TECHNICAL REPORT

ISO/IEC TR 15580

First edition
1998-12-01

Information technology — Programming languages — Fortran — Floating-point exception handling

Technologies de l'information — Langages de programmation — Fortran — Manipulation de l'exception du point flottant

Reference number
ISO/IEC TR 15580:1998(E)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The main task of technical committees is to prepare International Standards, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

ISO/IEC TR 15580, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This Technical Report specifies an extension of the programming language Fortran, specified by ISO/IEC 1539-1:1997.

It is the intention of ISO/IEC JTC 1/SC 22 that the semantics and syntax described in this Technical Report be incorporated in the next revision of ISO/IEC 1539-1:1997 exactly as they are specified here unless experience in the implementation and use of this feature has identified any errors which need to be corrected, or changes are required in order to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such integration changes on existing commercial implementations.

Introduction

Exception handling is required for the development of robust and efficient numerical software. In particular, it is necessary in order to be able to write portable scientific libraries. In numerical Fortran programming, current practice is to employ whatever exception handling mechanisms are provided by the system/vendor. This clearly inhibits the production of fully portable numerical libraries and programs. It is particularly frustrating now that IEEE arithmetic (specified by IEEE 754-1985 *Standard for binary floating-point arithmetic*, also published as IEC 559:1989, *Binary floating-point arithmetic for microprocessor systems)* is so widely used, since built into it are the five conditions: overflow, invalid, divide-by-zero, underflow, and inexact. Our aim is to provide support for these conditions.

We have taken the opportunity to provide support for other aspects of the IEEE standard through a set of elemental functions that are applicable only to IEEE data types.

This proposal involves three standard modules:

- `IEEE_EXCEPTIONS` contains a derived type, some named constants of this type, and some simple procedures. They allow the flags to be tested, cleared, set, saved, or restored.
- `IEEE_ARITHMETIC` behaves as if it contained a `USE` statement for all of `IEEE_EXCEPTIONS` and provides support for other IEEE features through further derived types, named constants, and simple procedures.
- `IEEE_FEATURES` contains some named constants that permit the user to indicate which IEEE features are essential in the application. Some processors may execute more slowly when certain features are requested.

To facilitate maximum performance, each of the proposed functions does very little processing of arguments. In many cases, a processor may generate only a few inline machine code instructions rather than library calls.

In order to allow for the maximum number of processors to provide the maximum value to users, we do **not** require IEEE conformance. A vendor with no IEEE hardware need not provide these modules and any request by the user for any of them with a `USE` statement will give a compile-time diagnostic. A vendor whose hardware does not fully conform with the IEEE standard may be unable to provide certain features. In this case, a request for such a feature will give a compile-time diagnostic. Another possibility is that not all flags are supported or that the extent of support varies according to the kind type parameter. The user must utilize an inquiry function to determine if he or she can count on a specific feature of the IEEE standard.

Note that an implementor should avoid a macro implementation, as IEEE conformance is often controlled by compiler switches. A processor which offers a switch to turn off a facility should adjust the values returned for these inquiries. For example, a processor which allows gradual underflow to be turned off (replaced with flush to zero) should return false for `IEEE_SUPPORT_DENORMAL(X)` when a source file is processed with that option on. Naturally it should return true when that option is not in effect.

