Non-Intrinsic IEEE Modules

January 9, 2013, July 31, 2013

R. J. Hanson

Abstract

Three non-intrinsic Fortran 2003 modules are implemented. These modules are IEEE_FEATURES, IEEE_EXCEPTIONS, and IEEE_ARITHMETIC. All codes assume the underlying machine architecture is the Intel or AMD x86. Machine instructions are used that have appeared since about 1999. The module procedures call C codes that use in-line assembler for instructions that must maintain the flags, status, and computation of the remainder function, IEEE_REM.

Contents

1 Introduction **1**

2 The Module IEEE_FEATURES **2**

3 The Module IEEE_EXCEPTIONS **3**

3.1 IEEE_STATUS_TYPE, IEEE_GET_STATUS, and IEEE_SET_STATUS . . . 3

3.1.1 Usage . . . 3

3.1.2 Interface . . . 3

3.2 IEEE_FLAG_TYPE, IEEE_GET_FLAG, and IEEE_SET_FLAG . . . 4

3.2.1 Usage . . . 4

3.3 IEEE_GET_HALTING_MODE, IEEE_SET_HALTING_MODE . . . 5

4 The Module IEEE_ARITHMETIC **5**

5 The Test Program IEEE_tests_gfortran **6**

6 Acknowledgments **7**

1 Introduction

This brief documentation for the IEEE modules is intended primarily for the team that will install them as part of the Fortran compiler, known by its common name *gfortran*. These notes might be useful to developers when the results are made available in a future compiler release.

There are sections below that describe additional details of the three modules. Readers will find the names and specification details in the final draft of the Fortran 2008 standard, **ISO/IEC JTC 1/SC 22/WG 5/N1830**. This will be referred to as **F2008**. Have a copy

of this document available.

Here are four principles that contributed to design and construction of this work:

1. All critical parts of the standard specification are implemented.
2. Every routine and derived type is written in Fortran, as far as possible.
3. Access to machine instructions is obtained by Fortran calling C routines with in-line assembler for these instructions. The Fortran 2003 standard specification for interoperability of Fortran and C is used.
4. Every routine in the modules is *thread-safe*, based on Fortran OpenMP multi-thread usage.

As a bonus, a derived type: TYPE(IEEE_X87_PRECISION_TYPE), is defined. It allows the run-time system to get, test and set the precision that the x87 floating-point unit employs in its internal arithmetic. Useful choices are: IEEE_SINGLE, IEEE_DOUBLE and IEEE_DOUBLE_EXTENDED.

A test program, IEEE_tests_gfortran.f90, calls routines from the package and checks their correctness. This testing software was first developed with the Intel XE 12.0 Fortran compiler. That allowed the C codes to be developed with Intel formats, compiled with MS VS C++, and using the in-line assembler. After this was working the in-line assembler codes were converted to use the *gcc* C compiler formats.

Certain routines, **F2008**, Table 14.1, are intended as *elemental* or *pure*. These declarations were sacrificed in favor of OpenMP thread-safety. For example a variable that is *threadprivate*, with respect to OpenMP, must be specified with a *save* attribute. But that attribute is not allowed in *elemental* or *pure* routines. It was felt that assuring thread safety, using *threadprivate* local variables, was more important than adhering to these declarations. *Users of the package can ignore this issue.*

2 The Module IEEE_FEATURES

See **F2008**, 14.2.4.
The module IEEE_FEATURES defines the type IEEE_FEATURES_TYPE, expressing the need for particular features. Its 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. There is an additional constant, IEEE_X87_ACCURACY, indicating support for precision control in the x87 floating- point unit.

All these features are available by default. So use-associating this module will have no effect on the performance obtained by additional use-association of the modules IEEE_ARITHMETIC and IEEE_EXCEPTIONS.

3 The Module IEEE_EXCEPTIONS

See **F2008**, Table 14.2, and 14.2.4.
The module IEEE_EXCEPTIONS defines the following types. IEEE_FLAG_TYPE is 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 module also defines arrays of these named constants: IEEE_USUAL = [IEEE_OVERFLOW, IEEE_DIVIDE BY ZERO, IEEE_INVALID] and IEEE ALL = [IEEE_USUAL, IEEE_UNDERFLOW, IEEE_INEXACT].

The inquiry functions IEEE_SUPPORT_FLAG and IEEE_SUPPORT_HALTING always return .TRUE., for all input parameters.

