

J3/02-277

**WORKING DRAFT
ISO IEC TECHNICAL REPORT 19767**

ISO/IEC JTC1 WG5 PROJECT 1.22.02.01.01.01

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1

18 September 2002

THIS PAGE TO BE REPLACED BY ISO-CS

Contents

0	Introduction	ii
0.1	Shortcomings of Fortran's module system	ii
0.1.1	Decomposing large and interconnected facilities	ii
0.1.2	Avoiding recompilation cascades	iii
0.1.3	Packaging proprietary software	iii
0.1.4	Easier library creation	iv
0.2	Disadvantage of using this facility	iv
1	General	1
1.1	Scope	1
1.2	Normative References	1
2	Requirements	2
2.1	Summary	2
2.2	Submodules	2
2.3	Separate module procedure and its corresponding module interface body	3
2.4	Examples of modules with submodules	3
2.5	Relation between modules and submodules	4
3	Required editorial changes to ISO/IEC 1539-1	5

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, most changes in modules occur in the implementation of those modules – in the procedures that implement the behavior of the modules and the private data they retain and share – not in the interfaces of the procedures of the modules, nor in the specification of publicly accessible types or data entities. Changes in the implementation of a module have no effect on the translation of other program units that access the changed module. The existing module facility, however, draws no structural distinction between interface and 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 other modules must be translated differently.

If a module is used only in the implementation of a second module, a third module accesses the second, and one changes the interface of the first module, utilities that examine the dates of files have no alternative but to conclude that a change may have occurred that could affect the translation of the third module.

Modules can be decomposed using facilities specified in this technical report so that a change in the interface of a module that is used only in a submodule has no effect on the parent of that submodule, and therefore no effect on the translation of other modules that use the second module. 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 withholding 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 inter-procedural optimizations if the program uses the facility specified in this technical report. When translator systems become able to do 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 nullified 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 1.1 Scope

2 This technical report specifies an extension to the module facilities of the programming language Fortran.
3 The current Fortran language is specified by the international standard ISO/IEC 1539-1 : Fortran. The
4 extension allows program authors to develop the implementation details of concepts in new program units,
5 called **submodules**, that cannot be accessed directly by use association. In order to support submodules,
6 the module facility of international standard ISO/IEC 1539-1 is changed by this technical report in such
7 a way as to be upwardly compatible with the module facility specified by international standard ISO/IEC
8 1539-1.

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

12 1.2 Normative References

13 The following standards contain provisions that, through reference in this text, constitute provisions of this
14 technical report. For dated references, subsequent amendments to, or revisions of, any of these publications
15 do not apply. Parties to agreements based on this technical report are, however, encouraged to investigate the
16 possibility of applying the most recent editions of the normative documents indicated below. For undated
17 references, the latest edition of the normative document referenced applies. Members of IEC and ISO
18 maintain registers of currently valid International Standards.

19 ISO/IEC 1539-1 : *Information technology - Programming Languages - Fortran*

1 2 Requirements

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

5 2.1 Summary

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

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

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

13 2.2 Submodules

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

17 An **ancestor** of a submodule is that submodule, or an ancestor of its parent. A **descendant** of a module
18 or submodule is that program unit, or a descendant of a child of that program unit.

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

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

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

2.3 Separate module procedure and its corresponding module interface body

A **module interface body** is different from an interface body defined by ISO/IEC 1539-1 in three respects. First, it has a **MODULE** prefix in the subroutine statement or function statement that introduces the interface body. Second, in addition to specifying a procedure's characteristics and dummy argument names, a module 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 has the same name as a module interface body declared in an ancestor module or submodule, the procedure is a **separate module procedure** that corresponds to the module interface body. 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.

Some or all of the characteristics and dummy argument names may be redeclared in the module subprogram that defines the separate module procedure. If any dummy arguments are redeclared, all shall be redeclared, and shall have the same names and characteristics as in the interface body. Any other characteristics of the module procedure that are declared in the module subprogram shall be the same as those declared in its interface body.