The most important use of a floating-point exception handling facility is to make possible the development of much more efficient software than is otherwise possible. The following 'hypotenuse' function, $\sqrt{x^2+y^2}$, illustrates the use of the facility in developing efficient software.

```
REAL FUNCTION HYPOT(X, Y)
! In rare circumstances this may lead to the signaling of IEEE_OVERFLOW
   USE, INTRINSIC :: IEEE_ARITHMETIC
   REAL X, Y
   REAL SCALED_X, SCALED_Y, SCALED_RESULT
   LOGICAL, DIMENSION(2) :: FLAGS
   TYPE (IEEE_FLAG_TYPE), PARAMETER, DIMENSION(2) :: &
          OUT_OF_RANGE = (/ IEEE_OVERFLOW, IEEE_UNDERFLOW /)
   INTRINSIC SQRT, ABS, EXPONENT, MAX, DIGITS, SCALE
! The processor clears the flags on entry
```

```
! Try a fast algorithm first
   HYPOT = SQRT( X**2 + Y**2 )
   CALL IEEE_GET_FLAG(OUT_OF_RANGE,FLAGS)
   IF ( ANY(FLAGS) ) THEN
      CALL IEEE_SET_FLAG(OUT_OF_RANGE,.FALSE.)
      IF ( X==0.0 .OR. Y==0.0 ) THEN
         HYPOT = ABS(X) + ABS(Y)
      ELSE IF ( 2*ABS(EXPONENT(X)-EXPONENT(Y)) > DIGITS(X)+1 ) THEN
         HYPOT = MAX( ABS(X), ABS(Y) )!  one of X and Y can be ignored
      ELSE     ! scale so that ABS(X) is near 1
         SCALED_X = SCALE( X, -EXPONENT(X) )
         SCALED_Y = SCALE( Y, -EXPONENT(X) )
         SCALED_RESULT = SQRT( SCALED_X**2 + SCALED_Y**2 )
         HYPOT = SCALE( SCALED_RESULT, EXPONENT(X) ) ! may cause overflow
      END IF
   END IF
! The processor resets any flag that was signaling on entry
END FUNCTION HYPOT
```

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation may involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if X and Y are both near the overflow limit).

If the overflow or underflow flag is signaling on entry, it is reset on return by the processor, so that earlier exceptions are not lost.

Can all this be accomplished without the help of an exception handling facility? Yes, it can – in fact, the alternative code can do the job, but of course it is much less efficient. That's the point. The HYPOT function is special, in this respect, in that the normal and alternative codes try to accomplish the same task. This is not always the case. In fact, it very often happens that the alternative code concentrates on handling the exceptional cases and is not able to handle all of the non-exceptional cases. When this happens, a program which cannot take advantage of hardware flags could have a structure like the following:


```
if ( in the first exceptional region ) then
    handle this case
else if ( in the second exceptional region ) then
    handle this case
  :
else
    execute the normal code
end
```

But this is not only inefficient, it also inverts the logic of the computation. For other examples, see Hull, Fairgrieve and Tang (1994) and Demmel and Li (1994).

The code for the HYPOT function can be generalized in an obvious way to compute the Euclidean norm, $\sqrt{x_1^2+x_2^2+...+x_n^2}$ of an *n*–vector; the generalization of the alternative code is not so obvious (though straightforward) and will be much slower relative to the normal code than is the case with the HYPOT function.

In connection with reliable computation, there is a need for intrinsic conditions further to those of the IEEE floating-point standard. Examples are:

- INSUFFICIENT_STORAGE for when the processor is unable to find sufficient storage to continue execution.
- INTEGER_OVERFLOW and INTEGER_DIVIDE_BY_ZERO for when an intrinsic integer operation has a very large result or has a zero denominator.

- `INTRINSIC` for when an intrinsic procedure has been unsuccessful.
- `SYSTEM_ERROR` for when a system error occurs.

This proposal has been designed to allow such enhancements in the future.

References

Demmel, J.W. and Li, X. (1994). Faster Numerical Algorithms via Exception Handling. IEEE Transactions on Computers, 43, no. 8, 983-992.

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

Information technology – Programming languages – Fortran – Floating-point exception handling

1 General

1.1 Scope

This Technical Report specifies an extension of the programming language Fortran, specified by the international standard ISO/IEC 1539-1:1997. Its main aim is to provide support for the five exceptions of the IEEE standard for floating-point arithmetic, but it also provides support for other features of the IEEE standard. A processor is permitted to provide partial support and there are facilities for enquiring about which features are supported or requiring support of certain features.

Clause 2 of this Technical Report contains a technical description of the features. It provides an overview and does not include all the fine details. Clause 3 contains the editorial changes to the standard and thereby provides a complete definition.

