fm				ivl						
matmul	fm	im	zm	rat	ivl						
max	fm	im		rat	ivl						
maxexponent	fm				ivl						
maxloc	fm	im	zm	rat							
maxval	fm	im	zm	rat	ivl						
min	fm	im		rat	ivl						
minexponent	fm				ivl						
minloc	fm	im	zm	rat							
minval	fm	im	zm	rat	ivl						
mod	fm	im		rat	ivl						
modulo	fm	im		rat	ivl						
multiply_mod		im									
nearest	fm				ivl						
nint	fm	im	zm	rat	ivl						
norm2	fm										
pochhammer	fm				ivl						
polygamma	fm				ivl						
power_mod		im									
precision	fm		zm		ivl						
product	fm	im	zm	rat	ivl						
psi	fm				ivl						
radix	fm	im	zm		ivl						
range	fm	im	zm		ivl						
rational_denominator				rat							
rational_numerator				rat							
real	fm	im	zm		ivl						
right_endpoint					ivl						
rrspacing	fm				ivl						
scale	fm		zm		ivl						
setexponent	fm				ivl						
sign	fm	im			ivl						
sin	fm		zm		ivl						
sinh	fm		zm		ivl						
sin_integral	fm				ivl						
sinh_integral	fm				ivl						
spacing	fm				ivl						
sqrt	fm		zm		ivl						
sum	fm	im	zm	rat	ivl						
tan	fm		zm		ivl						
tanh	fm		zm		ivl						
to_dp	fm	im	zm								
to_dpz	fm	im	zm								
to_fm	fm	im	zm	rat	ivl	int	sp	dp	spz	dpz	str

to_fm_interval	fm	im				int	sp	dp			str
to_fm_rational		im				int					str
to_im	fm	im	zm	rat	ivl	int	sp	dp	spz	dpz	str
to_int	fm	im	zm								
to_sp	fm	im	zm								
to_spz	fm	im	zm								
to_zm	fm	im	zm	rat	ivl	int	sp	dp	spz	dpz	str
tiny	fm	im	zm		ivl						
transpose	fm	im	zm	rat							

```
----------------------------------------------------------------------------------------------------
-------------------------   Converting a program to use the FM package   ---------------------------
```

0. If you want to write a program from scratch that uses FM, instead of converting an existing double precision version, consider writing a d.p. version first anyway. It is very useful to have a working d.p. version to compare the FM results and quickly locate large errors that might be caused by mistakes in the conversion.

1. Before any variable declarations in each program, subroutine, module, or function that will use multiple precision variables, insert

```
      use fmzm
```

   This module contains all the rules needed by the compiler for doing the multiple precision operations.

2. In all routines using multiple precision variables,
   change real or double precision declarations to type(fm)
   change complex or complex d.p. declarations to type(zm)
   if any integers need to be multiple precision, declare as type(im)

   For example, if the original main program had these declarations,

```
      real (kind(1.0d0)) :: x, y, a(50)
      complex (kind(1.0d0)) :: c, z(20)
```

   change them to this for the FM version.

```
      type(fm) :: x, y, a(50)
      type(zm) :: c, z(20)
```

3. Variables that were initialized in the declarations of the original program must be initialized separately as FM variables.

```
      double precision :: x = 1.2d0
```

   becomes

```
      type(fm), save :: x
      ...
      x = to_fm('1.2')
```

If this is in a subroutine and the value might change during one call and need to be remembered in a subsequent call, something like this can be done:

```
      type(fm), save :: x
      logical, save :: first_call = .true.
      ...
      if (first_call) then
          x = to_fm('1.2')
          first_call = .false.
      endif
```

4. At the beginning of the main program, call fm_set to set the FM precision.
   For example, to get 50 significant digits,

```
      call fm_set(50)
```

   Since increasing FM's internal precision level by one gives several extra base 10 significant digits, this call will actually set the user's precision to slightly more than 50 digits.

5. Check constants that are now part of multiple precision expressions and convert them.

```
      x = y/3     need not be converted (since integers are exact in binary), but

      x = y/3.7   should become   x = y/to_fm('3.7')
                  Since 3.7 is not represented exactly in the machine's single or double
                  precision, leaving the statement as x = y/3.7 would give x accurate only
                  the machine's single precision, even though x and y are multiple precision.
```

   Also, constants in routine argument lists that now refer to multiple precision variables must be converted.

```
      call sub(a,b,2.6d0,x)
```

   becomes

```
      c = to_fm('2.6')
      call sub(a,b,c,x)
```

6. Multiple precision variables in write statements can be handled in several ways:

   (a) If we use higher precision arithmetic for the calculations, but we only need to see the final output at double precision, the simplest option is to convert the multiple precision variables back to double for printing. Then no changes to formats are needed.

```
          write (*,"(' Step size = ',f15.6,'  tolerance = ',e15.7)"),h,t
```

   becomes

```
          write (*,"(' Step size = ',f15.6,'  tolerance = ',e15.7)"),to_dp(h),to_dp(t)
```

   now that h and t are type(fm) variables.

