Exception Handling in Fortran

Van Snyder
Jet Propulsion Laboratory, California Institute of Technology[1]
28 September 2019

NOTE

This is based upon a paper I wrote 13 February 2015. That paper described enumeration types. Enumeration types are part of the work plan for the next revision and are not described herein. This paper is not a proposal for a work item. It is intended to become part of a standing document that contains proposals that were at one time believed to be good ideas, but for which there was not sufficient time for them to be included in the work plan.

1 Introduction

Since 1966, Fortran has had only very rudimentary exception-handling mechanisms: END=, EOR=, ERR=, IOSTAT=, and STAT= specifiers, and alternate returns. Alternate returns are now considered to be obsolescent.

Block-structured exception handling has been provided in other languages for decades. According to *Programming Languages: Principles and Practice, 2nd edition*, by Kenneth C. Louden (a notable textbook on programming languages), "Exception handling was pioneered by the language PL/I in the 1960s and significantly advanced in CLU in the 1970s. However, it was only in the 1980s and early 1990s that design questions were largely resolved." Ada 83, for which development began in 1976, is probably the first language that provided block-structured exception handling in the modern form. It is generally agreed that exception handling is commonplace in all modern languages.

It has been observed that the CHANGE TEAM construct described in the 6 November 2014 draft of TS 18508 (ISO/IEC JTC1/SC22/WG5 paper N2033) is, in effect, an exception block, but with an intrinsic (and invisible) exception handler that only manages necessary synchronization and deallocations.

It's time for Fortran to have a complete block-structured exception handling mechanism. Block-structured exception handling has been proposed for Fortran, but has never been implemented.

Objections have been raised to block-structured exception handling, usually citing performance degradation. While some implementations of block-structured exception handling impose expense even if an exception does not occur, block-structured exception handling does not inevitably impose a significant execution-time penalty if an exception does not occur. For example, the Janus and Verdix Ada compilers' block-structured exception handling mechanisms impose very low cost if an exception does not occur, and the exception handling mechanism provided by the GNU Ada Translator (GNAT) imposes zero cost if an exception does not occur.

The cost of exception handling should not be confused with the cost of exception detection. An exception cannot be handled unless it is detected. Most processors have methods to specify whether certain exceptions, such as subscripts out of bounds, are detected. The cost of an exception handler is the additional cost to provide for handling an exception, if one is detected. Providing a mechanism to handle an exception does not require or imply that the processor is instructed to detect it. If the processor does detect it, the additional overhead to handle it is very small, or nonexistent, until the exception is detected. If the processor does not detect it, the additional overhead to handle an exception that cannot occur is very small, or nonexistent.

2 Proposal

Block-structured exception handling shall be provided in Fortran. Exception handlers shall be allowed in subprograms, BLOCK constructs, CHANGE TEAM constructs, CRITICAL constructs, and DO CONCURRENT constructs.

Exceptions shall be identified by enumerators of enumeration types, not integers. Enumeration types are therefore a prerequisite for a properly designed exception-handling mechanism.

An exception enumeration type, with specified enumerators, shall be described in the standard. To facilitate user-defined exceptions, the exception enumeration type shall be extensible. That is, it shall be possible to define new enumeration types, based upon existing enumeration types, with additional enumerators.

3 Specification – Exceptions and Exception Handlers

3.1 Exceptions

The standard shall specify events that cause exceptions, and the identifiers of enumerators that identify those exceptions. The enumerators shall be of a type EXCEPTION, defined in the intrinsic module ISO_FORTRAN_ENV. Every event that the standard specifies to be an error condition, or a condition that initiates normal or error termination, shall raise an exception, except that an exception shall not be raised if such a condition occurs during execution of a statement that has an END=, EOR=, ERR=, IOSTAT=, or STAT= specifier, or during execution of an intrinsic subroutine that has a present status argument. It shall be possible to specify whether additional conditions, such as subscripts that are not within bounds or references to intrinsic procedures with prohibited argument values, raise exceptions if they are detected. Enumerators identifying these exceptions shall be of a type OPTIONAL_EXCEPTION, an extension of type EXCEPTION, defined in the intrinsic module ISO_FORTRAN_ENV. Which of these conditions a processor is able to detect is processor dependent.

