ISO/IEC JTC1/SC22/WG5 **N1482**

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

22 July 2002

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 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 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 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 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 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 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 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, 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 0.1.2, 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.

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, additional syntax and semantics for one existing entity, and additional semantics for another existing entity.

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. The syntactic variation on an existing entity is the *separate interface body*. The semantic variation is an interpretation of a procedure as a *separate procedure* under conditions specified below.

By putting a separate interface body in a module and its corresponding separate procedure in a submodule, program units that access the separate 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 **descendant** of a module or submodule is that program unit, or a descendant of a child of that program unit. An **ancestor** of a submodule is that submodule, or an ancestor of its parent.

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 in a submodule are effectively PRIVATE, except for the names of separate procedures that correspond to public separate interface bodies in the parent 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. Thus, PRIVATE and PUBLIC declarations are not permitted in a submodule.

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

2.3 Separate interface body and its corresponding procedure body

A **separate interface body** is different from an interface body defined by ISO/IEC 1539-1 in two respects. First, it has a `SEPARATE` prefix in the subroutine statement or function statement that introduces the interface body. Second, in addition to specifying a procedure's characteristics, a separate interface body specifies that its corresponding procedure is in a descendant of the module or submodule in which it appears. Unlike an ordinary interface body, it accesses the module or submodule in which it is declared by host association.

A module procedure that has the same name as a separate interface body declared in an ancestor module or submodule is a **separate procedure**. Its characteristics are declared by its corresponding interface body. For purposes of argument association, its dummy argument names are declared by its corresponding interface body. The procedure is accessible by use association if and only if its interface body is accessible by use association. There are two possibilities concerning redeclaration of its characteristics and dummy argument names in the separate procedure's *function-stmt*, *subroutine-stmt* or *entry-stmt*:

1. The characteristics may be redeclared in the module procedure's *function-stmt*, *subroutine-stmt* or *entry-stmt*, and shall be the same as declared in the separate interface body. Its dummy argument names need not be the same as in the separate interface body.
2. Its characteristics, other than whether it is a subroutine or function, are entirely determined by its interface body. The specification whether it is a subroutine or function shall be the same in the separate subprogram as in its separate interface body. This possibility is indicated by the absence of a dummy argument list and, if it is a function, the absence of a declaration of the result type.

If it is a function, the result variable name is determined by the declaration of the separate procedure, 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

The example module `POINTS` below declares a type `POINT` and the interface of a separate 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 scope of the type name POINT extends into the submodule; it cannot be redefined in the submodule. The characteristics of the function POINT_DIST can be redeclared in the submodule function body, or taken from the separate interface body in the POINTS module. The fact that POINT_DIST is a separate procedure is deduced from the fact that there is a separate interface body of the same in an ancestor module.

```
SUBMODULE(POINTS) POINTS_A
  CONTAINS
    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 separate procedure POINT_DIST. The result variable name is POINT_DIST, even though the separate interface body specifies a different result variable name.

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

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 procedures for which there is a corresponding publicly accessible separate interface body.

The scoping unit of a submodule is an extension of the scoping unit of its parent module or submodule, but the scope of names of entities declared in the submodule does not extend into its parent module or submodule. Therefore, all entities accessible in a parent module or submodule, including private entities and separate interface bodies for separate procedures in different submodules, are accessible within each subsidiary submodule. Because this accessibility is not by host association, entities accessible in a submodule as a consequence of being accessible in its parent module or submodule cannot be redeclared or redefined.

3 Required editorial changes to ISO/IEC 1539-1

Except for the examples below, editorial changes to ISO/IEC 1539-1 are yet to be determined.

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`. 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 prodcedures. 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) :: 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 procedures for which 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 interfaces declared in the module

  color_point_del ! ( p )
    instance_count = instance_count - 1
    deallocate ( p )
  end subroutine color_point_del
  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
  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
  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
  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
```

Remarks (questions?) for WG5

WG5 may observe that the definitions of descendant and ancestor are reflexive. This allows a separate interface body and its corresponding procedure to be defined in the same module or submodule. This appears to be harmless, and is one of Lawrie Schonfelder's pet desires.

Variations and/or alternatives:

1. Instead of putting the `SEPARATE` prefix on the interface body, put it on the interface block. This decision could be cast in stone in the standard, or the standard could allow a program to use either one of these.
2. More radically, the `SEPARATE` prefix could be done away with entirely. If we allow an interface body in the *specification-part* of a module or submodule to have the same name as a module procedure in one of the descendant submodules of where it's declared, and state that the scope of the name is the module or submodule where the interface body is defined and all of its descendants, we don't need the `SEPARATE` prefix. This puts a perhaps-undesirable burden of inference on the compiler and the human reader. The BLUNDER in Fortran 90 of host association not working into interface bodies throws a monkey wrench into this plan, however: One would need to put an `IMPORT` statement in the interface body in order to have the same semantics as it would if it were after the *module-subprogram-part*.