16

Two approaches to exception handling in Fortran

J. K. Reid
Rutherford Appleton Laboratory,
Atlas Centre, Didcot, Oxon OX11 0RA, UK
Tel: (44)1235 446493. Fax: (44)1235 446626.
Email: jkr@rl.ac.uk

Abstract

We describe two approaches to exception handling that are being considered by the ISO Fortran Committee. The enable construct provides a mechanism for specifying the scope within which extra checks are required. The collection of procedures provides a mechanism for inspecting and altering the values of the exception flags and has the useful by-product of other support for the IEEE floating-point standard. The two mechanisms are analysed and illustrated with examples.

Keywords

Fortran 90, exceptions, IEEE floating-point standard.

1 INTRODUCTION

IFIP WG 2.5 has been campaigning for more than three years for the addition of exception handling to Fortran. The lack of any such facilities means that very defensive programming is needed if robust code is required. An extreme example of this is the work of Blue (1978), who spent several months developing a code for computing the 2-norm of a vector. I will illustrate with a simpler example, due to Hull, Fairgrieve, and Tang (1994).

Straightforward code for the 'hypotenuse' function, $\sqrt{x^2+y^2}$, is

```
HYPOT = SQRT( X**2 + Y**2 )
```

and a robust variant for all cases where the result does not overflow is

R. F. Boisvert (ed.), *Quality of Numerical Software*

```
IF ( X==0.0 .OR. Y==0.0 ) THEN
  HYPOT = ABS(X) + ABS(Y)
ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
  HYPOT = MAX( ABS(X), ABS(Y) ) ! one of X and Y can be ignored
ELSE     ! scale so that ABS(X) is near 1
  SCALED_X = SCALE( X, -EXPONENT(X) )
  SCALED_Y = SCALE( Y, -EXPONENT(X) )
  SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
  HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) )
END IF
```

If the case where the result overflows is to be covered, we need the test

```
IF( EXPONENT(SCALED_RESULT) + EXPONENT(X) <= MAX_EXPONENT(X) ) THEN
```

and to set some accessible flag in the exceptional case. We obviously pay for the robustness with obscure code that executes slowly.

Most computers nowadays have hardware that conforms to the IEEE floating-point standard (IEEE 754-1985), which specifies five flags for floating-point exceptions. Other computers have hardware for the two most important, overflow and divide-by-zero. What is lacking is a mechanism to test these flags so that we can run the simple straightforward code most of the time and use the complicated code only when necessary.

There were two attempts to provide such features during the development of Fortran 90. The first included tasking and was abandoned as too ambitious. The second was relegated to a Journal of Development with the pressure to keep Fortran 90 small and with the need to devote resources to the development of the rest of the language. It involved a new construct, the enable construct.

A simplified version of the second of these was developed during 1994 at the February and May meetings of X3J3 (the ANSI Fortran committee) and the August meetings of WG5 (the ISO Fortran committee) and X3J3 in Edinburgh and was the subject of an X3J3 letter ballot. Following this ballot, X3J3 decided that to continue to work on it would put the whole schedule for Fortran 95 in jeopardy and therefore stopped.

This decision was endorsed by WG5, but at its April 1995 meeting in Tokyo it decided that handling floating-point exceptions was too important to leave until Fortran 2000. It therefore decided to establish a development body to create a 'Type 2 Technical Report'. The intention is to finalize this in 1996. It will permit vendors to implement the feature as an extension of Fortran 95, confident that it will be part of Fortran 2000, unless experience in its implementation and use demand changes. It is a kind of beta-test facility for a new language feature.

The development body was asked to consider two approaches: a drastically simplified version of the enable proposal and a set of intrinsic procedures. I agreed to act a project editor and both approaches were considered in detail during 1995 by the development body in preparation for the November meeting of WG5. It was decided then that the enable construct was the right approach in the long term, but the instrinsic procedures were more suitable for the time scale of the Technical Report process. The procedures were considered in detail at the immediately succeeding meeting of X3J3.

