ISO/IEC JTC1/SC22/WG5 N1257

To: WG5
From: John Reid and Christian Weber
Subject: Enable revised
Date: 3 Febuary 1997

1. INTRODUCTION

Exception handling beyond that provided by the IEEE TR has come out with a high vote in the MISC ballot, see N1240. To help WG5 decide on this issue in Las Vegas, we collect here:

1. The requirement, as summarized in item 5b of the repository N1189.

2. The mechanism, as summarized by Christian Weber in his paper "Miscellaneous Requirements for Fortran2000", dated 28th Dec. 1996, which the subgroup used as the basis for its voting.

3. The technical specification of ENABLE (Section 2 of the 24 Oct. 1995 draft Technical Report).

4. The edits for ENABLE (Section 3 of the 24 Oct. 1995 draft Technical Report).

5. Christian Weber's ideas on modifications of ENABLE.

We suggest that the draft Technical Report be used as a starting point. The main difference in philosophy will be that it will be designed to handle situations where halting would otherwise occur, since IEEE_ARITHMETIC supports continued execution with the IEEE exceptions signaling. However, in order to allow for optimizations, we would have to retain the indeterminacy of the point at which the transfer to the handler is made. Section 5 contains the ideas of one of us in more detail.

2. THE REQUIREMENT (from N1189)

Number: 5b

Title: Condition Handling

Submitted By: US

Status: Being developed for 2000 Revision

References: Section F.4 of X3J3/S8.104 (Appendix F)
N900

Basic Functionality: Provide a structured way of dealing with relatively rare, synchronous events, such as error in input data or instability of an algorithm near a critical point.

Rationale: A structured approach to handling exceptional conditions would improve both the maintainability and robustness of Fortran programs.

Much of the error handling code which clutters up the expression of the underlying algorithm in current programs could instead be moved to a separate handler. Such handlers would, in turn, encourage programmers to write, and provide facilities to support, more complete code for dealing with exceptional conditions. A block-structured approach would preserve, for both the programmer and the compiler, a clear connection between the code expressing an algorithm and the associated exception handling code.

Estimated Impact:

Detailed Specification: The condition handling mechanism must have the following characteristics:

1. Automatic detection and signaling (when enabled) of a standard set of intrinsic conditions, including at least: numeric errors, subscript and substring bounds errors, I/O errors, end of file, and allocation errors. Additional, processor-dependent intrinsic conditions should also be allowed.

2. User-defined conditions, declared with a new specification statement and signaled explicitly by a new executable statement.

3. Dummy condition arguments.

4. Independent enabling and/or disabling of intrinsic conditions, on a granularity finer than that of a scoping unit; that is, it should be possible to enable an intrinsic condition only for a block of statements, possibly even a single statement, in a scoping unit.

5. Minimal impact on code performance within the innermost condition-handling block or scoping unit. In other words, entities in the innermost block or scoping unit should be permitted to become undefined when a condition is signaled when necessary to avoid extra processor overhead.

6. When a condition is signaled, a search should be made up the chain of nested, enabled condition-handling blocks and scoping units for an associated user-written handler, and control should be transferred to the first such handler found. If no handler is found, the result is processor dependent. A handler should be able to resignal the condition that activated it.

7. Default handlers to handle any condition.

8. Appropriate inquiry functions to determine whether a condition has been enabled and whether a handler has been established for it.

History: Request for investigation via B9/C5
Development Body established, July 1995, to produce a Technical Report in 1996 on handling floating point exceptions.

3. THE MECHANISM (Christian Weber, 28th Dec. 1996)

5.1 Exception Handling (5,5a,5b,5c)

Subject:
The requirement deals with enhancements in two directions:

1. Establish some high-level control flow constructs (such as ENABLE... / SIGNAL) to define - for a certain code region - a central (user-defined) exception handling mechanism which is invoked by the processor whenever an error (of a certain kind) occurs

   * within the designated piece of code, or
   * within a subroutine which is invoked (either directly or indirectly at any calling depth) from within the designated piece of code.

   The control flow construct should allow

   * block-structured nesting of exception handling definitions, and
   * the signaling of user-detected exceptions at any calling depth which will cause a jump ("longjump") to the nearest appropriate exception handling block further up the calling chain.

2. Establish some mechanism to control the behaviour of the processor in certain error situations where now (normally in a processor-dependent fashion)

   * the program is terminated with some error message, or
   * the error might be ignored, resulting in wrong program results.

   Examples for such error situations are:

   * IEEE floating point exceptions if the algorithm shall be halted in case of error (note that the IEEE TR covers *only* the case of continuation with not-a-number results after exceptions),
   * integer arithmetic exceptions,
   * lack of memory,
   * I/O errors if no IOSTAT... parameter has been specified,
   * access of array elements beyond the index bounds,
   * access of dummy arguments which are not present,

   etc.

Example how the requirement might be satisfied:
The last version of John Reid's ENABLE proposal would essentially satisfy all the requirements.
It introduced for the definition of "exception areas" a syntax like:

```
ENABLE (condition)
  ....
  Fortran code to be controlled
  ....
HANDLE
  ....
  Fortran code to deal with the exception
  ....
END ENABLE
```

For signalling a user-detected exception there was some new

```
SIGNAL (condition)
```

statement.

