J3/03-259

WORKING DRAFT
ISO IEC TECHNICAL REPORT 19767

ISO/IEC JTC1/SC22/WG5 PROJECT 22.02.01.05

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1:2004

21 October 2003

THIS PAGE TO BE REPLACED BY ISO-CS

Contents

Foreword

[General part to be provided by ISO CS]

This technical report specifies an extension to the module program unit facilities of the programming language Fortran. Fortran is specified by the international standard ISO/IEC 1539-1:2004. This document has been prepared by ISO/IEC JTC1/SC22/WG5, the technical working group for the Fortran language.

It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran standard (ISO/IEC 1539-1:2004) without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations.

0 Introduction

The module system of Fortran, as standardized by ISO/IEC 1539-1:2004, while adequate for programs of modest size, has shortcomings that become evident when used for large programs, or programs having large modules. The primary cause of these shortcomings is that modules are monolithic.

This technical report extends the module facility of Fortran so that program developers can optionally encapsulate the implementation details of module procedures in **submodules** that are separate from but dependent on the module in which the interfaces of their procedures are defined. If a module or submodule has submodules, it is the **parent** of those submodules.

The facility specified by this technical report is compatible to the module facility of Fortran as standardized by ISO/IEC 1539-1:2004.

0.1 Shortcomings of Fortran's module system

The shortcomings of the module system of Fortran, as specified by ISO/IEC 1539-1:2004, and solutions offered by this technical report, are as follows.

0.1.1 Decomposing large and interconnected facilities

If an intellectual concept is large and internally interconnected, it requires a large module to implement it. Decomposing such a concept into components of tractable size using modules as specified by ISO/IEC 1539-1:2004 may require one to convert private data to public data. The drawback of this is not primarily that an "unauthorized" procedure or module might access or change these entities, or develop a dependence on their internal details. Rather, during maintenance, one must then answer the question "where is this entity used?"

Using facilities specified in this technical report, such a concept can be decomposed into modules and submodules of tractable size, without exposing private entities to uncontrolled use.

Decomposing a complicated intellectual concept may furthermore require circularly dependent modules, but this is prohibited by ISO/IEC 1539-1:2004. It is frequently the case, however, that the dependence is between the implementation of some parts of the concept and the interface of other parts. Because the module facility defined by ISO/IEC 1539-1:2004 does not distinguish between the implementation and interface, this distinction cannot be exploited to break the circular dependence. Therefore, modules that implement large intellectual concepts tend to become large, and therefore expensive to maintain reliably.

Using facilities specified in this technical report, complicated concepts can be implemented in submodules that access modules, rather than modules that access modules, thus reducing the possibility for circular dependence between modules.

0.1.2 Avoiding recompilation cascades

Once the design of a program is stable, few changes to a module occur in its **interface**, that is, in its public data, public types, the interfaces of its public procedures, and private entities that affect their definitions. We refer to the rest of a module, that is, private entities that do not affect the definitions of public entities, and the bodies of its public procedures, as its **implementation**. Changes in the implementation have no effect on the translation of other program units that access the module. The existing module facility, however, draws no structural distinction between the interface and the implementation. Therefore, if one changes any part of a module, most language translation systems have no alternative but to conclude that a change might have occurred that could affect other modules that access the changed module. This effect cascades into modules that access modules that access the changed module, and so on. This can cause a substantial expense to retranslate and recertify a large program. Recertification can be several orders of magnitude more costly than retranslation.

Using facilities specified in this technical report, implementation details of a module can be encapsulated in submodules. Submodules are not accessible by use association, and they depend on their parent module, not vice-versa. Therefore, submodules can be changed without implying that a program unit accessing the parent module (directly or indirectly) must be retranslated.

It may also be appropriate to replace a set of modules by a set of submodules each of which has access to others of the set through the parent/child relationship instead of USE association. A change in one such submodule requires the retranslation only of its descendant submodules. Thus, compilation and certification cascades caused by changes can be shortened.

0.1.3 Packaging proprietary software

If a module as specified by international standard ISO/IEC 1539-1:2004 is used to package proprietary software, the source text of the module cannot be published as authoritative documentation of the interface of the module, without either exposing trade secrets, or requiring the expense of separating the implementation from the interface every time a revision is published.