The plan for this paper is to summarize the current thinking of the development body over both proposals, and illustrate how they help in the construction of robust and efficient software. Sections 2 and 3 are devoted to the enable proposal and examples of its use. Sections 4 and 5 are devoted to the intrinsic procedures proposal and examples of its use. Concluding remarks on both approaches are given in Section 6.

2 THE ENABLE PROPOSAL

For dealing with exceptional events, the enable proposal involves the addition of a new construct, some new statements, and an intrinsic module.

2.1 The module

The module `CONDITIONS` contains a derived type `CONDITION`, two constants of this type called `QUIET` and `SIGNALING`, and the following intrinsic variables of this type:

1. `OVERFLOW`, `UNDERFLOW`, `DIVIDE_BY_ZERO`, `INEXACT`, `INVALID`, as defined in the IEEE standard;
2. `INSUFFICIENT_STORAGE`;
3. `INTEGER_OVERFLOW`, `INTEGER_DIVIDE_BY_ZERO`;
4. `INTRINSIC`, for an exception in an intrinsic procedure;
5. `SYSTEM_ERROR`;
6. `ALLOCATION_ERROR`, `DEALLOCATION_ERROR`;
7. `IO_ERROR`, `END_OF_FILE`, `END_OF_RECORD`;
8. `BOUND_ERROR` for array subscript errors, `SHAPE` for shape mismatches in array assignments, `MANY_ONE` for an incorrect appearance of a many-one section, `NOT_PRESENT` for a reference to an absent optional argument, `UNDEFINED` for a reference to an undefined variable.

There has been some discussion over the allocation exceptions (6.) and the I/O exceptions (7.) since the language already has a mechansim for detecting these; and over the debugging exceptions (8.), for which there is little need in a production code.

The intrinsic procedures and operators always have access to this module. Otherwise, access is controlled in the usual way by the `USE` statement.

Following the lead set by the IEEE standard, the conditions are 'sticky', which means that an intrinsic procedure or operator never sets a signaling condition to the quiet value. This can be done only by explicit Fortran code:

```
USE, INTRINSIC :: CONDITIONS
:
OVERFLOW = QUIET
```

2.2 The enable construct

The new construct has the general form

enable statement
  [*enable block*]
[*handle statement*
  handle block]
end enable statement

where the brackets indicate optional items.

An enable block has a set of conditions that are said to be 'enabled' within it. If the processor detects the occurrence of one of the associated events, the condition is set to the value `SIGNALING` and there is a transfer to the handler. There is no requirement for the transfer to be immediate, since this would seriously impair performance on computers using pipe-lining or parallelisation. Indeed, the transfer may be delayed until execution of the enable

block is complete, and more than one enabled condition may be signaling when the handler is entered. Some conditions such as INSUFFICIENT_STORAGE are likely, however, to cause an immediate transfer of control.

We illustrate with the code for the hypotenuse function discussed in Section 1:

```
! Example A
USE, INTRINSIC :: CONDITIONS
:
ENABLE (OVERFLOW, UNDERFLOW)
  HYPOT = SQRT( X**2 + Y**2 ) ! The obvious and fast way.
HANDLE
  : ! The slow way.
END ENABLE
```

Here, the code in the enable block takes no precautions and will usually execute correctly. Should it fail with overflow or underflow, the alternative algorithm is used instead.

Different compiled code may be needed in the enable block according to which conditions are enabled. For example, extra code is likely to be needed to detect integer overflow. It would not be practical to require different code in a non-intrinsic procedure that is called from an enable block. The enabling of conditions in each such procedure is therefore subject to independent control by statements within it.

2.3 Nesting of enable constructs

Nesting of enable constructs is permitted. An enable or handle block may itself contain an enable construct. Also, nesting with other constructs is permitted, subject to the usual rules for proper nesting of constructs.

By default, any condition enabled in an enable block is enabled in any enable constructs nested in it. Outside the enable constructs of a procedure, no conditions are enabled unless there is a DEFAULT_ENABLE statement, the effect of which is like an enable construct with no handler, except that it applies also to any executable code in the specification statements (executed on entry to the procedure).

