WG5/N1537

WORKING DRAFT
ISO IEC TECHNICAL REPORT 19767

ISO/IEC JTC1/SC22/WG5 PROJECT
1.22.02.01.01.01

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1

15 July 2003

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. 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) 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, 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.

0.1 Shortcomings of Fortran's module system

The shortcomings of the module system of Fortran, as specified by ISO/IEC 1539-1, 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 may require one to convert private data to public data.

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. 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 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 the interface of one such submodule requires the retranslation only of its descendant submodules. Thus, compilation cascades caused by changes of interface can be shortened.

0.1.3 Packaging proprietary software

If a module as specified by international standard ISO/IEC 1539-1 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, 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 is easier for each program unit's author to control how module procedures are allocated among object files.

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 section 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 current Fortran language is specified by the international standard ISO/IEC 1539-1 : 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 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.

Clause 2 of this technical report contains a general and informal but precise description of the extended functionalities. Clause 3 contains detailed editorial changes that would implement the revised language specification if they were applied to the current international standard.

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 : *Information technology - Programming Languages - Fortran*

2 Requirements

The following subclauses contain a general description of the extensions to the syntax and semantics of the current 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 *separate interface body*, and a new variety of procedure, a *separate module procedure*, are described below.

By putting a separate 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 that submodule, or an ancestor of its parent. A **descendant** of a module or submodule is that program unit, or a descendant of a child of that program unit.

A submodule is introduced by a statement of the form `SUBMODULE (` *parent-name* `)` *submodule-name*, and terminated by a statement of the form `END SUBMODULE` *submodule-name*. The *parent-name* is the name of the parent module or submodule.

Identifiers declared in a submodule are effectively PRIVATE, except for the names of separate module procedures that correspond to public separate 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 permits PRIVATE and PUBLIC declarations only in a module, and this technical report does not propose to change that specification.

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

2.3 Separate module procedure and its corresponding separate interface body

2.3.1 As of J3 meeting 164

A **forward interface body** is different from an interface body defined by ISO/IEC 1539-1 in three respects. First, it is declared in an interface block that is introduced by a FORWARD INTERFACE statement. Second, in addition to specifying a procedure's characteristics and dummy argument names, a forward interface body specifies that its corresponding procedure body is in a descendant of the module or submodule in which it appears. Third, unlike an ordinary interface body, it accesses the module or submodule in which it is declared by host association.

If a module procedure is enclosed between IMPLEMENTATION and END IMPLEMENTATION statements, it is a **separate module procedure**. It shall have the same name as a forward interface body that is declared in a module or submodule that is an ancestor of the one in which the procedure is defined. Its characteristics and dummy argument names are declared by its corresponding interface body. The procedure is accessible if and only if its interface body is accessible.

The characteristics and dummy argument names may be redeclared in the module subprogram that defines the separate module procedure. If the characteristics and dummy argument names are redeclared, they shall be the same as in the interface body, except that the procedure's body may specify that the procedure is pure even if the interface body does not.

If the procedure is a function, the result variable name is determined by the declaration of the module subprogram, not by the forward interface body. If the forward interface body declares a result variable name different from the function name, that declaration is ignored, except for its use in specifying the result variable characteristics.

2.3.2 Revised for 2003 WG5 meeting

A **separate interface body** is different from an interface body defined by ISO/IEC 1539-1 in three respects. First, it is introduced by a *function-stmt* or *subroutine-stmt* that includes SEPARATE in its *prefix*. Second, in addition to specifying a procedure's characteristics and dummy argument names, a separate interface body specifies that its corresponding procedure body is in a descendant of the module or submodule in which it appears. Third, unlike an ordinary interface body, it accesses the module or submodule in which it is declared by host association.

If a module procedure is introduced by a *function-stmt* or *subroutine-stmt* that includes SEPARATE in its *prefix* it is a **separate module procedure**. It shall have the same name as a separate interface body that is declared in a module or submodule that is an ancestor of the one in which the procedure is defined. Its characteristics and dummy argument names are declared by its corresponding interface body. The procedure is accessible if and only if its interface body is accessible.