3.2 Enabling Exceptions

Detection of all exceptions that are identified by enumerators of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, is always enabled.

Whether conditions identified by enumerators of extensions of the type EXCEPTION are to be detected can be specified by an ENABLE suffix on a BLOCK, CHANGE TEAM, CRITICAL, DO CONCURRENT, FUNCTION, or SUBROUTINE statement. If detecting such a condition is not explicitly enabled or disabled, whether detecting such a condition is enabled is processor dependent.

An ENABLE suffix applies only to the *specification-part*, and the *block* or *execution-part*, corresponding to the statement, and enclosed constructs and subprograms. It does not apply to referenced subprograms that are not enclosed within the scoping unit or construct for which an ENABLE suffix is specified.

Detection of exceptions raised by RAISE statements is always enabled.

NOTE

ENABLE is a suffix rather than a statement so that it it is clear that it applies to the entire specification part of the subprogram or construct. If it were a statement, it would be necessary to explain whether it applies to the entire specification part, none of the specification part, or only statements within the specification part that appear after the ENABLE statement. It would be necessary to constrain against multiple ENABLE statements, or at least against specifying the same exception more than once within a specification part.

3.3 Exception Handlers

An exception handler is defined by a HANDLE statement, followed by an optional *specification-part*, and an *execution-part*, that appears after the *execution-part-construct*s of an *execution-part*, or after the *block* of a BLOCK construct, CHANGE TEAM construct, CRITICAL construct, or DO CONCURRENT construct.

An exception handler may handle any number of exceptions. Any number of exception handlers may appear, but separate exception handlers in a particular *execution-part*, or following a particular *block*, shall not specify handling the same exception.

Within an exception handler, the identity of the exception, and additional data associated with the exception, shall be available. The additional data are represented by a variable of a type EXCEPTION_DATA defined in the intrinsic module ISO_FORTRAN_ENV, or an extension of that type.

3.4 Exception Handling

Exceptions may be raised be the processor, or by execution of a RAISE statement.

When an exception is raised, execution of the statement in which the exception is raised is abandoned.

When an exception is raised within a *specification-part*, *block*, or *execution-part*, and there is a handler for the exception that is directly contained within the same subprogram or construct, control is transferred to the handler.

When an exception is raised within a construct, and there is no handler for the exception that is directly contained within the same construct, elaboration of its *specification-part* is abandoned, or execution of the construct is abandoned, and the exception is raised in the enclosing construct, if there is one, or in the enclosing subprogram otherwise.

When an exception is raised within a subprogram other than within a construct that could have a handler, and there is no handler for the exception that is directly contained within the same subprogram, elaboration of its *specification-part* is abandoned, or excecution of the subprogram is abandoned as if by execution of a *return-stmt*, and the exception is raised within the statement that invoked the procedure.

If an exception is not handled, the image shall initiate termination. If the exception was not raised by execution of a STOP statement, the processor shall initiate error termination and display a descriptive message on ERROR_UNIT.

If no exception occurs within a construct, and execution of the construct is completed by execution of the last *executable-construct* within its *block*, control is transferred to the END BLOCK, END CHANGE TEAM, END CRITICAL, or END DO statement of the construct. If no exception occurs within a program unit, and execution of the *execution-part* of the program unit is terminated by execution of the last *executable-construct* within that *execution-part*, execution proceeds from the last *executable-construct* to the END statement of the program unit.

When execution of a handler completes by execution of the last *executable-construct* within its *block*, control is transferred to the END BLOCK, END CHANGE TEAM, END CRITICAL, or END DO statement of its enclosing construct, or the END statement of its enclosing program unit.

4 Detailed Description – Exception Handlers

4.1 Defining Exception Handlers

An exception handler consists of a HANDLE statement and a following *block*.

An exception handler can appear after the *execution-part-construct*s of an *execution-part*, or after the *block* of a BLOCK, CHANGE TEAM, CRITICAL, or DO CONCURRENT construct. Syntax rules are modified accordingly.

Syntax rule numbers for *execution-part*, *block-construct*, *critical-construct*, and *do-construct* are the same as in 18-007r1:

R509 *execution-part* **is** *executable-construct*
[*execution-part-construct*] ...
[*exception-handler*] ...