2.4 Handling without enabling

A condition that is not enabled may nevertheless signal. This may happen if it is enabled in a called procedure and is not handled by that procedure. It may also happen if Fortran code assigns a signaling value to a condition. For this reason, there is an option on the handle statement to specify the handling of conditions that are not enabled. For example, we might use a library code that may cause underflow and want to handle this, while wanting to ignore underflow in our own code:

```
USE LIBRARY
   :
ENABLE
   :
   CALL LIBRARY_CODE
   :
HANDLE (UNDERFLOW)
   :
END ENABLE
```

2.5 Signaling without handling

In a procedure, it may not always be possible to handle a condition that occurs. For example, if a procedure is written for the hypotenuse function, it cannot handle the case where the true result would overflow. In such a case, a return is needed with the condition signaling, in the hope that the caller will be able to handle it, perhaps by using an alternative algorithm or perhaps by working with another kind of real. Therefore, the handle block is optional. For the same reason, the handle block is optional within a nested set of enable constructs. When an enabled condition signals, a transfer is made (not necessarily at once) to the innermost handler for the condition or a return (stop in a main program) if there is no such handler.

This feature is particularly important for defined operators. If we are using a module for extended precision, for example, we want to be able write code such as

```
ENABLE(OVERFLOW)
      :
      Z = SQRT(X*Y)
HANDLE
      :
      Z = SQRT(X)*SQRT(Y)
END ENABLE
```

when we are using extended precision, just as for normal precision.

2.6 Entering and leaving enable constructs

When an enable statement is encountered, if any signaling conditions are enabled or handled or are about to be enabled or handled, a transfer of control to the next outer handler for a signaling condition (or a return or stop) takes place. This ensures that all enabled and handled conditions are quiet on entering the enable block. Upon normal completion of the handle block, any signaling condition that it handles is reset to quiet.

If it is desired to leave a handler without resetting the handled conditions quiet (with the expectation that they will be handled by an outer handler or by the caller), this can be achieved with the statement

```
RESIGNAL
```

Similarly, there is a facility for causing a direct transfer to the handler from an enable block:

```
SIGNAL(DIVIDE_BY_ZERO)
```

Note that assignment to the signaling state does not cause an immediate transfer of control, but may cause a transfer on completion of an enable block or on encountering an enable statement. No other form of branching out of an enable construct is permitted. This limits the extent of uncertainty over which statements have been executed when a handler is entered.

The transfer to the handler may be made more precise by adding within the enable block a nested enable construct with no handler. If an enabled condition is signaling when the inner enable statement is executed, control is transferred to the handler. An example is given in Section 3.3.

Note that the enable, handle, signal, resignal, and end-enable statements provide effective barriers to code migration by an optimizing compiler.

3 ENABLE EXAMPLES

We now show some examples of code using the enable construct.

3.1 Dot product

The following module provides for dot products of real arrays of rank 1. If the sizes of the arrays do not match, the condition DOT_ERROR signals. Otherwise, the condition INTRINSIC is stored and restored so that it can be set quiet before the enable construct is entered. If it signals, the module procedure will signal the condition DOT_ERROR.

```
   MODULE DOT
! Module for dot product of two real arrays of rank 1.
      USE, INTRINSIC :: CONDITIONS
      TYPE (CONDITION) DOT_ERROR
   CONTAINS
      REAL FUNCTION MULT(A,B)
         REAL, INTENT(IN) :: A(:),B(:)
         TYPE (CONDITION) OLD_OVERFLOW
         IF (SIZE(A)/=SIZE(B)) SIGNAL(DOT_ERROR)
         OLD_OVERFLOW = OVERFLOW
         IF(OVERFLOW/=QUIET) OVERFLOW = QUIET
         ENABLE (OVERFLOW)
            MULT = DOT_PRODUCT(A, B)
         HANDLE
            DOT_ERROR = SIGNALING
         END ENABLE
         IF(OVERFLOW/=OLD_OVERFLOW) OVERFLOW = OLD_OVERFLOW
      END FUNCTION MULT
   END MODULE MATRIX
```