The characteristics and dummy argument names may be redeclared in the module subprogram that defines the separate module procedure. If the characteristics and dummy argument names are redeclared, they shall be the same as in the interface body, except that the procedure's body may specify that the procedure is pure even if the interface body does not.

If the procedure is a function, the result variable name is determined by the declaration of the module subprogram, not by the separate interface body. If the separate interface body declares a result variable name different from the function name, that declaration is ignored, except for its use in specifying the result variable characteristics.

2.4 Examples of modules with submodules

2.4.1 As of J3 meeting 164

The example module `POINTS` below declares a type `POINT` and a forward interface body for a module function `POINT_DIST`. Because the interface block includes the `FORWARD` prefix, the interface body within it accesses the scoping unit of the module by host association, without needing an `IMPORT` statement. The declaration of the result variable name `DISTANCE` serves only as a vehicle to declare the result characteristics; the name is otherwise ignored.

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

```

```
    FORWARD INTERFACE
      FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
        TYPE(POINT), INTENT(IN) :: A, B ! 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 name POINT is accessible in the submodule by host association. The characteristics of the function POINT_DIST can be redeclared in the module function body, or taken from the forward interface body in the POINTS module. The fact that POINT_DIST is accessible by use association results from the fact that there is a forward interface body of the same name in the ancestor module.

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

An alternative declaration of the example submodule POINTS_A shows that it is not necessary to redeclare the characteristics of the module procedure POINT_DIST. The result variable name is POINT_DIST, even though the forward interface body specifies a different result variable name. This is because any declarations in an interface body other than the characteristics of the procedure it declares are ignored; this technical report does not propose to change that specification.

```
  SUBMODULE ( POINTS ) POINTS_A
    CONTAINS
      IMPLEMENTATION POINT_DIST
        FUNCTION POINT_DIST
          POINT_DIST = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
        END FUNCTION POINT_DIST
      END IMPLEMENTATION POINT_DIST
  END SUBMODULE POINTS_A
```

2.4.2 Revised for 2003 WG5 meeting

The example module POINTS below declares a type POINT and a separate interface body for a module function POINT_DIST. Because the interface body includes the SEPARATE prefix, it accesses the scoping unit of the module by host association, without needing an IMPORT statement. The declaration of the result variable name DISTANCE serves only as a vehicle to declare the result characteristics; the name is otherwise ignored.

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

```
    INTERFACE
      SEPARATE FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
        TYPE(POINT), INTENT(IN) :: A, B ! 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 name `POINT` is accessible in the submodule by host association. The characteristics of the function `POINT_DIST` can be redeclared in the module function body, or taken from the separate interface body in the `POINTS` module. The fact that `POINT_DIST` is accessible by use association results from the fact that there is a separate interface body of the same name in the ancestor module.

```
  SUBMODULE ( POINTS ) POINTS_A
    CONTAINS
      SEPARATE REAL FUNCTION POINT_DIST ( P, Q ) RESULT ( HOW_FAR )
        TYPE(POINT), INTENT(IN) :: P, Q
        HOW_FAR = 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 characteristics of the module procedure `POINT_DIST`. The result variable name is `POINT_DIST`, even though the separate interface body specifies a different result variable name. This is because any declarations in an interface body other than the characteristics of the procedure it declares are ignored; this technical report does not propose to change that specification.

```
  SUBMODULE ( POINTS ) POINTS_A
    CONTAINS
      SEPARATE FUNCTION POINT_DIST
        POINT_DIST = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
      END FUNCTION POINT_DIST
  END SUBMODULE POINTS_A
```

2.5 Relation between modules and submodules

Public entities of a module, including separate interface bodies, can be accessed by use association. The only entities of submodules that are accessible by use association are separate module procedures for which there is a corresponding publicly accessible separate interface body.

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

3 Required editorial changes to ISO/IEC 1539-1

There are two sets of edits provided here. The first is for the proposal as of J3 meeting 163. The second is a revision based on earlier guidance to make the proposal as simple as possible.

3.1 As of J3 meeting 163

The changes described here refer to the 03-007 draft.

The following editorial changes, if implemented, would provide the facilities described in foregoing sections 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 11.2.3.] 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

[Replace the second sentence of 2.2.3.2 by the following sentence.] 12:27-29

A module procedure may be invoked from within any scoping unit that accesses its declaration (12.3.2.1) or definition (12.5) by use association or host association.