If the procedure is a function, the result variable name is determined by the declaration of the module subprogram, not by the module interface body. If the module 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

The example module **POINTS** below declares a type **POINT** and an 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. The declaration of the result variable name **DISTANCE** serves only as a vehicle to declare the result characteristics; the name is otherwise ignored.

```

26  MODULE POINTS
27      TYPE :: POINT
28          REAL :: X, Y
29      END TYPE POINT
30
31      INTERFACE
32          MODULE FUNCTION POINT_DIST ( A, B ) RESULT ( DISTANCE )
33              TYPE(POINT), INTENT(IN) :: A, B ! Accessed by host association
34              REAL :: DISTANCE
35          END FUNCTION POINT_DIST
36      END INTERFACE
37  END MODULE POINTS

```

1 The example submodule POINTS_A below is a submodule of the POINTS module. The scope of the type
 2 name POINT extends into the submodule; it cannot be redefined in the submodule. The characteristics of
 3 the function POINT_DIST can be redeclared in the module function body, or taken from the module interface
 4 body in the POINTS module. The fact that POINT_DIST is accessible by use association results from the fact
 5 that there is a module interface body of the same name in the ancestor module.

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

13 An alternative declaration of the example submodule POINTS_A shows that it is not necessary to redeclare the
 14 characteristics of the module procedure POINT_DIST. The result variable name is POINT_DIST, even though
 15 the module interface body specifies a different result variable name.

```
16  SUBMODULE ( POINTS ) POINTS_A
17    CONTAINS
18      FUNCTION POINT_DIST
19        TYPE(POINT), INTENT(IN) :: P, Q
20        POINT_DIST = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
21      END FUNCTION POINT_DIST
22  END SUBMODULE POINTS_A
```

23 2.5 Relation between modules and submodules

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

27 A submodule accesses the scoping unit of its parent module or submodule by host association. Therefore, all
 28 entities accessible in a parent module or submodule, including private entities and module interface bodies
 29 for separate module procedures in different submodules, are accessible within each subsidiary submodule.

1 **3 Required editorial changes to ISO/IEC 1539-1**

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

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

5 *or submodule*

6 [After syntax rule R1104, at 9:34+, add the following syntax rule. This is a quotation of the “real” syntax
 7 rule in subclause 11.2.3.]

8 *submodule* **is** *submodule-stmt*
 9 [*specification-part*]
 10 [*module-subprogram-part*]
 11 *end-submodule-stmt*

12 [In the second line of the first paragraph of subclause 2.2, at 11:42, insert “, a submodule” after “module”.]

13 [In the fourth line of the first paragraph of subclause 2.2, at 11:44, insert a new sentence:]

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

16 [In the sixth line of the first paragraph of subclause 2.2, at 11:46, insert “, a submodule” after “module”.]

17 [In the penultimate line of the first paragraph of subclause 2.2, at 11:48, insert “or submodule” after “mod-
 18 ular”.]

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

20 A module procedure may be invoked from within any scoping unit that accesses its declaration (12.3.2.1) or
 21 definition (12.5).

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

NOTE 2.2¹₂

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

1 [Insert a new subclause at 13:17+:]

2 **2.2.5 Submodule**

3 A **submodule** contains definitions (12.5) for procedures whose interfaces are declared (12.3.2.1) in its parent
4 module or submodule. It may also contain declarations and definitions of entities that are accessible to and
5 used by descendant submodules. An entity declared in a submodule is not accessible by use association, but
6 a procedure that is declared in a module and defined in one of that module's submodules is accessible by
7 use association.

8 [In the second line of the first row of Table 2.1 on page 14, insert “, SUBMODULE” after “MODULE”.]

9 [On page 14, change the heading of the third column of Table 2.2 from “Module” to “Module or Submodule”.]

10 [In the second footnote to Table 2.2 on page 14, insert “or submodule” after “module” and change “the
11 module” to “it”.]

12 [In the last line of 2.3.3, at 15:3, insert “, *end-submodule-stmt*” after “*end-module-stmt*”.]

13 [In the first line of the second paragraph of 2.4.3.1.1, at 17:4, insert “, submodule” after “module”.]

14 [At the end of 3.3.1, immediately before 3.3.1.1 on page 28, add “END SUBMODULE” to the list of adjacent
15 keywords where blanks are optional.]

16 [In the third line of the first paragraph of 4.5.1.8, at 50:22, replace “itself” by “and all of its descendant
17 submodules”.]

18 [In the last line of the second paragraph of 4.5.1.8, at 50:28, after “definition” add “and all of its descendant
19 submodules”.]

20 [In the last line of the fourth paragraph of 4.5.1.8, at 51:6, after “definition” add “and all of its descendant
21 submodules”.]