1.2 Normative references

The following standards contain provisions which, through references in this text, constitute provisions of this Technical Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred applies. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1:1997, *Information technology – Programming Languages – Fortran – Part 1: Base language.*

IEC 559:1989, *Binary floating-point arithmetic for microprocessor systems.*

Since IEC 559:1989 was originally IEEE 754-1985 *Standard for binary floating-point arithmetic*, and is widely known by this name, we refer to it as the **IEEE standard** in this Technical Report.

2 Technical specification

2.1 The model

This proposal is based on the IEEE model with flags for the floating-point exceptions (invalid, overflow, divide-by-zero, underflow, inexact), a flag for the rounding mode (nearest, up, down, to-zero), and flags for whether halting occurs following exceptions. It is not necessary for the hardware to have any such flags (they may be simulated by software) or for it to support all the modes. Inquiry procedures are available to allow a program to determine the extent of support. Inquiries are in terms of reals, but the same level of support is provided for the corresponding complex kind.

Some hardware may be able to provide no support of these features or only partial support. It may execute faster with compiled code that does not support all the features. This proposal therefore involves three intrinsic modules. `IEEE_EXCEPTIONS` is for the exceptions and the minimum requirement is for the support of overflow and divide-by-zero for all kinds of real and complex data. `IEEE_ARITHMETIC` behaves as if it contained a `USE` statement for all of `IEEE_EXCEPTIONS` and provides support for other IEEE features. `IEEE_FEATURES` contains some named constants that permit the user to indicate which features are essential in the application. A program is required to fail if a requested feature is not available. The modules contain five derived types (subclause 2.3), named constants to control the level of support (subclause 2.4), and a collection of procedures (subclauses 2.5 to 2.10). None of the procedures is permitted as an actual argument.

2.2 The USE statement for an intrinsic module

New syntax on the USE statement provides control over whether it is intended to access an intrinsic module:

```
USE, INTRINSIC :: IEEE_ARITHMETIC
```

or not:

```
USE, NON_INTRINSIC :: MY_IEEE_ARITHMETIC
```


The `INTRINSIC` statement is not extended. For the old form:

```
USE IEEE_ARITHMETIC
```

the processor looks first for a non-intrinsic module.

2.3 The derived types and data objects

The module `IEEE_EXCEPTIONS` contains the derived types:

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

The module `IEEE_ARITHMETIC` contains the derived types:

- `IEEE_CLASS_TYPE`, for identifying a class of floating-point values. Its only possible values are those of named constants defined in the module:
  `IEEE_SIGNALING_NAN`, `IEEE_QUIET_NAN`, `IEEE_NEGATIVE_INF`, `IEEE_NEGATIVE_NORMAL`, `IEEE_NEGATIVE_DENORMAL`, `IEEE_NEGATIVE_ZERO`, `IEEE_POSITIVE_ZERO`, `IEEE_POSITIVE_DENORMAL`, `IEEE_POSITIVE_NORMAL`, and `IEEE_POSITIVE_INF`.

- IEEE_ROUND_TYPE, for identifying a particular rounding mode. Its only possible values are those of named constants defined in the module: IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN for the IEEE modes; and IEEE_OTHER for any other mode.

The module IEEE_FEATURES contains the derived type:

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

2.4 The level of support

When IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessible, IEEE_OVERFLOW and IEEE_DIVIDE_BY_ZERO are supported in the scoping unit for all kinds of real and complex data. Which other exceptions are supported may be determined by the function IEEE_SUPPORT_FLAG, see subclause 2.6. Whether control of halting is supported may be determined by the function IEEE_SUPPORT_HALTING. The extent of support of the other exceptions may be influenced by the accessibility of the named constants IEEE_INEXACT_FLAG, IEEE_INVALID_FLAG, and IEEE_UNDERFLOW_FLAG of the module IEEE_FEATURES. If a scoping unit has access to IEEE_UNDERFLOW_FLAG of IEEE_FEATURES, the scoping unit must support underflow and return true from IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW,X) for at least one kind of real. Similarly, if IEEE_INEXACT_FLAG or IEEE_INVALID_FLAG is accessible, the scoping unit must support the exception and return true from the corresponding inquiry for at least one kind of real. Also, if IEEE_HALTING is accessible, the scoping unit must support control of halting and return true from IEEE_SUPPORT_HALTING(FLAG) for the flag.