R1107 *block-construct* **is** *block-stmt*
[*specification-part*]
handled-block
end-block-stmt

R1107a *handled-block* **is** *block*
[*exception-handler*] ...

R1111 *change-team-construct* **is** *change-team-stmt*
handled-block
end-change-team-stmt

R1116 *critical-construct* **is** *critical-stmt*
handled-block
end-critical-stmt

R1119 *do-construct* **is** *do-stmt*
handled-block
end-do-stmt

C1130a (R1119) An *exception-handler* shall not appear unless *loop-control* is CONCURRENT *concurrent-header*.

New syntax rules are introduced to define exception handlers.

R1190 *exception-handler* **is** *handle-stmt*
[*specification-part*]
block

R1191 *handle-stmt* **is** HANDLE [(*handle-spec-list*)] *exception-enumerator-list*
or HANDLE (*handle-spec-list*)

R1192 *handle-spec* **is** ID = *exception-identity-variable*
or DATA = *exception-data-variable*

R1193 *exception-enumerator* **is** *name*

C1190 (R1190) Within an *execution-part* or construct, every *exception-enumerator* in every *handle-stmt*

shall be distinct.

C1191 (R1191) Within an *execution-part* or construct, at most one *handle-stmt* without an *exception-enumerator-list* shall appear.

C1192 (R1191) Each *exception-enumerator* shall be an enumerator of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, or an extension of that type.

C1193 (R1192) No *handle-spec* shall appear more than once within a *handle-stmt*.

C1194 (R1192) If *exception-enumerator-list* does not appear, ID= shall appear.

C1195 (R1192) An *exception-identity-variable* shall be a scalar variable of an enumeration type that includes all enumerators in the *exception-enumerator-list*, if any. It may be declared within the *specification-part* of the exception handler. It shall not be a coarray or a coindexed object.

C1196 (R1192) An *exception-data-variable* shall be a scalar allocatable polymorphic variable of declared type EXCEPTION_DATA defined in the intrinsic module ISO_FORTRAN_ENV. It may be declared within the *specification-part* of the exception handler. It shall not be a coarray or a coindexed object.

C1197 (R1190) A branching statement within the block of an *exception-handler* shall not have a branch target within the *block* or *execution-part* in which the handler is appears.

4.2 Handling Exceptions

When an exception is raised, execution of the statement in which the exception is raised is terminated.

When an exception is raised within a *block* or *execution-part*, and there is a handler for the exception following the *block* or within the *execution-part*, control is transferred to the handler. The handler is the one that identifies the exception in the *exception-enumerator-list* of its HANDLE statement, if there is such a handler, or otherwise a handler for which its HANDLE statement does not include an *exception-enumerator-list*. When execution of the *block* of the handler is completed other than by execution of a branching, CYCLE, EXIT, RAISE, or RETURN statement, control is transferred to the *end-block-stmt*, *end-change-team-stmt*, *end-critical-stmt* or *end-do-stmt* of the construct in which the handler appears, if the handler appears within such a construct, or else to the END statement of the procedure in which the handler appears.

When an exception is raised within a *block* and there is no handler for the exception following the *block*, execution of the *block* is terminated and the exception is raised in the *block* of an enclosing construct, if there is one, or in the enclosing *execution-part*.

When an exception is raised but not handled within a CHANGE TEAM construct, execution of the *block* of the construct is terminated, and normal synchronization occurs at the END TEAM statement before the exception is raised in an enclosing *block* or *execution-part*. If this synchronization causes an exception, it is processor dependent which exception is raised within the enclosing *block* or *execution-part*.

When an exception is raised but not handled within a CRITICAL construct, execution of the *block* of the construct is terminated, and the construct is considered to be completed before the exception is raised within the enclosing *block* or *execution-part*.

When an exception is raised but not handled within a DO CONCURRENT construct, that iteration is terminated and the exception is raised within the enclosing *block* or *execution-part* after every iteration of the construct completes. If another exception occurs during another iteration before the construct completes, and is not handled, it is processor dependent which exception is raised within the enclosing *block* or *execution-part*.