Some reduced syntax (as Jerry Wagener has proposed) might be - as a first step - sufficient as well.

It should be possible to discuss separately the general high-level construct (ENABLE...) and the set of error situations which have to be detected by every Fortran processor.

W. Clodius has collected a list of aspects (which I think is quite comprehensive) which might have to be considered during the design of some exception handling facility: see attachment B.

Rationale:
The reasons for exception handling (in general) are described in the repository. It has been argued that the exception handling needs are now sufficiently satisfied by the IEEE TR. This is, however, only partially true:

- The IEEE deals with IEEE floating point exceptions only, and even with these only if the algorithm may be continued (with not-a-number result) in case of error.
  All the other exceptions (see above) cannot be handled so far.

- There is no high-level construct yet to deal with exceptions at a central point, especially if the exceptions occur further down the calling chain. Therefore, the handling of exceptions which may occur in subroutines must currently be programmed as follows:
  * the subroutine reports any exception by some return parameter value,
  * at each invocation of the subroutine the caller must inquire this return value, normally just to stop the further processing in case of error and to hand back a (different) return value to its own caller.

  This handling is very error prone and causes the production of a lot of lines of code which are unnecessary in other programming languages (such as C++ which *does* offer an appropriate exception handling feature).

Problems:
1. The routines along the call chain between the exception handling block and the "signalling" point must always be prepared for a termination by "longjump". This preparation may involve some additional calling overhead even if no exception ever occurs.

   Possible solutions:
   * accept the overhead (which is not all that large after all),
   * demand from the user that he marks all subroutines which should prepare to for a longjump termination with some new syntax construct (e.g. USE EXCEPTION_SUPPORT).

2. The detection of certain exceptions may be difficult to implement (e.g. INTEGER_OVERFLOW may not be supported by all hardware architectures).

3. The impact of exception handling on optimization must be carefully observed (I think, though, that John Reid's proposal has solved this problem adequately).

X3J3 status: Vote Yes=6, No=8

Amount of work for X3J3:
large, although a good part (most?) of the necessary work has already been done by John Reid's ENABLE proposal.

==========
Attachment B:

==========

Aspects of Exception Handling Constructs
(by W. Clodius)

1. Should the handler be a statement, a la CLU (=> the actual handling is done by an internal procedure), or a block, a la Ada, or both (similar to the alternatives of using an IF statement or an IF construct)?

NOTE: A statement handler is less distracting in terms of understanding the normal flow of control, a block construct can make it easier to understand the handling of the exceptional cases. Internal procedures make the handling of errors using a statement handler relatively easy.

2. Should there be a distinction between conditions and exceptions, where the optimizer is allowed to assume for exceptions, (but not for conditions) that their occurrence is sufficiently rare that it can optimize the unexceptional path to almost any detriment to the exceptional path?

3. Should there be a distinction between errors and warnings, where for errors execution of normal code should stop and control of flow should propagate instantly to the first available handler at the same level or up the call path at which time the handler is executed (typical exception handling behavior), while for warnings the behavior should be that normal execution continues until an appropriate handler at the same level or up the call path is encountered at which time the handler is executed?

Note: The behavior of warnings are similar to Fortran's floating point exceptions, or a typical set an integer flag and test.

4. Should the default upon entering the handler be to set the condition to quiet or to remain signaling?

Note: Leaving the condition active is probably the safest, setting it inactive probably reduces the amount of coding.

5. Should the default upon leaving the handler be to set the condition to quiet or to remain signaling?

Note: Leaving the condition active is probably the safest, setting it inactive probably reduces the amount of coding.

6. Can the condition carry with it additional data?

Note: Virtually all recent exception handling systems allow the condition to carry additional information that might prove useful in understanding the exception source. Ada lets it carry a string. Modula 3, C++, Java, CLOS, Dylan, let it carry almost arbitrary information.

7. What are the allowed propagation paths after the exception is cleared and handled?

a. To the code immediately after the handler

b. To the code immediately after the point where the exception was generated. Most people I think would rightly vote no, although it may be a useful capability during code debugging.

c. To the code immediately after the statement at the level of the handler which served as the ultimate source of the condition. Most people I think

would rightly vote no.

d. To the start of the code within which the condition was generated? (Eiffel RETRY semantics) I would vote yes.

8. Should the language include a condition that cannot be turned off? (Abort)

9. What sort of error propagation should be allowed for functions? Can they only propagate warnings or Abort?

Note: Standard exceptions are incompatible with Fortran's semantics for Functions.

10. Should a function always generate a "valid" result even if an exception is generated?

11. Should the code be required to document what conditions can propagate out of the scope of a programming unit? If so should it be documented at the subprogram or the module level? Does the propagation of an unlisted error result in the generation of Abort?

Note: most writers of code hate the documentation, most users appreciate it.

12. Should the exception handling system be integrated with an assertion (a la C) or requirement (a la Eiffel) construct?

Note: That the authors of Numerical Recipes in Fortran 90 found the creation of an assert subroutine to be useful and recommend it as an addition to the language.

13. The data initialization statements in specification part of Fortran program units, can result in stack over flows, various numerical error, etc., do we want the exception handling system to deal with those?

14. If constructors or destructors are added to the language and can generate exceptions, how do we deal with partially constructed objects?

Note: This has been a major problem with C++'s exception handling system.

