ISO IEC TECHNICAL REPORT

ISO/IEC JTC1 PDTR XX.XX.XX

Enhanced Module Facilities

in

Fortran

An extension to IS 1539-1:1997

13 March 2000

THIS PAGE TO BE REPLACED BY ISO-CS

Contents

Foreword

[General part to be provided by ISO CS]

This technical report specifies an extension to the module program unit facilities of the programming language Fortran. Fortran is specified by the international standard ISO/IEC 1539-1. 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 commercial 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 to extends the module facility of Fortran so that program developers can encapsulate the implementation details of module procedures in zero or more **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 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, the language translation system has no alternative but to conclude that a change may 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 re-translate and re-certify a large program.

Using facilities specified in this technical report, implementation details of a module can be encapsulated in submodules, so that they 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 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.

0.1.2 Packaging proprietary software

If a module as specified by the 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 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.

0.1.3 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.

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

Decomposing a complicated intellectual concept may furthermore require circularly dependent modules. The latter 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 international standard 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.

0.1.4 Easier library creation

Most Fortran translator systems produce a single file of computer instructions, called an *object file*, for each module. This is easier than producing a separate object file for the specification part and 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.

If modules are decomposed using facilities specified in this technical report, it would be 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 0.1.1, even if this facility is used. Although one advantage of this facility would 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 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.

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

1.2 Normative References

The following standards contain provisions which, 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. However, parties to agreements based on this technical report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 1539-1 : 1997 *Information technology - Programming Languages - Fortran*

2 Requirements

The following subsections contain a general description of the extensions to the syntax and semantics of the current Fortran programming language to provide facilities for submodules.

2.1 Modules

As specified in ISO/IEC 1539-1, a module consists of a specification part and a module subprogram part.

This technical report defines a **submodule specification part**, in which only the interfaces of procedures in submodules are declared. This part is introduced by a statement of the form `SUBMODULE ::` *submodule-name*. A submodule specification part extends from the SUBMODULE statement that introduces it to (but not including) the next CONTAINS, SUBMODULE or END MODULE statement. A **submodule procedure** is a module procedure for which the interface is specified in a submodule specification part, and the body is defined in a submodule.

A module or submodule may have any number of module subprogram parts, and any number of submodule specification parts, in any order. If several submodule specification parts have the same name, the effect is as if the specifications they contain were concatenated within a single submodule specification part. This allows one to put all module procedures into alphabetical order.

Within a submodule specification part, procedure interface declarations specify procedures in the specified subsidiary submodule that can be accessed. This interface is syntactically identical to an interface body, but semantically different in that entities of the host environment of the interface are accessible within the interface by host association. Because of this difference, a procedure interface declaration within a submodule specification part is called a **procedure interface declaration** instead of an interface body.

2.1.1 Example of a submodule specification part

```
SUBMODULE :: POINTS_A
  REAL FUNCTION POINT_DIST ( A, B )
  ! Compute the distance between the points A and B
    TYPE(POINT) :: A, B
  END FUNCTION POINT_DIST
```

The submodule specification part in the above example specifies that there is a submodule, named `POINTS_A`, and that there is a function named `POINT_DIST`, with the specified interface, that can be accessed from that submodule. If the program unit containing the submodule specification part is a module, and `POINT_DIST` is public, then `POINT_DIST` can be accessed by use association of that module.

2.2 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to a module or another submodule. If a module or submodule has subsidiary submodules, it is the **parent** of those subsidiary submodules.

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*.

A submodule may have a specification part, zero or more submodule specification parts, and zero or more module procedure parts.

Everything is a submodule is effectively PRIVATE except for those submodule procedures that were declared to be PUBLIC in the parent module. It is not possible to access entities declared in the specification part of a submodule because a USE statement must specify a module, not a submodule. Thus, PRIVATE and PUBLIC declarations are not permitted in a submodule.

2.2.1 Completing a procedure declared in a parent module or submodule

If a procedure interface declaration appears in the parent program unit, the procedure shall be defined in the specified submodule, either within a module procedure part or a submodule specification part.

