To: X3J3
From: John Reid
Subject: Enable proposal
Date: 10 August 1994

1. RATIONALE

If an operator invokes a process (in hardware or in a procedure for a defined operator) and hits a problem with which it cannot deal, such as overflow, it needs to quit and ask the caller to do something else. A simple example of this proposal is

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

If the multiply is intrinsic and an overflow occurs, a transfer of control is made to the block of code following the HANDLE statement. Similarly, if the multiply is a defined operator, it can be arranged that the OVERFLOW signals in comparable circumstances. The handle block may contain very carefully written code that is slow to execute but circumvents the problem, or may arrange for a graceful termination.

2. TECHNICAL SPECIFICATION

For dealing with exceptional events, this proposal involves the addition of integer-valued intrinsic conditions, a new construct, and some new statements. The intrinsic values are all positive, but negative values may be set by execution of a SIGNAL statement. For the definition of the conditions, see the proposed new section 15 at the end of this paper. Also, there are more examples in the proposed new sub-section 8.1.5.5.

The enable construct has the general form

```
enable statement
    enable block
[handle statement
    handle block]
end enable statement
```

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.

The enable statement lists the names of the conditions to be signaled. If any of these conditions signals during the execution of the enable block, control is transferred to the handle block. A simple example is the following:

```
! Example 1
ENABLE (OVERFLOW)
 ! First try a fast algorithm for inverting a matrix.
  :
HANDLE
 ! Fast algorithm failed; use slow one.
  :
END ENABLE
```

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

The transfer to the handle block is imprecise in order to allow for optimizations such as vectorization. Any variable that is defined or redefined in a statement of the enable block becomes undefined. In Example 1, a copy of the matrix itself would need to be available for the slow algorithm.

The transfer may be made more precise by adding within the enable block a nested enable construct without a handler. If any of the conditions is signaling when its enable statement is executed, control is transferred to the handle block. This reduces the imprecision to the statements within the inner construct or outside it. Adding such a construct to the code of Example 1 gives:

```
! Example 2
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.

If there is no handler for a signaling condition (for example, if a condition signals outside any enable construct for the condition), a transfer of control as for a return statement takes place in a procedure or as for a stop statement takes place in a main program. The condition continues to signal.

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

There is an option on the enable statement to specify that some of the conditions enabled are 'immediate'. Any 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)
```

For some conditions (mainly those that may require additional code, for example, BOUND_ERROR), the processor is required to signal the condition only within the statements of the enable block. Whether such a condition signals outside any enable block for the condition is processor dependent. There is no requirement to signal such a condition in a procedure that is called from within an enable block.

There is an option on the handle statement to specify the handling of further conditions. For example,

```
HANDLE (ALL)
```

specifies that any condition that signals during the execution of the enable block be handled, including those that the processor handles outside enable blocks.

There is a facility for making a specified condition signal with a specified value. This is done with the SIGNAL statement. An example is

SIGNAL(OVERFLOW, -3)
! Negative values of intrinsic conditions can be set this way.

It causes a transfer to the handler if in an enable block that has a handler for the condition or a return (stop in a main program) otherwise. This may also be used to set conditions quiet. For example,

SIGNAL(ALL, 0)

sets all conditions quiet. In this case, there is no transfer of control.

In a handler, if it is desired to resignal the signaling conditions, this can be achieved with the pair of statements

ENABLE
ND ENABLE

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

There is a facility for finding the value of a condition. This is done with the CONDITION_INQUIRE statement. An example is

CONDITION_INQUIRE(OVERFLOW, I)

which stores the value of the overflow condition in the variable I. Another form of the statement:

CONDITION_INQUIRE(CHAR_ARRAY)

returns the names of the conditions that are signaling in the character array variable CHAR_ARRAY.

Each condition has a default integer value. The scoping rules for intrinsic conditions are as for intrinsic procedures. A future enhancement might allow the declaration of user conditions with scoping rules similar to those for variables.

If a condition is still signaling when the program stops, the processor must issue a warning on the default output unit.

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.