3.2 Matrix inversion

In the following example, the function FAST_INV may cause a condition to signal. If it does, another try is made with SLOW_INV. If this still fails, a message is printed and the program stops. Note that if either condition is signaling when the enable construct is encountered, a transfer to a handler is provoked and the enable construct is not executed.

```
   ENABLE (OVERFLOW,DIVIDE_BY_ZERO)
      MATRIX1 = FAST_INV (MATRIX) ! FAST_INV does not alter MATRIX
   HANDLE
    ! "Fast" algorithm failed; try "slow" one:
      OVERFLOW = QUIET; DIVIDE_BY_ZERO = QUIET;
      ENABLE (OVERFLOW,DIVIDE_BY_ZERO)
         MATRIX1 = SLOW_INV (MATRIX)
      HANDLE
         WRITE (*, *) 'Cannot invert matrix'
         STOP
      END ENABLE
   END ENABLE
```

3.3 Matrix inversion using in-line code

Here the code for matrix inversion is in line and the transfer is made more precise by adding extra tests.

```
ENABLE (OVERFLOW,DIVIDE_BY_ZERO)
  ENABLE
  :
  END ENABLE
  DO K = 1, N
    ENABLE
    :
    END ENABLE
  END DO
  ENABLE
  :
  END ENABLE
HANDLE
! Alternative code which knows that K-1 steps have executed normally.
:
END ENABLE
```

3.4 Hypoteneuse function

Here is code using the enable construct for the hypotenuse function discussed in Section 1. If the final result overflows, execution of the SIGNAL statement without a handler makes the condition OVERFLOW signal to the caller.

```
REAL FUNCTION HYPOT(X, Y)
   USE, INTRINSIC :: CONDITIONS
   REAL X, Y
   REAL SCALED_X, SCALED_Y, SCALED_RESULT
   TYPE (CONDITION) OLD_OVERFLOW, OLD_UNDERFLOW
   OLD_OVERFLOW = OVERFLOW; OVERFLOW = QUIET
   OLD_UNDERFLOW = UNDERFLOW; UNDERFLOW = QUIET
   ENABLE(OVERFLOW, UNDERFLOW)    ! Try the fast algorithm
      HYPOT = SQRT( X**2 + Y**2 )
   HANDLE
      OVERFLOW = QUIET
      ENABLE(OVERFLOW)
         : ! The slow code shown in Section 1
      HANDLE
         UNDERFLOW = OLD_UNDERFLOW
         SIGNAL(OVERFLOW)! Signal overflow to the caller
      END ENABLE
   END ENABLE
   UNDERFLOW = OLD_UNDERFLOW; OVERFLOW = OLD_OVERFLOW
END FUNCTION HYPOT
```

4 INTRINSIC PROCEDURES PROPOSAL

The intrinsic procedures proposal is based on the IEEE model with flags for the floating-point exceptions (invalid, overflow, divide-by-zero, underflow, inexact), a flag for the rounding mode (nearest, up, down, to zero), and flags for whether halting occurs following exceptions. It is not necessary for the hardware to have any such flags (they may be simulated by software) or for it to support all the modes. Inquiry procedures are available for determining the extent of

support. The minimum requirement is for the support of overflow and divide-by-zero.

4.1 The modules

Some hardware may be able to provide no support of these features or only partial support. It may execute faster with compiled code that does not support all the features. This proposal therefore involves three intrinsic modules. `IEEE_EXCEPTIONS` is for the exceptions and the minimum requirement is for the support of overflow and divide-by-zero for all kinds of real and complex data. `IEEE_ARITHMETIC` includes all of `IEEE_EXCEPTIONS` and provides support for other IEEE features. `IEEE_FEATURES` contains some constants that permit the user to indicate which features are essential in the application.

