Exception handling in Fortran

John Reid
Rutherford Appleton Laboratory
jkr@rl.ac.uk

Presented to BCS Fortran Group, 17 October 1995

1. Introduction

It is a disgrace that we do not have facilities for exception handling in Fortran. This lack means that very defensive programming is needed if robust code is required.

Most computers nowadays have hardware that conforms to the IEEE standard (IEEE 754-1985), which specifies five flags for floating—point exceptions. Other hardwares have 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. We need something like this

```
         ENABLE (overflow)
            [enable block] ! the simple and quick code
         HANDLE
            [handle block] ! complicated and slow code for when
                           ! the simple code causes overflow
         END ENABLE
```

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.

A simplified version of the second of these was developed during 1994 at the February and May meetings of X3J3 and the August meetings of WG5 and X3J3 in Edinburgh. I represented IFIP WG 2.5 (Numerical Software), which is strongly in favour of the provision of such facilities, at all these meetings. The final version, which I will call the Edinburgh proposal, 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 soon (to be ready for formal balloting by April 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 their implementation and use demand changes. It is a kind of beta—test facility for a new language feature.

I have agreed to act a project editor and this article describes the current thinking of the development body. Comments will be welcome, particularly on the open issues listed in section 2.10. They should be sent to me, preferably by email. The Tokyo decision was for a facility for floating—point exceptions, using two alternative approaches:
(1) based on the enable construct, and
(2) based on a set of simple procedures.
The first is described in section 2 and the second in section 3.

2. The enable proposal

The development body has decided that it must develop a full enable proposal at least in outline in order to be sure that the limited proposal is consistent with it. The full proposal is based on the Edinburgh proposal, but with an important addition: as well as intrinsic conditions, conditions may be declared by Fortran code. This was a wish expressed strongly by WG5 in Edinburgh, but no way was seen at that time to satisfy this requirement without significantly enlarging the proposal. However, the suggestion of Richard Hanson to use a derived type in an intrinsic module (see section 2.5), provides this and is actually an overall simplification. Because it is a simplification, we have retained it in the subset.

This article is aimed as a description of the subset proposal, but we summarize the differences in section 2.9.

2.1. The enable construct

The current thinking of the development body is based of the Edinburgh proposal, which itself was based the work in the Fortran 90 Journal of Development. At the heart of this is the enable construct:

```
         ENABLE (list of conditions)
              [enable block] ! the simple and quick code
         HANDLE
              [handle block] ! the complicated and slow code
         END ENABLE
```

The conditions listed are "enabled" during execution of the simple code. If an enabled condition signals during the execution of the enable block, control is transferred to the handle block. The transfer to the handle block is imprecise in order to allow for optimizations such as vectorization. Unfortunately, this means that the value of any variable that is defined or redefined in a statement of the enable block cannot be relied upon and has to be regarded as undefined. However, there is a mechanism for reducing the imprecision (see section 2.6). The enabling is confined to intrinsic operations and intrinsic procedure calls and does not extend to any other procedures that are invoked; for them, it is controlled by their own enable constructs.

Nesting of enable constructs is permitted. An enable or handle block may itself contain an enable construct. By default, any condition enabled in an enable block is enabled in any enable constructs nested in it. However, the set of enabled conditions may be altered using optional syntax on the enable statement. Also, nesting with other constructs is permitted, subject to the usual rules for proper nesting of 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.

Neither a handle statement nor an end—enable statement is permitted to be a branch target. A handle—block is intended for execution only following the signaling of a condition that it handles, and an end—enable statement is not a sensible target because it would permit skipping the handling of a condition.

Branching out of an enable construct is not permitted. This limits the extent of uncertainty over which statements have been executed when a handler is entered.

2.2. 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 sqrt(x*x+y*y), 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.

