Subject: First draft of edits for parameterized modules
From: Van Snyder
Reference: 04-383

1 Introduction

Assuming parameterized modules get onto the J3 work plan, the reason for this paper is to get a running start on the edits. Nothing is said here about submodules. All that needs to be said is that parameterized modules that are global entities, internal modules, and instances, do not have submodules. The editor's guidance will be needed concerning how to specify where the edits apply.

2 Edits

Edits refer to 04-007. Page and line numbers are displayed in the margin. Absent other instructions, a page and line number or line number range implies all of the indicated text is to be replaced by associated text, while a page and line number followed by + (-) indicates that associated text is to be inserted after (before) the indicated line. Remarks are noted in the margin, or appear between [and] in the text.

R204	*specification-part*	**is**	*global-use-association-stmt*	9:38
		or	*other-use-stmt*	10:6+
		or	*internal-module*	10:11+
		or	*module-interface-block*	
		or	*other-use-stmt*	

[Editor: Replace Table 2.1. Notice that row 4 is gone — because it was wrong!] 14

Table 2.1: **Requirements on statement ordering**

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">Global use association statements</td></tr>
<tr><td colspan="3">IMPORT statements</td></tr>
<tr><td rowspan="3">FORMAT and ENTRY statements</td><td rowspan="2">Other USE statements and PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>Derived-type definitions,
internal modules,
interface blocks,
module interface blocks,
type declaration statements,
DATA statements,
enumeration definitions,
procedure declarations,
specification statements,
and statement function statements</td></tr>
<tr><td colspan="2">Executable constructs and DATA statements</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

We don't add parameterized modules or internal modules to Table 2.2 because they don't fit. Should they be added? Keep in mind we need to try to shoehorn submodules, too. I'd be happy to delete Table 2.2. 14 *J3 question*

C433a (R429) A *sequence-stmt* shall not appear in a *derived-type-def* that declares a type module parameter (11.2.2). 46:15+

---

or INITIALIZATION 71:22+

[Editor: Insert ", a module parameter" after "function".] 72:33

C514a (R501) The INITIALIZATION attribute shall not be specified except in the declaration of a data entity module parameter (11.2.2). 72:33+

C514b (R501) If the INITIALIZATION attribute is specified, the ALLOCATABLE, ASYNCHRONOUS, EXTERNAL, INTRINSIC, POINTER or VOLATILE attribute shall not be specified.

[Editor: Insert ", a module parameter" after "result".] 72:39

[Editor: Insert ", a module parameter" after "result".] 73:11

[Editor: Insert "that does not declare a module parameter" after "*entity-decl*".] 73:31

[Editor: Insert "and is not a module parameter" after "block".] 73:34

[Editor: Replace "A **module**" by the following:] 250:3

Modules are characterized by two independent factors. One is whether they are defined within another scoping unit, the other by whether they have parameters. A module that is defined within another scoping unit is an **internal module**. An internal module is not a program unit. A module that has parameters is a **parameterized module**. A module that is neither a parameterized module nor an internal module is referred to simply as a **module**. It

---

R1105 *module-stmt* **is** *module-name* [(*module-param-list*)] 250:11

If a *module-param-list* appears in a *module-stmt*, the module it introduces is a parameterized module (11.2.2). 250:25+ New ¶

[Editor: Replace "The" by "If the module is not an internal module, the".] 250:25+2

[Editor: Insert new subclauses and renumber subsequent ones (TeX-o-matic):] 251:4+

11.2.1 Internal modules

An **internal module** is a module that is defined within a program unit or subprogram. The scoping unit of an internal module accesses the scoping unit in which it is defined by host association.

R1108a *internal-module* **is** *module*

C1107a An *internal-module* shall not be defined within a block data program unit, an interface body, or another internal module.

11.2.2 Parameterized modules

A parameterized module is a module that has a *module-param-list* in its *module-stmt*. It serves as a template for creating instances (11.2.3) by substituting entities for its parameters. Parameters may be data entities, types, procedures, generic identifiers, or modules.

The **interface** of a parameterized module determines how it can be instantiated. It consists of the names of its parameters and their characteristics as module parameters.

The characteristics of a data entity module parameter are its type, type parameters, shape, the exact dependence of its type, type parameters or array bounds on other entities, whether it has the ALLOCATABLE, ASYNCHRONOUS, INITIALIZATION, POINTER, TARGET or VOLATILE attribute, whether it is polymorphic, whether the shape is assumed, and which if any of its type parameters are assumed.