[Insert the following note at the end of 2.2.3.2.] 12:30+

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.

[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 entities that are accessible to 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.

[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 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".] 44:27

[In the last line of Note 4.19, after "defined" add ", and its descendant submodules".] 45

[In the last line of the fourth paragraph of 4.5.3.6, after "definition", add "and its descendant submodules".] 54:6

[In the last line of Note 4.41, after "module" add ", and its descendant submodules".] 54

[In the last line of Note 4.42, after "definition" add "and its descendant submodules".] 54

[In the last line of the paragraph before Note 4.45, after "definition" add ", and its descendant submodules".] 57:3

[In the third and fourth lines of the second paragraph of 4.5.5.2 insert "or submodule" after "module" twice.] 58:11-12

[In the second paragraph of Note 4.49, insert "or submodule" after "module" twice.] 58

[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.] 84:12,14

[After the second paragraph after constraint C581 insert the following note.] 93:4+

NOTE 5.33$\frac{1}{2}$

> I have no idea what I had in mind for this note. 02-277, 02-277r1 and 03-123 all had instructions to insert a note, but no note. Maybe it was about the possibility that a separate interface and a separate procedure might have different IMPLICIT rules in effect.

[In the third line of the penultimate paragraph of 6.3.1.1 replace "or a subobject thereof" by "or sub- 113:22

module, or a subobject thereof,".]

[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".] 251:3

[In the first line of the second paragraph of Section 11 insert ", submodules" after "modules".] 251:4

[After the second right-hand side for R1108 add:] 252:17+

or *implementation*

[Within the first paragraph of 11.2.1, at its end, insert the following sentence:] 253:8

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.

[After constraint C1109 insert an additional constraint:] 253:30+

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

[Insert a new subclause immediately before 11.3:] 255:1-

11.2.3 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-name* in the *submodule-stmt*. A submodule is a **child** of its parent. An **ancestor** of a module or 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 accesses the scoping unit of its parent module or submodule by host association.

A submodule may provide implementations for module procedures that are declared by separate interface bodies within ancestor program units, 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-name*) *submodule-name*

R1115c *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]

C1114a (R1115a) The *parent-name* shall be the name of a submodule or a nonintrinsic module.

C1114b (R1115a) An automatic object shall not appear in the *specification-part* of a submodule.

C1114c (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*.

c1114d R1115a) A submodule *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt* or a *format-stmt*.

c1114e R1115a) If an object of a type for which *component-initialization* is specified (R438 appears 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 third line of the first paragraph of 12.3 replace ", but" by ". If the dummy arguments are redeclared in a separate module procedure body (12.5.2.5) they shall have the same names as in the corresponding interface body (12.3.2.1); otherwise".] 259:12

[Replace the first line of syntax rule R1203 with the following:] 260:11

R1203 *interface-stmt* **is** [FORWARD] INTERFACE [*generic-spec*]

[Add a new constraint after C1204:] 261:9+

C1204a (R1203) FORWARD shall not appear except in the *specification-part* of a module or submodule.

[Add a new constraint after C1209:] 261:19+

C1209a (R1206) A *procedure-stmt* shall not appear in an interface block that is introduced by a FORWARD INTERFACE statement.

[Add a new constraint after constraint C1211:] 261:21+

C1211a (R1209) An IMPORT statement shall not appear within an interface body that is declared within an interface block that is introduced by a FORWARD INTERFACE statement.

[After the third paragraph after constraint C1211 insert the following paragraph and note.] 261:30+

A **forward interface body** is an interface body that appears in an interface block introduced by a FORWARD INTERFACE statement. It declares the interface for a separate module procedure (12.5.2.5). 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 has the PUBLIC attribute. If the definition of its procedure body does not appear within the *module-subprogram-part* of the program unit in which the module interface body is declared, or one of its descendant submodules (11.2.3), the interface may be used but the procedure shall not be used in any way.

A **forward interface** is declared by a forward interface body.

NOTE 12.3$\frac{1}{2}$

A forward interface body shall not appear except within an interface block within the *specification-part* of a module or submodule.