Using facilities specified in this technical report, one can easily publish the source text of the module as authoritative documentation of its interface, while witholding publication of the source text of the submodules that contain the implementation details, and the trade secrets embodied within them.

0.1.4 Easier library creation

Most Fortran translator systems produce a single file of computer instructions and data, frequently called an *object file*, for each module. This is easier than producing an object file for the specification part and one for each module procedure. It is also convenient, and conserves space and time, when a program uses all or most of the procedures in each module. It is inconvenient, and results in a larger program, when only a few of the procedures in a general purpose module are needed in a particular program.

Modules can be decomposed using facilities specified in this technical report so that it is easier for each program unit's author to control how module procedures are allocated among object files. One can then collect sets of object files that correspond to a module and its submodules into a library.

0.2 Disadvantage of using this facility

Translator systems will find it more difficult to carry out global inter-procedural optimizations if the program uses the facility specified in this technical report. Interprocedural optimizations involving procedures in the same module or submodule will not be affected. When translator systems become able to do global inter-procedural optimization in the presence of this facility, it is likely that requesting inter-procedural optimization will cause compilation cascades in the first situation mentioned in subclause 0.1.2, even if this facility is used. Although one advantage of this facility could perhaps be reduced in the case when users request inter-procedural optimization, it would remain if users do not request inter-procedural optimization, and the other advantages remain in any case.

Information technology – Programming Languages – Fortran

Technical Report: Enhanced Module Facilities

1 General

1.1 Scope

This technical report specifies an extension to the module facilities of the programming language Fortran. The Fortran language is specified by international standard ISO/IEC 1539-1:2004 : Fortran. The extension allows program authors to develop the implementation details of concepts in new program units, called **submodules**, that cannot be accessed directly by use association. In order to support submodules, the module facility of international standard ISO/IEC 1539-1:2004 is changed by this technical report in such a way as to be upwardly compatible with the module facility specified by international standard ISO/IEC 1539-1:2004.

Clause 2 of this technical report contains a general and informal but precise description of the extended functionalities. Clause 3 contains detailed instructions for editorial changes to ISO/IEC 1539-1:2004.

1.2 Normative References

The following standards contain provisions that, through reference 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. Parties to agreements based on this technical report are, however, 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 referenced applies. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1:2004 : *Information technology – Programming Languages – Fortran; Part 1: Base Language*

2 Requirements

The following subclauses contain a general description of the extensions to the syntax and semantics of the Fortran programming language to provide facilities for submodules, and to separate subprograms into interface and implementation parts.

2.1 Summary

This technical report defines a new entity and modifications of two existing entities.

The new entity is a program unit, the *submodule*. As its name implies, a submodule is logically part of a module, and it depends on that module. A new variety of interface body, a *module procedure interface body*, and a new variety of procedure, a *separate module procedure*, are introduced.

By putting a module procedure interface body in a module and its corresponding separate module procedure in a submodule, program units that access the interface body by use association do not depend on the procedure's body. Rather, the procedure's body depends on its interface body.

2.2 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to a module or another submodule. A module or submodule may have several subsidiary submodules. If it has subsidiary submodules, it is the **parent** of those subsidiary submodules, and each of those submodules is a **child** of its parent. A submodule accesses its parent by host association.

An **ancestor** of a submodule is its parent, or an ancestor of its parent. A **descendant** of a module or submodule is one of its children, or a descendant of one of its children.

A submodule is introduced by a statement of the form `SUBMODULE (` *parent-identifier* `)` *submodule-name*, and terminated by a statement of the form `END SUBMODULE` *submodule-name*. The *parent-identifier* is either the name of the parent module or is of the form *ancestor-module-name* : *submodule-name*, where *submodule-name* is the name of a submodule that is a descendant of the module named *ancestor-module-name*.

Identifiers declared in a submodule are effectively PRIVATE, except for the names of separate module procedures that correspond to public module procedure interface bodies (2.3) in the ancestor module. It is not possible to access entities declared in the specification part of a submodule by use association because a USE statement is required to specify a module, not a submodule. ISO/IEC 1539-1:2004 permits PRIVATE and PUBLIC declarations only in a module, and this technical report does not propose to change this.

Submodule identifiers are global identifiers, but since they consist of a module name and a descendant submodule name, the name of a submodule can be the same as the name of another submodule so long as they do not have the same ancestor module.