2.3. 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
```

2.4. The intrinsic conditions

The proposed intrinsic conditions are those of the IEEE standard:

OVERFLOW: This condition occurs when the result for an intrinsic real or complex operation has a very large processor—dependent absolute value.

UNDERFLOW: This condition occurs when the result for an intrinsic real or complex operation has a very small processor-dependent absolute value.

DIVIDE_BY_ZERO: This condition occurs when a real or complex division has a nonzero numerator and a zero denominator.

INEXACT: This condition occurs when the result of a real or complex operation is not exact.

INVALID: This condition occurs when a real or complex operation is invalid.

A processor that does not conform to IEEE 754-1985 is not required to detect UNDERFLOW, INEXACT, or INVALID in an intrinsic operation or procedure, even if the condition is enabled.

2.5. The intrinsic module and the derived type

A significant change from the Edinburgh proposal is that conditions are of the derived type

```
   TYPE CONDITION
      SEQUENCE; PRIVATE
      INTEGER VALUE = O
   END TYPE CONDITION
```

which is defined in an intrinsic module.

The value zero is reserved for the quiet state. Specifying this for the component (a feature of Fortran 95 not present in Fortran 90) means that all conditions are initially quiet. Having an integer value, rather than a Boolean value means that extra information about signaling conditions may be encoded.

The type is a sequence type for consistency with the full proposal, which has many more intrinsic conditions and has equivalenced arrays to permit shorthands for long lists of conditions.

The module also contains the constants

```
   TYPE(CONDITION), PARAMETER :: QUIET = CONDITION(0), &
                                 DEFAULT_SIGNALING = CONDITION(-1)
```

and the intrinsic conditions

```
    TYPE(CONDITION) :: OVERFLOW, INVALID, DIVIDE_BY_ZERO, &
                       UNDERFLOW, INEXACT
```

Only positive values are set by the processor for the intrinsic conditions so that a negative value may be used by Fortran code to give a clear indication that the condition has not been given the signaling state by the processor.

The module also contains definitions for the operations of comparisons of conditions with integers and of assignments of conditions to and from integers.

Having conditions of derived type means that non—intrinsic conditions are available, which allow codes to signal a condition when something occurs that they cannot handle. Here is an example:

```
      MODULE DOT ! Module for dot product of two real arrays of rank 1.
       USE CONDITIONS
       TYPE (CONDITION) DOT_ERROR
       INTERFACE OPERATOR(.dot.)
          MODULE PROCEDURE MULT
       END INTERFACE
      CONTAINS
          REAL FUNCTION MULT(A,B)
             REAL, INTENT(IN) :: A(:),B(:)
             INTEGER I
             IF (SIZE(A)/=SIZE(B)) SIGNAL(DOT_ERROR,1)
             ENABLE (OVERFLOW)
                MULT = 0.
                DO I = 1, SIZE(A)
                  MULT = MULT + A(I)*B(I)
                END DO
             HANDLE
                SIGNAL(DOT_ERROR)
             END ENABLE
          END FUNCTION MULT
       END MODULE DOT
```

This module provides the dot product of two real arrays of rank 1. If the sizes of the arrays are different, an immediate return occurs with DOT_ERROR signaling with value 1. If OVERFLOW occurs during the actual calculation, the module procedure will signal it together with the condition DOT_ERROR with value -1.

2.6. Reducing the imprecision of the transfer to the handler

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. This reduces the imprecision to either the statements within the inner construct or those outside the inner construct. Adding such a construct to the code of Example A gives:

```
   ! Example B
   USE CONDITIONS
      :
      ENABLE (OVERFLOW)
   ! First try a fast algorithm for inverting a matrix.
      : ! Code that cannot signal overflow
      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
```

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

2.7. Signal and resignal

There is a facility for making a specified condition signal with the default value -1 or a specified value. This is done with the SIGNAL statement:

```
      SIGNAL(DIVIDE_BY_ZERO)
      SIGNAL(OVERFLOW, -3)
```

It causes a transfer to the handler if in an enable block that has a handler for the condition; otherwise, it causes a return in a subprogram or a stop in a main program. This may not be used to set conditions quiet.

In a handler, if it is desired to leave 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
```

A transfer of control to the next outer handler for a signaling handled condition (or a return or stop) occurs without the values of the conditions changing.