[In the first sentence of the fourth paragraph after constraint C1211 insert ", that is not a forward interface body," after "block".] 261:31

[Insert a new subclause before 12.5.2.4 and renumber succeeding subclauses appropriately.] 285:1-

12.5.2.4 Separate module procedures

A **separate module procedure** is a module procedure for which the interface is declared by a forward interface body (12.3.2.1) in the *specification-part* of a module or submodule and the procedure body is defined by an *implementation* in a descendant of the program unit in which the interface body is declared.

NOTE $\mathbf{12.40\frac{1}{3}}$

> A separate module procedure can be accessed by use association if and only if its interface body can be accessed by use association. 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.

A module subprogram that defines a separate module procedure may respecify the characteristics declared in its interface body. If they are respecified, they shall be identical to those specified in its interface body, except that the module procedure may be specified to be pure even if the interface body does not so specify, in which case the procedure is pure.

NOTE $\mathbf{12.40\frac{2}{3}}$

> As with an external procedure, if a separate module procedure is declared to be pure, it shall satisfy all the requirements for pure procedures. If the interface is not declared to be pure, the invocations using that interface cannot take advantage of the properties of purity.

> The exception for pure procedures was consciously modeled on [261:40-41]. Module procedures are, however, different from external procedures. Do we want this exception for separate procedures? There is no requirement that dummy arguments have the same names in the separate procedure body as in the corresponding separate interface body. This too is modeled on external procedures. Do we want to require the dummy argument names to be the same?

WG5 questions

R1233a	*implementation*	**is**	*implementation-stmt*
			[*implementation-body*]
			end-implementation-stmt

R1233b *implementation-stmt* **is** IMPLEMENTATION *subprogram-name*

C1252b (R1233b) The *subprogram-name* shall be identical to the name of a forward interface that is declared in an ancestor module or submodule of the scoping unit in which the *implementation* appears.

R1233c *end-implementation-stmt* **is** END [IMPLEMENTATION [*subprogram-name*]]

C1107a (R1233c) If a *subprogram-name* appears in the *end-implementation-stmt*, it shall be identical to the *subprogram-name* specified in the *implementation-stmt*.

R1233d	*implementation-body*	**is**	*function-impl*
		or	*subroutine-impl*

R1233e *function-impl* **is** *function-subprogram*

or *subprogram-body*

R1233f *subprogram-body* **is** [*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]

C1252c (R1233e) If *function-impl* is *function-subprogram* the *function-name* shall be identical to the *subprogram-name* specified in the *implementation-stmt*.

C1252d (R1233e) If *function-impl* is *function-subprogram* interface declared by *function-impl* shall be identical to the interface declared by the interface body for the *subprogram-name*, except that it may specify PURE even if the interface declared by the interface body does not.

R1233g *subroutine-impl* **is** *subroutine-subprogram*
or *subprogram-body*

C1252g (R1233g) If *subroutine-impl* is *subroutine-subprogram* the *subroutine-name* shall be identical to the *subprogram-name* specified in the *implementation-stmt*.

C1252h (R1233g) If *subroutine-impl* is *subroutine-subprogram* the interface declared by *subroutine-impl* shall be identical to the interface declared by the interface body for the *subprogram-name*, except that it may specify PURE even if the interface declared by the interface body does not.

C1258a (R1234) An *entry-stmt* shall not appear in an *implementation-body*.

[In the first line of the first paragraph after syntax rule R1236 in 12.5.2.6 insert ", submodule" after "module",] 286:37

[In item (1) in the first numbered list in 16.2, after "abstract interfaces" insert ", forward interfaces".] 408:6

[After "(4.5.9)" insert ", and a separate module procedure shall have the same name as its corresponding separate interface body".] 408:16

[In the first line of the first paragraph of 16.4.1.3 insert ", a forward interface body" after "module subprogram". In the second line, insert "that is not a forward interface body" after "interface body".] 412:30

[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".] 413:26

[Insert a new item after item (5)(d) in the list in 16.4.2.1.3:] 417:6+

(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 the second line of item 2 of 16.5.6 replace "or in a" by ", submodule,".] 423:48

[In item (3)(c) of the list in 16.5.6 insert "and its descendant submodules" after the first "module" and insert "or any of its descendant submodules" after the second "module".] 424:8-9

[Replace Note 16.18 by the following.] 424

NOTE 16.18

A module subprogram inherently references the module or submodule that is its host. Therefore,

NOTE 16.18 (cont.)