The characteristics of a type module parameter are its type parameters, its component names, the characteristics its components would have if they were data entity module parameters, the interfaces of its type-bound procedures, the generic identifiers of its generic bindings, and which type-bound procedures are bound to each generic binding.

The characteristics of a procedure module parameter are its abstract interface and whether it is a procedure pointer.

The characteristics of a generic identifier module parameter are the characteristics as procedure module parameters of the interfaces specified by its interface bodies.

The characteristics of a module module parameter are whether it is parameterized, and, if so, its interface as a parameterized module.

Every parameter shall be declared. A data entity module parameter shall be declared by a *type-declaration-stmt*. A type module parameter shall be declared by a *derived-type-def*. A procedure module parameter shall be declared by a *procedure-declaration-stmt*, an *external-stmt* or an *interface-body*. A generic identifier module parameter shall be declared by an *interface-block*. A module module parameter shall be declared by a *module-interface*.

```
R1108b module-interface-block      is  INTERFACE
                                          module-interface
                                          [ module-interface ] ...
                                          END INTERFACE

R1108c module-interface            is  module-stmt
                                          [ specification-part ]
                                          end-module-stmt
```

C1107b (R1108c) The *module-name* in the *module-stmt* shall be the name of a module module parameter of the scoping unit containing the *module-interface-block*.

11.2.3 Instances of parameterized modules

An **instance** of a parameterized module is a module. It is created by a USE statement that specifies entities to be substituted for its module parameters. It is a local entity of the scoping unit in which it is instantiated. If the parameterized module from which the instance is created is an internal module, the instance accesses the scoping unit in which the parameterized module is defined by host association. An entity other than a module parameter in one instance is distinct from the corresponding entity in a different instance. A module parameter in one instance might or might not be distinct from the corresponding module parameter in a different instance, depending upon whether their corresponding instance parameters are distinct. Distinct entities in different instances might nonetheless be associated.

11.2.4 The USE statement 251:5-8

The **USE statement** specifies use association or creates an instance of a parameterized module. A USE statement is a **module reference** to the module it specifies. A module shall not reference itself, either directly or indirectly.

```
R1109  global-use-association-stmt is  USE [ [ , module-nature ] :: ] module-name ■      251:18-20
                                        ■ [ ( instance-parameter-spec-list ) ] module-ref-specialization
R1109a other-use-stmt              is  USE [ [ , module-nature ] :: ] module-name ■
                                        ■ [ ( instance-parameter-spec-list ) ] module-ref-specialization
                                   or  USE [ [ , module-nature ] :: ] local-module-name => ■
                                        ■ module-name ( instance-parameter-spec-list )
```

```
R1110a module-ref-specialization   is  [ , rename-list ]                                   251:22+
                                   or  , ONLY : [ only-list ]
R1110b instance-parameter-spec     is  [ keyword = ] instance-parameter
R1110c instance-parameter          is  expr
                                   or  declaration-type-spec
                                   or  procedure-name
                                   or  generic-identifier
                                   or  module-name
```

C1109a (R1109) The *module-name* shall be the name of a global nonparameterized module or a nonparameterized module module parameter. 251:32+

C1109b (R1109a) The *module-name* shall be the name of a global parameterized module, a parameterized module module parameter, an internal module that is accessed by host association, previously accessed within the same scoping unit by use association, or previously defined within the same scoping unit, or an instance of a parameterized module that is accessed by host association, previously accessed within the same scoping unit by use association, or previously instantiated within the same scoping unit.

C1110b (R1109) An *instance-parameter-spec-list* shall appear if and only if the *module-name* specifies a parameterized module. 251:34+

C1110c (R1110b) The *keyword* = shall not be omitted from an *instance-parameter-spec* unless it is omitted from each preceding *instance-parameter-spec* in the *instance-parameter-spec-list*.

C1110d (R1110b) Each *keyword* shall be the name of a parameter of the module specified by *module-name*.

C1110e (R1109a, R1110b) The *instance-parameter* shall not identify *module-name*, either directly or indirectly.

11.2.4.1 Instantiation of parameterized modules

252:7+

A USE statement in which an *instance-parameter-spec-list* appears creates an **instance** of a parameterized module by substituting entities for corresponding module parameters. The *instance-parameter-spec-list* identifies the correspondence between the instance parameters specified and the parameters of the module. This correspondence may be established either by keyword or by position. If an instance parameter keyword appears, the instance parameter corresponds to the module parameter whose name is the same as the instance parameter keyword. In the absence of an instance parameter keyword, the instance parameter corresponds to the module parameter occupying the corresponding position in the module parameter list; that is, the first instance parameter corresponds to the first module parameter, the second instance parameter corresponds to the second module parameter, etc.