22 [In the last line of the first paragraph after Note 4.34, at 51:8, after “definition” add “and all of its descendant
23 submodules”.]

24 [In the last line of Note 4.37 on page 51, after “module” add “and all of its descendant submodules”.]

- 1 [In the last line of Note 4.38 on page 51, after “defined” add “, and all of its descendant submodules”.]

- 2 [In the last line of Note 4.39 on page 52, after “definition” add “and all of its descendant submodules”.]

- 3 [In the third line of the second paragraph of 4.5.10.1, at 60:19, insert “or submodule” after “module”. In
4 the third and fourth line, replace “referencng the module” by “that has access to that program unit”.]

- 5 [In the first line of the second paragraph of Note 4.58, on page 61, insert “or submodule” after “module”.]

- 6 [In constraint C531, at 69:33, insert “or submodule” after “module”.]

- 7 [In the first line of the second paragraph of 5.1.2.12, at 81:26, insert “, or any of its descendant submodules”
8 after “attribute”.]

- 9 [In the first line of the second paragraph of 5.1.2.13, at 82:9, insert “or any of its descendant submodules”
10 after “module”.]

- 11 [In constraint C558, at 85:10, insert “or submodule” after “module”.]

- 12 [After the second paragraph after constraint C580, at 91:7+, insert the following note.]

- 13 [In the third line of the penultimate paragraph of 6.3.1.1, at 111:15, replace “or a subobject thereof” by “or
14 submodule, or a subobject thereof.”.]

- 15 [In the first line of the first paragraph after Note 6.22, at 113:9, insert “or submodule” after “module”.]

- 16 [In the fourth item in the list in 6.3.3.2, at 115:10, insert “or submodule” after the first “module”.]

- 17 [In the second line of the first paragraph of Section 11, at 245:3, insert “, a submodule” after “module”.]

- 18 [In the first line of the second paragraph of Section 11, at 245:4, insert “, submodules” after “modules”.]

- 19 [In constraint C1105, at 246:20, insert “or submodule” after “module”.]

- 20 [In constraint C1106, at 246:22, insert “or submodule” after “module”.]

- 21 [In constraint C1107, at 246:24, insert “or submodule” after “module”.]

1 [Within the first paragraph of 11.2.1, at its end at 247:4, insert the following sentence:]

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

3 [Then insert the following note:]

NOTE 11.6¹₂

It is possible for submodules of different modules to access each others' ancestor modules.

4 [After constraint C1109, at 247:36+, insert an additional constraint:]

5 Constraint: If the USE statement appears within a submodule, *module-name* shall not be the name of the
6 ancestor module of the submodule.

7 [Insert a new subclause immediately before 11.3, at 249:6-:]

8 **11.2.3 Submodules**

9 A **submodule** is a program unit that depends on a module or another submodule. It may provide definitions
10 for module procedures that are declared in or accessible by host association in the module or submodule on
11 which it depends, and declarations and definitions of other entities that are accessible by host association in
12 submodules subsidiary to it.

```

13 submodule                                is submodule-stmt
14                                           [ specification-part ]
15                                           [ module-subprogram-part ]
16                                           end-submodule-stmt

17 submodule-stmt                          is SUBMODULE ( parent-name ) submodule-name
18 end-submodule-stmt                      is END [ SUBMODULE [ submodule-name ] ]

```

19 Constraint: The *parent-name* shall be the name of a submodule or a nonintrinsic module.

20 Constraint: The *submodule-name* shall not be the same as *parent-name*.

21 Constraint: If a *submodule-name* is specified in the *end-submodule-stmt*, it shall be identical to the *submodule-*
22 *name* specified in the *submodule-stmt*.

23 The program unit on which a submodule depends is its **parent** module or submodule; its parent is specified
24 by the *parent-name* in the *submodule-stmt*. A submodule is a **child** of its parent. An **ancestor** of a
25 submodule is that submodule or an ancestor of its parent. A **descendant** of a module or submodule is that
26 program unit or a descendant of one of its child submodules.

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

NOTE 11.12¹₂

A procedure in a module or submodule has access to every entity in its ancestor program units. Even if no other program unit has access to the module or submodule, there may be an active procedure invoked by way of a procedure pointer or by means other than Fortran that has access to it. This may affect finalization (4.5.10.1) or undefinition (6.3.3.2, 16.4.2.1.3, 16.5.6).