Within a module procedure part of the subsidiary submodule, the procedure body shall be introduced by a statement of the form `SUBMODULE FUNCTION` *function-name* or `SUBMODULE SUBROUTINE` *subroutine-name*, depending on the declaration in the parent program unit. The interface of the procedure shall not be repeated in the submodule.The procedure body is logically an extension of its interface declaration; it does not access its interface declaration by host assciation.

Within a submodule specification part of the subsidiary submodule, the same statement may be used to indicate that definition of the body of the procedure is deferred to a yet more subsidiary submodule. In this case, neither an interface nor body shall follow the statement. The procedure shall be defined in the submodule specified in the submodule specification part of the subsidiary submodule, either within a module procedure part or a submodule specification part. This facility may be used to place the body of a public procedure in a submodule two or more steps subsidiary to the module, so that it may share implementation-dependent data or procedures in an intermediate subsidiary submodule with procedures in different subsidiary submodules. If the procedures in the intermediately subsidiary submodule are not specified in the module, they cannot be accessed by use association, and therefore either their interfaces or bodies can be changed without affecting the translation of a program unit that accesses the module by use association.

Example of a submodule

```
SUBMODULE(POINTS) POINTS_A
  CONTAINS
    SUBMODULE FUNCTION POINT_DIST RESULT(HOW_FAR)
```

```
      ! Don't re-declare dummy arguments, or result type
        HOW_FAR = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
      END FUNCTION POINT_DIST
  END SUBMODULE POINTS_A
```

Example of submodules with a deferred procedure body

```
  SUBMODULE(POINTS) POINTS_A
    ! Type and data declarations shared by submodules of POINTS_A (but not
    ! accessible anywhere else:
    ...
    SUBMODULE :: SUB_POINTS_A
      SUBMODULE FUNCTION POINT_DIST
      ! No body, because it's in a SUBMODULE specification part
      ...
    ! Other submodule or contains parts
  END SUBMODULE POINTS_A

  SUBMODULE(POINTS_A) SUB_POINTS_A
    CONTAINS
      SUBMODULE FUNCTION POINT_DIST RESULT(HOW_FAR)
      ! Don't re-declare dummy arguments, or result type
      ! Entities in POINTS_A and POINTS can be accessed
        HOW_FAR = SQRT( (A%X-B%X)**2 + (A%Y-B%Y)**2 )
      END FUNCTION POINT_DIST
  END SUBMODULE SUB_POINTS_A
```

2.3 Relation between modules and submodules

Public entities of a module, including procedure interface declarations in submodule specification parts, can be accessed by use association. Submodules contain no public entities. Public procedure interface declarations in submodule specification parts of modules imply that the procedure bodies in the specified submodules are indirectly accessible, by use association of the module.

All entities of a parent module or submodule, including private entities, declarations of interfaces to procedures implemented in different submodules, and entities accessed from a parent module or submodule by host association, are accessible within each subsidiary submodule by host association.

A procedure body in a submodule is logically a continuation of its interface in its parent program unit; it does not access its interface by host association.

3 Required editorial changes to ISO/IEC 1539-1 : 1997

The following suggested editorial changes illustrate that the extension in this technical report is not a large change to Fortran. While every effort has been made to cover all the bases, they will undoubtedly be a few additional changes necessary. Depending on the schedule of implementation of this technical report, it may also be necessary to convert the changes to refer to the 200x standard instead of the 1997 standard.

[Page and line numbers in brackets refer to ANSI/NCITS/J3 document 97-007r2.]

[10:30+] Add a new syntax rule in section **2.1 High level syntax**, after rule R213:

submodule-specification-part	**is** *submodule-specification-stmt* *submodule-procedure-declaration* [*submodule-procedure-declaration*] ...

[11:35] In the second line of **2.2 Program unit concepts**, add ", a submodule" after "a module".

[11:45] In item (2) of the list in section **2.2 Program unit concepts**, replace "body" by "declaration (12.3.2)".