When an exception is raised within an *execution-part* and there is no handler for the exception within the *execution-part*, execution of the *execution-part* is terminated as if by execution of a *return-stmt*. If the *execution-part* is the *execution-part* of a procedure that was invoked by a program unit defined by means of Fortran, the exception is raised within the statement that invoked the procedure. If the procedure was invoked by a procedure defined by means other than Fortran, it is processor dependent whether the exception is handled by the invoking procedure, or raised instead at the most-recently executed subroutine or function reference in a Fortran program unit that resulted in the Fortran subprogram being invoked.

NOTE

If an exception is raised but not handled within a final subroutine, the statement in which it is thereupon raised might not be a CALL statement.

If an exception is raised during execution of an exception handler, and it is not handled by a contained handler, or one in an invoked procedure, it is raised in the *block* or *execution-part* that contains the *block* that contains the exception handler, if the exception handler is within a construct, or it is raised within the statement that caused the procedure to be invoked, as described above, if the exception handler is within an *execution-part*.

If an exception is not handled by any handler, the image shall initiate termination. If the exception is not identified by the value of the enumerator STOP_STATEMENT defined in the intrinsic module ISO_FORTRAN_ENV, the processor shall initiate error termination and display a descriptive message on ERROR_UNIT.

Data relating to the exception are provided in the *exception-data-variable*.

4.3 Raising Exceptions

An exception may be raised by the processor.

A program may raise an exception by executing a RAISE statement.

R1194 *raise-stmt* **is** RAISE [(*exception-data-expr*)] *exception-expr*

R1195 *exception-data-expr* **is** *expr*

R1196 *exception-expr* **is** *expr*

C1198 (R1194) *exception-data-expr* shall be of the type EXCEPTION_DATA defined in the intrinsic module ISO_FORTRAN_ENV, or an extension of that type.

C1199 (R1195) *exception-expr* shall be of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, or an extension of that type.

Executing a RAISE statement causes an exception to be raised. The identity of the exception is specified by the *exception-expr*.

Data relating to the exception may be specified by *exception-data-expr*. If *exception-data-expr* appears, its value is assigned to an *exception-data-variable*, if one appears, in the handler that handles the exception, as if by execution of an assignment statement.

If *exception-data-expr* does not appear and the handler that handles the exception has a DATA= specifier, the *exception-data-variable* becomes deallocated.

4.4 Enabling Exception Detection

Detection of all exceptions that are identified by enumerators of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, is always enabled.

Detection of exceptions that are not identified by enumerators of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, such as subscript values not within array bounds, or prohibited values of arguments to intrinsic functions, can be specified to be enabled or disabled by an ENABLE statement.

Exceptions that are not identified by enumerators of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, are identified by enumerators of extensions of that type. The type OPTIONAL_EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV, an extension of the type EXCEPTION, identifies exceptions corresponding to conditions or events specified by the standard to be prohibited.

R1197 *enable-suffix* **is** ENABLE (*enable-item-list*)

R1198 *enable-item* **is** [−] *exception-enumerator*

C1199a (R1197) No *exception-enumerator* shall appear more than once in *enable-item-list*.

C1199b (R1198) Each *exception-enumerator* shall be an enumerator of a type that is an extension of the type EXCEPTION defined in the intrinsic module ISO_FORTRAN_ENV.

R1108 *block-stmt* **is** [*block-construct-name* :] BLOCK [*enable-suffix*]

R1112 *change-team-stmt* **is** [*team-construct-name*] CHANGE TEAM (*team-variable* ■ ■ [, *coarray-association-list*] [, *sync-stat-list*]) ■ ■ [*enable-suffix*]

R1117 *critical-stmt* **is** [*critical-construct-name* :] CRITICAL [*enable-suffix*]

R1123 *loop-control* **is** ...
or [,] CONCURRENT *concurrent-header* [*enable-suffix*]

R1532 *suffix* **is** ...
or *enable-suffix*

If an *enable-item* is an *exception-enumerator* not preceded by −, detection of the specified exception is enabled within the construct or subprogram introduced by the statement in which the *enable-suffix* appears, and in contained *block*s or subprograms, unless explicitly disabled.

If an *enable-item* is an *exception-enumerator* preceded by −, detection of the specified exception is disabled within the construct or subprogram introduced by the statement in which the *enable-suffix* appears, and in contained *block*s or subprograms, unless explicitly enabled.