15. If parallel execution is allowed, how do we deal with multiple simultaneously generated exceptions?

4. TECHNICAL SPECIFICATION PART OF THE ENABLE DRAFT TR

(24 Oct. 1995, Section 2)

For dealing with exceptional events, this proposal involves the addition of a new construct, some new statements, and an intrinsic module containing a derived type for conditions, some operations for this derived type, and a set of intrinsic conditions.

The new 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.

An enable block is has a set of conditions that are said to be 'enabled' within it. A processor normally required to detect the occurrence of the associated events, which may mean that different compiled code is needed or that a mode flag has to be set. If such an event is detected, the condition is set to a nonzero (signaling) value and there is a transfer to the handler, but there is no requirement that this be done immediately. Indeed, on a processor that conforms to IEEE 754-1985, the expectation is that execution will continue after a floating-point exception, but with a flag raised and that the flag will be tested when execution of the enable block is complete. Some conditions, such as INSUFFICIENT_STORAGE are likely to cause an immediate transfer of control.

Similarly, a handle block has a set of conditions that are said to be 'handled' within it. Normally, the set of conditions that are enabled in the enable block are handled in the handle block, but there is syntax to allow additional conditions to be handled. When an enable construct is nested in a handle block, a handled condition may be enabled once again. For any particular statement, a condition is either enabled or handled or neither. If a condition is enabled in an enable block and handled on its enable statement (because of nesting), a copy of the condition is made and is restored when the construct finishes execution unless the condition signals in the construct and is not handled there.

A processor that does not conform to IEEE 754-1985 is not required to detect UNDERFLOW, INEXACT, or INVALID.

The derived type is

```
   TYPE CONDITION
      SEQUENCE
      PRIVATE
      INTEGER VALUE = 0 ! All conditions are initially quiet.
   END TYPE CONDITION
```

The quiet value is zero and all nonzero values indicate signaling. All conditions are initialized by default to the value zero. The signaling values set by the processor for intrinsic conditions are all positive, but negative values may be set by Fortran code. For the definition of the module and the intrinsic 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 type is a sequence type to allow intrinsic conditions to have equivalenced arrays, thereby permitting shorthands for long lists of intrinsic conditions.

If a non-intrinsic procedure is invoked from a statement for which a condition is enabled, this has no effect on the enabling of the condition in the procedure invoked.

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. Outside all enable constructs, no conditions are enabled unless there is a DEFAULT_ENABLE stat

If an enabled condition signals during the execution of the enable

block, control is transferred to the handle block. A simple example is the following:

```
! Example A
USE CONDITIONS
   :
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 A, a copy of the matrix itself would need to be available for the slow algorithm.

If there is no handler for an enabled condition that is signaling, 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. If the condition becomes undefined (14.7.6) (that is, if it runs out of scope), the processor issues a message on the unit identified by * in a WRITE statement and stops, unless the condition is UNDERFLOW or INEXACT. The message must indicate which conditions are signaling and their values.

A condition that is not enabled may nevertheless signal. This may happen if it is intrinsic or if it is enabled in a called procedure and is not handled by that procedure. For this reason, there is an option on the handle statement to specify the handling of conditions that are not enabled. For example,

```
    HANDLE (ALL_CONDITIONS)
```

specifies that any intrinsic condition that signals during the execution of the enable block be handled.

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.

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.

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)
```

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

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.

Assignment from a value of type default integer may be used to alter the value of a condition. For example,

```
   ALL_CONDITIONS = 0
```

sets all intrinsic conditions quiet. Assignment to a 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. The module contains the constants

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

to allow statements such as

```
   DIVIDE_BY_ZERO = QUIET;  OVERFLOW = DEFAULT_SIGNALING
```

It may be useful to set values other than -1 to give more information about the circumstance.

Assignment may also be used to place the value of a condition in a variable of type default integer:

```
   I = OVERFLOW
```

and conditions may be tested for equality with another condition or a value of type default integer:

```
   IF (OVERFLOW==QUIET .OR. DIVIDE_BY_ZERO/=-1) THEN
```

Conditions are like other derived types except in these respects:

1. An intrinsic operation or procedure that is invoked from a statement for which a set of intrinsic conditions is enabled

behaves as if those conditions were accessed in the intrinsic's code by a use statement for those conditions. This may also happen for intrinsic conditions that are not enabled.

2. There is a special relationship between enabled or handled conditions and the control flow.
3. In a pure procedure, any conditions that are accessible outside the procedure are treated in a special way. They are enabled by default outside enable constructs, 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.
   [Note: The intention is to permit the concurrent execution of the procedure on several independent processors, each with its own separate condition handling hardware. If one or more of them signals a condition and fails to handle it, the caller is told about one such occurrence.]

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.

If a condition is signaling when the program stops, the processor must issue a warning on the unit identified by * in a WRITE statement. The message must indicate which conditions are signaling and their values.

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.

5. EDITS PART OF THE ENABLE DRAFT TR (24 Oct. 1995, Section 3)

4/30. Add

An executing program is permitted to violate a prohibition or restriction where this corresponds to an intrinsic condition (2.4.8) that has been enabled (8.1.5.1). A prohibition or restriction expressed as a constraint to the syntax rules is never relaxed in this way. The intrinsic conditions are specified in 15.1.