   If the FM automatic tracing option is on (see ntrace below), some Fortran compilers might generate a "recursive write" error message here, since another write statement would be executed during the to_dp call. A fix is to turn the tracing off before this write statement. This can also happen if an FM error message is written during the to_dp call.

(b) Format the writes for multiple precision.

```
      write (*,"(' Step size = ',f15.6,'  tolerance = ',e15.7)"),h,t
```

becomes

```
      write (*,"(' Step size = ',a,'  tolerance = ',a)"),  &
            trim(fm_format('f15.6',h)),trim(fm_format('e15.7',t))
```

fm_format is a formatting function used when the number of digits being shown is small enough to fit on one line.

Often after converting to multiple precision, we want to see more digits, so here f15.6 and e15.7 might become f35.20 and e35.25 in the FM version.

(c) Subroutine fm_form does similar formatting, but we supply a character string for the formatted result. After declaring the strings at the top of the routine, as with

```
      character(80) :: st1,st2
```

the write above could become

```
      call fm_form('f15.6',h,st1)
      call fm_form('e15.7',t,st2)
      write (*,"(' Step size = ',a,'  tolerance = ',a)") trim(st1),trim(st2)
```

fm_form must be used instead of fm_format when there are more than 200 characters in the formatted string. These longer numbers usually need to be broken into several lines.

fm_form should also be used when the FM trace option is on, since some compilers may generate an error message about a "recursive i/o reference" if a trace write executes from within another write statement via fm_format.

(d) To use the current FM default format and handle any line breaks automatically, subroutine fm_print can be used. Calling fm_set to set precision at the beginning of the program also initializes this format. For example, fm_print displays 50 significant digits after call fm_set(50). See the discussion of FM's settings for jform1, jform2, and kswide for changing the default format. The FM numbers will print on separate lines.

```
      write (*,*) ' Step size = '
      call fm_print(h)
      write (*,*) ' Tolerance = '
      call fm_print(t)
```

7. Multiple precision variables in read statements can be done with FM's free-format input:

(a) Read the line as a character string then convert using to_fm.

```
      read (*,*) a,b,c
```

becomes this (with st1 declared at the top of the routine as character with length large enough to hold each input data line).

```
      read (*,"(a)") st1
      call fmscan(st1,1,ja,jb)
      a = to_fm(st1(ja:jb))
      j1 = jb
      call fmscan(st1,j1,ja,jb)
      b = to_fm(st1(ja:jb))
      j1 = jb
      call fmscan(st1,j1,ja,jb)
      c = to_fm(st1(ja:jb))
```

Where fmscan is defined by:

```
      subroutine fmscan(string,jstart,ja,jb)
!  Scan string from position jstart and return ja as the next non-blank and
!  jb as the next blank after ja.
      character (*) :: string
      ja = 0
      jb = 0
      do j = jstart, len(string)
         if (ja == 0) then
             if (string(j:j) /= ' ') ja = j
         else
             if (string(j:j) == ' ') then
                 jb = j
                 return
             endif
         endif
      enddo
      end subroutine fmscan
```

This assumes all three numbers are on one line. If they could appear on two or three lines, more code would be needed to check for that.

It also assumes that blanks separate the numbers. If input records use commas to separate numbers, repeat counts on input items, or slashes, then either the code above can be made more elaborate to handle those cases, or the data file can be edited so the simpler code works.

(b) Formatted reads can be converted directly to calls to to_fm without scanning to find where each number appears on the line.

```
      read (*,"(f20.15,e26.16,e20.10)") a,b,c
```

becomes this

```
      read (*,"(a)") st1
      a = to_fm(st1( 1:20))
      b = to_fm(st1(21:46))
      c = to_fm(st1(47:66))
```

(c) Declare double precision variables so the original read statement still works, then convert to multiple precision.

```
      read (*,*) a,b,c
```

becomes this

```
            read (*,*) a_dp,b_dp,c_dp
            a = a_dp
            b = b_dp
            c = c_dp
```

where a_dp,b_dp,c_dp are double precision and a,b,c are multiple precision.