If detection of an exception is not explicitly enabled or disabled by an *enable-item* in an *enable-suffix* in the statement that introduces a construct, subprogram, containing construct, or containing subprogram, whether detection of the exception is enabled is processor dependent.

Detection of exceptions raised by RAISE statements is always enabled.

5 Type Definitions in ISO_FORTRAN_ENV

5.1 EXCEPTION Type

The ordered enumeration type EXCEPTION shall be defined in the intrinsic module ISO_FORTRAN_-ENV.

Type EXCEPTION shall include the following enumeration literals. Whether it includes additional enumeration literals is processor dependent.

ALLOCATE_ALLOCATED An allocated allocatable variable was allocated by an ALLOCATE statement (6.7.1.3p1).

ALLOCATE_FAILURE An otherwise unspecified error occured during execution of an ALLOCATE statement (6.7.1.1).

ALLOCATE_TYPE_PARAM The value of a type parameter specified by a *type-spec* in an ALLOCATE statement is not correct (6.7.1.1p6).

AUTOMATIC_FAILURE One or more automatic variables were not created.

BACKSPACE_ERROR An error occurred during execution of a BACKSPACE statement.

CLOSE_ERROR An error occurred during execution of a CLOSE statement.

DEALLOCATE_DEALLOCATED A deallocated allocatable variable or a disassociated pointer was deallocated by a DEALLOCATE statement (6.7.1.3p1, 6.7.3.3p1).

DEALLOCATE_FAILURE An otherwise unspecified error occured during execution of a DEALLOCATE statement (6.7.3.1).

EMPTY_REDUCE The initial sequence for a REDUCE intrinsic function is empty, and the IDENTITY argument is not present.

END_FILE_ERROR An error condition occurred during execution of an END FILE statement.

END_OF_FILE An end of file condition occurred during execution of a READ statement.

END_OF_RECORD An end of record condition occurred during execution of a READ statement.

ERROR_STOP_STATEMENT An ERROR STOP statement was executed.

FLUSH_ERROR An error occurred during execution of a FLUSH statement.

INQUIRE_ERROR An error condition other than INQUIRE_INTERNAL_UNIT occurred during execution of an INQUIRE statement.

INQUIRE_INTERNAL_UNIT The *file-unit-number* in an INQUIRE statement specifies an internal unit (9.10.2.1p2).

LOCK_ERROR An error condition other than LOCK_LOCKED or LOCK_LOCKED_OTHER occurred during execution of a LOCK statement (8.5.7p3).

LOCK_LOCKED A lock variable in a LOCK statement is already locked by the executing image (8.5.6p6).

LOCK_LOCKED_OTHER A lock variable in a LOCK statement is already locked by another image (8.5.7p3).

OPEN_ERROR An error occurred during execution of an OPEN statement.

READ_ERROR An exception other than READ_FORMAT_ERROR occurred during execution of a READ statement.

READ_FORMAT_ERROR An input field does not have the form specified for the format item and is not acceptable to the processor (10.7.2.2p3, 10.7.2.3.2p8, 10.7.2.4p3, 10.7.3p2, 10.10.3p1, 10.11.3.2p2).

RESUME_STALLED_IMAGE A stalled image resumed execution at the end of the *block* of a CHANGE TEAM construct.

REWIND_ERROR An error occurred during execution of a REWIND statement.

STOP_STATEMENT A STOP statement was executed.

SYNC_ERROR An exception other than SYNC_STOPPED_IMAGE occurred in an image control statement (8.5.7p2).

SYNC_STOPPED_IMAGE A SYNC ALL or SYNC IMAGES statement initiated synchronization with a stopped image (8.5.7p2).

UNLOCK_ERROR An exception other than UNLOCK_UNLOCKED occurred during execution of an UNLOCK statement (8.5.7p3).

UNLOCK_UNLOCKED A lock variable in an UNLOCK statement is not already locked by the executing image (8.5.6p6).

VALUE_FAILURE One or more dummy variables with the VALUE attribute were not created.

WAIT_ERROR An error occurred during execution of a WAIT statement.

5.2 OPTIONAL_EXCEPTION Type

The ordered enumeration type OPTIONAL_EXCEPTION shall be defined in the intrinsic module ISO_FORTRAN_ENV.

Type OPTIONAL_EXCEPTION shall include the following enumeration literals. Whether it includes additional enumeration literals is processor dependent.