.....

10/47+. Add

<<or>> <enable-construct>

.....

11/2-30. Add to R216 (in alphabetic positions) the lines
<<or>> <resignal-stmt>
<<or>> <signal-stmt>
.....

14/35. After 'DO constructs,', add 'ENABLE constructs,'.
.....

14/38+. Add:
(4) Execution of a signal or resignal 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 scalar variable of derived type CONDITION (15) and is associated with the occurrence of an exceptional event. The value of a condition is the value of its only component, which is of type default integer. The value 0 corresponds to the <<quiet>> state and this is its initial value. Nonzero values correspond to the <<signaling>> state. Values set by the processor to indicate signaling of an intrinsic condition (15.1) are positive and otherwise processor dependent.

[Note: The reason for specifying that conditions have integer values is that this leaves open the possibility of providing detailed information about the condition. 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.]

[Note: Although multitasking is not part of Fortran 90, the interaction of this proposal with multitasking extensions has been considered. A model is that each virtual processor has a separate instance of each condition. If an enable construct contains statements that spawn tasks, enable, handle, and end-enable statements act as barriers at which the condition values are merged; if any is signaling, one of the signaling values is taken. Condition handling is therefore permissible within a pure procedure.]
.....

25/33+ Add
END ENABLE
.....

78/19. After 'terminated', add 'unless the ALLOCATION_ERROR condition is enabled'.
.....

80/25. After 'terminated', add 'unless the DEALLOCATION_ERROR condition is enabled'.
.....

119/9+ Add
(4) ENABLE construct
.....

95/35. Add 'Transfers from an enable or a handle block may occur only through execution of a SIGNAL, RESIGNAL, or ENABLE statement.'.

.....

129/2+. Add

<<8.1.5 Condition handling>>

A condition is a data object of the derived type CONDITION of the intrinsic module CONDITIONS (15). The value of a condition is the value of its integer component. The value zero corresponds to the normal or 'quiet' state and nonzero values correspond to exceptional circumstances. All intrinsic 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 in a statement for which the condition is enabled. The processor may signal an intrinsic condition that is not enabled. When the processor signals an intrinsic condition, it has a positive value.

Conditions are like other derived types except in these respects:

1. An intrinsic operation or procedure that is invoked from a statement for which a set of intrinsic conditions is enabled behaves as if those conditions were accessed in the intrinsic's code by a use statement for those conditions. This may also happen for intrinsic conditions that are not enabled.
2. There is a special relationship between enabled or handled conditions and the control flow.
3. 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.

[Note: The proposal allows the in-lining of procedures with no change to the enable constructs. This may cause additional conditions to signal.]

[Note: On many processors, it is expected that some intrinsic 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.]