The modules `IEEE_EXCEPTIONS` and `IEEE_ARITHMETIC` contain the derived types:

- `IEEE_FLAG_TYPE`, for identifying a particular exception flag. Its only possible values are those of constants defined in the module:
  `IEEE_INVALID`, `IEEE_OVERFLOW`, `IEEE_DIVIDE_BY_ZERO`, `IEEE_UNDERFLOW`, and `IEEE_INEXACT`. The modules also contains the array constants
  `IEEE_USUAL = (/ IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_INVALID /)` and
  `IEEE_ALL = (/ IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT /)`.
- `IEEE_STATUS_TYPE`, for saving the current floating point status.

The module `IEEE_ARITHMETIC` contains the derived types:

- `IEEE_CLASS_TYPE`, for identifying a class of floating-point values. Its only possible values are those of constants defined in the module: `IEEE_NEGATIVE_INF`, `IEEE_NEGATIVE_NORMAL`, `IEEE_NEGATIVE_DENORMAL`, `IEEE_NEGATIVE_ZERO`, `IEEE_POSITIVE_ZERO`, `IEEE_POSITIVE_DENORMAL`, `IEEE_POSITIVE_NORMAL`, `IEEE_POSITIVE_INF`, `IEEE_SIGNALING_NAN`, and `IEEE_QUIET_NAN`.
- `IEEE_ROUND_TYPE`, for identifying a particular rounding mode. Its only possible values are those of constants defined in the module: `IEEE_NEAREST`, `IEEE_TO_ZERO`, `IEEE_UP`, `IEEE_DOWN`, and `IEEE_OTHER`.

The module `IEEE_FEATURES` contains the derived type:

- `IEEE_FEATURES_TYPE`, for expressing the need for particular IEEE features. Its only possible values are those of constants defined in the module: `IEEE_DATATYPE`, `IEEE_DENORMAL`, `IEEE_DIVIDE`, `IEEE_HALTING`, `IEEE_INEXACT_FLAG`, `IEEE_INF`, `IEEE_INVALID_FLAG`, `IEEE_NAN`, `IEEE_ROUNDING`, `IEEE_SQRT`, and `IEEE_UNDERFLOW_FLAG`.

The flags are initially clear and are set when an exception occurs. The value of a flag is determined by the elemental subroutine

```
IEEE_GET_FLAG(FLAG,FLAG_VALUE)
```

where `FLAG` is of type `IEEE_FLAG_TYPE` and `FLAG_VALUE` is of type logical. Being elemental allows an array of flag values to be obtained at once and obviates the need for a list of flags. It is not a function because such a function would not be pure (its value could vary without the argument value varying) and with a subroutine there is a clear separation between statements that can change the flag values and those that can access them.

Flags may be cleared or made to signal by the elemental subroutine

```
IEEE_SET_FLAG(FLAG,FLAG_VALUE)
```

4.2 Inquiry functions

The modules `IEEE_EXCEPTIONS` and `IEEE_ARITHMETIC` contain the following inquiry functions. If the optional argument `X` is present, it must be real and the inquiry is about reals of this kind; otherwise it is about all kinds of reals.

- `IEEE_SUPPORT_FLAG(FLAG[,X])` Inquire if the processor supports an exception.
- `IEEE_SUPPORT_HALTING(FLAG)` Inquire if the processor supports control of halting after an exception.

The module `IEEE_ARITHMETIC` contains the following inquiry functions.

- `IEEE_SUPPORT_DATATYPE([X])` Inquire if the processor supports IEEE arithmetic.
- `IEEE_SUPPORT_DENORMAL([X])` Inquire if the processor supports IEEE denormalized numbers.
- `IEEE_SUPPORT_DIVIDE([X])` Inquire if the processor supports IEEE divide.
- `IEEE_SUPPORT_INF([X])` Inquire if processor supports the IEEE infinity.
- `IEEE_SUPPORT_NAN([X])` Inquire if processor supports the IEEE Not-A-Number.
- `IEEE_SUPPORT_ROUNDING(ROUND_VALUE[,X])` Inquire if processor supports a particular rounding mode.
- `IEEE_SUPPORT_SQRT([X])` Inquire if the processor supports IEEE square root.
- `IEEE_SUPPORT_STANDARD([X])` Inquire if processor supports all the facilities.