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 or by means other than Fortran.

[In item 3d of 16.5.6 insert "or submodule" after the first "module" and replace the second "module" by "that scoping unit". 424:10-11

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

ancestor (11.2.3) : A module, a submodule, or an ancestor of the parent of that submodule. 427:15+

child (11.2.3) : A submodule, when considered in its relation to the module or submodule upon which it depends. 428:43+

descendant (11.2.3) : A module or submodule, or a descendant of a child of that module or submodule. 430:28+

forward interface (12.3.2.1) : An interface defined by an interface body in an interface block introduced by a FORWARD INTERFACE statement. It declares the interface for a module procedure that has a separately-defined body. 432:9+

parent (11.2.3) : A module or submodule, when considered in its relation to the submodules that depend upon it. 434:36+

submodule (2.2.5, 11.2.3) : A program unit that depends on a module or another submodule; it extends the program unit on which it depends. 437:15+

[Insert a new subclause immediately before C.9:] 479:33+

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 forward interface body that is specified in a module has public accessibility, and its corresponding implementation 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, one can change an implementation in a submodule without any possibility of changing the interface of the procedure. If a tool for automatic program translation is used, and even if it exploits the relative modification times of files as opposed to comparing the result of translating the module to the result of a previous translation, modifying a submodule cannot result in the tool deciding to reprocess program units that access the module by use association.

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 to affect 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 ought not to be exposed to users of the module and preventing cascades of reprocessing.

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. Except for the characteristics and dummy argument names of implementations that have forward interface bodies that are accessible 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 forward prefix in the interface bodies would serve equally well as documentation – but the procedures would be external procedures. It wouldn't make any 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

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

```
    forward interface           ! Interface for a procedure with a separate
                                ! body in submodule color_points_b
     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 forward interfaces declared
           ! in the module

    implementation color_point_del ! ( p )
      instance_count = instance_count - 1
      deallocate ( p )
    end implementation color_point_del
    implementation color_point_dist
      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
    end color_point_dist
    implementation color_point_new ! ( p )
      instance_count = instance_count + 1
      allocate ( p )
    end implementation 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 parent submodule
    implementation color_point_draw ! ( p )
    ! Its interface is defined in an ancestor.
      type(palette) :: MyPalette
      ...; call inquire_palette ( p, MyPalette ); ...
    end implementation color_point_draw

    implementation inquire_palette
      ... implementation of inquire_palette
```

```
    end implementation 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
```

Multilevel submodule systems 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.

3.2 Revised for 2003 WG5 meeting

The changes described here refer to the 03-007 draft.

The following editorial changes, if implemented, would provide the facilities described in foregoing sections 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 11.2.3.] 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

[Replace the second sentence of 2.2.3.2 by the following sentence.] 12:27-29

A module procedure may be invoked from within any scoping unit that accesses its declaration (12.3.2.1) or definition (12.5) by use association or host association.

[Insert the following note at the end of 2.2.3.2.] 12:30+

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.

[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 entities that are accessible to 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.

[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 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”.] 44:27

[In the last line of Note 4.19, after “defined” add “, and its descendant submodules”.] 45

[In the last line of the fourth paragraph of 4.5.3.6, after “definition”, add “and its descendant submodules”.] 54:6

[In the last line of Note 4.41, after “module” add “, and its descendant submodules”.] 54

[In the last line of Note 4.42, after “definition” add “and its descendant submodules”.] 54

[In the last line of the paragraph before Note 4.45, after “definition” add “, and its descendant submodules”.] 57:3

[In the third and fourth lines of the second paragraph of 4.5.5.2 insert “or submodule” after “module” twice.] 58:11-12

[In the second paragraph of Note 4.49, insert “or submodule” after “module” twice.] 58

[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.] 84:12,14

[After the second paragraph after constraint C581 insert the following note.] 93:4+

NOTE 5.33$\frac{1}{2}$

I have no idea what I had in mind for this note. 02-277, 02-277r1 and 03-123 all had instructions to insert a note, but no note. Maybe it was about the possibility that a separate interface and a separate procedure might have different IMPLICIT rules in effect.