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

In a sequence of statements that contains no condition handling statements, if the execution of a process would cause a condition to signal but after execution of the sequence no value of a variable depends on the process, whether the condition signals is processor dependent. For example, when Y has the value zero, whether the code

```
          X = 1.0/Y
          X = 3.0
signals DIVIDE_BY_ZERO is processor dependent.

A condition must not signal if the signal could arise only during
execution of a process further to those required or permitted by the
standard.  For example, the intrinsic LOG in the statement
          IF (F(X)>0.) Y = LOG(Z)
must not signal a condition when both F(X) and Z are negative and for the
statement
          WHERE(A>0.) A = LOG (A)
negative elements of A must not cause signaling. On the other hand, when
X has the value 1.0 and Y has the value 0.0, the expression
          X>0.00001 .OR. X/Y>0.00001
is permitted to cause the signaling of DIVIDE_BY_ZERO.

[Note: In general, it is intended that implementations be free within
enable constructs to use the code motion techniques that they use outside
enable constructs.]

If execution of a RETURN or END statement causes a condition other than
INEXACT or UNDERFLOW to become undefined (14.7.6), the processor issues
a warning on the unit identified by * in a WRITE statement and stops.
The message must indicate which conditions are signaling and their values.

[Note: INEXACT and UNDERFLOW may signal when there is no serious
problem.]

<<8.1.5.1. The enable construct>>

The ENABLE construct specifies a (possibly empty) 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 [(<main-condition-list>)] #
                              # [,-(<minus-condition-list>)]     #
                              # [,IMMEDIATE (<immediate-condition-list>)]

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

R835d <handle-stmt>      <<is>> HANDLE [(<handled-condition-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: Each <main-condition>, <minus-condition, and <handled-condition>
must be a condition variable.

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

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

Constraint: An <enable-construct> must not appear in a scoping unit
unless the scoping unit has access to the module CONDITIONS.

The conditions listed in the <main-condition-list> are <<enabled>>
during execution of the enable block. If the enable construct is nested
within an enable block, the conditions enabled in the outer block
except those listed in the <minus-condition-list> are also enabled
in the inner block. If there is a handle-block, the conditions listed
in the <main-condition-list> and the <handled-condition-list>
are <<handled>> in the enable construct. A condition is also handled
in an enable construct if it is handled in a construct within which it
is nested. A handle-block handles only those conditions listed in the
<main-condition-list> and the <handled-condition-list> of its
construct.

The intrinsic condition INSUFFICIENT_STORAGE is implicitly enabled
throughout any scoping unit, including the specification part, within
which it is accessible and the presence of its name on an enable
statement or a handle statement controls only its handling.

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

[Note: 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.]

[Note: 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 a
signaling condition is enabled or handled or is about to be enabled or
handled. If the enable statement is nested in an enable block that has
a handler for such 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.

[Note: On return to the caller, the condition will be signaling. If
the invocation is within an enable block in which the condition is
enabled or handled, there will be a transfer to the handler (or a

return or stop), but not necessarily until the execution of the block is complete. Otherwise, the processor continues normal execution.]

[Note: In an enable block, the pair of statements
ENABLE
END ENABLE
has a checking effect. If any enabled or handled condition is signaling, there will be a transfer of control to a handler (or a stop or return).]

[Note: In a function subprogram it is very desirable to ensure that the function value is defined even if an error condition has been diagnosed and 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.]

[Note: If a condition handled by a handler signals again during execution of the handler, this second signal will be indistinguishable from the first. If it is desired to handle it separately, it must be set to the quiet value and a nested enable must be provided.]

The value of each condition handled by a handle block is set to the quiet value upon normal completion of execution of the block.

<<8.1.5.2 Execution of an 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 condition enabled or handled in the <enable-construct> signals. The <<uncertainty-scope>> of an enable block consists of the statements of the block that lie outside any enable construct that is nested within the enable block. If a condition handled in the <enable-construct> signals in its uncertainty scope, there is a transfer of control to a handler (or a return or stop). This transfer of control may take place on completion of execution of the enable block or may take place sooner after the signaling of the condition. Any variable that might be defined or redefined by execution of a statement of the uncertainty scope or of a procedure invoked in such a statement is undefined, any pointer whose pointer association might be altered has undefined pointer association status, any allocatable array that might be allocated or deallocated may have been allocated or become unallocated, and the file position of any file specified in an input/output statement that might be executed is processor dependent.

[Note: The transfer of control is imprecise in order to allow for optimizations such as vectorization. As a consequence, some variables become undefined. In Example 3 of 8.1.5.5, a copy of the matrix itself would need to be available for the slow algorithm.]

Branching out of an enable construct is not permitted. A CYCLE or EXIT statement is not permitted in an enable construct unless the do construct to which it belongs is nested within the enable construct. An alternate return specifier in an enable construct must not specify the label of a statement outside the construct. An ERR=, END=, or EOR= specifier in a statement in an enable construct must not be the label of a statement outside the construct. A RETURN or STOP statement is permitted in an enable construct.

[Note: 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 both preceded and followed by an <enable-construct> with an empty enable block and no handler. In this context, an IF statement is treated as an <if-construct> containing a single <action-stmt>, a WHERE statement is treated as if preceded by an <assignment-stmt> that assigns the <mask-expr> to a temporary array, and a <where-construct> is treated as such an assignment followed by a sequence of where statements involving the temporary array.

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

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

[Note: 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 condition may signal during the execution of a statement for which it is not enabled. This causes no immediate transfer of control, but may cause a transfer on later execution of an ENABLE or RESIGNAL statement.

<<8.1.5.4 Signal and resignal statements>>

R835g <signal-stmt> <<is>> SIGNAL (<condition-variable>,[<int-expr>])

Constraint: The <condition-variable> must be a variable of derived type CONDITION.

Constraint: The <int-expr> must be of type default integer and be scalar or have the same shape as the condition.

The SIGNAL statement changes the value of the condition it names to that of the expression it contains or to the default value -1 if the expression is not present. The value shall be nonzero. Execution causes a transfer of control. If the statement is in an enable block of an enable construct that has a handler for a condition given a signaling value, the transfer is to the handler of the innermost such enable construct. Otherwise, it is as for a return if in a subprogram, or a stop if in a main program.

R835h <resignal-stmt> <<is>> RESIGNAL

Constraint: A <resignal-stmt> must lie within a <handle-block>.

The RESIGNAL statement causes a transfer of control without changing the value of any condition. If the statement is in an enable block of an enable construct that has a handler for a signaling condition, the transfer is to the handler of the innermost such enable construct. Otherwise, it is as for a return if in a subprogram, or a stop if in a

main program.

<<8.1.5.5 Examples of ENABLE constructs>>

Example 1:

```
   MODULE MATRIX
! Module for matrix multiplication of real arrays of rank 2.
      USE CONDITIONS
      TYPE (CONDITION) MATRIX_ERROR
      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(MATRIX_ERROR)
         END ENABLE
      END FUNCTION MULT
   END MODULE MATRIX
```

This module provides matrix multiplication for real arrays of rank 2. Since the condition INSUFFICIENT_STORAGE is always enabled when accessible, if there is insufficient storage for the necessary temporary array, the module procedure will signal the condition INSUFFICIENT_STORAGE. If an INTRINSIC or OVERFLOW condition occurs, the module procedure will signal it together with the condition MATRIX_ERROR with value -1.

Example 2:

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

```
USE CONDITIONS
ENABLE (USUAL)
 ! First try the "fast" algorithm for inverting a matrix:
   MATRIX1 = FAST_INV (MATRIX)
                     ! MATRIX is not altered during execution of FAST_INV.
HANDLE
 ! "Fast" algorithm failed; try "slow" one:
   USUAL = QUIET
   ENABLE (USUAL)
      MATRIX1 = SLOW_INV (MATRIX)
   HANDLE
      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:

```
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
```