In all other respects, a submodule is identical to a module.

2.3 Separate module procedure and its corresponding interface body

A **module procedure interface body** specifies the interface for a separate module procedure. It is different from an interface body defined by ISO/IEC 1539-1:2004 in three respects. First, it is introduced by a *function-stmt* or *subroutine-stmt* that includes MODULE in its *prefix*. Second, it specifies that its corresponding procedure body is in the module or submodule in which it appears, or one of its descendant submodules. Third, it accesses the module or submodule in which it is declared by host association.

A **separate module procedure** is a module procedure whose interface is declared in the same module or submodule, or is declared in one of its ancestors and is accessible from that ancestor by host association. The module subprogram that defines it may redeclare its characteristics, whether it is recursive, and its binding label. If any of these are redeclared, the characteristics, corresponding dummy argument names, whether it is recursive, and its binding label if any, shall be the same as in its module procedure interface body. The procedure is accessible by use association if and only if its interface body is accessible by use association. It is accessible by host association if and only if its interface body or procedure body is accessible by host association.

If the procedure is a function and its characteristics are not redeclared, the result variable name is determined by the FUNCTION statement in the module procedure interface body. Otherwise, the result variable name is determined by the FUNCTION statement in the module subprogram.

2.4 Examples of modules with submodules

The example module `POINTS` below declares a type `POINT` and a module procedure interface body for a module function `POINT_DIST`. Because the interface body includes the MODULE prefix, it accesses the scoping unit of the module by host association, without needing an IMPORT statement; indeed, an IMPORT statement is prohibited.

```
MODULE POINTS
  TYPE :: POINT
    REAL :: X, Y
  END TYPE POINT

  INTERFACE
    MODULE FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
      TYPE(POINT), INTENT(IN) :: A, B ! POINT is accessed by host association
      REAL :: DISTANCE
    END FUNCTION POINT_DIST
  END INTERFACE
END MODULE POINTS
```

The example submodule `POINTS_A` below is a submodule of the `POINTS` module. The type `POINT` and the interface `POINT_DIST` are accessible in the submodule by host association. The characteristics of the function `POINT_DIST` are redeclared in the module function body, and the dummy arguments have the same names. The function `POINT_DIST` is accessible by use association because its module procedure interface body is in the ancestor module and has the PUBLIC attribute.

```
SUBMODULE ( POINTS ) POINTS_A
  CONTAINS
    REAL MODULE FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
      TYPE(POINT), INTENT(IN) :: A, B
      DISTANCE = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
    END FUNCTION POINT_DIST
END SUBMODULE POINTS_A
```

An alternative declaration of the example submodule POINTS_A shows that it is not necessary to redeclare the properties of the module procedure POINT_DIST.

```
SUBMODULE ( POINTS ) POINTS_A
```

```
    CONTAINS
      MODULE PROCEDURE POINT_DIST
        DISTANCE = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
      END PROCEDURE POINT_DIST
  END SUBMODULE POINTS_A
```

3 Required editorial changes to ISO/IEC 1539-1:2004

The following editorial changes, if implemented, would provide the facilities described in foregoing clauses of this report. Descriptions of how and where to place the new material are enclosed between square brackets.

[After the third right-hand-side of syntax rule R202 insert:] 9:12+

or *submodule*

[After syntax rule R1104 add the following syntax rule. This is a quotation of the "real" syntax rule in subclause 11.2.2.] 9:34+

R1115a *submodule* **is** *submodule-stmt*
[*specification-part*]
[*module-subprogram-part*]
end-submodule-stmt

[In the second line of the first paragraph of subclause 2.2 insert ", a submodule" after "module".] 11:41

[In the fourth line of the first paragraph of subclause 2.2 insert a new sentence:] 11:43

A submodule is an extension of a module; it may contain the definitions of procedures declared in a module or another submodule.

[In the sixth line of the first paragraph of subclause 2.2 insert ", a submodule" after "module".] 11:45

[In the penultimate line of the first paragraph of subclause 2.2 insert "or submodule" after "module".] 11:47

[In the second sentence of 2.2.3.2, insert "or submodule" between "module" and "containing".] 12:28

[Insert a new subclause:] 13:17+

2.2.5 Submodule

A **submodule** is a program unit that extends a module or another submodule. It may provide definitions (12.5) for procedures whose interfaces are declared (12.3.2.1) in an ancestor module or submodule. It may also contain declarations and definitions of other entities, which are accessible in descendant submodules. An entity declared in a submodule is not accessible by use association unless it is a module procedure whose interface is declared in the ancestor module.