If a scoping unit does not access IEEE_EXCEPTIONS or IEEE_ARITHMETIC, the level of support is processor dependent, and need not include support for any exceptions. If a flag is signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

For processors with IEEE arithmetic, further IEEE support is available through the module IEEE_ARITHMETIC. The extent of support may be influenced by the accessibility of the named constants of the module IEEE_FEATURES. If a scoping unit has access to IEEE_DATATYPE of IEEE_FEATURES, the scoping unit must support IEEE arithmetic and return true from IEEE_SUPPORT_DATATYPE(X) (see subclause 2.6) for at least one kind of real. Similarly, if IEEE_DENORMAL, IEEE_DIVIDE, IEEE_INF, IEEE_NAN, IEEE_ROUNDING, or IEEE_SQRT is accessible, the scoping unit must support the feature and return true from the corresponding inquiry function for at least one kind of real. In the case of IEEE_ROUNDING, it must return true for all the rounding modes IEEE_NEAREST, IEEE_TO_ZERO, IEEE_UP, and IEEE_DOWN.

Execution may be slowed on some processors by the support of some features. If IEEE_EXCEPTIONS or IEEE_ARITHMETIC is accessed but IEEE_FEATURES is not accessed, the vendor is free to choose which subset to support. The processor's fullest support is provided when all of IEEE_FEATURES is accessed:

```
USE IEEE_ARITHMETIC; USE IEEE_FEATURES
```

but execution may then be slowed by the presence of a feature that is not needed. In all cases, the extent of support may be determined by the inquiry functions of subclause 2.6.

If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return.

If a flag is quiet on entry to a procedure with access to IEEE_EXCEPTIONS or IEEE_ARITHMETIC and is signaling on return, the processor will not restore it to quiet.

In a procedure, the processor ensures that the flags for halting have the same values on return as on entry.

In a procedure, the processor ensures that the flags for rounding have the same values on return as on entry.

2.5 The exception flags

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

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

where `FLAG` is of type `IEEE_FLAG_TYPE` and `FLAG_VALUE` is of type default `LOGICAL`. Being elemental allows an array of flag values to be obtained at once and obviates the need for a list of flags.

Flag values may be assigned by the elemental subroutine

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

An exception must not signal if this could arise only during execution of a process further to those required or permitted by the standard. For example, the statement

```
IF (F(X)>0.0) Y = 1.0/Z
```

must not signal `IEEE_DIVIDE_BY_ZERO` when both `F(X)` and `Z` are zero and the statement

```
WHERE(A>0.0) A = 1.0/A
```

must not signal `IEEE_DIVIDE_BY_ZERO.` 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 `IEEE_DIVIDE_BY_ZERO`.

2.6 Inquiry functions for the features supported

The module `IEEE_EXCEPTIONS` contains the following inquiry functions:

- `IEEE_SUPPORT_FLAG(FLAG[,X])` True if the processor supports an exception flag for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_HALTING(FLAG)` True if the processor supports the ability to control during program execution whether to abort or continue execution after an exception.

The module `IEEE_ARITHMETIC` contains the following inquiry functions:

- `IEEE_SUPPORT_DATATYPE([X])` True if the processor supports IEEE arithmetic for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`. Here support means employing an IEEE data format and performing the operations of `+`, `–`, and `*` as in the IEEE standard whenever the operands and result all have normal values.
- `IEEE_SUPPORT_DENORMAL([X])` True if the processor supports the IEEE denormalized numbers for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_DIVIDE([X])` True if the processor supports divide with the accuracy specified by the IEEE standard for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_INF([X])` True if the processor supports the IEEE infinity facility for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_NAN([X])` True if the processor supports the IEEE Not-A-Number facility for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_ROUNDING(ROUND_VALUE[,X])` True if the processor supports a particular rounding mode for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.

Here, support includes the ability to change the mode by

```
CALL IEEE_SET_ROUNDING_MODE(ROUND_VALUE)
```

- `IEEE_SUPPORT_SQRT([X])` True if the processor supports IEEE square root for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.
- `IEEE_SUPPORT_STANDARD([X])` True if the processor supports all the IEEE facilities defined in this Technical Report for all reals (`X` absent) or for reals of the same kind type parameter as the argument `X`.