3. EDITS TO THE STANDARD

6/18+. Add

IEC 559:1989, <Binary floating-point arithmetic for microprocessor systems> (also ANSI/IEEE 754-1985, IEEE standard for binary floating-point arithmetic).

.....

8/45+. Add

<<or>> <enable-construct>

.....

10/4-24. Add to R216 (in alphabetic positions) the lines

<<or>> <condition-inquire-stmt>
<<or>> <signal-stmt>

.....

12/50. After 'CASE constructs,', add 'ENABLE constructs,'.

.....

12/53+. Add:

(4) Execution of a signal statement (8.1.5.4) may change the execution sequence.
(5) Execution of an enable statement (8.1.5.1) may change the execution sequence.

.....

15/33+ Add

<<2.4.8 Condition>>

A <<condition>> is a default integer flag associated with the occurrence of an exceptional event. The value 0 corresponds to the quiet state and this is its initial value. Nonzero values correspond to signaling states. Negative values can occur only through execution of the SIGNAL statement. There is one value for all scoping units and it may be found by execution of a CONDITION_INQUIRE statement or altered by execution of the SIGNAL statement.

[Footnote: The reason for specifying that conditions have integer values is that this leaves open the possibility of providing detailed information about the condition. This will be useful when a procedure (for example, in a library) signals a condition so that it can indicate the cause of the problem. The intrinsic values are forced to be positive so that a negative value can be seen to be created by source code and not by the system.]

[Footnote: Although multitasking is not part of Fortran 90, X3J3 has considered the interaction of this proposal with multitasking extensions. Its model is that each virtual processor has a flag for each condition. Condition handling is permissible within a pure procedure. Enable, handle, and end-enable statements act as barriers and a transfer of control takes place if any processor is signaling.]

.....

22/23+ Add to the Blanks Optional column:

END ENABLE

.....

67/39. After 'terminated', add 'unless the ALLOCATION_ERROR condition is enabled'.

.....

68/40. After 'terminated', add 'unless the DEALLOCATION_ERROR condition is enabled'.

.....

80/2. After 'program', add ', except in an enable block for a suitable condition'.

.....

95/10+ Add

(4) ENABLE construct

.....

95/19. Delete 'three'.

.....

107/0+. Add

<<8.1.5 Condition handling>>

A condition has a name with the same scoping rules as for intrinsic procedures and a default integer value. The value zero corresponds to the normal or 'quiet' state and nonzero values correspond to exceptional circumstances. All conditions have initial value zero. The processor is required to signal a condition if the associated circumstance occurs during execution of an intrinsic operation or an intrinsic procedure call specified in the scope of an

enable block for the condition. Some conditions are also required to signal when the circumstance occurs outside an enable block, but whether other conditions signal outside an enable block is processor dependent. Which is which is specified in 15. When the processor signals a condition, it has a positive value. The SIGNAL statement (8.1.5.4) may be used to give it a negative value.

[Footnote: The proposal allows the in-lining of procedures with no change to the enable constructs. The effect is as if any condition enabled in the calling procedure but not in the called procedure is a condition that always signals.]

[Footnote. On many processors, it is expected that some conditions will cause no alteration to the flow of control when they signal and that they will be tested only when the enable block completes or another enable statement is encountered. Thus the overheads of testing the condition are confined precisely to the places where the programmer has requested a test. On other processors, this may be very expensive. They may instead cause a transfer of control to the handler (or a return or stop) as soon as the condition signals or soon thereafter.]

[Footnote: If additional code is needed (for example, for integer overflow), this is required only within the scope of the enable block.]

<<8.1.5.1. The enable construct>>

The ENABLE construct specifies a set of conditions, an enable block, and (optionally) a handle block with (optionally) a further set of conditions. The handle block is executed only if execution of the enable block leads to the signaling of one or more of the conditions.