NOTE 2.2$\frac{1}{2}$

The scoping unit of a submodule accesses the scoping unit of its parent module or submodule by host association.

[In the second line of the first row of Table 2.1 insert ", SUBMODULE" after "MODULE".] 14

[Change the heading of the third column of Table 2.2 from "Module" to "Module or Submodule".] 14

[In the second footnote to Table 2.2 insert "or submodule" after "module" and change "the module" to "it".] 14

[In the last line of 2.3.3 insert ", *end-submodule-stmt*," after "*end-module-stmt*".] 15:2

[In the first line of the second paragraph of 2.4.3.1.1 insert ", submodule," after "module".] 17:4

[At the end of 3.3.1, immediately before 3.3.1.1, add "END PROCEDURE" and "END SUBMODULE" into the list of adjacent keywords where blanks are optional, in alphabetical order.] 28

[In the second line of the third paragraph of 4.5.1.1 after "definition" insert ", and its descendant submodules".] 46:10

[In the last line of Note 4.18, after "defined" add ", and its descendant submodules".] 46

[In the last line of the fourth paragraph of 4.5.3.6, after "definition", add "and its descendant submodules".] 55:10

[In the last line of Note 4.40, after "module" add ", and its descendant submodules".] 55

[In the last line of Note 4.41, after "definition" add "and its descendant submodules".] 56

[In the last line of the paragraph before Note 4.44, after "definition" add ", and its descendant submodules".] 58:8

[In the third and fourth lines of the second paragraph of 4.5.5.2 insert "or submodule" after "module" twice.] 59:23-24

[In the second paragraph of Note 4.48, insert "or submodule" after "module" twice.] 60

[In the first line of the second paragraph of 5.1.2.12 insert ", or any of its descendant submodules" after "attribute".] 84:3

[In the first and third lines of the second paragraph of 5.1.2.13 insert "or submodule" after "module" twice.] 84:14,16

[In the third line of the penultimate paragraph of 6.3.1.1 replace "or a subobject thereof" by "or submodule, or a subobject thereof,".] 113:18

[In the first two lines of the first paragraph after Note 6.23 insert "or submodule" after "module" twice.] 115:9-10

[In the second line of the first paragraph of Section 11 insert ", a submodule" after "module".] 249:3

[In the first line of the second paragraph of Section 11 insert ", submodules" after "modules".] 249:4

[Add another alternative to R1108] 250:17+

or *separate-module-subprogram*

[Within the first paragraph of 11.2.1, at its end, insert the following sentence:] 251:8

A submodule shall not reference its ancestor module by use association, either directly or indirectly.

[Then insert the following note:]

NOTE 11.6$\frac{1}{2}$

> It is possible for submodules with different ancestor modules to access each others' ancestor modules by use association.

[After constraint C1109 insert an additional constraint:] 251:30+

C1109a (R1109) If the USE statement appears within a submodule, *module-name* shall not be the name of the ancestor module of that submodule (11.2.2).

[Insert a new subclause immediately before 11.3:] 253:6-

11.2.2 Submodules

A **submodule** is a program unit that extends a module or another submodule. The program unit that it extends is its **parent** module or submodule; its parent is specified by the *parent-identifier* in the *submodule-stmt*. A submodule is a **child** of its parent. An **ancestor** of a submodule is its parent or an ancestor of its parent. A **descendant** of a module or submodule is one of its children or a descendant of one of its children.

NOTE 11.6$\frac{2}{3}$

> A submodule has exactly one ancestor module and may optionally have one or more ancestor submodules.

A submodule accesses the scoping unit of its parent module or submodule by host association.

A submodule may provide implementations for module procedures, each of which is declared by a module procedure interface body (12.3.2.1) within that submodule or one of its ancestors, and declarations and definitions of other entities that are accessible by host association in descendant submodules.

R1115a *submodule* **is** *submodule-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-submodule-stmt

R1115b *submodule-stmt* **is** SUBMODULE (*parent-identifier*) *submodule-name*

R1115c *parent-identifier* **is** *module-name*
  or *submodule-identifier*

R1115d *submodule-identifier* **is** *ancestor-module-name* : *submodule-name*

R1115e *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]

C1114a (R1115c) The *module-name* shall be the name of a nonintrinsic module.

