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)
   ```

   Local multiple precision variables in a subprogram are those that are not input arguments, module variables, or function names. These local variables should be saved, because a compiler might create a new instance of the local variable each time the routine is called. That would not cause an error, but it would leak memory and possibly cause the program to fail later for lack of memory. So if the original code was part of a subroutine,

   ```
   SUBROUTINE SOLVE(A,Z,C)
   REAL (KIND(1.0D0)) :: X, Y, A(50)
   COMPLEX (KIND(1.0D0)) :: C, Z(20)
   ```

   then the only local variables were X and Y, so the new declaration becomes

   ```
   TYPE (FM) :: X, Y, A(50)
   TYPE (ZM) :: C, Z(20)
   SAVE :: X, Y
   ```

   or equivalently,

   ```
   TYPE (FM), SAVE :: X, Y
   TYPE (FM) :: A(50)
   TYPE (ZM) :: C, Z(20)
   ```

Using SAVE for variables in the main program is ok, but not required.

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. At the beginning of every function subprogram that returns a multiple precision value insert

```
      CALL FM_ENTER_USER_FUNCTION(your_function_name)
```

where "your_function_name" is replaced by the actual name of the function.

Before each RETURN statement and before the END FUNCTION statement if it is not immediately preceded by a RETURN, insert

```
      CALL FM_EXIT_USER_FUNCTION(your_function_name)
```

These are needed so that FM does not discard temporary variables too soon when they are created by the FMZM interface routines. The function name itself is used as a variable name to return the function value, so it is one of these temporary variables.

If a function subprogram uses multiple precision variables but returns a value that is not multiple precision, then the function name is not a temporary multiple precision object, but FM still needs to be notified not to discard temporaries within the function until that function is done. In this case, use

```
      CALL FM_ENTER_USER_ROUTINE
```

and

```
      CALL FM_EXIT_USER_ROUTINE
```

6. 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)
```

Arithmetic expressions in subroutine calls should also be removed if they will use multiple precision in the new version, since they also create temporary FM variables.

```
      CALL SUB(A,B,C+3*D,X)
```

becomes

```
      T = C+3*D
      CALL SUB(A,B,T,X)
```

If many such argument expressions occur in a program, an alternate way to convert them is by leaving the temporary object in the argument list, like

```
      CALL SUB(A,B,TO_FM('2.6'),X)
      CALL SUB(A,B,C+3*D,X)
```

and then inserting calls to FM_ENTER_USER_ROUTINE and FM_EXIT_USER_ROUTINE into subroutine SUB as described in section 5 above.

7. If a routine uses allocatable type FM, ZM, or IM arrays and allocates and deallocates with each call, then a call to FM_DEALLOCATE should be made before actually deallocating each array, to avoid leaking memory in FM's internal variable database. For example:

```
      DEALLOCATE(T)
```

becomes:

```
      CALL FM_DEALLOCATE(T)
      DEALLOCATE(T)
```

8. 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-95 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)
```

9. 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 6 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.