4.3 Elemental functions

The module `IEEE_ARITHMETIC` contains the following elemental functions for reals `X` and `Y` for which `IEEE_SUPPORT_DATATYPE(X)` and `IEEE_SUPPORT_DATATYPE(Y)` are true:

- `IEEE_CLASS(X)` Returns the IEEE class (see Section 4.1 for the possible values).
- `IEEE_COPY_SIGN(X,Y)` IEEE copysign function, that is `X` with the sign of `Y`.
- `IEEE_IS_FINITE(X)` IEEE finite function. True if the value of `X` is normal or denormal.
- `IEEE_IS_NAN(X)` True if the value is IEEE Not-a-Number.
- `IEEE_IS_NEGATIVE(X)` True if the value is negative.
- `IEEE_IS_NORMAL(X)` True if the value is a normal number.
- `IEEE_LOGB(X)` IEEE logb function, that is, the unbiased exponent of `X`.
- `IEEE_NEXT_AFTER(X,Y)` Next representable neighbor of `X` in the direction toward `Y`.
- `IEEE_RINT(X)` Round to an integer value according to the current rounding mode.
- `IEEE_SCALB(X,I)` Returns `X*2**I`.
- `IEEE_UNORDERED(X,Y)` IEEE unordered function. True if `X` or `Y` is a NaN.
- `IEEE_VALUE(X,CLASS)` Generate an IEEE value.

4.4 Elemental subroutines

The modules `IEEE_EXCEPTIONS` and `IEEE_ARITHMETIC` contain the elemental subroutines:

- `IEEE_GET_FLAG(FLAG,FLAG_VALUE)` Get an exception flag.
- `IEEE_GET_HALTING_MODE(FLAG,HALTING)` Get halting mode for an exception. Halting is not necessary immediate, but normal processing does not continue.
- `IEEE_SET_FLAG(FLAG,FLAG_VALUE)` Set an exception flag.
- `IEEE_SET_HALTING_MODE(FLAG,HALTING)` Whether to halt on an exception.

4.5 Non-elemental subroutines

The modules `IEEE_EXCEPTIONS` and `IEEE_ARITHMETIC` contain the following non-elemental subroutines:

- `IEEE_GET_STATUS(STATUS_VALUE)` Get the current values of the set of flags that define the current state of the floating point environment.
- `IEEE_SET_STATUS(STATUS_VALUE)` Restore the values of the set of flags that define the current state of the floating point environment.

The module `IEEE_ARITHMETIC` contains the following non-elemental subroutines:

- `IEEE_GET_ROUNDING_MODE(ROUND_VALUE)` Get the current IEEE rounding mode.
- `IEEE_SET_ROUNDING_MODE(ROUND_VALUE)` Set the current IEEE rounding mode. If this is invoked, `IEEE_SUPPORT_ROUNDING(ROUND_VALUE,X)` must be true for any `X` such that `IEEE_SUPPORT_DATATYPE(X)` is true.

4.6 Transformational function

The modules contain the following transformational function:

- `IEEE_SELECTED_REAL_KIND([P,][R])` As for `SELECTED_REAL_KIND` but gives an IEEE kind.

5 INTRINSIC PROCEDURES EXAMPLES

We now show some examples of code using intrinsic procedures exception handling.

