

# **SystemVerilog**

## **Modports:**

### **a suggestion for enhancement**

version 0.2

# Contents

0. Revision information.....	3
1. Introduction .....	4
2. Highlights of suggested changes .....	5
3. Modport as a connection façade .....	6
3.1. Intent .....	6
3.2. Declaring a modport .....	6
4. Modport connectors and receptacles .....	8
4.1. Modport polarity.....	8
4.2. Modports in Std.1800-2005 .....	8
4.3. Adding modport receptacles to the body of a module .....	9
4.4. Adding modport connectors to the body of an interface .....	10
4.5. Connectors and receptacles in generate constructs.....	10
4.6. Modules and interfaces can both have connectors and receptacles .....	11
4.7. Modports in programs .....	11
5. Alias names in receptacles.....	12
6. Virtual interface variables .....	13
6.1. Virtual interface variables and modports .....	13
6.2. Virtual receptacles and alias names.....	13
7. Generalisation of modport expressions .....	14
7.1. Modport expression to rename a clocking block .....	14
7.2. Modports and <i>import</i> subprograms.....	15
7.3. Modports and <i>export</i> subprograms.....	15
8. Modport inheritance - <i>incomplete</i> .....	17
8.1. Introduction .....	17
8.2. Derived modport declaration .....	17
8.3. Connectors and receptacles of derived modport type .....	18

## 0. Revision information

Version 0.1 Jonathan Bromley 18 May 2007	Initial draft, released for comment.
Version 0.2 Jonathan Bromley 12 June 2007	Minor errata fixes. Make it clear that a virtual interface is a receptacle and therefore must hold a reference to a modport connection. Add alias expressions to virtual interface.modports. (Section 6) Add incomplete suggestion for extensible modports. (Section 8)

# 1. Introduction

The `interface` construct in SystemVerilog provides a means to encapsulate complicated interconnect. However, experience has shown that the definition of interfaces in IEEE Std.1800-2005 fails to meet some real user needs, particularly the creation of parameterised design IP and the abstraction of interconnect architectures. This document suggests a major overhaul of the modports mechanism in SystemVerilog (IEEE Std.1800-2005) to improve its fit with user requirements.

The changes suggested here were motivated by a variety of concerns, of which the most obvious and urgent is the need to find a clean, backward-compatible means to specify configurable port lists on design IP modules. Various fixes to this problem have been proposed, but the reworking of the modport mechanism described here seems to offer a consistent and flexible approach.

A second motivation was the perceived need to provide for hardware designers the same level of flexibility in specifying interconnection as is already available to verification engineers through transaction-level modelling. There are many well-known frameworks for designing transaction-level interconnections, but all depend on the notion of an abstract interface class that captures an interface to a module *without any dependence on the module's contents*. The existing interface and modport mechanism in SystemVerilog has some features that approach this goal, but it falls short in many ways. The most important of these shortcomings is that the view of an interface provided by a modport cannot be abstracted out of the interface for use in other contexts. This suggestion aims to redress that limitation by redefining a modport as an abstract interconnection, in the same way that the shapes and pin layouts of a plug and socket are the definition of a hardware interconnection. The new suggestion makes a modport look like an abstraction of part of the port list of a module, in a way that is completely independent of the module itself. The modport becomes a re-usable connector component. It has always been possible to achieve this with a modport embedded in an interface intended purely as a wrapper for the modport, but that earlier approach is clumsy and is typically intractable for synthesis because it puts the modport inside an extra layer of interface hierarchy.

Throughout the suggestion, care has been taken to ensure that existing 1800-2005 language features will continue to work without change in the new framework. This has been achieved by redefining all the existing behaviour and syntax of modports as a strict subset of the new language features.

A table in the following section highlights the most important and radical of the suggested changes. The order in which these changes have been described, both in the table and in the body of the document, has been chosen to develop the new ideas incrementally. It may well be that this order should be abandoned and replaced with a different exposition, giving priority to conciseness and precision, for any new LRM document.