R835a <enable-construct> <<is>> <enable-stmt>
<enable-block>
[<handle-stmt>
<handle-block>]
<end-enable-stmt>

R835b <enable-stmt> <<is>> [<enable-construct-name>:] #
ENABLE [(<condition-name-list>)]
[,IMMEDIATE (<condition-name-list>)]

R835c <enable-block> <<is>> <block>

R835d <handle-stmt> <<is>> HANDLE [(<condition-name-list>)] #
[<enable-construct-name>]

R835e <handle-block> <<is>> <block>

R835f <end-enable-stmt> <<is>> END ENABLE [<enable-construct-name>]

Constraint: If the <enable-stmt> of an <enable-construct> is identified by an <enable-construct-name>, the corresponding <end-enable-stmt> must specify the same <enable-construct-name>. If the <enable-stmt> of an <enable-construct> is not identified by an <enable-construct-name>, the corresponding <end-enable-stmt> must not specify an <enable-construct-name>. If the <handle-stmt> is identified by an <enable-construct-name>, the corresponding <enable-stmt> must specify the same <enable-construct-name>.

Constraint: A condition name must not appear more than once in an <enable-stmt>.

Constraint: A condition name must not appear more than once in an <handle-stmt>.

The set of conditions enabled during execution of the enable block consists of all those named on the enable statement. The set of conditions handled by the handle block consists of all those named on the enable statement or on the handle statement.

An <enable-stmt> may be a branch target statement (8.2).

[Footnote: 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.]

[Footnote: 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.]

Execution of an enable statement causes a transfer of control if any condition is signaling. If the enable statement is nested in an enable block that has a handler for a signaling condition, the transfer is to the handler of the innermost such enable block. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program. The values of the conditions are not altered.

[Footnote: Note that in a function subprogram it is very desirable to ensure that the function value is defined even if an error condtion has been diagnosed and this is expected to be handled in the calling subprogram. If the function value is not defined, further conditions will probably be signaled during the evaluation of the expression that gave rise to the function call, which may mask the condition that was the root cause.]

The value of each condition handled by the construct is set to the quiet value upon completion of execution of the <handle-block>. If a transfer of control out of the <handle-block> occurs, the conditions retain their values.

<<8.1.5.2 Enable construct>>

Execution of an <enable-construct> begins with the first executable construct of the <enable-block>, and continues to the end of the block unless a branching statement is executed or an handled condition is signaled. If a condition handled by the construct is signaling on completion of execution of the <enable-block>, control is transferred to the <handle-block>. Transfer of control to the <handle-block> may also take place sooner after the signaling of the condition. If the <enable-block> contains no nested enable constructs, any variable that might be defined or redefined by execution of a statement of the enable block is undefined. If the <enable-block> contains one or more nested enable constructs, the variables that are undefined are those that might be defined or redefined by execution of a certain subset of the statements of the enable block. The subset consists of statements that lie within an enable block but do not lie within any enable constructs nested inside it.

Branching out of an enable construct is not permitted.

[Footnote: The ban on branching out of an enable construct limits the extent of uncertainty over which statements have been executed when a handler is entered.

Any <executable-construct> of the enable block that might signal one or more of the conditions in the immediate list on the enable statement is treated as if it were followed by an <enable-construct> with an empty block and no handler.

Execution of the <handle-block> completes the execution of the <enable-construct>.

[Footnote: The transfer to the handle block is imprecise in order to allow for optimizations such as vectorization. As a consequence, some variables become undefined. In Example 3 of 8.1.5.6, a copy of the matrix itself would need to be available for the slow algorithm.]

If no condition handled by the construct is signaling on completion of execution of the <enable-block>, the execution of the entire construct is complete.

[Footnote: Nested enable constructs without handlers can be employed to reduce the imprecision of an interrupt. Note that enable, handle, and end-enable statements provide effective barriers to code migration by an optimizing compiler.]

<<8.1.5.3 Signaling conditions that are not enabled>>

A processor may signal a condition while executing a statement that is not in an enable block for the condition. If in a subprogram, a return is executed without alteration of the values of the conditions. If in a main program, a stop is executed and the processor must issue a warning on the default output unit.