C1115a (R1109) Every instance parameter specified in a USE statement shall correspond with a module parameter of the specified module, and every module parameter of the specified module shall have a corresponding instance parameter.

C1115b (R1109) An instance parameter that corresponds to a data entity module parameter that does not have the INITIALIZATION attribute shall be a variable that has the same characteristics as the characteristics of its corresponding module parameter.

C1115c (R1109) An instance parameter that corresponds to a data entity module parameter that has the INITIALIZATION attribute shall be an initialization expression that has the same characteristics as the characteristics of its corresponding module parameter.

C1115d (R1109) An instance parameter that corresponds to a type module parameter shall at least have components that have the same names and characteristics as the public components of the type module parameter, and shall at least have type-bound procedures and generic bindings that have the same identifiers and characteristics as the public type-bound procedures and generic bindings of the type module parameter.

An instance parameter that corresponds to a type module parameter may have additional components or type-bound procedures or generic bindings. For purposes of correspondence between instance parameters and module parameters, intrinsic operations are considered to be type-bound procedures of intrinsic types.

NOTE 11.8$\frac{1}{2}$

It is possible for a type module parameter to require its corresponding instance parameter to have a generic binding with particular interfaces without requiring its type-bound procedures to have specified names by making the generic binding of the type module parameter public and the type-bound procedures of the generic binding private.

C1115e (R1109) An instance parameter that corresponds to a procedure module parameter shall be a procedure. If the module parameter declaration specifies a function, the corresponding instance parameter shall be a function with the same result type. If the module parameter declaration specifies a subroutine, the corresponding instance parameter shall be a subroutine. If the module parameter has explicit interface, the corresponding instance parameter shall have the same abstract interface.

C1115f (R1109) An instance parameter that corresponds to a generic identifier module parameter shall be a generic identifier. It shall at least have specific procedures with the same abstract interfaces as the specific interfaces specified by the corresponding module parameter. If any part of the module parameter is declared by a nonabstract interface, the names of the specific procedures of the instance parameter shall be the same as the names of the specific procedures of the generic identifier module parameter that have the same abstract interfaces.

An instance parameter that corresponds to a generic identifier module parameter may have additional specific procedures.

C1115g (R1109) An instance parameter that corresponds to a module module parameter shall be a module that has the same interface as the corresponding module parameter.

If the USE statement has a *local-module-name* it creates an instance named by the *local-module-name* but does not access it by use association. The created instance is a module that may be accessed by use association. If the USE statement does not have a *local-module-name* it creates an instance that does not have a name, and accesses it by use association. Since the instance does not have a name, it cannot be referenced by a different USE statement.

11.2.4.2 Use association

[Editor: Replace "The **USE statement**" at 251:9 by "**Use association**". Then move 251:9-17 and Note 11.7 to here.]

A USE statement without an *instance-parameter-spec-list* specifies use association.

C1235a (R1224) The *function-name* shall not be the name of a function that has the ABSTRACT prefix. 279:25+

---

or ABSTRACT 280:3+

C1242a (R1227 A *prefix* shall not specify ABSTRACT unless it is within a *function-stmt* or *subroutine-stmt* that introduces an interface body within an interface block that declares a module parameter (11.2.2). 280:7+

C1247a (R1232) The *subroutine-name* shall not be the name of a subroutine that has the ABSTRACT prefix. 282:10+

[Editor: Insert "module parameters," before "dummy".] 406:5

[Editor: After "module subprogram" insert ", an internal module".] 411:2

[Editor: After "body." insert "An instance of a parameterized module has access via host association to the scoping unit where the parameterized module is defined."] 411:4

---

instance of a parameterized module (11.2.3, 11.2.4.1) A module that is created by substituting entities for a parameterized module's module parameters. 430:35+

---

interface of a parameterized module (11.2.2) : The names of the modules parameters and their characteristics as module parameters. 431:6+

---

internal module (11.2.1) : A module that is defined within another scoping unit. 431:9+

---

parameterized module (11.2.2) : A module whose initial statement has a *module-param-list*. It serves as a template for creating instances by substituting entities for its parameters. 433:3+

C.8.4 Parameterized modules (11.2.2) 477:29+

A parameterized module is a template that may be used to create specific instances by substituting entities for its module parameters.