C1114b (R1115d) The *ancestor-module-name* shall be the name of a nonintrinsic module; the *submodule-name* shall be the name of a descendant of that module.

C1114c (R1115a) An automatic object shall not appear in the *specification-part* of a submodule.

C1114d (R1115c) If a *submodule-name* is specified in the *end-submodule-stmt*, it shall be identical to the *submodule-name* specified in the *submodule-stmt*.

C1114e (R1115a) A submodule *specification-part* shall not contain a *format-stmt* or a *stmt-function-stmt*.

C1114f (R1115a) If an object of a type for which *component-initialization* is specified (R444) is declared in the *specification-part* of a submodule and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

[In the last line of the first paragraph of 12.3 after "units" add ", except that for a separate module procedure body (12.5.2.4), the dummy argument names, binding label, and whether it is recursive shall be the same as in its corresponding module procedure interface body (12.3.2.1)".] 257:13

[In C1210 insert "that is not a module procedure interface body" after "*interface-body*".] 259:20

[After the third paragraph after constraint C1211 insert the following paragraphs and constraints.] 259:30+

A **module procedure interface body** is an interface body in which the *prefix* of the initial *function-stmt* or *subroutine-stmt* includes MODULE. It declares the interface for a separate module procedure (12.5.2.4). A separate module procedure is accessible by use association if and only if its interface body is declared in the specification part of a module and its name has the PUBLIC attribute. If its separate module procedure body is not defined, the interface may be used to specify an explicit specific interface but the procedure shall not be used in any way. *Meaning what?*

A **module procedure interface** is declared by a module procedure interface body.

C1211a (R1205) A scoping unit in which a module procedure interface body is declared shall be a module or submodule.

C1212b (R1205) A module procedure interface body shall not appear in an abstract interface block.

[Add a right-hand-side to R1228:] 280:3+

or MODULE

[Add constraints after C1242:] 280:7+

C1242a (R1227) MODULE shall appear only within the initial *function-stmt* or *subroutine-stmt* of an interface body that is declared in the scoping unit of a module or submodule, or of a module subprogram.

C1242b (R1227) If MODULE appears within the *prefix* in a module subprogram, a module procedure interface having the same name as the subprogram shall be declared in the module or submodule in which the subprogram is defined, or in an ancestor of that program unit and be accessible by host association from that ancestor.

C1242c (R1227) If MODULE appears within the *prefix* in a module subprogram, the subprogram shall specify the same names, type, kind type parameters and rank for corresponding dummy arguments, and the same binding label if any, as in its corresponding module procedure interface body.

C1242c (R1227) If MODULE appears within the *prefix* in a module subprogram, RECURSIVE shall appear if and only if RECURSIVE appears in the *prefix* in the corresponding module procedure interface body.

C1242e (R1227) If MODULE appears within the *prefix* in a module function subprogram, the subprogram shall specify the same type, kind type parameters and rank for the result variable as in its corresponding module procedure interface body.

[Insert the following new subclause before the existing subclause 12.5.2.4 and renumber succeeding subclauses appropriately:] 283:1-

12.5.2.4 Separate module procedures

A **separate module procedure** is a module procedure defined by a *separate-module-subprogram*, by a *function-subprogram* in which the *prefix* of the initial *function-stmt* includes MODULE, or by a *subroutine-subprogram* in which the prefix of the initial *subroutine-stmt* includes MODULE. Its interface is declared by a module procedure interface body (12.3.2.1) in the *specification-part* of the module or submodule in which the procedure is defined, or in an ancestor module or submodule.

R1234a *separate-module-subprogram* **is** MODULE PROCEDURE *procedure-name*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-sep-subprogram-stmt

R1234b *end-sep-subprogram-stmt* **is** END [PROCEDURE [*procedure-name*]]

C1251a (R1234a) The *procedure-name* shall be the same as the name of a module procedure interface that is declared in the module or submodule in which the *separate-module-subprogram* is defined, or in an ancestor of that program unit and be accessible by host association from that ancestor.

C1251b (R1234b) If a *procedure-name* appears in the *end-sep-subprogram-stmt*, it shall be identical to the *procedure-name* in the MODULE PROCEDURE statement.