[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,”.] 113:22

[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”.] 251:3

[In the first line of the second paragraph of Section 11 insert ", submodules" after "modules".] 251:4

[Within the first paragraph of 11.2.1, at its end, insert the following sentence:] 253:8

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.

[After constraint C1109 insert an additional constraint:] 253:30+

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

[Insert a new subclause immediately before 11.3:] 255:1-

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-name* in the *submodule-stmt*. A submodule is a **child** of its parent. An **ancestor** of a module or 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 accesses the scoping unit of its parent module or submodule by host association.

A submodule may provide implementations for module procedures that are declared by separate interface bodies within ancestor program units, 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-name*) *submodule-name*

R1115c *end-submodule-stmt* **is** END [SUBMODULE [*submodule-name*]]

C1114a (R1115a) The *parent-name* shall be the name of a submodule or a nonintrinsic module.

C1114b (R1115a) An automatic object shall not appear in the *specification-part* of a submodule.

C1114c (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*.

C1114d (R1115a) A submodule *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt* or a *format-stmt*.

C1114e (R1115a) If an object of a type for which *component-initialization* is specified (R438) appears 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 third line of the first paragraph of 12.3 replace ", but" by ". If the dummy arguments are redeclared in a separate module procedure body (12.5.2.5) they shall have the same names as in the corresponding interface body (12.3.2.1); otherwise".] 259:12

[After the third paragraph after constraint C1211 insert the following paragraph and note.] 261:30+

A **separate interface body** is an interface body in which the *prefix* of the initial *function-stmt* or *subroutine-stmt* includes SEPARATE. It declares the interface for a separate module procedure (12.5.2.5). 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 has the PUBLIC attribute. If the definition of its procedure body does not appear within the *module-subprogram-part* of the program unit in which the separate interface body is declared, or one of its descendant submodules (11.2.3), the interface may be used but the procedure shall not be used in any way.

A **separate interface** is declared by a separate interface body.

C1211a (R1205) A scoping unit that specifies a separate interface body shall be a module or submodule.

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

[Add a right-hand-side to R1228:] 282:5+

or SEPARATE

[Add constraints after C1242:] 282:9+

C1242a (R1227) SEPARATE shall appear only within the initial *function-stmt* or *subroutine-stmt* of an interface body or module subprogram.

C1242b (R1227) If SEPARATE appears within the initial *function-stmt* or *subroutine-stmt* of a module subprogram, a separate interface for the module subprogram shall appear in the module or submodule in which the subprogram appears or an ancestor of it.

This is *not the same* as accessing the interface by host association. Host association would include getting the interface into an ancestor by use association.	*Note to WG5*

[Insert a new subclause before 12.5.2.4 and renumber succeeding subclauses appropriately.] 285:1-

12.5.2.4 Separate module procedures

A **separate module procedure** is a module procedure whose interface is declared by a separate interface body (12.3.2.1) in the *specification-part* of a module or submodule.

A separate module procedure and a separate interface body **correspond** if they have the same name, and the separate module procedure is defined in the program unit where the separate interface is defined or a descendant of that program unit. At most one separate module procedure shall correspond to a separate interface.

If a separate interface body does not have a corresponding separate module procedure it shall not be invoked or used as a *proc-target*.

NOTE 12.40$\frac{1}{3}$

> 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 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.

A module subprogram that defines a separate module procedure may respecify the characteristics declared in its interface body. If any characteristic is specified, or if the *prefix* has any *prefix-spec* other than SEPARATE, all characteristics shall be specified. If the characteristics are respecified, they shall be identical to those specified in its interface body, except that the module procedure may be specified to be pure even if the corresponding interface does not so specify, in which case the procedure is pure.

NOTE 12.40$\frac{2}{3}$

> As with an external procedure, if a separate module procedure is declared to be pure, it shall satisfy all the requirements for pure procedures. If the interface is not declared to be pure, the invocations using that interface cannot take advantage of the properties of purity.