A possible drawback to this method is that the values are read as double precision, so after conversion to FM they are usually accurate only to double precision. As with the to_fm example in section 5 above, a number such as 3.7 is not exactly representable in binary, so if that is the value being read for a in this example, reading it as a string in (b) and then converting the string gives full multiple precision accuracy for 3.7, but reading it as in (c) gives double precision accuracy.

------------------------------------   fmzm Interface Notes   ------------------------------------

The fmzm module extends the definition of the basic Fortran arithmetic and function operations so they also apply to multiple precision numbers.

There are three multiple precision data types:

```
    fm  (multiple precision real)
    im  (multiple precision integer)
    zm  (multiple precision complex)
```

A routine using any of these types needs this statement at the top:

```
    use fmzm
```

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

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

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

-----------------------------   Initialization and internal names   --------------------------------

Initialization: The default precision for the multiple-precision numbers is about 50 significant digits.

To set precision to a different value, put this

```
      call fm_set(n)
```

in the main program before any multiple precision operations are done, with n replaced by the number of decimal digits of accuracy to be used.

Routine names: For each multiple precision operation there are several routines with related
                   names that perform variations of that operation. For example, the addition
                   operation has these forms:

                   Using the fmzm interface module to perform real (floating-point) multiple
                   precision addition, declare the variables as FM derived types with

```
      use fmzm
      type(fm) a,b,c
```

                   and then after values are assigned to a and b, doing a multiple precision
                   addition looks the same as if the variables were real or double.

```
      c = a + b
```

                   Normally, using the interface module avoids the need to know the name of the
                   FM routine being called. For some operations, usually those that are not
                   numerical Fortran functions (such as formatting a number), a direct call may be
                   needed. For the addition above there is no reason to write it as a call in the
                   user's program, but it could be done as

```
      call fm_add(a,b,c)
```

                   Routines with names starting with fm_ in the fmzm module (file fmzm90.f95) then
                   call the low-level arithmetic routines in file fm.f95. The low-level routines
                   are not usually called directly by the user's program, since they do not operate
                   on the derived type variables that the user sees, but on the internal components
                   of the types.

                   The low-level routines in fm.f95 usually have similar names to those in
                   fmzm90.f95, but with no underscore after the first two letters. In this case
                   the low-level routine is named fmadd.

                   For a few routines that don't have multiple precision arguments, like fm_set and
                   fm_setvar, the corresponding low-level names fmset and fmsetvar are also
                   available, and either form can be used.

------------------------------------   Conversion functions   --------------------------------------

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

to_im converts to type(im), and to_zm converts to type(zm).

Functions are also supplied for converting the three multiple precision types to the other numeric data types:

to_int	converts to machine precision integer
to_sp	converts to single precision
to_dp	converts to double precision
to_spz	converts to single precision complex
to_dpz	converts to double precision complex

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

-------------------------------------   Inquiry functions   ---------------------------------------

is_overflow, is_underflow, and is_unknown are logical functions for checking whether a multiple precision number is in one of the exception categories. Testing to see if a type(fm) number is in the +overflow category by directly using an if can be tricky. When x is +overflow, the statement

```
           if (x == to_fm(' +overflow ')) then
```

might return false if the comparison routine does not see that two different overflowed results would have been equal if the overflow threshold had been higher. Instead, use

```
           if (is_overflow(x)) then
```

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

------------------------   Multiple precision operations and functions   --------------------------

For each of the operations =, == , /= , < , <= , > , >= , +, -, *, /, and **, the fmzm interface module defines all mixed mode variations involving one of the three multiple precision derived types and another argument having one of these types:

```
    { integer, real, double, complex, complex double, fm, im, zm }
```

So mixed mode expressions such as

```
      x = 12
      x = x + 1
      if (abs(x) > 1.0d-23) then
```

are handled correctly.