C.8.4.1 Examples of definition of parameterized modules

C.8.4.1.1 Sort module with < accessed by host association

This is an example of the beginning of a generic sort module in which the < operator with an appropriate interface must be accessed from the scoping unit in which the parameterized module is defined, is intrinsic, is defined via host association, or is bound to the type of its operands. In general, the processor cannot check that one with an appropriate interface is accessible until the module is instantiated. There is no requirement on the parameters of the type module parameter `MyType`. The quality of message announced in the event `MyType` does not have a suitable < operator is less than would be the case if the < operator were defined by a generic identifier module parameter, or explicitly required to be bound to the type of a type module parameter..

```
module Sorting ( MyType )
  type :: MyType
  end type MyType
  ....
```

C.8.4.1.2 Sort module with < specified by generic interface module parameter

The < operator is given by a module parameter. When the module is instantiated, a generic identifier for an interface with a specific consistent with the `less` function interface shown here, shall be provided as an instance parameter.

```
module SortingP ( MyType, Operator(<) )
  type :: MyType
  end type MyType
  interface operator (<)
    pure logical abstract function Less ( A, B ) ! "less" is purely an abstraction
      type(myType), intent(in) :: A, B
    end function Less
  end interface
  ....
```

The ABSTRACT attribute for the `less` function means that the associated instance parameter for `operator(<)` only needs to have a specific with the specified interface, but the name isn't required to be `less`. Indeed, `less` can't be accessed by that name within `SortingP` or by use association from an instance of `SortingP`.

The instance parameter corresponding to `operator(<)` need not have the same generic identifier. For example, if it's `operator(>)` (with the obvious semantics), the instantiated sort routine would sort into reverse order.

C.8.4.1.3 Sort module with < specified by type-bound generic interface

This illustrates a module parameter that is a type that is required to have a particular type-bound generic identifier. The type shall have a type-bound generic identifier with a particular interface, but if entities are declared by reference to the name `MyType` or a local name for it after it is accessed from an instance, the specific type-bound procedure cannot be invoked by name; it can only be accessed by way of the type-bound generic. The `private` attribute does this.

```
module SortingTBP ( MyType )
  type :: MyType
  contains
    procedure(less), private :: Less ! Can't do "foobar%less".  "Less" is only
      ! a handle for the interface for the "operator(<)" generic
    generic operator(<) => Less ! Type shall have this generic operator
  end type MyType
  ! Same explicit interface for "less" as in previous example
  ....
```

C.8.4.1.4 Module with type module parameter having at least a specified component

```
module LinkedLists ( MyType )
  type :: MyType
    type(myType), pointer :: Next! "next" component is required.
    ! Type is allowed to have other components, and TBPs.
  end type MyType
  ....
```

C.8.4.1.5 Module with type module parameter having separately-specified kind parameter

```
module LinkedLists ( MyType, ItsKind )
  type :: MyType(itsKind)
    integer, kind :: itsKind
  end type MyType
  integer, kind :: ItsKind
  ....
```

C.8.4.1.6 BLAS definition used in instantiation examples in C.8.4.2

```
module BLAS ( KIND )
  integer, kind :: KIND
  interface NRM2; module procedure GNRM2; end interface NRM2
  ....
contains
  pure real(kind) function GNRM2 ( Vec )
  ....
```

C.8.4.1.7 Ordinary module with private instance count and internal parameterized module

```
module ModuleWithInternalGeneric
  integer, private :: HowManyInstances
  module InternalGeneric ( MyType )
    ! Instances of InternalGeneric access HowManyInstances by host association
    ....
```

C.8.4.2 Examples of instantiation of parameterized modules

The following subclauses illustrate how to instantiate a parameterized module.

C.8.4.2.1 Instantiating a noninternal parameterized module

Instantiate a noninternal parameterized module `BLAS` with `kind(0.0d0)` and access every public entity from the instance:

```
use BLAS(kind(0.0d0))
```

Instantiate a parameterized module `BLAS` with `kind(0.0d0)` and access only the `GNRM2` function from the instance:

```
use BLAS(kind(0.0d0)), only: GNRM2
```

Instantiate a parameterized module `BLAS` with `kind(0.0d0)` and access only the `GNRM2` function from the instance, with local name DNRM2:

```
use BLAS(kind(0.0d0)), only: DNRM2 => GNRM2
```

C.8.4.2.2 Instantiate within a module, and then use from that module

This is the way to get only one single-precision and only one double-precision instance of BLAS; instantiating them wherever they are needed results in multiple instances. This also illustrates two ways to make generic interfaces using specific procedures in parameterized modules. The first one creates the generic interface from specific procedures accessed from the instances:

```
  module DBLAS
    use BLAS(kind(0.0d0))
  end module DBLAS
  module SBLAS
    use BLAS(kind(0.0e0))
  end module SBLAS
  module B
    use DBLAS, only: DNRM2 => GNRM2
    use SBLAS, only: SNRM2 => GNRM2
    interface NRM2
      module procedure DNRM2, SNRM2
    end interface
  end module B
```