3.1 IEEE_STATUS_TYPE, IEEE_GET_STATUS, and IEEE_SET_STATUS

3.1.1 Usage

The type IEEE_STATUS_TYPE is for representing the floating-point status or state. This object has a C binding and a single *private* component, an array **INTEGER(C_INT) :: STATE(128+27)**. This component contains the 512 bytes of the SSE together with the 108 bytes of the x87 floating-point unit. This array holds the snapshot of the entire state of both floating-point units, when IEEE_GET_STATUS is called. A fragment of an example:

```
   TYPE(IEEE_STATUS_TYPE) ::  SAVE_STATE
...
CALL IEEE_GET_STATUS(SAVE_STATE)
! Do some computing, test and set flags, ...
!  Return floating point units to initial status.
CALL IEEE_SET_STATUS(SAVE_STATE)
```

3.1.2 Interface

The subroutine IEEE_GET_STATUS calls the C routine `GET_STATES`. This C routine copies the x87 and the SSE states into its integer array component. The machine instruction for saving the status of the SSE is *fxsave s;*. The target of the operation is the memory location *int s[128]*. The alignment of this array is required by the hardware to have its memory address divisible by 16. Thus align this intermediate array: *int s[128] __attribute__((aligned(16)));*. Following the save of the SSE state to *s*, the contents are then copied to the component of the derived type *SAVE_STATE* that may not have this alignment constraint. There is no restriction on the memory address alignment for saving the x87 state: *fsave s[27];*.

The subroutine IEEE_SET_STATUS calls the C routine `SET_STATES` with the C binding name `__GFORTRAN__set_states`. This C routine reverses the copy into the x87 and the SSE states from its integer array component. There is a corresponding copy-in, copy-out requirement for restoring the status of the SSE. State saving and restore are relatively expensive, but as they should only be called rarely this should not be an issue.

3.2 IEEE_FLAG_TYPE, IEEE_GET_FLAG, and IEEE_SET_FLAG

See **F2008, 14.3**. The exceptions are the following.

- IEEE_OVERFLOW occurs when the result for an intrinsic real operation or assignment has an absolute value greater than a processor-dependent limit, or the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value greater than a processor-dependent limit. These limits are provided by the environmental parameters *HUGE(X)*, where *X* has a *KIND* value that depends on the accuracy of the operation.

- IEEE_DIVIDE_BY_ZERO occurs when a real or complex division has a nonzero numerator and a zero denominator.

- IEEE_INVALID occurs when a real or complex operation or assignment is invalid; possible examples are SQRT (X) when X is real and has a nonzero negative value, and conversion to an integer (by assignment, an intrinsic procedure, or a procedure defined in an intrinsic module) when the result is too large to be representable.

- IEEE_ UNDERFLOW occurs when the result for an intrinsic real operation or assignment has an absolute value less than a processor-dependent limit and loss of accuracy is detected, or the real or imaginary part of the result for an intrinsic complex operation or assignment has an absolute value less than a processor- dependent limit and loss of accuracy is detected. These limits are provided by the environmental parameters *TINY(X)*, where *X* has a *KIND* value that depends on the accuracy of the operation.

- IEEE_INEXACT occurs when the result of a real or complex operation or assignment is not exact.

3.2.1 Usage

The Fortran subroutine IEEE_GET_FLAG(FLAG, YES_NO) calls the C routines `GETSWX87` and `GETSWSSE`. The C routines get the status words for the x87 and the SSE. The argument FLAG is one of the above named exceptions of type IEEE_FLAG_TYPE. The argument YES_NO is a Fortran logical, indicating that this exception is signaling in either the x87 floating-pint unit or the SSE.

The subroutine IEEE_SET_FLAG (FLAG, YES_NO) calls the C routines `GETSWX87` and `GETSWSSE`. If changes occur after setting or clearing the exception flags, the C routines `SETCWSSE` and `SETSWS87` are called. Both the x87 and SSE status flags are set to the same logical value. The argument FLAG can be scalar or an array. The argument YES_NO can be scalar or an array of Fortran logicals. When an array argument is used for FLAG, any repeats in the entries will use the flag with the highest index.

A fragment of an example:

```
LOGICAL DIV_BY_Z
! ...  Do some computing with floating-point divides:
```

```
CALL IEEE_GET_FLAG(IEEE_DIVIDE_BY_ZERO, DIV_BY_Z)
IF( DIV_BY_Z) THEN
! ...  Take any needed action and clear the flag.
CALL IEEE_SET_FLAG(IEEE_DIVIDE_BY_ZERO, .FALSE.)
END IF
...
```

3.3 IEEE_GET_HALTING_MODE, IEEE_SET_HALTING_MODE

This subroutine guides the occurrence of an exception to either quietly set a status flag or else cause an interrupt that calls an external handler function or terminates the execution. This is illustrated with an example. Suppose a code is compiled to never tolerate an INVALID operation. A part of the code is allowed to encounter this exception and not terminate execution. After this code is finished, and the exception is handled, revert to the initial halting mode for this exception.