## 2. Highlights of suggested changes

topic	outline	section
Modport as a new data type	Modports can be declared stand-alone, typically in a package, as a new kind of data type. Modport declarations can have type and value parameters, just like class declarations. Named specialisations of parameterised modport declarations can be created using <code>typedef</code> .	3.2
Distinguish connectors from receptacles	Like hardware plugs and sockets, modports are considered to have dual complementary polarities, <i>connector</i> and <i>receptacle</i> . The declaration of a modport specifies signal directions and subprogram import/export direction from the point of view of the receptacle. The matching connector is derived from it automatically by reversing the direction of all items in the modport. This fits with existing Std.1800-2005 and is only a matter of terminology and clarification.	4
Connectors and receptacles are module items	Given a stand-alone declaration of a modport type, connectors and receptacles of that type can be created as interface or module items. Both connectors and receptacles can appear within generate constructs, making it possible for modules to have ports whose existence and configuration is controlled by parameters. Such ports <i>do not</i> appear in the module's conventional port list, but are ports nonetheless.	4.3, 4.4
Backward compatibility with 1800-2005	The existing (Std.1800-2005) definitions of modports and ports of modport type are redefined as a subset of these suggestions. Modport definition in an interface, which was the only way to create a modport in 1800-2005, is redefined as the combination of a modport's declaration and its use as a modport connector. A modport appearing in a module's port list is redefined to be one of two possible ways to obtain a modport receptacle.	4.2
Modport expression shorthand	Modport expressions can use port connection shorthand such as <code>.*</code> to simplify the creation of modport connectors. Items in a modport connector may be explicitly left un-associated.	4.4
Item renaming across a receptacle	Using new syntax similar to modport expressions, a mechanism is provided to rename modport items across a modport receptacle. A module having a modport receptacle port can then access the items of that port with simple names, instead of the dotted names of the form <code>port_name.item_name</code> that are currently required.	5
Modport expressions generalised	Modport expressions are generalised so that they can be used to rename not only signals, but also subprograms and clocking blocks, across a modport connector.	7
Interface vs. module	The suggested changes sweep away many of the differences between interfaces and modules. Modules can have modport receptacles (ports of modport type), but can also expose modport connectors and therefore can act in almost every way as interfaces. The only remaining distinguishing characteristics of interfaces are that interfaces can be referenced using variables of virtual interface type, and can be made visible in their entirety through a port of interface type. The distinction between module and interface may remain useful for synthesis; in particular, it may be appropriate for synthesis tools to support modport connectors in interfaces but not in modules.	4.6

## 3. Modport as a connection façade

A SystemVerilog `interface` can be thought of as an encapsulation of all the machinery needed to link some modules together. On such an interface, a `modport` provides a view of the interface that is exported by the interface so that a module may connect to it.

This idea of a modport as a view or aspect of an interface is sufficiently useful in its own right that modports should be capable of declaration as a new kind of data type. A modport declaration does not in itself create or define any data objects, but merely provides a view of a set of objects. The existing (Std.1800-2005) usage and meaning of `modport` is redefined as a subset of the new definition, and is unaffected.

### 3.1. Intent

A modport declaration specifies a connection façade.

It may be helpful to think of a modport declaration as an abstraction of part of the port list of a module. Alternatively, it may be appropriate to think of it as the data-sheet specification of the mating face of a connector. In the world of hardware, the mating face of a connector defines a connection façade in the following ways:

- It has a well-defined structure that is independent of the equipment using it, so that it can be re-used (possibly for diverse purposes) on many different items of equipment.
- Each terminal of the connector has specified electrical characteristics (data type) and may, optionally, be given a direction (input, output, inout).
- Each terminal of the connector has a name defined by the connector's specification. This name is independent of name of the internal signal that is made available on that terminal, although of course the electrical characteristics of the internal signal must conform to the connector's specification.

In hardware, connectors can be specified in this way without any reference to the equipment or cable harness of which they form part. The suggested new form of modport declaration provides SystemVerilog with the same capability to specify an abstract connector, without regard to the details of any module or interface that may use one or more such connector.

### 3.2. Declaring a modport

A `modport` may be declared in a package or in an interface. Typically, it is declared in a package, in which case it represents an abstract connection façade as described in section 3.1 above. A modport declaration may occur in an interface, in which case special rules apply to maintain compatibility with Std.1800-2005.

The new stand-alone declaration syntax closely follows the existing syntax for declaring a modport within an interface, except that each modport item must have its data type specified. Note how this differs from modports in Std.1800-2005 interfaces; the modport items in such modports are invariably references to things of known data type that already exist within the interface. In an abstract (stand-alone) modport declaration the modport items cannot refer to an existing thing, and therefore they must be explicitly typed.