[Footnote: On return to the caller, the condition will be signaling. If the invocation is within an enable block that has a handler for the condition, there will be a transfer to the handler (or a return or stop), but not necessarily until the execution of the block is complete. If the invocation is not within an enable block that has a handler for the condition, there may be a return (or stop) at once, or the processor may continue executing.]

<<8.1.5.4 Signal statement>>

R835g <signal-stmt> <<is>> SIGNAL (<condition-name>,<scalar-int-expr>)

Constraint: If the condition name is DEFAULT, IO, FLOATING, INTEGER, or ALL, the <scalar-int-expr> must be the literal constant 0.

The SIGNAL statement changes the value of the condition it names to that of the expression it contains. If the value is nonzero, it causes a transfer of control. If the statement is in an enable block that has a handler for the condition, the transfer is to the handler of the next outer outer such handler. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program.

[Footnote: In a handler, the pair of statements
ENABLE
END ENABLE
has the effect of a resignal statement. It causes a transfer of control if any condition is signaling. If the pair of statements is in an enable block that has a handler for a signaling condition, the transfer is to the next outer such handler. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program. The values of the conditions are not altered.]

<<8.1.5.5 Examples of ENABLE constructs>>

Example 1:

```
  MODULE MATRIX
! Module for matrix multiplication of real arrays of rank 2.
    INTERFACE OPERATOR(.mul.)
      MODULE PROCEDURE MULT
    END INTERFACE
  CONTAINS
    FUNCTION MULT(A,B)
      REAL, INTENT(IN) :: A(:,:),B(:,:)
```

```
    REAL MULT(SIZE(A,1),SIZE(B,2)
    ENABLE (INTRINSIC, OVERFLOW)
      MULT = MATMUL(A, B)
    HANDLE
      SIGNAL(INEXACT, 1)
    END ENABLE
  END FUNCTION MULT
END MODULE MATRIX
```

This module provides matrix multiplication for real arrays of rank 2. If there is insufficient storage for the necessary temporary array, it signals the condition INSUFFICIENT_STORAGE. If an INTRINSIC or OVERFLOW condition occurs, it signals the condition INEXACT with value 1.

Example 2:

```
IO_CHECK: ENABLE (IO_ERROR, END_OF_FILE)
        :
        READ (*, '(I5)') I
        READ (*, '(I5)', END = 90) J
        :
   90   J = 0
      HANDLE
        CONDITION_INQUIRE(END_OF_FILE,K)
        IF (K/=0) THEN
          WRITE (*, *) 'Unexpected END-OF-FILE when reading ', &
                'the real data for a finite element'
        ELSE
          CONDITION_INQUIRE(IO_ERROR,K)
          IF (K /= 0) THEN
            WRITE (*, *) 'I/O error when reading ', &
                'the real data for a finite element'
          END IF
        END IF
        STOP
      END ENABLE IO_CHECK
```

In this example, if an input/output error occurs in either of the READ statements or if an end-of-file is encountered in the first READ statement, the appropriate condition will be signaled and the handler will receive control, print a message, and terminate the program. However, if an end-of-file is encountered in the second READ statement, no condition will be signaled and control will be transferred to the statement indicated in the END= specifier.

Example 3:

```
ENABLE
 ! First try the "fast" algorithm for inverting a matrix:
  MATRIX1 = FAST_INV (MATRIX)
HANDLE (ALL)
 ! "Fast" algorithm failed; try "slow" one:
  SIGNAL (ALL, 0)
  ENABLE
    MATRIX1 = SLOW_INV (MATRIX)
  HANDLE (ALL)
    WRITE (*, *) 'Cannot invert matrix'
    STOP
  END ENABLE
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 4:

```
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
```

Here the code for matrix inversion is in line and the transfer is made more precise by adding to the enable block two enable constructs without handlers.

Example 5:

The following subroutine finds a zero of <f(x)> on an interval [<a,b>]. It is limited to take one second of real time as measured by the system clock. If it fails to obtain the requested accuracy after this time, the condition INEXACT signals with the value -1.