2.8. Termination of execution

If any conditions are signaling when the program terminates, the processor is required to print a message on the default output unit indicating which conditions are signaling. It is also required to stop with such a message if a return statement causes a signaling condition to become undefined, that is, if a signaling condition goes out of scope. Such a circumstance means that something untoward has occurred and not been handled; it would not be appropriate to destroy the evidence and continue execution.

2.9. How the full proposal differs

The full proposal differs mainly in that it offers more intrinsic conditions:

```
   ALLOCATION_ERROR, DEALLOCATION_ERROR
   INSUFFICIENT_STORAGE
   BOUND_ERROR, SHAPE, MANY_ONE, NOT_PRESENT, UNDEFINED
   IO_ERROR, END_OF_FILE, END_OF_RECORD
   INTEGER_OVERFLOW, INTEGER_DIVIDE_BY_ZERO
   INTRINSIC
   SYSTEM_ERROR
```

These are equivalenced to the rank—one arrays:
STORAGE, IO, FLOATING, INTEGER, USUAL, ALL_CONDITIONS
to allow convenient shorthands.

The other significant difference is that there is an option on the enable statement to specify that some of the enabled conditions are "immediate". Any <executable—construct> of the enable block that might signal one of the immediate conditions is treated as if it were followed by an enable construct with an empty body and no handler. An example of such an enable statement is

```
   ENABLE, IMMEDIATE (OVERFLOW)
```

2.10. Issues under discussion

The development body is considering the following possible changes:

a. Add a DEFAULT_ENABLE statement, with a syntax like the ENABLE statement, and allowed only in a scoping unit with access to the module. All the conditions named would be enabled, without a handler, outside enable blocks. It would apply in the scoping unit and any nested scoping units without their own DEFAULT_ENABLE statements.
b. Remove the SIGNAL statement. Assignment has a similar effect, but the transfer of control need not be immediate.
c. As in the Edinburgh proposal, the processor is permitted to signal a condition from a statement for which it is not enabled. We might introduce the concept of "default enabling". In a scoping unit with access to the module:
   1. Any accessible condition of the USUAL set is enabled by default outside enable constructs.
   2. In a pure procedure, all accessible conditions are enabled by default outside enable constructs.
   3. Any other accessible condition is enabled only within an enable block for which it is explicitly enabled.
   4. No intrinsic condition signals during an intrinsic operation or intrinsic procedure call in a statement of the scoping unit unless it it is enabled there.

   In a scoping unit with no access to the module, whether intrinsic conditions signal would remain processor dependent.
d. Add CHECK inside an enable block as an alias for ENABLE; END ENABLE

e. Add a statement to interface blocks to indicate that the procedure accesses a condition by use association and may signal it without handling it. This was suggested on an X3J3 ballot.
f. Remove some all of the conditions BOUND_ERROR, SHAPE, MANY_ONE, NOT_PRESENT, UNDEFINED. These are really debugging conditions and were criticized in the X3J3 ballot. It has also been suggested that we add a condition for a READ_WRITE error for an INTENT(IN) or INTENT(OUT) argument.
g. We have not finalized how pure procedures should be treated. One possibility is that in a pure procedure, any conditions that are accessible outside the procedure are treated in a special way. The first executable statement must be an enable statement for them, so there is an immediate return if any is signaling. They are treated as having been declared locally. If the condition is signaling on return and the global value is not, the local value is copied to the global value.
h. Change conditions to Boolean and allow only tests against the values DEFAULT_SIGNALING and QUIET.

Comments on any of these would be welcome.

3. The procedures approach

The procedures approach is less well developed at this time and is based on work of Keith Biermann of SUN. It is for support of the whole of the IEEE standard, not just exception handling. Obviously, a subset might be limited to exception handling (see section 3.1). The idea is to have an intrinsic module containing the types:

1. IEEE_FLAG, with the constants INVALID, UNDERFLOW, OVERFLOW, INEXACT, DIVIDE_BY_ZERO, COMMON. (COMMON combines INVALID, OVERFLOW, and DIVIDE_BY_ZERO.)