- 2 [In the third line of the second paragraph of 12.3, at 253:15, replace “, but” by “. If the dummy arguments
3 are redeclared in a separate module procedure body (12.5.2.5) they shall have the same names as in the
4 corresponding module interface body (12.3.2.1); otherwise”.]

- 5 [Add a new constraint after Constraint C1211 on page 255, at 255:26+.]

- 6 Constraint: (R1209) An IMPORT statement shall not appear within a module procedure interface body.

- 7 [After the third paragraph after constraint C1211 on page 255, at 255:35+, insert the following paragraph
8 and note.]

- 9 An interface body introduced by a *function-stmt* or *subroutine-stmt* that has a *prefix* in which MODULE
10 appears is a **module interface body**. A separate module procedure (12.5.2.5) is accessible if and only if
11 its corresponding module interface body is accessible. If its definition does not appear within the *module-*
12 *subprogram-part* of the program unit in which the module interface body is declared, or one of its descendant
13 submodules (11.2.3), the interface may be used but the procedure shall not be used in any way.

NOTE 12.3¹₂

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

- 14 [In the first sentence of the fourth paragraph after constraint C1211 on page 255, at 255:36, insert “, that is
15 not a module interface body,” after “block”.]

- 16 [In the first paragraph on page 256, move the sentence “An interface for a procedure named by an ENTRY
17 statement may be specified by using the entry name as the procedure name in the interface body” to be a
18 paragraph in its own right after Note 12.4.]

- 19 [In the first paragraph after Note 12.6 on page 257, at 257:3-4, delete the sentence “The characteristics of
20 module procedures are not given in interface blocks, but are assumed from the module subprograms.”]

- 21 [After the first paragraph after Note 12.7 on page 258, at 258:2+, insert the following paragraph.]

1 The characteristics of a module procedure may be specified by a module interface body, by the module
2 procedure's declaration, or both.

3 [Add a fourth right-hand side for syntax rule R1228 on page 275, at 275:34+.]

4
$$\text{or MODULE [(*module-or-submodule-name*)]}$$

5 [Add the following constraint after constraint C1243 on page 275, at 275:38+.]

6 Constraint: (R1227) MODULE shall not appear in a *prefix* except within *subroutine-stmt* or *function-stmt*
7 that introduces an interface body that is directly within a specific or generic interface block that
8 is in turn directly within the *specification-part* of a module.

9 [Insert a new paragraph and a note immediately before Note 12.38 on page 276, at 276:33+]

10 The optional *module-or-submodule-name* in the MODULE prefix shall be the name of the module or sub-
11 module in which the separate module procedure definition appears (12.5.2.5).

NOTE 12.37¹₂

The purpose of the optional *module-or-submodule-name* in the MODULE prefix is to allow processors to inline module procedures that have separate bodies. It is also useful to those maintaining a program. If it is not specified, the separate module procedure definition may not yet exist, it may be in the same program unit as the interface body, or it may be in an unspecified submodule – which makes it difficult to find.

12 [Insert a new subclause before 12.5.2.5 on page 280, at 280:14-, and renumber succeeding subclauses appro-
13 priately.]

14 **12.5.2.5 Separate module procedure definition**

15 A **separate module procedure** is a module procedure that has the same name as a module interface body
16 (12.3.2.1) in an ancestor module or submodule of the scoping unit in which the procedure is defined. That
17 module interface body is the **corresponding module interface body** of the separate module procedure.

NOTE 12.40¹₃

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

18 A module subprogram that defines a separate module procedure may respecify any of the characteristics
19 declared in its corresponding module interface body. If any dummy arguments are respecified in the module
20 subprogram, they shall all be respecified, in the same order and each with the same name and characteristics
21 as specified in the corresponding module interface body. If a characteristic is specified in the corresponding

1 module interface body but it is not specified in the module subprogram, the module procedure nonetheless
 2 has that characteristic. A characteristic that is specified in the module subprogram that defines a separate
 3 module procedure shall be specified in its corresponding module interface body.

NOTE 12.40²₃

As specified in 12.3.2.1, specifications within an interface body that do not specify characteristics or dummy argument names have no effect. Therefore, if a separate module procedure is to be recursive, or it is to have a result name different from the function name, these properties must necessarily be specified within the module subprogram.