5.1 Dot product

The following module corresponds to the module of Section 3.1 and the coding parallels it.

```
   MODULE DOT
! Module for dot product of two real arrays of rank 1.
      USE, INTRINSIC :: IEEE_EXCEPTIONS
      LOGICAL DOT_ERROR
   CONTAINS
      REAL FUNCTION MULT(A,B)
         REAL, INTENT(IN) :: A(:),B(:)
         LOGICAL OVERFLOW,OLD_OVERFLOW
         IF (SIZE(A)/=SIZE(B)) THEN
            DOT_ERROR = .TRUE.
            RETURN
         END IF
         CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OLD_OVERFLOW)
         IF(OLD_OVERFLOW) CALL IEEE_SET_FLAG(IEEE_OVERFLOW,.FALSE.)
         MULT = DOT_PRODUCT(A,B)
         CALL IEEE_GET_FLAG(IEEE_OVERFLOW,OVERFLOW)
         IF(OVERFLOW) DOT_ERROR = .TRUE.
         IF(OVERFLOW.NEQV.OLD_OVERFLOW)    &
            CALL IEEE_SET_FLAG(IEEE_OVERFLOW,OLD_OVERFLOW)
      END FUNCTION MULT
   END MODULE DOT
```

5.2 Matrix inversion

Given the following declarations

```
TYPE(IEEE_FLAG_TYPE),PARAMETER, DIMENSION(2) :: &
         FLAGS = (/IEEE_OVERFLOW,IEEE_DIVIDE_BY_ZERO/)
LOGICAL, DIMENSION(2) :: FLAG_VALUES
```

the following code corresponds to the enable construct of Section 3.2. If either condition is already signaling, we return to the caller, which simulates the enable case without nesting. Note the use of arrays to allow the values of the two conditions to be found or altered with one elemental procedure call.

```
CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
IF(ANY(FLAG_VALUES)) RETURN
MATRIX1 = FAST_INV(MATRIX) ! This must not alter MATRIX.
CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
IF (ANY(FLAG_VALUES)) THEN
  ! "Fast" algorithm failed; try "slow" one:
  CALL IEEE_SET_FLAG(FLAGS,.FALSE.)
  MATRIX1 = SLOW_INV(MATRIX)
  CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
  IF (ANY(FLAG_VALUES)) THEN
    WRITE (*, *) 'Cannot invert matrix'
    STOP
  END IF
END IF
```

5.3 Matrix inversion using in-line code

Using the specifications of the previous section, the following corresponds to the code of Section 3.3.

```
CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
IF(ANY(FLAG_VALUES))RETURN
:
CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
IF (.NOT.ANY(FLAG_VALUES)) THEN
  DO K = 1, N
    :
    CALL IEEE_GET_FLAG(FLAGS,FLAG_VALUES)
    IF (ANY(FLAG_VALUES)) EXIT
  END DO
END IF
IF (ANY(FLAG_VALUES)) THEN
! Alternative code which knows that K-1 steps have executed normally.
:
```

5.4 Hypoteneuse function

The following corresponds to the code of Section 3.3.

```
REAL FUNCTION HYPOT(X, Y)
  USE, INTRINSIC :: IEEE_ARITHMETIC
  USE, INTRINSIC :: IEEE_FEATURES, ONLY: IEEE_UNDERFLOW_FLAG
  REAL X, Y
  REAL SCALED_X, SCALED_Y, SCALED_RESULT
  LOGICAL, DIMENSION(2) :: FLAGS, OLD_FLAGS
  TYPE(IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
```

```
         OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
! Store the old flags and set them quiet
   CALL IEEE_GET_FLAG(OUT_OF_RANGE,OLD_FLAGS)
   CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
   HYPOT = SQRT( X**2 + Y**2 ) ! Try the fast algorithm
   CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)
   IF ( ANY(FLAGS) ) THEN
      CALL IEEE_SET_FLAG(OUT_OF_RANGE,(/.FALSE.,OLD_FLAGS(2)/))
      : ! The slow code shown in Section 1
      CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAGS(1))
      IF(FLAGS(1) RETURN
   END IF
   IF(ANY(OLD_FLAGS)) CALL IEEE_SET_FLAG(OUT_OF_RANGE,OLD_FLAGS)
END FUNCTION HYPOT
```

6 CONCLUDING REMARKS

We have described the two approaches and illustrated them on four examples needing access to the IEEE flags. Both are able to cope.