If the procedure is a function and its characteristics are not redeclared, the result variable name is determined by the FUNCTION statement in the module procedure interface body. Otherwise, the result variable name is determined by the FUNCTION statement in the module subprogram.

A separate module procedure and a module procedure interface body **correspond** if they have the same name, and the module procedure interface is declared in the same program unit as the separate module procedure or is declared in an ancestor of the program unit in which the separate module procedure is defined and is accessible by host association from that ancestor.

NOTE 12.40$\frac{1}{2}$

A separate module procedure can be accessed by use association if and only if its interface body is declared in the specification part of a module and its name has the PUBLIC attribute. A separate module procedure that is not accessible by use association might still be accessible by way of a procedure pointer, a dummy procedure, or a type-bound procedure.

If a separate module procedure is defined by a subroutine subprogram or a function subprogram, its characteristics as a procedure (12.2), its dummy argument names, and its binding label if any shall be identical to those specified by its corresponding module procedure interface body. The subroutine or function subprogram shall be specified to be recursive if and only if RECURSIVE appears in the *prefix* of the initial *subroutine-stmt* or *function-stmt* of its corresponding module procedure interface body.

[In constraint C1253 replace "*module-subprogram*" by "a *module-subprogram* that does not define a separate module procedure".] 283:7

[In the first line of the first paragraph after syntax rule R1237 in 12.5.2.6 insert ", submodule" after "module",] 284:37

[In the second sentence of the first paragraph of 16.1, insert "non-submodule" before "program unit.] 405:19,22

[After the second sentence of the first paragraph of 16.1, insert a new sentence "The submodule identifier is a global identifier and shall not be the same as any other submodule identifier.] 405:22

NOTE 16.2$\frac{1}{2}$ 406:1-

> Submodule identifiers are global identifiers, but since they consist of a module name and a descendant submodule name, the name of a submodule can be the same as the name of another submodule so long as they do not have the same ancestor module.

[In item (1) in the first numbered list in 16.2, after "abstract interfaces" insert ", module procedure interfaces".] 406:6

[In the paragraph immediately before Note 16.3, after "(4.5.9)" insert ", and a separate module procedure shall have the same name as its corresponding module procedure interface body".] 406:20

[In the first line of the first paragraph of 16.4.1.3 insert ", a module procedure interface body" after "module subprogram". In the second line, insert "that is not a module procedure interface body" after "interface body".] 411:2,3

[In the third line of the first paragraph of 16.4.1.3, after the second instance of "interface body", insert a new sentence: "A submodule has access to the named entities of its parent by host association."] 411:4

[In the third line after the sixteen-item list in 16.4.1.3 insert "that does not define a separate module procedure" after "subprogram".] 411:33

[In the first line of Note 16.9, after "interface body" insert "that is not a module procedure interface body".] 412:1+2

[Insert a new item after item (5)(d) in the list in 16.4.2.1.3:] 415:15+

(d$\frac{1}{2}$) Is in the scoping unit of a submodule if any scoping unit in that submodule or any of its descendant submodules is in execution.

[In item (3)(c) of 16.5.6 insert "or submodule" after "module" twice.] 422:14-15

[Replace Note 16.18 by the following.] 422

NOTE 16.18

> A module subprogram inherently references the module or submodule that is its host. Therefore, for processors that keep track of when modules or submodules are in use, one is in use whenever any procedure in it or any of its descendant submodules is active, even if no other active scoping units reference its ancestor module; this situation can arise if a module procedure is invoked via a procedure pointer, a type-bound procedure, or by means other than Fortran.

[In item (3)(d) of 16.5.6 insert "or submodule" after "module" twice.] 422:16-17

[Insert the following definitions into the glossary in alphabetical order:]

ancestor (11.2.2) : Of a submodule, its parent or an ancestor of its parent. 425:15+

child (11.2.2) : A submodule is a child of its parent. 426:43+

descendant (11.2.2) : Of a module or submodule, one of its children or a descendant of one of its children. 428:28+

module procedure interface (12.3.2.1) : An interface defined by an interface body in which MODULE appears in the *prefix* of the initial *function-stmt* or *subroutine-stmt*. It declares the interface for a separate module procedure. 432:9+

parent (11.2.2) : Of a submodule, the module or submodule specified by the *parent-identifier* in its *submodule-stmt*. 432:36+

separate module procedure (12.5.2.4) : A module procedure defined by a subprogram in which MODULE appears in the *prefix* of the initial *function-stmt* or *subroutine-stmt*. 434:26+

submodule (2.2.5, 11.2.2) : A program unit that depends on a module or another submodule; it extends the program unit on which it depends. 435:15+

[Insert a new subclause immediately before C.9:] 477:29+

C.8.3.9 Modules with submodules

Each submodule specifies that it is the child of exactly one parent module or submodule. Therefore, a module and all of its descendant submodules stand in a tree-like relationship one to another.