2. IEEE__RND__VALUE, with the constants NEAREST, TO_ZERO, TO_INFINITY.

3. IEEE_FSR_VALUE, for saving the current state of the floating point environment (usually the floating point status register).

There will be the following procedures:

A. Inquiry functions:

IEEE_SUPPORT_ALL() Inquire if processor supports all the IEEE facilities defined in this standard.

IEEE_SUPPORT_NAN(X) Inquire if processor supports the IEEE Not—A—Number facility for reals of the same kind type parameter as the argument.

IEEE_SUPPORT_INF(X) Inquire if processor supports the IEEE infinity facil for reals of the same kind type parameter as the argument.

IEEE_SUPPORT_FLAG(flag) Inquire if the processor supports flag.

IEEE_SUPPORT_DENORMAL(X) Inquire if the processor supports IEEE gradual underflow for reals of the same kind type parameter as the argument.

IEEE_SUPPORT_SQRT(X) Inquire if the processor supports IEEE square root for reals of the same kind type parameter as the argument.

IEEE_SUPPORT_HALTING() Inquire if the processor supports the ability to control
during program execution whether to abort or continue execution after IEEE exceptions.

IEEE_SUPPORT_ROUNDING (RND_VALUE) Inquire if processor supports a particular IEEE rounding mode. RND_VALUE is of type IEEE_RND_VALUE, with value NEAREST, TO_ZERO, or TO_INFINITY.

IEEE_SUPPORT_FSR() Inquire whether the processor supports getting and setting of the set of flags that define the current state of the floating point environment (usually in the floating point status register).

IEEE_DATATYPE (X) Inquire if the processor supports IEEE arithmetic for reals of the same kind type parameter as the argument.

B. Elemental functions:

IEEE_SQRT(X) Square root as defined in the IEEE standard.

IEEE_IS_NAN(X) Determine if value is IEEE Not—a—Number.

IEEE_IS_INF(X) Determine if value is IEEE Infinity.

IEEE_IS_VALID(X) Determine if value is neither an Infinity nor a NaN.

IEEE_IS_DENORMAL(X) Determine if value is IEEE denormalized.

IEEE_COPYSIGN(X,Y) IEEE copysign function.

IEEE_NEXTAFTER(X,Y) Returns the next representable neighbor of X in the direction toward Y.

IEEE_LOGB(X) Unbiased exponent in the IEEE floating point format.

IEEE_SCALB (X,I) Returns X * 2**I.

IEEE_INFINITY(X) Generate IEEE infinity with the smae sign as X.

IEEE_NAN(X) Generate IEEE Not-a—Number.

C. Scalar functions:

IEEE_FLAG_GET(FLAG) Get an IEEE flag (also known as exception or sticky bit), where FLAG is of type IEEE_FLAG and value INVALID, UNDERFLOW, OVERFLOW, INEXACT, or DIVIDE_BY_ZERO.

IEEE_GET_FSR() Save the current values of the set of flags that define the current state of the floating point environment (usually the floating point status register). The result is of type IEEE_FSR_VALUE.

IEEE_GET_ROUNDING_MODE() Save the current IEEE rounding mode. The result is of type IEEE_RND_VALUE.

D. Subroutines:

IEEE_FLAG_SET(flag—list) Set flags listed.

IEEE_FLAGS_CLEAR(flag—list) Clear flags listed.

IEEE_SET_FSR(FSR_VALUE) Restore the values of the set of flags that define the current state of the floating point environment (usually the floating point status register). FSR_VALUE is of type IEEE_FSR_VALUE and has been Set by a call of IEEE_GET_FSR.

IEEE_SET_ROUNDING_MODE(RND_VALUE) Set the current IEEE rounding mode. RND_VALUE is Of type IEEE_RND_VALUE, with value NEAREST, TO_ZERO, or TO_INFINITY.

IEEE_HALT(FLAG,HALTING) Controls continuation or halting on exceptions. FLAG is of type IEEE_FLAG and Value INVALID, UNDERFLOW, OVERFLOW, INEXACT, or DIVIDE_BY_ZERO.