2.7 Elemental functions

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

- `IEEE_CLASS(X)` Returns the IEEE class (see subclause 2.3 for the possible values).
- `IEEE_COPY_SIGN(X,Y)` IEEE copysign function, that is `X` with the sign of `Y`.
- `IEEE_IS_FINITE(X)` IEEE finite function. True if `IEEE_CLASS(X)` has one of the values `IEEE_NEGATIVE_NORMAL`, `IEEE_NEGATIVE_DENORMAL`, `IEEE_NEGATIVE_ZERO`, `IEEE_POSITIVE_ZERO`, `IEEE_POSITIVE_DENORMAL`, `IEEE_POSITIVE_NORMAL`.
- `IEEE_IS_NAN(X)` True if the value is IEEE Not-a-Number.
- `IEEE_IS_NEGATIVE(X)` True if the value is negative (including negative zero).
- `IEEE_IS_NORMAL(X)` True if the value is a normal number.
- `IEEE_LOGB(X)` IEEE *logb* function, that is, the unbiased exponent of `X`.
- `IEEE_NEXT_AFTER(X,Y)` Returns the next representable neighbor of `X` in the direction toward `Y`.
- `IEEE_REM(X,Y)` The IEEE REM function, that is `X – Y*N`, where `N` is the integer nearest to the exact value `X/Y`.
- `IEEE_RINT(X)` Round to an integer value according to the current rounding mode.
- `IEEE_SCALB` (X,I) Returns $2^{I}X$.
- `IEEE_UNORDERED(X,Y)` IEEE unordered function. True if `X` or `Y` is a NaN and false otherwise.
- `IEEE_VALUE(X,CLASS)` Generate a value of a given IEEE class. The value of `CLASS` is permitted to be
  - `IEEE_SIGNALING_NAN` or `IEEE_QUIET_NAN` if `IEEE_SUPPORT_NAN(X)` has the value true,
  - `IEEE_NEGATIVE_INF` or `IEEE_POSITIVE_INF` if `IEEE_SUPPORT_INF(X)` has the value true,
  - `IEEE_NEGATIVE_DENORMAL` or `IEEE_POSITIVE_DENORMAL` if `IEEE_SUPPORT_DENORMAL(X)` has the value true,
  - `IEEE_NEGATIVE_NORMAL`, `IEEE_NEGATIVE_ZERO`, `IEEE_POSITIVE_ZERO` or `IEEE_POSITIVE_NORMAL`.

  Although in most cases the value is processor dependent, the value does not vary between invocations for any particular `X` kind type parameter and `CLASS` value.

2.8 Elemental subroutines

The module IEEE_EXCEPTIONS contains the following elemental subroutines:

- IEEE_GET_FLAG(FLAG,FLAG_VALUE) Get an exception flag.
- IEEE_GET_HALTING_MODE(FLAG,HALTING) Get halting mode for an exception. The initial halting mode is processor dependent. Halting is not necessarily immediate, but normal processing does not continue.
- IEEE_SET_FLAG(FLAG,FLAG_VALUE) Set an exception flag.
- IEEE_SET_HALTING_MODE(FLAG,HALTING) Controls continuation or halting on exceptions.

2.9 Non-elemental subroutines

The module IEEE_EXCEPTIONS contains the following non-elemental subroutines:

- IEEE_GET_STATUS(STATUS_VALUE) Get the current values of the set of flags that define the current state of the floating point environment. STATUS_VALUE is of type IEEE_STATUS_TYPE.
- IEEE_SET_STATUS(STATUS_VALUE) Restore the values of the set of flags that define the current state of the floating point environment (usually the floating point status register). STATUS_VALUE is of type IEEE_STATUS_TYPE and has been set by a call of IEEE_GET_STATUS.

The module IEEE_ARITHMETIC contains the following non-elemental subroutines:

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

2.10 Transformational function

The module IEEE_ARITHMETIC contains the following transformational function:

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