4 [In the first line of the first paragraph after syntax rule R1236 in 12.5.2.6, at 280:27, insert “, submodule”
 5 after “module”,]

6 [In the first line of the first paragraph of 16.4.1.3, at 400:32, insert “, module interface body” after “module
 7 subprogram”. In the second line, insert “that is not a module interface body” after “interface body”.]

8 [In item 2 of 16.5.6, at 411:30, insert “or submodule” after “module”.]

9 [In item 4c of 16.5.6, at 411:38-39, insert “or submodule” after the first “module” and replace the second
 10 “module” by “that scoping unit”.]

11 [Replace Note 16.18 by the following.]

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.

12 [In item 4d of 16.5.6, at 411:40-41, insert “or submodule” after the first “module” and replace the second
 13 “module” by “that scoping unit”.]

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

- 15 **ancestor** (11.2.3) : A module or submodule, or an ancestor of the parent of that module or submodule. 415:12+
- 16 **child** (11.2.3) : A submodule, when considered in its relation to the module or submodule upon which it 416:40+
- 17 depends.
- 18 **descendant** (11.2.3) : A module or submodule, or a descendant of a child of that module or submodule. 418:22+

1 **parent** (11.2.3) : A module or submodule, when considered in its relation to the submodules that depend
2 upon it.

3 **submodule** (2.2.5, 11.2.3) : A program unit that depends on a module or another submodule; it extends
4 the program unit on which it depends.

5 [Insert a new subclause immediately before C.9, at 465:33+:]

6 **C.8.3.9 Modules with submodules**

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

9 If a module interface body is specified in a module, and it has public accessibility, and its corresponding
10 separate module procedure is defined by a subprogram in a descendant of that module, the procedure can
11 be accessed by use association. No other entity in a submodule can be accessed by use association. Each
12 program unit that accesses a module by use association depends on it, and each submodule depends on its
13 ancestor module. Therefore, one can change a procedure body in a submodule without any possibility of
14 changing the interface of the procedure. If a tool for automatic program translation is used, and even if it
15 exploits the relative modification times of files as opposed to comparing the result of translating the module
16 to the result of a previous translation, modifying a submodule cannot result in the tool deciding to reprocess
17 program units that access the module by use association.

18 This is not the end of the story. By constructing taller trees, one can put entities at intermediate levels
19 that are shared by submodules at lower levels, and have no possibility to affect anything that is accessible
20 from the module by use association. Developers of modules that embody large complicated concepts can
21 exploit this possibility to organize components of the concept into submodules, while preserving the privacy
22 of entities that ought not to be exposed to users of the module and preventing cascades of reprocessing.

23 The following example illustrates a module, `color_points`, with a submodule, `color_points_a`, that in turn
24 has a submodule, `color_points_b`. Public entities declared within `color_points` can be accessed by use
25 association. Except for the characteristics and dummy argument names of separate module procedures that
26 have module interface bodies that are accessible by use association, the submodules `color_points_a` and
27 `color_points_b` can be changed without causing the appearance that the module `color_points` might have
28 changed.

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

```
35 module color_points  
36  
37     type color_point
```

```

1     private
2     real :: x, y
3     integer :: color
4 end type color_point
5
6 interface      ! Interfaces for procedures with separate
7               ! bodies in the submodule color_points_a
8 module subroutine color_point_del ( p ) ! Destroy a color_point object
9     type(color_point) :: p
10 end subroutine color_point_del
11 ! Distance between two color_point objects
12 real module function color_point_dist ( a, b )
13     type(color_point) :: a, b
14 end function color_point_dist
15 module subroutine color_point_draw ( p ) ! Draw a color_point object
16     type(color_point) :: p
17 end subroutine color_point_draw
18 module subroutine color_point_new ( p ) ! Create a color_point object
19     type(color_point) :: p
20 end subroutine color_point_new
21 end interface
22
23 end module color_points

```

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

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

```

33 submodule ( color_points ) color_points_a ! Submodule of color_points
34
35     integer, save :: instance_count = 0
36
37     interface      ! Interface for a procedure with a separate
38                   ! body in submodule color_points_b
39 module subroutine inquire_palette ( pt, pal )
40     use palette_stuff      ! palette_stuff, especially submodules
41                           ! thereof, can access color_points by use
42                           ! association without causing a circular