A modport declaration may optionally have type parameters and value parameters. These parameters act in much the same way as parameters associated with a class declaration, in the sense that they may be overridden to create a specialisation of the modport. Parameters of the modport may be used in the *modport\_typed\_items*, for example to specify bit widths, item data types, or the argument types of subprograms.

There shall be at least one *modport\_typed\_item* in a modport declaration. A *modport\_typed\_item* can be a list of port connections, import/export subprogram prototypes, or clocking blocks. Subprogram and clocking items are discussed more fully in section 7.

```
modport_declaration ::=
    modport identifier
        [ parameter_port_list ]
        ( modport_typed_item { , modport_typed_item } );
```

A modport declaration creates a new data type, and therefore may be used together with `typedef`. This combination can provide a convenient way to create named specialisations of a parameterised modport.

#### Examples:

```
package P_3_2;
    // Modport suitable for connection to an RS-232 serial link
    modport rs_232 (
        input logic RXD, DSR, CTS, DCD,
        output logic TXD, RTS, DTR );
    // Modport representing a general-purpose test point of any type
    modport testpoint #(parameter type T = logic) (output T TP);
    // Specialisation of testpoint for data type "byte"
    typedef testpoint #(byte) byte_TP;
endpackage : P_3_2
```

The `typedef` at the end of the example above is equivalent to a modport declaration

```
modport byte_TP (output byte TP);
```

## 4. Modport connectors and receptacles

Given the declaration of an abstract modport, it becomes possible to equip any module or interface with one or more connections of the type specified by such a modport. These connections may form part of the port list. Alternatively, they may appear in the body of the module, thereby providing additional external connectivity that is not specified in the module's regular port list. A module that has such connections may or may not have a regular port list.

### 4.1. Modport polarity

Modport items have a direction, loosely corresponding to male and female terminals on a physical (hardware) connector. Each modport declaration, therefore, can be thought of as defining both a plug and a socket. Given one form of the connector, the other form is implicit. Only one form is defined, and its complementary mating form follows inevitably from that definition.

The relationship between the two polarities of modport is slightly different than the plug/socket relationship found in hardware. A modport declaration defines the polarity suitable for inclusion in a module's port list. We describe such a modport as a *receptacle*. If a module has such a receptacle, either in its port list or in its body, the receptacle represents an obligation that must be satisfied by binding the receptacle during elaboration to a suitable instance of a *connector*, which has the opposite polarity of the modport.

Modport *receptacles*, then, have item directions exactly as they appear in the modport's declaration. Modport *connectors* automatically have the opposite item directions. In the next few sub-sections we will see how to create modport receptacles and connectors. First we will examine how this was achieved in the original SystemVerilog standard, and then we will define extensions that allow a more flexible approach.

### 4.2. Modports in Std.1800-2005

The terminology outlined in the previous sub-clause can be applied to the existing SystemVerilog interface and modport mechanism. In the example below, modport `mp` defined in the interface is a *modport connector*, whereas port `P` in the module's port list is a *modport receptacle*. In instance `inst_m` of the module, the obligation imposed by modport receptacle `P` is met by binding it to instance `inst_i.mp` of the appropriate modport connector. Note that, in this case, the `modport` statement within interface `I_4_2` both declares the modport `mp` and creates a modport connector with the same name. This backward compatibility mechanism is discussed in more detail in a later section.

```

interface I_4_2;
    logic L;
    modport mp(output L);
    // declares modport type mp, creates connector mp of type mp
endinterface : I_4_2

module M_4_2 (I_4_2.mp P); // port list has modport receptacle P
    assign P.L = 0;
endmodule : M_4_2

module top_4_2;
    I_4_2 inst_i(); // instance of interface, with its connector
    M_4_2 inst_m(.P(inst_i.mp)); // bind connector to receptacle
endmodule : top_4_2

```

### 4.3. Adding modport receptacles to the body of a module

Modport receptacles may appear as part of a module's port list. Alternatively they may appear in the body of the module, thereby providing additional external connectivity that is not specified in the module's regular port list. A module that has such connections may or may not have a regular port list.

The examples and discussion in the previous sub-section considered only the modport receptacles specified in a module's port list. Modport receptacles may also appear as module items in the body of a module. However, modports that were defined in the body of an interface cannot be used in this way. Only abstract modports (those having stand-alone declaration, outside any interface) may be used as module items.