The most important difference is with respect to scope. The enable construct permits the programmer to express the exact scope within which conditions are to be enabled. This allows the implementor to optimize the code according to exactly what conditions are required to be enabled. With the intrinsic procedures, such optimization is limited to two levels and the scope has to be that of a procedure.

Another important advantage for the enable approach is that it handles other exceptions. The most important is probably `INSUFFICIENT_STORAGE`, since there is currently no mechanism to recover if there is insufficient storage for an automatic array or for a temporary array needed in an array expression.

On the other hand, the possible transfers of control caused by the new statements can be a source of misunderstanding. The procedures model is far easier to understand in this respect. Also, of course, it provides needed further support for the IEEE standard.

For these reasons, the long-term plan is for both modules to be included. The decision as to which to use for the Technical Report was finely balanced, and the procedures approach chosen on the grounds that it is conceptually simpler. In particular, the following were seen as problems:

- Differing perceptions as to how to seek a handler through multiple levels of procedure calls.
- Whether it is adequate to have each exception flagged with a single global data object.
- The interaction with multi-processing.

7 REFERENCES

Blue, J. L. (1978). A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Softw.* **4**, 15-23

Hull, T. E., Fairgrieve, T. F., and Tang, T. P. T. (1994). Implementing complex elementary functions using exception handling. *ACM Trans. Math. Softw.* **20**, 215-244.

DISCUSSION

Speaker : J. Reid

S. Hammarling : Does the ENABLE proposal define the state of variables on entry to a HANDLE block?

J. Reid : When an exception causes a transfer from an enable block to a handler, any variable that might be defined or redefined by any statement in the ENABLE block (or any procedure that might be called from it) is undefined. In the example, K is defined because the transfer can only be from one of the inner ENABLE blocks represented by colons.

W.M. Gentleman : You commented several times that a certain condition should cause a compile time error. I believe that in general, the best we can hope for is a run time error. Of course if the hardware does not support something, a compile time error can be detected. However, even if the hardware is capable of supporting something, the operating system may not, and since we often use the same compiler for all operating systems on that hardware the compiler cannot know.

J. Reid : IEEE-FEATURES allows the programmer to express a definite need for a certain feature. If the lack of this is not detectable until run time, there will have to be a test at load time or at the commencement of execution. Alternatively, we could weaken the facility to a request to provide the feature if this is possible; the programmer would then need to use the inquiry functions to find out whether the feature has been provided.

I. Philips : Is WG5 going to preserve both methods for exception handling?

J. Reid : The present plan is for enable as the long-term solution. A decision will be made at the meeting in Dresden later this month on the content of Fortran 2000.

W. Walter : How do you perceive the future coexistence of the intrinsic procedure and the ENABLE block approach to exception handling in Fortran?

J. Reid : I do not think there is a conflict. The two can coexist. However, I wonder whether the Fortran community has the resources to develop ENABLE.

W.V. Snyder : Les Hatton argues that the size of a language has a negative effect on reliability, caused by the inability of practitioners to understand exactly what they write. Wouldn't it be better to keep the "size" of the language smaller by settling now on the ultimately desired mechanism, rather than on one that is orthogonal to, and can't be extended to, the ultimately desired mechanism?

J. Reid : The IEEE model can be extended to other exceptions such as insufficient storage. All that is missing is the limitation of the scope that ENABLE provides.

N. Highman : You might like to consider using the terminology "subnormal numbers" rather than "denormalized numbers" in your proposed IEEE arithmetic modules. While the latter is the term used in the IEEE standard, the former is arguably more descriptive and is now recommended by Velvel Kahan.

J. Reid : I have tried to keep to the terms used in the IEEE standard itself to minimize confusion.

J. Rice : Can you explain why exception handling is so complicated?

J. Reid : I think it is because of the profusion of differences between hardware and the different perceptions of how the systems should respond. An example is the point that Morven Gentleman raised, which we have not considered, and needs to be clarified.

R. Hanson : Exception handling seems to hold promise to create better quality. Can you respond?

J. Reid : I agree.