A fragment: `LOGICAL ::  HALT, INVLD`

```
! ...Save the initial halting mode:
CALL IEEE_GET_HALTING_MODE(IEEE_INVALID, HALT)
! ...Allow INVALID exceptions without halting.
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, .FALSE.)
!...  Execute code or routines that may set IEEE_INVALID
...
CALL IEEE_GET_FLAG(IEEE_INVALID, INVLD)
IF (INVLD) THEN
! ...  Take any needed action and clear the flag.
CALL IEEE_SET_FLAG(IEEE_INVALID, .FALSE.)
END IF
! ...  Restore the initial halting mode.
CALL IEEE_SET_HALTING_MODE(IEEE_INVALID, HALT)
...
```

Similar coding to that of the above fragment can be used for any of IEEE_INVALID, IEEE_OVERFLOW, IEEE_DIVIDE_BY_ZERO, IEEE_UNDERFLOW, or IEEE_INEXACT.

4 The Module IEEE_ARITHMETIC

See **F2008**, Table 14.2, and section 14.11.
The module IEEE_EXCEPTIONS is use-associated by IEEE_ARITHMETIC. This is done because the routine IEEE_GET_FLAG is called by some of the module subprograms that implement the procedures listed in **F2008**, 14.11.1 - 14.11.18. Consequently a user routine can use-associate just IEEE_ARITHMETIC to have access to both modules.

Several of these procedure names are generic and refer to specific module procedures, depending on the type, kind and rank of their arguments. The inquiry functions **F2008**, 14,11.24-14.11.27, and 14.11.29-14.11.35, are generic with respect to single and double pre-

cision scalar arguments. They are generic with respect to arrays of these precisions, up to and including rank 7.

An initial version of this module, and the test program IEEE_tests_gfortran, contained INTEGER variables that required the high order bit, number 31, to be set. For example the IEEE single precision value for $-\infty$ was initialized to the value 'FF800000'. But gfortran and the Nag compiler give a default compiler error. The issue is that the size of this integer value exceeds huge(1)=2**31-1. That compile error was avoided by initializing the value to '7F800000', and replacing the role of IEEE $-\infty$ as the intrinsic function value IBSET('7F800000',31). Similar code changes were made for other IEEE_CLASS types that have bit 31 set.

5 The Test Program IEEE_tests_gfortran

Testing anything approaching all ways to use the modules is practically impossible. But several routines are tested, in both single and double precision. The main program unit IEEE_tests_gfortran.f90 does a small amount of consistency checking for the following routines and overloaded comparisons. To test exceptions signaled by the x87 there are C functions `S87` and `D87` that compute the function $d = \sqrt{a/b + c}$, for inputs a, b, c. The operations use the x87. There is an additional x87 function `GETPI` that returns a value of π, with double precision accuracy. Printing of the status of results uses subroutine `messy`, [1].

1. IEEE_GET_STATUS
2. IEEE_SET_STATUS
3. IEEE_GET_HALTING_MODE
4. IEEE_SET_HALTING_MODE
5. IEEE_GET_FLAG
6. IEEE_SET_FLAG
7. IEEE_COPY_SIGN
8. IEEE_LOGB
9. IEEE_NEXT_AFTER
10. IEEE_REM
11. IEEE_RINT
12. IEEE_SCALB
13. IEEE_GET_X87_PRECISION_MODE
14. IEEE_SET_X87_PRECISION_MODE
15. Logical comparisons "==" and "/="

6 Acknowledgments

W. Van Snyder provided insight into aspects of the Fortran 2008 standard. F. T. Krogh kindly made his *gcc* and *gfortran* compilers available. He also provided use of his machine for testing. Additionally he improved the documentation with several suggestions. Tim Hopkins made changes to the codes that allowed the Nag Fortran compiler to be successfully used. Dominique d'Humieres provided a fix for a difference in printed output that resulted with the IEEE rounding mode set to round toward zero.

References

[1] F. T. Krogh. A Fortran Message Processor. *ACM Transactions on Mathematical Software*, V(N):X–X+5, mmm 201x.