ARGUMENT_VALUE The value of an argument to an intrinsic procedure is not within the allowed range.

COSUBSCRIPT_ERROR The value of a cosubscript is not within the cobounds of the codimension in which it is used, or the value of a final cosubscript is such that it does not specify a valid image.

DEALLOCATED_ARGUMENT A deallocated allocatable variable is an actual argument corresponding to a nonoptional nonallocatable dummy argument (6.7.1.3p1).

DISASSOCATED_ARGUMENT A disassociated pointer is an actual argument corresponding to a nonoptional pointer dummy argument.

ENUM_RANGE The argument to a constructor for an enumeration type was of type integer, and its value was not the value of an enumerator of the type.

ENVIRONMENT_VARIABLE_STATUS The GET_ENVIRONMENT_VARIABLE intrinsic subroutine was executed, an error condition occurred, and the STATUS argument was not present (13.7.68p3).

EXECUTE_COMMAND_CMDSTAT The EXECUTE_COMMAND_LINE intrinsic subroutine was executed, an error condition occurred, and the CMDSTAT argument was not present (13.7.58p3).

IEEE_DIVISION_BY_ZERO An arithmetic operation or intrinsic function invocation caused the IEEE_DIVIDE_BY_ZERO (14.1) to be set.

IEEE_INEXACT_RESULT An arithmetic operation or intrinsic function invocation caused the IEEE_INEXACT_FLAG (14.1) to be set.

IEEE_INF_RESULT The result of an arithmetic operation or intrinsic function invocation was IEEE Inf, or an item in an input list in a READ statement was IEEE Inf.

IEEE_INVALID_RESULT An arithmetic operation or intrinsic function invocation caused the IEEE_INVALID (14.1) to be set.

IEEE_OVERFLOW_RESULT An arithmetic operation, intrinsic function invocation, or execution of a READ statement, caused the IEEE_OVERFLOW (14.1) to be set.

IEEE_SIGNALING_NAN_RESULT The result of an arithmetic operation or intrinsic function invocation was an IEEE signaling NaN, or an item in an input list in a READ statement was an IEEE signaling NaN.

IEEE_UNDERFLOW_RESULT An arithmetic operation, intrinsic function invocation, or execution of a READ statement, caused the IEEE_UNDERFLOW (14.1) to be set.

INTEGER_DIVIDE_BY_ZERO During execution of an integer divide operation, the divisor was zero.

INTEGER_OVERFLOW Overflow occurred during an arithmetic operation or intrinsic function invocation whose result is of type integer, or during execution of a READ statement while reading an input item of type integer.

PARENT_IO An OPEN, CLOSE, BACKSPACE, ENDFILE, or REWIND statement was executed while a parent data transfer was active on the same unit.

REAL_OVERFLOW Overflow occurred during an arithmetic operation or intrinsic function invocation whose result is of type real or complex, or during execution of a READ statement while reading an input item of type real or complex.

RECURSIVE_IO A function referenced within an input or output list caused data transfer using the same unit.

RECURSIVE_REF A nonrecursive procedure was referenced recursively.

SUBSCRIPT_ERROR The value of a subscript is not within the bounds of the dimension in which it is used.

UNDERFLOW Underflow occurred during an arithmetic operation or intrinsic function invocation whose result is of type real or complex, or during execution of a READ statement while reading an input item of type real or complex.

ZERO_DIVIDE The value of the second operand of a divide operation, or the second argument of a MOD or MODULO intrinsic function, was zero.

It is processor dependent which of the exceptions identified by these enumerators a processor is able to detect.

If several of these exceptions might be raised by the processur during execution of an arithmetic operation or invocation of an intrinsic function, for example IEEE_SIGNALING_NAN_RESULT and ARGUMENT_VALUE, which exception is raised is processor dependent.

5.3 EXCEPTION_DATA Type

The extensible derived type EXCEPTION_DATA shall be defined in the intrinsic module ISO_FORTRAN_ENV.

It shall have a type-bound procedure for defined formatted output, which shall not require a *char-literal-constant* to follow a DT format specifier.

It shall have the following public components, and may may have additional processor-dependent public or private components. These might represent, for example, the file name, program unit name, and line number where the exception occurred.