If a module procedure interface body that is specified in a module has public accessibility, and its corresponding separate module procedure is defined in a descendant of that module, the procedure can be accessed by use association. No other entity in a submodule can be accessed by use association. Each program unit that accesses a module by use association depends on it, and each submodule depends on its ancestor module. Therefore, if one changes a separate module procedure body in a submodule but does not change its corresponding module procedure interface, a tool for automatic program translation would not need to reprocess program units that access the module by use association. This is so even if the tool exploits the relative modification times of files as opposed to comparing the result of translating the module to the result of a previous translation.

This is not the end of the story. By constructing taller trees, one can put entities at intermediate levels that are shared by submodules at lower levels, and have no possibility of affecting anything that is accessible from the module by use association. Developers of modules that embody large complicated concepts can exploit this possibility to organize components of the concept into submodules, while preserving the privacy of entities that are shared by the submodules and that ought not to be exposed to users of the module. Putting these shared entities at an intermediate level also prevents cascades of reprocessing and recertification if some of them are changed.

The following example illustrates a module, `color_points`, with a submodule, `color_points_a`, that in turn has a submodule, `color_points_b`. Public entities declared within `color_points` can be accessed by use association. The submodules `color_points_a` and `color_points_b` can be changed without causing the appearance that the module `color_points` might have changed.

The module `color_points` does not have a *contains-part*, but a *contains-part* is not prohibited. The module could be published as definitive specification of the interface, without revealing trade secrets contained within `color_points_a` or `color_points_b`. Of course, a similar module without the `module` prefix in the interface bodies would serve equally well as documentation – but the procedures would be external procedures. It would make little difference to the consumer, but the developer would forfeit all of the advantages of modules.

```
module color_points

  type color_point
    private
    real :: x, y
    integer :: color
  end type color_point

  interface              ! Interfaces for procedures with separate
                         ! bodies in the submodule color_points_a
    module subroutine color_point_del ( p ) ! Destroy a color_point object
      type(color_point), allocatable :: p
    end subroutine color_point_del
    ! Distance between two color_point objects
    real module function color_point_dist ( a, b )
      type(color_point), intent(in) :: a, b
    end function color_point_dist
    module subroutine color_point_draw ( p ) ! Draw a color_point object
      type(color_point), intent(in) :: p
    end subroutine color_point_draw
    module subroutine color_point_new ( p ) ! Create a color_point object
      type(color_point), allocatable :: p
    end subroutine color_point_new
  end interface

end module color_points
```

The only entities within `color_points_a` that can be accessed by use association are separate module procedures for which corresponding module procedure interface bodies are provided in `color_points`. If the procedures are changed but their interfaces are not, the interface from program units that access them by use association is unchanged. If the module and submodule are in separate files, utilities that examine the time of modification of a file would notice that changes in the module could affect the translation of its submodules or of program units that access the module by use association, but that changes in submodules could not affect the translation of the parent module or program units that access it by use association.

The variable `instance_count` is not accessible by use association of `color_points`, but is accessible within `color_points_a`, and its submodules.

```
submodule ( color_points ) color_points_a ! Submodule of color_points

  integer, save :: instance_count = 0

  interface                   ! Interface for a procedure with a separate
                              ! body in submodule color_points_b
    module subroutine inquire_palette ( pt, pal )
      use palette_stuff       ! palette_stuff, especially submodules
                              ! thereof, can access color_points by use
                              ! association without causing a circular
                              ! dependence because this use is not in the
                              ! module.  Furthermore, changes in the module
                              ! palette_stuff are not accessible by use
                              ! association of color_points
```

```
      type(color_point), intent(in) :: pt
      type(palette), intent(out) :: pal
    end subroutine inquire_palette

  end interface

contains ! Invisible bodies for public module procedure interfaces
         ! declared in the module

  module subroutine color_point_del ( p )
    type(color_point), allocatable :: p
    instance_count = instance_count - 1
    deallocate ( p )
  end subroutine color_point_del
  real module function color_point_dist ( a, b ) result ( dist )
    type(color_point), intent(in) :: a, b
    dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
  end function color_point_dist
  module subroutine color_point_new ( p )
    type(color_point), allocatable :: p
    instance_count = instance_count + 1
    allocate ( p )
  end subroutine color_point_new

end submodule color_points_a
```