The syntax for adding a modport receptacle as a module item requires the use of the prefix `interface.` before the modport type name. The use of this prefix ensures that there is no confusion between a modport receptacle and a modport connector (described in the next section).

Modport receptacles in the body of a module have the same status as modport receptacles appearing in the module's port list. In particular, they must be bound to a modport connector instance (of the same modport type) when the module is instanced.

```

package P_4_3;
  modport mp_4_3(input logic R, output logic W);
endpackage : P_4_3

module M_4_3 (input bit clk);
  import P_4_3::*; // makes declaration of mp_4_3 visible
  interface mp_4_3 rw_recep; // creates a port named rw_recep
    always_ff @(posedge clk) rw_recep.W <= rw_recep.R;
  endmodule : M_4_3

```

#### 4.4. Adding modport connectors to the body of an interface

A modport that has been declared stand-alone can be used to create a modport connector in an interface, exposing items in the interface through the connector. When the interface is instantiated, such a modport connector can then be used to meet the obligation represented by a modport receptacle in an instance of some other module.

Every item in the modport connector must be associated with an expression of equivalent type in the enclosing interface. Each modport item merely provides visibility of a data object in the enclosing interface. The connector does not create any new data objects for its modport items.

Items of the modport connector are associated with expressions using the same syntax as port connection lists on a module instance. The connection shorthand *.name* is acceptable as an abbreviation for the association *.name(name)*, and the connection shorthand *.\** is acceptable for wildcard association of all items not otherwise associated.

Items can be explicitly specified to have no association by using the empty association syntax *.name()*.

```

interface I_4_4();
  import P_4_3::*; // makes declaration of mp_4_3 visible
  logic R, W, Lire, Ecrire; // data objects in the interface
  mp_4_3 rw_conn1( .W(Ecrire), .R ); // .R(R) is implied
  mp_4_3 rw_conn2( .*, .R(Lire) ); // .W(W) is implied
endinterface : I_4_4

```

#### 4.5. Connectors and receptacles in generate constructs

Std.1800-2005 already permits modport constructs in an interface to appear within a generate construct. Similarly, in this suggestion we permit modport connectors to appear within a generate. Thus it is possible to create arrays of connectors, and optional connectors.

This document allows for modport receptacles to appear as a modport item, outside the module's port list but nevertheless providing external connectivity for the module. We also allow a modport receptacle to appear in a generate construct. This mechanism makes it possible for a module to have external connectivity (a set of ports) that is controlled by the module's parameters. Alias expressions in modport receptacles (described in section 5) provide a means to ascribe different names to the modport items on each receptacle in a generated array.

#### **4.6. Modules and interfaces can both have connectors and receptacles**

The discussion so far has assumed that modport receptacles will provide ports on a module, and modport connectors will expose internal signals, subprograms and clocking blocks of an interface. However, the converse can also be true. Modules may have modport connectors, and interfaces may have modport receptacles. The only restriction is that every receptacle must be bound at elaboration to a matching connector.

For synthesis, however, it may be preferable to add the restriction that only interfaces can have modport connectors. This restriction means that it remains possible for synthesis tools to synthesise the contents of an interface instance as if those contents were to appear directly in the instance's enclosing module, just as they do at present. Such synthesis restrictions of course do not form part of the language specification.

#### **4.7. Modports in programs**

Programs may have modport receptacles, but shall not have modport connectors. Modport receptacles in a program must be of a modport type that has no exported subprograms. These two rules make it possible for programs to use virtual interfaces and to have static connections to modport connectors, while preserving the existing restriction that there can be no access to the internal contents of a program from outside that program. See section 6 for further discussion of virtual interfaces and modports.

## 5. Alias names in receptacles

When a module (or interface or program) has a port that is a modport receptacle, all items of the modport are visible within the module. To make reference to any such item, code in the module must use a two-part dotted name of the form *port\_name.item\_name*. Clearly this is clumsy if such an item is referenced from many places within the module. It also makes it troublesome and error-prone to convert a legacy Verilog module to replace some or all of its ports with a modport receptacle. Furthermore, the modport's item names are not necessarily the most appropriate names for use within the code of the module itself.

To alleviate these difficulties we add *alias names* to modport receptacles. They allow any item of the modport receptacle to be renamed to an appropriate simple name within the module. The presence or absence of such alias names has no effect on the modport's semantics.