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 SOLVER_ERROR signals with the value -1.

```
SUBROUTINE ZERO_SOLVER (A, B, X, TOLERANCE, F, SOLVER_ERROR)
   USE CONDITIONS
   TYPE(CONDITION) SOLVER_ERROR
```

```
      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 SYSTEM_ERROR.
      IF (COUNT > START+RATE) SIGNAL (SOLVER_ERROR,-1)
      :
   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, SOLVER_ERROR)
   HANDLE (SOLVER_ERROR)

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

Example 6:

```
REAL FUNCTION HYPOT(X, Y)
   USE CONDITIONS
   REAL X, Y
   REAL SCALED_X, SCALED_Y, SCALED_RESULT
   INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
quick: ENABLE(OVERFLOW, UNDERFLOW)  ! try a fast algorithm first
      HYPOT = SQRT( X**2 + Y**2 )
   HANDLE quick
       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 )
         OVERFLOW = QUIET; UNDERFLOW = QUIET
         ENABLE(OVERFLOW)  ! possibility of overflow in unscaling result
            HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) )
            SIGNAL(OVERFLOW)! if overflow does occur here, it is
         END ENABLE         ! signaled to the caller
      END IF
   END ENABLE quick
END FUNCTION HYPOT
```

This illustrates the setting of a special condition value when the problem really has a result that overflows. It also illustrates use of the constant QUIET.

Example 7:

```
      MODULE LIBRARY
      USE CONDITIONS
      ...
      CONTAINS
         SUBROUTINE B
            ...
            X = Y*Z(I) ! Neither OVERFLOW nor BOUND_ERROR is 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) may 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:

```
   USE CONDITIONS
   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 enable construct is equivalent to

```
      ENABLE (OVERFLOW)
          A = B*C
          ENABLE;  END ENABLE
          WHERE(RAINING)  X(:) = X(:)*A
          ENABLE;  END ENABLE
          WHERE(.NOT.RAINING) Y(:) = Y(:)*A
          ENABLE;  END ENABLE
    HANDLE
       .....
    END ENABLE
```

Example 9:

```
   SUBROUTINE LONG
      USE CONDITIONS
      REAL, ALLOCATABLE :: A(:), B(:,:)
      : ! Other specifications
      ENABLE
          :
          ! Lots of code, including many procedure calls
          :
      HANDLE (ALL_CONDITIONS)
          ! Fix-up, including deallocation of any allocated arrays
          IF(ALLOCATED(A)) DEALLOCATE (A)
          IF(ALLOCATED(B)) DEALLOCATE (B)
          :
      END ENABLE
   END SUBROUTINE LONG
```

This illustrates the use of a handle statement with additional conditions.
Here the enable block enables no conditions because fast execution is desired,
but if anything goes wrong (for example, in one of the procedures invoked),
fix-ups are performed, including deallocation of any local allocated arrays.
.....

129/7. After '<end-do-stmt,>' add 'an <enable-stmt>,'.
.....

130/11. Add: 'If any condition is signaling, the processor must issue a
warning on the unit identified by * in a WRITE statement, indicating
which conditions are signaling and their values.'.
.....

145/2-3. Replace sentence by
   If an error condition (9.4.3) occurs during execution of an input/output
   statement that contains an ERR= specifier or lies in an enable block for
   the IO_ERROR condition:
.....

145/10. After 'specifier' add ', or with the handler (or a return or stop)
   when there is no ERR= specifier'.
.....

145/12-13. 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 contains an
   END= specifier or lies in an enable block for the END_OF_FILE condition:
.....

145/19. After 'specifier' add ', or with the handler (or a return or
   stop) when there is no END= specifier'
.....

145/21-22. 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
   contains an EOR= specifier or lies in an enable block for the
   END_OF_RECORD condition:
 .....

145/32. After 'specifier' add ', or with the handler (or a return or stop) when

```
   there is no EOR= specifier'
.....................................................................

148/20. Before 'contains' add 'is not in a enable block for the IO_ERROR
        condition and '.
.....................................................................

148/21.  Before 'contains' add 'is not in a enable block for the END_OF_FILE
         condition and '.
.....................................................................

148/23.  Before 'contains' add 'is not in a enable block for the END_OF_RECORD
         condition and '.
.....................................................................

211/4. After 'for' add 'conditions (15) or for'.
.....................................................................

288+. Add

<<15. CONDITIONS>>

In this section, the intrinsic module CONDITIONS and the intrinsic
conditions supported by the standard are specified. The module is

MODULE CONDITIONS
   TYPE CONDITION
      SEQUENCE
      PRIVATE
      INTEGER VALUE = 0 ! All conditions are initially quiet.
   END TYPE CONDITION

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

   TYPE(CONDITION) STORAGE(3)
   TYPE(CONDITION) ALLOCATION_ERROR, DEALLOCATION_ERROR, INSUFFICIENT_STORAGE
      EQUIVALENCE (STORAGE(1), ALLOCATION_ERROR)
      EQUIVALENCE (STORAGE(2), DEALLOCATION_ERROR)
      EQUIVALENCE (STORAGE(3), INSUFFICIENT_STORAGE)