```
SUBROUTINE ZERO_SOLVER (A, B, X, TOLERANCE, F)
  REAL A, B, X, TOLERANCE
  INTERFACE; REAL FUNCTION F(X); REAL X; END INTERFACE

  INTEGER  COUNT, RATE, START ! Local variables
  CALL SYSTEM_CLOCK(START, RATE)
  :
! The following code is executed every iteration
  CALL SYSTEM_CLOCK(COUNT)
! If time has run out, return, signaling condition INEXACT.
  IF (COUNT > START+RATE) THEN
     SIGNAL (INEXACT,-1)
  END IF
  :
END SUBROUTINE ZERO_SOLVER
```

The application code handles the exception in a way that only it knows. An example is:

```
:
ENABLE
   CALL ZERO_SOLVER (A, B, X, TOLERANCE, F)
```

```
HANDLE (INEXACT)

! Exceeded time limit. Fix up and go on.
  :
END ENABLE
:
```

Example 6:

```
    REAL FUNCTION CABS (Z)
      COMPLEX Z
! Calculate the complex absolute value, using a scaled algorithm
!  if the straightforward calculation underflows or overflows. Set the
!  overflow condition to the value -1 if the result is too large to
!  be representable.

      REAL S, ZI, ZR
      INTRINSIC REAL, AIMAG, SQRT, ABS, MAX

      ZR = REAL(Z)
      ZI = AIMAG(Z)

quick: ENABLE(OVERFLOW, UNDERFLOW)

!       This is the quick and usual calculation.
        CABS = SQRT(ZR**2 + ZI**2)

      HANDLE quick

!       Will try again using a scaled equivalent method.
        S = MAX(ABS(ZR),ABS(ZI))
        SIGNAL (OVERFLOW,0) ; SIGNAL (UNDERFLOW,0)
 slow: ENABLE(OVERFLOW, UNDERFLOW)
         CABS = S*SQRT( (ZR/S)**2 + (ZI/S)**2 )
       HANDLE slow
         CONDITION_INQUIRE(OVERFLOW,K)
         IF (K/= 0) THEN
!          The result is too large to be representable.
           SIGNAL(OVERFLOW, -1)
         ELSE
           CONDITION_INQUIRE(UNDERFLOW,K)
           IF (K/= 0)  CABS = S
         END IF
       END ENABLE slow

     END ENABLE quick

   END FUNCTION CABS
```

This illustrates the setting of a special condition value when the problem really has a result that overflows.

Example 7:

```
   MODULE LIBRARY
   ...
   CONTAINS
```

```
  SUBROUTINE B
    ...
    X = Y*Z(I) ! No condition enabled.
    IF(X>10.)SIGNAL(OVERFLOW, 1)
    ...
  END SUBROUTINE B
END MODULE LIBRARY

SUBROUTINE A
  USE LIBRARY
  ENABLE
    CALL B
  HANDLE (OVERFLOW)
    ...
  END ENABLE
END SUBROUTINE A
```

This illustrates the use of a library module that may signal the condition OVERFLOW. The signal statement causes a transfer to the handler in the calling subroutine A.

This also illustrates the effect of an intrinsic condition that is not enabled. An overflow in Y*Z(I) would cause OVERFLOW to signal and hence a transfer to the handler in the calling subroutine A. An out-of-range subscript value I might or might not signal BOUND_ERROR, but it would not be handled by subroutine A.

Example 8:

```
ENABLE, IMMEDIATE (OVERFLOW)
    A = B*C
    WHERE(RAINING)
       X(:) = X(:)*A
    ELSEWHERE
       Y(:) = Y(:)*A
    END WHERE
HANDLE
  .....
END ENABLE
```

This illustrates the use of IMMEDIATE. The construct is equivalent to