The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared therein. It is not, however, accessible by use association, because its interface is not declared in the module, `color_points`. Since the interface is not declared in the module, changes in the interface cannot affect the translation of program units that access the module by use association.

```
submodule ( color_points_a ) color_points_b ! Subsidiary**2 submodule

contains
  ! Invisible body for interface declared in the ancestor module
  module subroutine color_point_draw ( p )
    use palette_stuff, only: palette
    type(color_point), intent(in) :: p
    type(palette) :: MyPalette
    ...; call inquire_palette ( p, MyPalette ); ...
  end subroutine color_point_draw

  ! Invisible body for interface declared in the parent submodule
  module procedure inquire_palette
    ... implementation of inquire_palette
  end procedure inquire_palette

  subroutine private_stuff ! not accessible from color_points_a
    ...
  end subroutine private_stuff

end submodule color_points_b
```

```
module palette_stuff
  type :: palette ; ... ; end type palette
contains
  subroutine test_palette ( p )
  ! Draw a color wheel using procedures from the color_points module
    type(palette), intent(in) :: p
    use color_points ! This does not cause a circular dependency because
                     ! the "use palette_stuff" that is logically within
                     ! color_points is in the color_points_a submodule.
    ...
  end subroutine test_palette
end module palette_stuff
```

There is a `use palette_stuff` in `color_points_a`, and a `use color_points` in `palette_stuff`. The `use palette_stuff` would cause a circular reference if it appeared in `color_points`. In this case, it does not cause a circular dependence because it is in a submodule. Submodules are not accessible by use association, and therefore what would be a circular appearance of `use palette_stuff` is not accessed.

```
program main
  use color_points
  ! "instance_count" and "inquire_palette" are not accessible here
  ! because they are not declared in the "color_points" module.
  ! "color_points_a" and "color_points_b" cannot be accessed by
  ! use association.
  interface draw                    ! just to demonstrate it's possible
    module procedure color_point_draw
  end interface
  type(color_point) :: C_1, C_2
  real :: RC
  ...
  call color_point_new (c_1)        ! body in color_points_a, interface in color_points
  ...
  call draw (c_1)                   ! body in color_points_b, specific interface
                                    ! in color_points, generic interface here.
  ...
  rc = color_point_dist (c_1, c_2)  ! body in color_points_a, interface in color_points
  ...
  call color_point_del (c_1)        ! body in color_points_a, interface in color_points
  ...
end program main
```

A multilevel submodule system can be used to package and organize a large and interconnected concept without exposing entities of one subsystem to other subsystems.

Consider a `Plasma` module from a Tokomak simulator. A plasma simulation requires attention at least to fluid flow, thermodynamics, and electromagnetism. Fluid flow simulation requires simulation of subsonic, supersonic, and hypersonic flow. This problem decomposition can be reflected in the submodule structure of the `Plasma` module:

```
                          Plasma module
                                |
           |---------------------|---------------------|
           |                     |                     |
     Flow submodule      Thermal submodule     Electromagnetics
```

```
                          |                                             Submodule
        |-------------------|-------------------|
        |                   |                   |
    Subsonic            Supersonic          Hypersonic
```

Entities can be shared among the `Subsonic, Supersonic`, and `Hypersonic` submodules by putting them within the `Flow` submodule. One then need not worry about accidental use of these entities by use association or by the `Thermal` or `Electromagnetics` modules, or the development of a dependency of correct operation of those subsystems upon the representation of entities of the `Flow` subsystem as a consequence of maintenance. Since these these entities are not accessible by use association, if any of them are changed, it cannot affect program units that access the `Plasma` module by use association, and the answer to the question "where are these entities used" is confined to the set of descendant submodules of the `Flow` submodule.