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

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

--------------   Multiple precision versions of Fortran operations and functions   -----------------

=			
+			
-			
*			
/			
**			
==			
/=			
<			
<=			
>			
>=			
abs	real	integer	complex
acos	real		complex
acosh	real		complex
aimag			complex
aint	real		complex
anint	real		complex
asin	real		complex
asinh	real		complex
atan	real		complex
atanh	real		complex
atan2	real		
btest		integer	
ceiling	real	integer	complex
cmplx	real	integer	
conjg			complex
cos	real		complex
cosh	real		complex
dble	real	integer	complex
digits	real	integer	complex
dim	real	integer	
dint	real		complex
epsilon	real		
exp	real		complex
exponent	real		
floor	real	integer	complex
fraction	real		complex
huge	real	integer	complex
hypot	real		
int	real	integer	complex
log	real		complex
log10	real		complex
max	real	integer	

maxexponent	real		
min	real	integer	
minexponent	real		
mod	real	integer	
modulo	real	integer	
nearest	real		
nint	real	integer	complex
norm2	real		
precision	real		complex
radix	real	integer	complex
range	real	integer	complex
real	real	integer	complex
rrspacing	real		
scale	real		complex
setexponent	real		
sign	real	integer	
sin	real		complex
sinh	real		complex
spacing	real		
sqrt	real		complex
tan	real		complex
tanh	real		complex
tiny	real	integer	complex

----------------------------- Conversion and inquiry functions ---------------------------------

to_fm	real	integer	complex	string	other
to_im	real	integer	complex	string	other
to_zm	real	integer	complex	string	other
to_int	real	integer	complex		
to_sp	real	integer	complex		
to_dp	real	integer	complex		
to_spz	real	integer	complex		
to_dpz	real	integer	complex		
is_overflow	real	integer	complex		
is_underflow	real	integer	complex		
is_unknown	real	integer	complex		

----------------------------------- Formatting functions ---------------------------------------

fm_format	real		
im_format		integer	
zm_format			complex

------------------------------------ Integer functions ---

binomial		integer
factorial		integer
gcd		integer
multiply_mod		integer
power_mod		integer

```
-----------------------------------   Special functions   ---------------------------------------


   bernoulli(n)              real
   bessel_j0(x)              real
   bessel_j1(x)              real
   bessel_jn(n,x)            real
   bessel_jn(n1,n2,x)        real
   bessel_y0(x)              real
   bessel_y1(x)              real
   bessel_yn(n,x)            real
   bessel_yn(n1,n2,x)        real
   beta(a,b)                 real
   binomial(a,b)             real                complex
   cos_integral(x)           real
   cosh_integral(x)          real
   erf(x)                    real                complex
   erfc(x)                   real                complex
   erfc_scaled(x)            real                complex
   exp_integral_ei(x)        real
   exp_integral_en(n,x)      real
   factorial(x)              real                complex
   fresnel_c(x)              real
   fresnel_s(x)              real
   gamma(x)                  real                complex
   incomplete_beta(x,a,b)    real
   incomplete_gamma1(a,x)    real
   incomplete_gamma2(a,x)    real
   log_erfc(x)               real
   log_gamma(x)              real                complex
   log_integral(x)           real
   pochhammer(x,n)           real
   polygamma(n,x)            real
   psi(x)                    real
   sin_integral(x)           real
   sinh_integral(x)          real


Several of these functions are described in more detail below.


-----------------   Subroutines that do not correspond to any function above   ---------------------


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

   These are subroutines instead of functions, so they are invoked as with
       call fm_cos_sin(ma,mb,mc)

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

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

   fm_euler(ma)               ma = Euler's constant ( 0.5772156649... )
```

```
fm_equ(ma,mb,na,nb)        mb = ma    where precision is being changed.
                                      ma is defined with ndig = na digits and
                                      mb will be defined having nb digits.
                                      mb is rounded if nb < na
                                      mb is zero-padded if nb > na

fm_flag(k)                 k = kflag  get the value of the FM condition flag -- stored in
                                      the internal FM variable kflag in module fmvals.

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

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

fm_pi(ma)                  ma = pi

fm_print(ma)               Print ma on unit kw using the current default format.

fm_random_number(x)        x is returned as a double precision random number, uniformly
                           distributed on the open interval (0,1).  It is a high-quality,
                           long-period generator based on 49-digit prime numbers.
                           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.

fm_rational_power(ma,k,j,mb)
                           mb = ma**(k/j)  Rational power.
                           Faster than mb = ma**(to_fm(k)/j) for functions like the cube root.

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

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

fm_setvar(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 fm_setvar(' kswide = 120 ')
                           The variables that can be changed and the options they control are
                           listed in sections 2 through 6 of the "fm.f95 Notes" section below.
                           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 quotes in string:
                                call fm_setvar(' cmchar = e ')

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

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

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


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

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

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

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

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

   im_print(ma)           Print ma on unit kw using the current default format.

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

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


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

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

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

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

   zm_form(form1,form2,ma,string)
                          string = ma
```