```
ENABLE (OVERFLOW)
   A = B*C
   ENNABLE
   END ENABLE
   WHERE(RAINING)
      X(:) = X(:)*A
   ELSEWHERE
      Y(:) = Y(:)*A
   END WHERE
   ENABLE
   END ENABLE
HANDLE
  .....
END ENABLE
```

Note that the statements of a WHERE construct are not tested separately.

.....

107/5. After '<end-do-stmt,>' add 'an <enable-stmt>,'.
.....

122/17-18. Replace sentence by
If an error condition (9.4.3) occurs during execution of an input/output statement that lies in an enable block for the IO_ERROR condition or contains an ERR= specifier:
.....

122/25. After 'continues with' add 'the handle block or'
.....

122/27-28. Replace sentence by
If an end-of-file condition (9.4.3) occurs and no error condition (9.4.3) occurs during execution of an input/output statement that lies in an enable block for the END_OF_FILE condition or contains an END= specifier:.....

122/34. After 'continues with' add 'the handle block or'
.....

122/37-38. Replace sentence by

If an end-of-record condition (9.4.3) occurs and no error condition (9.4.3) occurs during condition of an input/output statement that lies in an enable block for the END_OF_FILE condition or contains an EOR= specifier:
.....

123/6. After 'continues with' add 'the handle block or'
.....

125/10. Before 'contains' add 'is not in a enable block for the IO_ERROR condition and '.
.....

125/11. Before 'contains' add 'is not in a enable block for the END_OF_FILE condition and '.
.....

125/13. Before 'contains' add 'is not in a enable block for the END_OF_RECORD condition and '.
.....

241/25. After 'procedures,' add 'intrinsic conditions,'.
.....

241/35. After 'procedure,' add 'or condition'.
.....

<<15. CONDITIONS>>

In this section, the conditions supported by the standard and a statement for obtaining the value of a condition are specified.

The CONDITION_INQUIRE statement returns the value of a condition.

R835i <condition-inquire-stmt> <<is>> CONDITION_INQUIRE (<condition-name>, #
[STAT=]<scalar-default-int-variable>)
<<or>> CONDITION_INQUIRE (<conditions-array>)

R835j <conditions-array> <<is>> <default-char-variable>

Constraint: The condition name must not be DEFAULT, IO, FLOATING, INTEGER, or ALL.

Constraint: The <conditions-array> must be an array.

The STAT= variable is defined with the value 0 if the condition named is quiet and a nonzero value otherwise. Negative values can occur only following execution of a SIGNAL statement.

The <conditions-array> is defined with the names of signaling conditions and blanks according to the rules of default assignment. If there are <s> conditions signaling, the first <s> elements are defined with the names of these conditions and the remaining elements are given the value blank. If there are more signaling conditions than the length of the array, all elements are defined with condition names and which are chosen is processor dependent.

<<15.1 Storage and addressing conditions>>

ALLOCATION_ERROR
This occurs when the processor is unable to perform an allocation requested by an ALLOCATE statement (6.3.1) containing no STAT= specifier. It is not signaled by an ALLOCATE statement containing a STAT= specifier. The signaling values are the same as the STAT values. It signals outside enable blocks.

DEALLOCATION_ERROR
This occurs when the processor detects an error when executing a DEALLOCATE statement (6.3.1) containing no STAT= specifier. It is not signaled when executing a DEALLOCATE statement containing a STAT= specifier. The signaling values are the same as the STAT values. It signals outside enable blocks.

INSUFFICIENT_STORAGE
This indicates that the processor is unable to find sufficient storage to continue execution. It may occur prior to the execution of the first executable statement of a main program or procedure and it may occur during the execution of an executable statement. It need not signal if ALLOCATION_ERROR signals. It signals outside enable blocks.

BOUND_ERROR
This occurs when an array subscript, array section subscript, or substring range violates its bounds. This does not include violations of the requirements derived from the size of an assumed-size array. Whether it signals outside enable blocks is processor dependent.

SHAPE
This occurs when an array operation or assignment does not conform in shape. Whether it signals outside enable blocks is processor dependent.

MANY_ONE
This occurs when a many-one array section (6.2.2.3.2) appears on the left of the equals in an assignment statement or as an input item in a READ statement. Whether it signals outside enable blocks is processor dependent.