The following example is functionally equivalent to that of section 4.3, but uses an alias expression on the module's modport receptacle. Note that modport `mp_4_3` is unchanged from that earlier example.

```
package P_4_3;
    modport mp_4_3(input logic R, output logic W);
endpackage : P_4_3

module M_5 (input bit clk);
    import P_4_3::*;
    interface mp_4_3 rw_recep( .R(Lire), .W(W) );
    always_ff @(posedge clk)
        W <= Lire; // equivalent to rw_recep.W <= rw_recep.R;
endmodule : M_5
```

Shorthand forms of named association syntax also work in this context. For any given item,

```
receptacle_name( .item_name )
```

is equivalent to:

```
receptacle_name( .item_name(item_name) )
```

Wildcard association `.*` can also be used to provide implicit aliases for all items whose association is not otherwise specified. An empty association `.item_name()` can be used to suppress wildcard association explicitly for individual items.

Regardless of what form of association is used, it shall be an error if any name created by this association is the same as another identifier appearing anywhere in the module scope.

## 6. Virtual interface variables

### 6.1. Virtual interface variables and modports

A virtual interface is a variable that can hold a reference to an interface or modport instance. Virtual interfaces of interface type (without a modport specification) are unaffected by this suggestion, and as in Std.1800-2005 they hold a reference to an existing interface instance. However, this suggestion has an impact on virtual interface variables whose declarations are qualified with a modport type.

A virtual interface variable that is declared with the keyword `interface` followed by a modport type name, thus:

```
virtual interface.modport_name variable_name;
```

holds either the special value `null` or a reference to an instance of a modport connector. The referenced connector instance must be within an interface, not within a module.

A virtual interface variable of this kind is effectively a modport receptacle whose binding can be set at runtime. In the remainder of this section, such a variable is called a *virtual receptacle* to avoid any confusion with the traditional form of virtual interface.

### 6.2. Virtual receptacles and alias names

Just like static modport receptacles, virtual receptacles can have alias names. In the absence of alias names, items in the receptacle can be accessed using a two-part dotted name as usual.

## 7. Generalisation of modport expressions

In Std.1800-2005, a modport in an interface can have *modport expressions*. They allow the named modport item to be mapped to any appropriate expression. In this way, signals seen via the modport can have names that are appropriate to the connection, whilst giving access to signals or expressions in the interface that are appropriate to their implementation.

Unfortunately, this useful and powerful feature is limited only to members of the modport that are signals, and is not applicable to subprograms or clocking blocks visible through the modport. Here we attempt to lift those restrictions.

We suggest an extension of the modport expression mechanism to subprograms and clocking blocks so that they, too, can be given different names from the point of view of a user module than they have within the interface itself. The same generalisations apply also to alias expressions on a modport receptacle.

### 7.1. Modport expression to rename a clocking block

Consider a modport declaration having an item of type `clocking`. In a modport connector, that item can be mapped on to an actual clocking block in the connector's enclosing module or interface. In a modport receptacle, the item can be given a local simple name so that a two-level dotted reference is not required.

```

package P_6;
    modport MP_6 ( clocking mp_cb );
endpackage : P_6

interface I_6_1 ( input bit clk );
    import P_6::*;
    clocking intf_cb @(posedge clk);
    ...
    endclocking : intf_cb
    MP_6 mp ( .mp_cb(intf_cb) );
endinterface : I_6_1;

program PR_6_1;
    import P_6::*;
    interface.MP_6 mp( .mp_cb(synch) ); // "synch" is an alias name
    initial begin
        @(synch); // equivalent to @(mp.mp_cb)
        ...
    endprogram : PR_6_1

```

## 7.2. Modports and *import* subprograms

An `import` task or function in a modport provides access, through a modport receptacle in a client module, to a subprogram declared in the module or interface enclosing the modport connector.