[186:17-34] Replace the normative text of section **11.3 Modules** (but not subsidiary sections or the notes) with the following:

A **module** contains specifications and definitions that may be accessible to other program units.

module	**is** *module-stmt* [*specification-part*] [*procedure-part*] ... *end-module-stmt*
module-stmt	**is** MODULE *module-name*
procedure-part	**is** *module-subprogram-part* **or** *submodule-specification-part*
submodule-specification-stmt	**is** SUBMODULE :: *submodule-name*
submodule-procedure-declaration	**is** *procedure-interface-declaration* **or** *submodule-procedure-stmt*
submodule-procedure-stmt	**is** SUBMODULE FUNCTION *function-name* **or** SUBMODULE SUBROUTINE *subroutine-name*
end-module-stmt	**is** END MODULE [*module-name*]

Constraint: If *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* in the *module-stmt*.

Constraint: A *specification-part* in a module or submodule shall not contain a *stmt-function-stmt*, an *entry-stmt* or a *format-stmt*.

Constraint: If an object of a type for which *component-initialization* (R429) is specified appears in the *specification-part* of a module or submodule and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

Constraint: A *module-name* shall not be the same as any other name in the program unit.

Constraint: A *submodule-name* shall not be the same as any other name in the program unit, except that two *submodule-specification-parts* may have the same name.

Constraint: The *function-name* or *subroutine-name* in a *submodule-procedure-stmt* shall be declared to be a function or subroutine, respectively, in a *submodule-procedure-declaration* in a *submodule-specification-part* of the parent module or submodule that names the submodule in which the *submodule-procedure-stmt* appears.

Constraint: A *submodule-procedure-stmt* shall not appear except within a submodule.

A module name is a global name, and shall not be the same as the name of any other program unit, external procedure, or common block in the program.

If a module has submodules ([new section] 11.3.1), it is the **parent module** of those submodules.

A *submodule-name* specified in a *submodule-specification-stmt* shall be the same as the name of exactly one submodule ([new section] 11.3.1) in the program.

Every procedure that is named in a *submodule-specification-part* and is not a dummy procedure is a submodule procedure ([new section] 12.5.2.1), and shall be declared in a *submodule-procedure-stmt*, a *submodule-function-stmt*, or a *submodule-subroutine-stmt* in the submodule specified by the *submodule-name* in the *submodule-specification-stmt*.

If the same *submodule-name* appears on more than one *submodule-specification-stmt*, the effect is as though the *submodule-specification-parts* introduced by those statements were concatenated.

[187:2+] Insert the following before the existing section **11.3.1 Module reference**, and renumber subsequent sections:

11.3.1 Submodules

A **submodule** is a program unit that is dependent on and subsidiary to its parent module or submodule. Its parent module or submodule is its host environment.

submodule	**is**	*submodule-stmt*
		[*specification-part*]
		[*procedure-part*] ...
		end-submodule-stmt
submodule-stmt	**is**	SUBMODULE (*parent-name*) *submodule-name*
end-submodule-stmt	**is**	END SUBMODULE [*submodule-name*]

Constraint: The *submodule-name* in the *submodule-stmt* shall appear in a *submodule-specification-stmt* in the module or submodule named by the *parent-name*.

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

Constraint: The *submodule-name* specified in the *submodule-stmt* shall not be the same as any other name in the program unit.

A *submodule-name* is a global name, and shall not be the same as the name of any other program unit, external procedure, or common block in the program.

If a submodule has submodules, it is the **parent submodule** of those submodules.

Note

Related submodules and their parent module or submodule stand in a tree-like hierarchical relationship one to another, with the module at the root. For each submodule, its parent module or submodule is its parent with respect to the tree, and its submodules are children with respect to the tree.

[193:25] In the first sentence of section **12.3.2 Specification of the procedure interface**, add ", submodule" after "module".

[193:27-28] In the last sentence of the first paragraph of section **12.3.2 Specification of the procedure interface**, change the first occurrence of "in an interface block" to "as a procedure interface declaration," and change the second "interface block" to "procedure interface declaration."