In the second one the parameterized module has the generic interface named NRM2 that includes the GNRM2 specific:

```
  module DBLAS
    use BLAS(kind(0.0d0))
  end module DBLAS
  module SBLAS
    use BLAS(kind(0.0e0))
  end module SBLAS
  module B
    use DBLAS, only: NRM2    ! Generic; GNRM2 specific not accessed
    use SBLAS, only: NRM2, & ! Generic
      &    SNRM2 => GNRM2    ! Specific
  end module B
```

C.8.4.2.3 Instantiate and access twice in one scoping unit, augmenting generic interface

```
  module B
    use BLAS(kind(0.0d0)), only: NRM2    ! Generic; GNRM2 specific not accessed
    use BLAS(kind(0.0e0)), only: NRM2, & ! Generic NRM2 grows here
      &                SNRM2 => GNRM2    ! Specific
  end module B
```

The method in C.8.4.2.2 above might be desirable so as not accidentally to have multiple identical instances of BLAS in different scoping units.

C.8.4.2.4 Instantiate and give the instance a name, then access from it

```
  ! Instantiate BLAS with kind(0.0d0) and call the instance DBLAS, which is
  ! a local module.
  use :: DBLAS => BLAS(kind(0.0d0))
  ! Access GNRM2 from the instance DBLAS and call it DNRM2 here
  use DBLAS, only: DNRM2 => GNRM2
```

C.8.4.2.5 Instantiate two named instances in one module, then use one elsewhere

```
module BlasInstances
  ! Instantiate instances but do not access from them by use association
  use :: DBLAS => BLAS(kind(0.0d0)), SBLAS => BLAS(kind(0.0d0))
end module BlasInstances
module NeedsSBlasNRM2
  use BlasInstances, only: SBLAS  ! gets the SBLAS instance module, not its contents
  use SBLAS, only: SNRM2 => GNRM2 ! Accesses GNRM2 from SBLAS
end module  NeedsSBlasNRM2
```

C.8.4.2.6 Instantiate sort module with generic interface instance parameter

```
type :: OrderedType
  ...
end type OrderedType
interface operator (<)
  pure logical function Less ( A, B )
    type(orderedType), intent(in) :: A, B
  end function Less
end interface
! Notice relaxed statement ordering.
use SortingP(orderedType,operator(<)), only: OrderedTypeQuicksort => Quicksort
....
```

C.8.4.2.7 Instantiate sort module with type-bound Less procedure

```
use SortingTBP(real(kind(0.0d0))), only: DoubleQuicksort => Quicksort
```

Notice that this depends on < being a “type-bound generic” that is bound to the intrinsic double precision type. Here’s one with a user-defined type that has a user-defined type-bound < operator.

```
type MyType
  ! My components here
contains
  procedure, private :: MyLess => Less
  generic operator ( < ) => myLess
end type MyType

use SortingTBP(myType), only: MyTypeQuicksort => Quicksort
```

The interface for `less` is given in C.8.4.1.2. The name of the specific type-bound procedure bound to < need not be `less`.

Notice that the USE statement comes *after* the type definition and the TBP’s function definition.

C.8.4.2.8 Example of consistent type and type-bound procedure

This example illustrates how to create a type with type-and-kind consistent type-bound procedures, for any kind. This cannot be guaranteed by using parameterized types.

```
module SparseMatrices ( Kind )
  integer, kind :: Kind
  type Matrix
    ! Stuff to find nonzero elements...
    real(kind) :: Element
  contains
    procedure :: FrobeniusNorm
    ....
  end type

contains
  subroutine FrobeniusNorm ( TheMatrix, TheNorm )
    type(matrix), intent(in) :: TheMatrix
    real(kind), intent(out) :: TheNorm
    ....
  end subroutine FrobeniusNorm
  ....
end module SparseMatrices

....

use SparseMatrices(selected_real_kind(28,300)), & ! Quad precision
  & only: QuadMatrix_T => Matrix, QuadFrobenius => Frobenius, &
  &       QuadKind => Kind ! Access instance parameter by way of generic parameter

....

type(quadMatrix_t) :: QuadMatrix
real(quadKind) :: TheNorm

....

call quadFrobenius ( quadMatix, theNorm )
```