I think the minimum requirement is for the support of OVERFLOW and DIVZERO. All the inquiry functions need to be there (but could all return false) and the procedures for getting, setting, and clearing the flags need to be there.

Something needs to be said about what happens in PURE procedures. My current thought is that if a processor spawns tasks in a FORALL:

a. the processor should store the flags before executing the FORALL and copy them to each processor;
b. at each join (when a processor is done) the stored flags should be OR’d with the processor values.

3.1 Examples

Example 1:

```
   USE IEEE_ARITHMETIC
   TYPE(IEEE_FRS_YALUE) FSR_VALUE
      FSR_VALUE = IEEE_GET_FSR()
      CALL IEEE_FLAGS_CLEAR (COMON)
    ! First try the "fast" algorithm for inverting a matrix:
      MATRIXl = FAST_INV (MATRIX)
               ! MATRIX is not altered during execution of FAST_INV.
      IF (IEEE_FLAG_GET(COMMON)) THEN
    ! "Fast" algorithm failed; try "slow" one:
      CALL IEEE_FLAGS_CLEAR (OVERFLOW, INVALID, DIVIDE_BY_ZERO)
      MATRIXl = SLOW_INV (MATRIX)
      IF (IEEE_FLAG_GET(COMMON) ) THEN
         WRITE (*, *) ’Cannot invert matrix’
         STOP
      END IF
      CALL IEEE_SET_FSR(FSR_VALUE)
   END ENABLE
```

In this 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 the use of nested enable constructs. Note, also, that it is important to set the signals to 'quiet’ before the inner enable. If this is not done, a condition will still be signaling when the inner ENABLE is encountered, which will cause an immediate transfer to an outer handler (or a stop or return).

Example 2:

```
REAL FUNCTION HYPOT(X, Y)
! In rare circumstances this may lead to the setting of the OVERFLOW flag
   USE IEEE_ARITHMETIC
   REAL X, Y
   REAL SCALED_X, SCALED_Y, SCALED_RESULT
   LOGICAL OLD_OVERFLOW, OLD_UNDERFLOW
   INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
! Store the old flags and clear them
   OLD_OVERFLOW = IEEE_FLAG_GET(OVERFLOW)
   OLD_UNDERFLOW = IEEE_FLAG_GET(UNDERFLOW)
   CALL IEEE_FLAGS_CLEAR(UNDERFLOW, OVERFLOW)
! Try a fast algorithm first
   HYPOT = SQRT( X**2 + Y**2 )
   IF (IEEE_FLAG_GET(UNDERFLOW) .OR. IEEE_FLAG_GET(OVERFLOW) ) THEN
      CALL IEEE_FLAGS_CLEAR(UNDERFLOW, OVERFLOW)
      IF ( X==0.0 .OR. Y==0.0 ) THEN
         HYPOT = ABS(X) + ABS(Y)
      ELSE IF ( 2*ABS(EXPONENT(X)—EXPONENT(Y)) > DIGITS(X)+l ) 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) ) ! may cause overflow
      END IF
   END IF
   IF(OLD_OVERFLOW) CALL IEEE_FLAG_SET(OVERFLOW)
   IF(OLD_UNDERFLOW) CALL IEEE_FLAG_SET(UNDERFLOW)
END FUNCTION HYPOT
```

An attempt is made to evaluate this function directly in the fastest possible way. (Note that with hardware support, exception checking is very efficient; without language facilities, reliable code would require programming checks that slow the computation significantly.) The fast algorithm will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation may involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if X and Y are both near the overflow limit). If the overflow or underflow flag is set on entry, it is reset on return, so that earlier exceptions are not lost.

3.2 Subset for exceptions only

A subset for exceptions only would require support of the exception flags, and the procedures IEEE_FLAG_GET, IEEE_FLAG_SET, IEEE_FLAGS_CLEAR, IEEE_GET_FSR, and IEEE_SET_FSR.