STATUS is a default integer scalar. When an exception is raised by the processor, its value is the value that would have been assigned to the variable in a STAT= or IOSTAT= specifier, if the statement that caused the exception could have had such a specifier, or a status argument to an intrinsic procedure, if the procedure has such an argument; otherwise, its value is zero.

MESSAGE is an allocatable default character variable with deferred length. When an exception is raised by the processor, its value is the value that would have been assigned to the variable in an ERRMSG= or IOMSG= specifier, if the statement that caused the exception could have had such a specifier, or a message argument to an intrinsic procedure, if the procedure has such an argument, or the text of the *stop-code* of a STOP or ERROR STOP statement, if any; otherwise, its value and allocation status are processor dependent. If it is allocated, its value shall not be undefined.

5.4 ARITHMETIC_EXCEPTION_DATA Type

The extensible derived type ARITHMETIC_EXCEPTION_DATA, an extension of the type EXCEPTION_DATA defined in the intrinsic module ISO_FORTRAN_ENV, shall be defined in the intrinsic module ISO_FORTRAN_ENV.

When an exception is raised by the processor during an intrinsic operation or intrinsic function invocation whose result is of type real or complex, or during excecution of a READ statement in which reading an input item of type real or complex resulted in an exception, a value of this type shall be assigned to the *exception-data-variable*, if any, in the handler.

It shall have one kind type parameter:

KIND When an exception is raised by the processor, the value of the kind type parameter is the kind of the result of the operation or intrinsic function invocation whose result is of type real or complex and whose execution resulted in the exception, or the kind of an input item in a READ statement for which reading the item resulted in the exception.

It shall have a type-bound procedure for defined formatted output that overrides the one for type EXCEPTION_DATA, which shall not require a *char-literal-constant* to follow a DT format specifier.

It shall have the following public component, and may have additional processor-dependent public or private components.

INFO is of type real and of kind specified by the kind type parameter. When an exception is raised by the processor, the component shall be defined with a processor-dependent value that might be useful in explaining the exception, for example, a representation of the payload in a signaling NaN.

6 Example

This example illustrates the use of an exception handler to construct an efficient NORM2 function, that produces a result that is within the range of the result kind, even if overflow or underflow occurs, unless the result would overflow or underflow.

```
module NORM2_m
  use ISO_Fortran_Env, only: IEEE_Invalid_Result, IEEE_Underflow_Result, &
    & Real_Overflow, Underflow
  implicit NONE
  private
  public :: NORM2
  interface NORM2
    module procedure DNRM2_1, SNRM2_1, DNRM2_2 ! ... for other kinds and ranks
  end interface
contains
  pure double precision function DNRM2_1 ( X ) &
    & enable ( IEEE_Invalid_Result, IEEE_Underflow_Result, Real_Overflow, &
    & Underflow )
    double precision, intent(in) :: X(:)
    ! This will be zero, without an exception, if every element of X is zero.
    dnrm2_1 = sqrt ( dot_product(x,x) )
  handle IEEE_Underflow_Result, Real_Overflow, Underflow
    ! We don't care which exception occurred, and don't need information
    ! about it, so the HANDLE statement does not need ID= or DATA= specifiers.
    ! Efficiency is not an issue here, because these exceptions are expected
    ! to be rare.
    ! We need to turn off detecting overflow and underflow, so we do not
    ! raise another exception if the result is out of range.
    block enable (-IEEE_Underflow_Result, -Real_Overflow, -Underflow )
      double precision XMAX
      xmax = maxval ( abs ( x ) )
      dnrm2_1 = xmax * sqrt ( dot_product(x/xmax,x/xmax ) )
    end block
  handle IEEE_Invalid_Result
    use IEEE_Arithmetic, only: IEEE_Is_Finite, IEEE_Is_NaN, &
      & IEEE_Positive_Inf, IEEE_Quiet_Nan, IEEE_Value
    if ( any ( IEEE_Is_NaN(x) ) ) then
      dnrm2_1 = IEEE_Value ( dnrm2_1, IEEE_Quiet_Nan )
    else
      dnrm2_1 = IEEE_Value ( dnrm2_1, IEEE_Positive_Inf )
    end if
  end function DNRM2_1
  ! and similarly for SNRM2_1, and for other ranks
end module NORM2_m
```