```

```

1           ! dependence because this use is not in the
2           ! module. Furthermore, changes in the module
3           ! palette_stuff are not accessible by use
4           ! association of color_points
5           type(color_point), intent(in) :: pt
6           type(palette), intent(out) :: pal
7           end subroutine inquire_palette
8
9           end interface
10
11          contains ! Invisible bodies for public interfaces declared in the module
12
13          subroutine color_point_del ! ( p )
14              instance_count = instance_count - 1
15              deallocate ( p )
16          end subroutine color_point_del
17          function color_point_dist(result(dist)) ! ( a, b )
18              dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
19          end function color_point_dist
20          subroutine color_point_new ! ( p )
21              instance_count = instance_count + 1
22              allocate ( p )
23          end subroutine color_point_new
24
25          end submodule color_points_a

```

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

```

30          submodule ( color_points_a ) color_points_b ! Subsidiary**2 submodule
31
32          contains ! Invisible body for interface declared in the parent submodule
33              subroutine color_point_draw ! ( p )
34                  ! Its interface is defined in an ancestor.
35                  type(palette) :: MyPalette
36                  ...; call inquire_palette ( p, MyPalette ); ...
37          end subroutine color_point_draw
38          subroutine inquire_palette
39              ! "use palette_stuff" not needed because it's in the parent submodule
40              ... implementation of inquire_palette
41          end subroutine inquire_palette
42          subroutine private_stuff ! not accessible from color_points_a
43              ...

```

```

1     end subroutine private_stuff
2
3     end submodule color_points_b
4
5     module palette_stuff
6         type :: palette ; ... ; end type palette
7     contains
8         subroutine test_palette ( p )
9             ! Draw a color wheel using procedures from the color_points module
10            type(palette), intent(in) :: p
11            use color_points ! This does not cause a circular dependency because
12                ! the "use palette_stuff" that is logically within
13                ! color_points is in the color_points_a submodule.
14            ...
15        end subroutine test_palette
16    end module palette_stuff

```

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

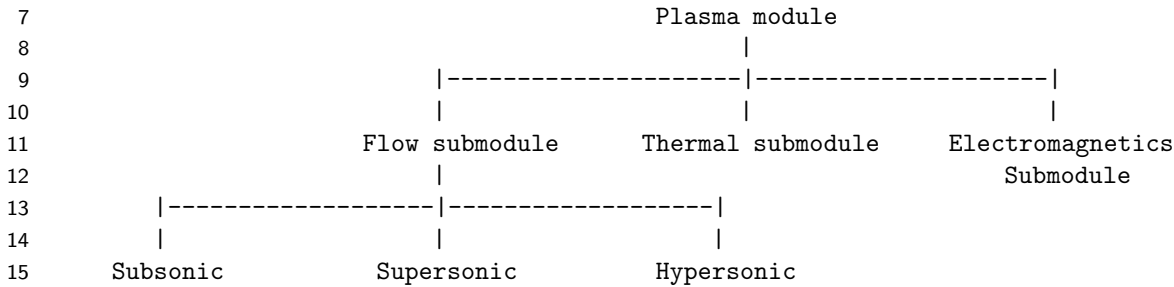
```

21
22     program main
23         use color_points
24         ! "instance_count" and "inquire_palette" are not accessible here
25         ! because they are not declared in the "color_points" module.
26         ! "color_points_a" and "color_points_b" cannot be accessed by
27         ! use association.
28         interface ( draw ) ! just to demonstrate it's possible
29             module procedure color_point_draw
30         end interface
31         type(color_point) :: C_1, C_2
32         real :: RC
33         ...
34         call color_point_new (c_1)          ! body in color_points_a, interface in color_points
35         ...
36         call draw (c_1)                    ! body in color_points_b, specific interface
37                                           ! in color_points, generic interface here.
38         ...
39         rc = color_point_dist (c_1, c_2) ! body in color_points_a, interface in color_points
40         ...
41         call color_point_del (c_1)        ! body in color_points_a, interface in color_points
42         ...
43     end program main

```

1 Multilevel submodule systems can be used to package and organize a large and interconnected concept
 2 without exposing entities of one subsystem to other subsystems.

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



16 Entities can be shared among the **Subsonic**, **Supersonic**, and **Hypersonic** submodules by putting them
 17 within the **Flow** submodule. One then need not worry about accidental use of these entities by the **Thermal**
 18 or **Electromagnetics** modules, or the development of a dependency of correct operation of those subsystems
 19 upon the representation of entities of the **Flow** subsystem as a consequence of maintenance.