Standalone modport declarations (i.e. those in a package) must fully specify the prototype of any imported subprogram item:

```
package P_6_2;
  modport MP_6_2 ( import function int MP_F ( input byte B ) );
endpackage : P_6_2
```

When such a modport is used to create a connector, the enclosing interface must provide a suitable subprogram. It is not necessary for the actual subprogram to have the same name as the corresponding modport item, because the name can be mapped through the extended form of modport expression:

```
interface I_6_2;
  import P_6_2::*;
  function int f ( input byte B );
  ...
endfunction : f
MP_6_2 f_conn ( .MP_F(f) ); // expose function f() to modport
endinterface : I_6_2
```

Similarly, the modport receptacle in a client module can have an alias expression to re-name the imported subprogram:

```
module M_6_2;
  import P_6_2::*;
  interface.MP_6_2 f_recep ( .MP_F(g) );
  // receptacle provides function g() to this module
  initial
    $display( g(8'h55) );
endmodule : M_6_2
```

## 7.3. Modports and *export* subprograms

`export` subprograms are a special case because they provide visibility in the opposite direction to all other modport items. For variables, nets, clocking blocks and imported subprograms, a module containing the receptacle is given access to something in the interface that contains the connector. For export subprogram items, the relationship is reversed: an interface containing the connector is given access to a subprogram in the module containing the receptacle. For these

items, therefore, the rules for imported subprograms apply, but with the roles of receptacle and connector exchanged. The following example adds an `export` subprogram to the example of section 7.2. In this example, the task call `t(y)` in interface `I_6_3` is redirected, through the modport, to task `foo()` in an instance of the module `M_6_3`. Note that, thanks to the renaming provided by extended modport expressions, it is no longer necessary for the task `foo` (exported from module `M_6_3` through modport `MP_6_3`) to have a two-part dotted name.

```

package P_6_3;
  modport MP_6_3 (
    import function int MP_F ( input byte B ),
    export task MP_T (ref int X ) );
endpackage : P_6_3

interface I_6_3;
  import P_6_3::*;
  function int f ( input byte B );
  ...
endfunction : f
MP_6_3 f_conn ( .MP_F(f), .MP_T t );
initial begin
  int y = 5;
  t(y);
end
endinterface : I_6_3

module M_6_3;
  import P_6_3::*;
  task foo ( ref int R); ...; endtask
  interface.MP_6_3 f_recep ( .MP_F(g), .MP_T(foo) );
  initial $display( g(8'h55) );
endmodule : M_6_3

```

## 8. Modport inheritance - *incomplete*

### 8.1. Introduction

This section is incomplete and may have to be abandoned. The idea of deriving a more complicated modport definition from a simpler one by something akin to class inheritance is superficially attractive, but it is hard to see how it would work out in practice. For it to be useful, we need something like up- and down-casting.

A module with a derived modport receptacle needs to be able to connect to an interface with a base modport connector. Similarly, we need to go the other way: a legacy module with a base modport receptacle needs to be able to connect to a newer interface with a derived modport connector. In both cases there is a risk that signals might remain undriven, and tasks or functions unimplemented. It might be possible to work around this if a derived modport were to supply default values for its terminals, and default return values for its functions; but this solution seems complicated and error-prone.

Consequently, this section merely suggests an extension mechanism so that new modport declarations can be based on existing ones, thereby avoiding cut-and-paste in some situations. Section 8.3 therefore insists that every modport receptacle must be bound to a modport connector instance of exactly the same type, forbidding any kind of up- or down-casting of modports. This is a very un-ambitious goal!

### 8.2. Derived modport declaration

A modport may be declared as an extension of an existing modport using similar syntax to class inheritance. For example, the APB (AMBA Peripheral Bus) specification exists in two variants: the original form APB2, and the extended form APB3 which adds two further signals to the bus. Modports to represent these two standards could be created thus:

```

modport mp_APB2_master          // Original form
#(parameter Abits = 16, Dbits = 16)
(
    input  bit    PCLK,
    output logic PSEL, PENABLE, PWRITE,
    output logic [Abits-1:0] PADDR,
    output logic [Dbits-1:0] PWDATA,
    input  logic [Dbits-1:0] PRDATA
);

modport mp_APB3_master extends mp_APB2_master
#(parameter bit dummy = 1'b0)
( input logic PREADY, PSLVERR );

```

The new, extended modport definition has all the parameters and items of the existing base definition, but also has the specified additional parameters and items. In the example above, parameter `dummy` is not useful, but has been included to illustrate how the parameter list may be extended.

### 8.3. Connectors and receptacles of derived modport type

The new, derived modport type behaves in every way as if it were a standard (base) modport type. In particular, every receptacle must be bound to a connector of exactly the same modport type.