> The exception for pure procedures was consciously modeled on [261:40-41]. Module procedures are, however, different from external procedures. Do we want this exception for separate procedures? *WG5 questions*
> There is no requirement that dummy arguments have the same names in the separate procedure body as in the corresponding separate interface body. This too is modeled on external procedures. Do we want to require the dummy argument names to be the same?

[In constraint C1253 replace "*module-subprogram*" by "a *module-subprogram* that does not define a separate module procedure".] 285:7

[In the first line of the first paragraph after syntax rule R1236 in 12.5.2.6 insert ", submodule" after "module",] 286:37

[In item (1) in the first numbered list in 16.2, after "abstract interfaces" insert ", separate interfaces".] 408:6

[After "(4.5.9)" insert ", and a separate module procedure shall have the same name as its corresponding separate interface body".] 408:16

[In the first line of the first paragraph of 16.4.1.3 insert ", a separate interface body" after "module subprogram". In the second line, insert "that is not a separate interface body" after "interface body".] 412:30,31

[After **host association** insert a new sentence: "A submodule has access to the named entities of its parent by host association."] 412:31

[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".] 413:26

[Insert a new item after item (5)(d) in the list in 16.4.2.1.3:] 417:6+

(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 the second line of item 2 of 16.5.6 replace "or in a" by ", submodule, or".] 423:48

[In item 3c of 16.5.6 insert "or submodule" after "module" twice.] 424:8-9

[Replace Note 16.18 by the following.] 424

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 or by means other than Fortran.

[In item 3d of 16.5.6 insert "or submodule" after "module" twice.] 424:10-11

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

ancestor (11.2.3) : Of a submodule, its parent or an ancestor of its parent. 427:15+

child (11.2.3) : A submodule is a child of its parent. 428:43+

descendant (11.2.3) : Of a module or submodule, one of its children or a descendant of one of its children. 430:28+

parent (11.2.3) : Of a submodule, the module or submodule specified by the *parent-name* in its *submodule-stmt*. 434:36+

separate interface (12.3.2.1) : An interface defined by an interface body in which SEPARATE appears in the initial *function-stmt* or *subrotine-stmt*. It declares the interface for a module procedure that has a separately-defined body. 436:26+

submodule (2.2.5, 11.2.3) : A program unit that depends on a module or another submodule; it extends the program unit on which it depends. 437:15+

[Insert a new subclause immediately before C.9:] 479:33+

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 separate interface body that is specified in a module has public accessibility, and its corresponding separate 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, one can change a separate procedure body in a submodule without any need to change its corresponding separate interface. If a tool for automatic program translation is used, and even if it exploits the relative modification times of files as opposed to comparing the result of translating the module to the result of a previous translation, modifying a submodule need not result in the tool deciding to reprocess program units that access the module by use association.

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 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 `separate` prefix in the interface bodies would serve equally well as documentation – but the procedures would be external procedures. It wouldn't make any 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
    separate subroutine color_point_del ( p ) ! Destroy a color_point object
      type(color_point) :: p
    end subroutine color_point_del
    ! Distance between two color_point objects
    real separate function color_point_dist ( a, b )
      type(color_point), intent(in) :: a, b
    end function color_point_dist
    separate subroutine color_point_draw ( p ) ! Draw a color_point object
      type(color_point) :: p
    end subroutine color_point_draw
    separate subroutine color_point_new ( p ) ! Create a color_point object
      type(color_point) :: 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 procedures for which corresponding separate 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
      separate 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 forward interfaces declared
           ! in the module

    separate subroutine color_point_del ! ( p )
      instance_count = instance_count - 1
      deallocate ( p )
    end subroutine color_point_del
    separate 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
    separate subroutine color_point_new ! ( 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 parent submodule
    separate subroutine color_point_draw ! ( p )
    ! Its interface is defined in an ancestor.
      type(palette) :: MyPalette
      ...; call inquire_palette ( p, MyPalette ); ...
    end subroutine color_point_draw

    separate subroutine inquire_palette
      ... implementation of inquire_palette
    end subroutine 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
```

Multilevel submodule systems 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. If any of them are changed, it cannot affect program untis that access the `Plasma` module by use association.