   TYPE(CONDITION) IO(3)
   TYPE(CONDITION) IO_ERROR, END_OF_FILE, END_OF_RECORD
      EQUIVALENCE (IO(1), IO_ERROR)
      EQUIVALENCE (IO(2), END_OF_FILE)
      EQUIVALENCE (IO(3), END_OF_RECORD)

   TYPE(CONDITION) FLOATING(3)
   TYPE(CONDITION) OVERFLOW, INVALID, DIVIDE_BY_ZERO
      EQUIVALENCE (FLOATING(1), OVERFLOW)
      EQUIVALENCE (FLOATING(2), INVALID)
      EQUIVALENCE (FLOATING(3), DIVIDE_BY_ZERO)

   TYPE(CONDITION) INTEGER(2)
   TYPE(CONDITION) INTEGER_OVERFLOW, INTEGER_DIVIDE_BY_ZERO
      EQUIVALENCE (INTEGER(1), INTEGER_OVERFLOW)
      EQUIVALENCE (INTEGER(2), INTEGER_DIVIDE_BY_ZERO)

   TYPE(CONDITION) USUAL(10)
   TYPE(CONDITION) INTRINSIC
```

```
      EQUIVALENCE (USUAL(1), STORAGE(1))
      EQUIVALENCE (USUAL(4), IO(1))
      EQUIVALENCE (USUAL(7), FLOATING(1))
      EQUIVALENCE (USUAL(10), INTRINSIC)

   TYPE(CONDITION) ALL_CONDITIONS(20)
   TYPE(CONDITION) BOUND_ERROR, SHAPE, MANY_ONE, NOT_PRESENT, UNDEFINED
   TYPE(CONDITION) UNDERFLOW, INEXACT
   TYPE(CONDITION) SYSTEM_ERROR
      EQUIVALENCE (ALL_CONDITIONS(1), USUAL(1))
      EQUIVALENCE (ALL_CONDITIONS(11), BOUND_ERROR)
      EQUIVALENCE (ALL_CONDITIONS(12), SHAPE)
      EQUIVALENCE (ALL_CONDITIONS(13), MANY_ONE)
      EQUIVALENCE (ALL_CONDITIONS(14), NOT_PRESENT)
      EQUIVALENCE (ALL_CONDITIONS(15), UNDEFINED)
      EQUIVALENCE (ALL_CONDITIONS(16), UNDERFLOW)
      EQUIVALENCE (ALL_CONDITIONS(17), INEXACT)
      EQUIVALENCE (ALL_CONDITIONS(18), SYSTEM_ERROR)
      EQUIVALENCE (ALL_CONDITIONS(19), INTEGER(1))

   INTERFACE OPERATOR (==)
      MODULE PROCEDURE EQ_CI, EQ_IC
   END INTERFACE
   PRIVATE EQ_CI, EQ_IC

   INTERFACE OPERATOR (/=)
      MODULE PROCEDURE NE_CI, NE_IC
   END INTERFACE
   PRIVATE NE_CI, NE_IC

   INTERFACE ASSIGNMENT (=)
      MODULE PROCEDURE ASSIGN_CI, ASSIGN_IC
   END INTERFACE
   PRIVATE ASSIGN_CI, ASSIGN_IC

CONTAINS

   LOGICAL ELEMENTAL FUNCTION EQ_CI(C,I)
   TYPE(CONDITION), INTENT(IN) :: C
   INTEGER, INTENT(IN) :: I
      EQ_CI = C%VALUE==I
   END FUNCTION EQ_CI

   LOGICAL ELEMENTAL FUNCTION EQ_IC(I,C)
   INTEGER, INTENT(IN) :: I
   TYPE(CONDITION), INTENT(IN) :: C
      EQ_IC = C%VALUE==I
   END FUNCTION EQ_IC

   LOGICAL ELEMENTAL FUNCTION NE_CI(C,I)
   TYPE(CONDITION), INTENT(IN) :: C
   INTEGER, INTENT(IN) :: I
      NE_CI = C%VALUE/=I
   END FUNCTION NE_CI

   LOGICAL ELEMENTAL FUNCTION NE_IC(I,C)
   INTEGER, INTENT(IN) :: I
   TYPE(CONDITION), INTENT(IN) :: C
      NE_IC = C%VALUE/=I
   END FUNCTION NE_IC
```

```
   ELEMENTAL SUBROUTINE ASSIGN_CI(C,I)
   TYPE(CONDITION), INTENT(OUT) :: C
   INTEGER, INTENT(IN) :: I
     C%VALUE = I
   END SUBROUTINE ASSIGN_CI

   ELEMENTAL SUBROUTINE ASSIGN_IC(I,C)
   INTEGER, INTENT(OUT) :: I
   TYPE(CONDITION), INTENT(IN) :: C
      I = C%VALUE
   END SUBROUTINE ASSIGN_IC

END MODULE CONDITIONS


<<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.

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.

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 is always enabled when
accessible.  Insufficient storage occurring prior to execution of the
first executable statement of a procedure may cause SYSTEM_ERROR to
signal.

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.

SHAPE
This occurs when an array operation or assignment does not conform in shape.

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.

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.