[193:29-33] Remove note 12.3 – it is modified and moved to section 12.3.2.1. Replace it by:

procedure-interface-declaration **is** *function-stmt*
[*specification-part*]
end-function-stmt
or *subroutine-stmt*
[*specification-part*]
end-subroutine-stmt

Constraint: A *procedure-interface-declaration* for a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, or procedure arguments.

Constraint: A *procedure-interface-declaration* shall not contain an *entry-stmt, data-stmt, format-stmt,* or *stmt-function-stmt.*

A procedure interface declaration specifies all of the procedure's characteristics.

[193:42-6] Replace the definition of *interface-body* (R1205) and the following constraint by:

interface-body **is** *procedure-interface-declaration*

Note 12.3

An interface body cannot be used to describe the interface of an internal procedure, a module procedure, of an intrinsic procedure because the interfaces of such procedures are already explicit. The name of a module procedure may, however, appear in a MODULE PROCEDURE statement in an interface block.

[194:11-12] Remove the first constraint after syntax rule R1207 (it has been moved to section 12.3.2, with revised wording).

[194:32] In the first sentence of the paragraph of text immediately before note 12.4, replace "An interface body specifies all of the procedure's characteristics and these" by "The procedure characteristics specified by an interface body".

[206:21+] Add a new section 12.5.2.1 subsidiary to section 12.5.2 and renumber subsequent subsections:

12.5.2.1 Submodule procedures

A **submodule procedure** is a module procedure for which the interface is declared in a parent module (11.3) or submodule (11.3.1), and the body is defined in a submodule of that parent program unit. A submodule procedure body is logically a continuation of its procedure interface declaration in the parent module or submodule; it does not access the interface by host association.

A submodule procedure is accessible in its parent module or submodule. If the parent program unit is a module, and the procedure declared in the *submodule-specification-part* is public, it can be accessed by use association.

Note

It is possible to place specifications in a submodule declaration that do not contribute to specification of the interface. Unlike in an interface body, these specifications are part of the procedure.

[206:34] In section **12.5.2.2 Function subprogram** change the first line of the syntax rule (R1216) for *function-subprogram* to:

function-subprogram **is** *function-header*

[206:38+] In section **12.5.2.2 Function subprogram** add the following before the syntax rule (R1217) for *function-stmt*:

function-header **is** *function-stmt*
or *submodule-function-stmt*

[206:42+] In section **12.5.2.2 Function subprogram** add the following after the syntax rule (R1217) and constraint for *function-stmt*:

submodule-function-stmt **is** SUBMODULE FUNCTION *function-name* ■
■ [RESULT (*result-name*)]

Constraint: A *submodule-function-stmt* shall not appear except within a submodule.
Constraint: The *function-name* shall be declared, in a *submodule-procedure-declaration* in a *submodule-*

specification-part of the parent module or submodule of the submodule in which the *submodule-function-stmt* appears, to be a function that is not a dummy procedure. The *submodule-specification-stmt* that introduces that *submodule-specification-part* shall name the submodule in which the *submodule-function-stmt* appears.

Constraint: The *result-name* shall not be specified in both the *procedure-interface-declaration* and the *submodule-function-stmt*.

[207:14+] In section **12.5.2.2 Function subprogram**, in the last constraint after the syntax rule (R1220) for *end-function-stmt*, change *function-stmt* to *function-header*.

[208:25] In section **12.5.2.3 Subroutine subprogram** change the first line of the syntax rule (R1221) for *subroutine-subprogram* to:

subroutine-subprogram **is** *subroutine-header*

[208:29+] In section **12.5.2.3 Subroutine subprogram** add the following before the syntax rule (R1222) for *subroutine-stmt*:

subroutine-header **is** *subroutine-stmt*
or *submodule-subroutine-stmt*

[208:32+] In section **12.5.2.3 Subroutine subprogram** add the following after the syntax rule (R1222) and constraint for *subroutine-stmt*:

submodule-subroutine-stmt **is** SUBMODULE SUBROUTINE *subroutine-name*

Constraint: A *submodule-subroutine-stmt* shall not appear except within a submodule.