NOT_PRESENT
This occurs when a dummy argument that is not present is accessed as if it were present; that is, when one of the restrictions of 12.5.2.8 is violated. Whether it signals outside enable blocks is processor dependent.

UNDEFINED
This occurs when a value that is required for an operation is detected by the processor to be undefined. Whether it signals outside enable blocks is processor dependent.

[Footnote: This wording is intended to allow the processor to be as thorough as it chooses with respect to the detection of undefined values.]

<<15.2 Input/output conditions>>

IO_ERROR
This occurs when an input/output error (9.4.3) is encountered in an input/output statement containing no IOSTAT= or ERR= specifier. It is not signaled when executing an input/output statement containing an IOSTAT= or ERR= specifier. The signaling values are the same as the IOSTAT values. It signals outside enable blocks.

END_OF_FILE
This occurs when an end-of-file condition (9.4.3) is encountered in an input statement containing no IOSTAT= or END= specifier. It is not signaled when executing an input statement containing an IOSTAT= or ERR= specifier. It signals outside enable blocks.

END_OF_RECORD
This occurs when an end-of-record condition (9.4.3) is encountered in an input statement containing no IOSTAT= or EOR= specifier. It is not signaled when executing an input statement containing an IOSTAT= or ERR= specifier. It signals outside enable blocks.

<<15.3 Floating-point conditions>>

OVERFLOW
This condition occurs when the result for an intrinsic real or complex operation has a very large processor-dependent absolute value. Whether it signals outside enable blocks is processor dependent. Enabling INTRINSIC may cause this condition to signal.

UNDERFLOW
This condition occurs when the result for an intrinsic real or complex operation has a very small processor-dependent absolute value. A processor that does not conform to IEC 559:1989 is required to set this condition when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

DIVIDE_BY_ZERO
This condition occurs when a real or complex division has a nonzero numerator and a zero denominator. Whether it signals outside enable blocks is processor dependent.

INEXACT
This condition occurs when the result of a real or complex operation is not exact. A processor that does not conform to IEC 559:1989 is required to set this condition when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

INVALID
This condition occurs when a real or complex operation is invalid. A processor that does not conform to IEC 559:1989 is required to set this condition for real or complex division of zero by zero and when requested to do so by a SIGNAL statement, but is not required to set it otherwise. Whether it signals outside enable blocks is processor dependent.

<<15.4 Integer conditions>>

INTEGER_OVERFLOW
This condition occurs when the result for an intrinsic integer operation has a very large processor-dependent absolute value. Whether it signals outside enable blocks is processor dependent.

INTEGER_DIVIDE_BY_ZERO
This condition occurs when an integer division has a zero denominator. Whether it signals outside enable blocks is processor dependent.

<<15.5 Intrinsic procedure condition>>

INTRINSIC
This condition indicates that an intrinsic procedure or operation has been unsuccessful. An unsuccessful intrinsic procedure may signal other conditions instead of INTRINSIC. Whether it signals outside enable blocks is processor dependent. If an intrinsic procedure is an actual argument in a procedure call within an enable block for the INTRINSIC condition, the condition must signal if the procedure is invoked through the argument association.

<<15.6 System error conditions>>

SYSTEM_ERROR
This condition occurs as a result of a system error. Whether it signals outside enable blocks is processor dependent.

<<15.7 Combination conditions>>

Each of the following conditions may be specified on an enable, handle, signal statement and is equivalent to specifying a list of conditions.

DEFAULT
This condition is equivalent to the list: ALLOCATION_ERROR, DEALLOCATION_ERROR, INSUFFICIENT_STORAGE, IO_ERROR, END_OF_FILE, END_OF_RECORD, OVERFLOW, DIVIDE_BY_ZERO, INTEGER_DIVIDE_BY_ZERO, and INTRINSIC.

IO
This condition is equivalent to listing all the input/output conditions.

FLOATING
This condition is equivalent to listing all the floating-point conditions.

INTEGER
This condition is equivalent to listing the two integer conditions.

ALL
This condition is equivalent to listing all the conditions.