UNDEFINED
This occurs when a value that is required for an operation is detected by the
processor to be undefined.
```

[Note: 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.

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 END= specifier.

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 EOR= specifier.

<<15.3 Floating-point conditions>>

OVERFLOW
This condition occurs when the result for an intrinsic real operation has a very large processor-dependent absolute value, or the real or imaginary part of the result for an intrinsic complex operation has a very large processor-dependent absolute value.

UNDERFLOW
This condition occurs when the result for an intrinsic real operation has a very small processor-dependent absolute value, or the real or imaginary part of the result for an intrinsic complex operation has a very small processor-dependent absolute value. A processor that does not conform to IEEE 754-1985 is not required to detect this condition in an intrinsic operation or procedure.

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. A processor that does not conform to IEEE 754-1985 is not required to detect this condition in an intrinsic operation or procedure.

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 this condition in an intrinsic operation or procedure.

[Note: It is expected that by default the conditions UNDERFLOW and INEXACT will not signal except when enabled.]

<<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.

INTEGER_DIVIDE_BY_ZERO
This condition occurs when an integer division has a zero denominator.

<<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. 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.

[Note: If an exceptional event occurs during the execution of an intrinsic procedure, the associated condition should not be signaling on return if the event is known to be harmless to the results of the procedure.]

<<15.6 System error conditions>>

SYSTEM_ERROR
This condition occurs as a result of a system error.

<<15.7 Array-valued conditions>>

The array-valued conditions and the equivalences of their elements to the scalar conditions allow lists of conditions to be specified conveniently on a use, enable, handle, or signal statement.

6. IDEAS CONCERNING DESIRABLE MODIFICATIONS OF THE CURRENT ENABLE TR

(by Christian Weber, 2.2.1997)

1.) Establish a USE EXCEPTIONS: any new keyword is intrduced only in a scoping unit which USE's this module (to prevent keyword pollution).

2.) Establish the syntax:

```
ENABLE [(condition-name-1,-2,...)]
... Fortran code
[HANDLE [(condition-name-a,-b....)]
.... handling code]
END ENABLE
```

with similar syntax / semantics as in the current draft TR, but the following changes:

(1) it should be made clear that the ENABLE block - with or without condition-list - is there primarily to designate a piece of code for exception control.

(2) it should be made clear (the current text is a bit hard to understand in this) that the set of conditions in ENABLE has *nothing* to do with the set in HANDLE:

- the set in ENABLE simply specifies the minimum set of conditions which *must* be detected by the processor; the processor may - however - be able to detect far more than the conditions specified. I would be in favor to demand that certain exceptions (STORAGE, IEEE if halting is active, INTEGER_DIVIDE, exceptions explicitly raised by SIGNAL) are *always* "enabled". (The name "ENABLE" somewhat blurs that the real purpose of this statement is to designate exception control: the stating of some minimum exception set is only an additional purpose).
- the set in HANDLE denotes the conditions which the handler block wants to handle,
- I would decouple both sets entirely to enhance the clarity of the concept,
- an empty condition list in ENABLE means that no additional conditions (than are "enabled" all the time anyway) are enabled.

3.) Establish the following condition names:

FLOAT_OVERFLOW (may be changed to the IEEE-name if people wish), FLOAT_DIVIDE_BY_ZERO, FLOAT_INVALID, INTEGER_OVERFLOW, INTEGER_DIVIDE_BY_ZERO, INTRINSIC, INSUFFICIENT_STORAGE, SYSTEM_ERROR, perhaps (debatable): UNDEFINED (since this is the only debugging exception which cannot be replaced by explicit Fortran code), and USER_ERROR (to have one exception which is *never* signalled by the system, and which may be used for entirely user-specific errors).

These condition names may be used:

- in ENABLE / HANDLE
- in a new SIGNAL statement:
  SIGNAL (condition),
- in a new inquiry intrinsic
  SIGNALLING (condition) => TRUE/FALSE.

I would *not* define a concept for condition values etc.: this can be left to be processor-specific (since there wasn't much specified anyway).

If there is strong objection to a particular exception, let's do the trick from the IEEE TR and introduce the inquiry intrinsic:

- SUPPORT_EXCEPTION (debatable-condition-name) => TRUE/FALSE.

If the answer is "false", then the user knows that the corresponding exception may never be detected nor signalled by the processor (the corrsponding ENABLE... syntax will always be accepted, though).

I would *not* want to allow SUPPORT..=false for all exceptions, but at most for FLOAT_INVALID, INTEGER_OVERFLOW, INTRINSIC, UNDEFINED.

4.) The rules when a condition is set to quiet / signalling could stay as present; I would make clear, though, that the transfer of control (to the handler, or RETURN/STOP) is always *immediate*, although the point where the exception is actually caused and therefore detected may be indeterminate due to optimizations (with the obvious rule that ENABLE, HANDLE, SIGNAL, calls of SIGNALLING, and END ENABLE are barriers to optimization).

5.) Exception-supportive subprograms:

A simple USE EXCEPTIONS without additional use of ENABLE... stuff causes a subprogram to become "exception-supportive" in the following sense:

any exception which is detected within the subprogram (because it's always enabled) or within a routine called from the subprogram will be transferred to the nearest appropriate exception handler further up the call chain, or will cause a program-halt with error message if
- no such handler exists, or
- the caller is not "exception-supportive".

A USE EXCEPTIONS is therefore equivalent to a ENABLE / END ENABLE around the whole code body.

That's it, essentially.