Constraint: The *subroutine-name* shall be declared, in a *submodule-procedure-declaration* in a *submodule-specification-part* of the parent module or submodule of the submodule in which the *submodule-subroutine-stmt* appears, to be a subroutine that is not a dummy procedure. The *submodule-specification-stmt* that introduces that *submodule-specification-part* shall name the submodule in which the *submodule-subroutine-stmt* appears.

[208:41+] In section **12.5.2.3 Subroutine subprogram**, in the last constraint after the syntax rule (R1224) for *end-subroutine-stmt*, change *subroutine-stmt* to *subroutine-header*.

[282:24] In **14.6.1.3 Host association** add "a procedure interface declaration that is not an interface body, a subsidiary submodule of a module" after "module subprogram" in the first sentence.

[283:8] In the last line of the first paragraph after the list of entities that can become inaccessible as a result of host association, replace "subprogram" by "scoping unit".

[298:30+] Add **parent module or submodule** to the glossary:

parent module or submodule (11.3): A *module* or *submodule* that has *submodules*.

[299:20] In the glossary entry for **program unit**, add "a *submodule*" after "*module*".

[300:24+] Add **submodule** and **submodule procedure** to the glossary:

submodule (11.3.1): A *program unit* that is logically an extension of a *module* or *submodule*, but cannot be accessed directly by *use association*.

submodule procedure (12.5.2.1): A *module procedure* for which the interface is declared in a *parent module or submodule*, and the body is defined in a *submodule* of that parent program unit.

[334:17+] Add a new section subsidiary to section **C.8.3 Examples of the use of modules**:

C.8.3.9 Modules with submodules

This 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 module `color_points` does not have a *contains-part*, but a *contains-part* is not prohibited. The module `color_points` could be published as definitive specification of the interface, without revealing trade secrets contained within `color_points_a` or `color_points_b`.

```
module color_points
  type color_point
    private
    real :: x, y
    integer :: color
  end type color_point
submodule :: color_points_a ! 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
  real function color_point_dist ( a, b ) ! Distance between two color_point objects
    type(color_point) :: 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 module color_points
```

The only entities within `color_points_a` that can be accessed by use association are procedures declared in submodule specification parts of `color_points` (in this case, there is only one submodule specification part). If the procedures' bodies 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 date of modification of a file would notice that changes in the module could affect the translations of program units that access the module by use association, but that changes in submodules could not.

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
! Procedure names are in alphabetical order
contains ! Invisible bodies for public interfaces declared in the module
  submodule subroutine color_point_del ! ( p )
    instance_count = instance_count - 1
    deallocate ( p )
  end subroutine color_point_del
  submodule function color_point_dist result(dist) ! ( a, b )
    dist = sqrt( (b%x - a%x)**2 + (b%y - a%y)**2 )
  end function color_point_dist
submodule :: color_points_b
  submodule subroutine color_point_draw ! ( p )
  ! "submodule" prefix indicates the interface is defined in the parent, not here.
  ! Being in a submodule specification part means the body is not here, either.
contains
  submodule subroutine color_point_new ! ( p )
    instance_count = instance_count + 1
    allocate(p)
  end subroutine color_point_new
submodule :: color_points_b ! continuation of above.
                            ! 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 submodule color_points_a
```

The subroutine `inquire_palette` is accessible within `color_points_a` because its interface is declared within a submodule specification part therein. It is not, however, accessible by use association, because its interface is not declared in a submodule specification part of 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
  submodule subroutine color_point_draw ! ( p )
  ! "submodule" prefix indicates the interface is defined in the parent, not here.
  ! Being in a contains part means the body is here.
    type(palette) :: MyPalette
    ...; call inquire_palette ( p, MyPalette ); ...
  end subroutine color_point_draw
  submodule subroutine inquire_palette
  ! "use palette_stuff" not needed because it's in the parent submodule
    ... 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
```