

## 6. e Ports 1

This clause describes ports, an *e* unit member that enhances the portability and inter-operability of verification environments by making separation between an *e* unit and its interface possible. 5

### 6.1 Introduction to e Ports

A port is an *e* unit member that makes a connection between an *e* unit and its interface to another internal or external entity. There are two ways to use ports: 10

- Internal ports (*e2e* ports) connect an *e* unit to another *e* unit.
- External ports connect an *e* unit to a simulated object. 15

External ports are a generic way to access simulated objects of various kinds. An external port is bound to a simulated object, e.g., an HDL signal in the DUT. Then all access to that signal is made via the port. The port can be used to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. 20

NOTE—A *simulator* is any hardware or software agent that runs in parallel with an *e* program, and models the behavior of any part of the design under test (DUT) or its environment. 25

#### 6.1.1 Advantages of using ports

Although previous HDL access mechanisms are still supported, ports have the following advantages over the old access mechanisms: 25

- Ports support modularity and encapsulation by explicitly declaring interfaces to *e* units.
- They are typed. 30
- They improve performance of accessing DUT objects with configurable names.
- They can pass not only single values but also other kinds of information, such as events and queues.
- They can be accompanied in *e* with generic or simulator-specific attributes which can be used to specify information needed for enhanced access to DUT objects.
- They are suitable for use with a publicly available procedural External Simulator Interface (ESI).
- Some new simulator interfaces, such as SystemC, require the use of ports. 35

#### 6.1.2 Creating port instances

A port type is defined by three aspects: 40

- a) the kind of port, either simple port, buffer port, or event port:
  - 1) Simple ports access data directly.
  - 2) Buffer ports implement an abstraction of queues with blocking get and put.
  - 3) Event ports transfer events between *e* units or between an *e* unit and a simulator. 45
- b) direction, either input or output (or inout for simple and event ports)
- c) data element, the *e* type that can be passed through this port.

Ports can only be instantiated within units using a unique instance name and the port type (direction, port kind, and a kind-specific type specifier). Like units, port instances are generated during pre-run generation and cannot be created, modified, or removed during a run. 50

The generic syntax for ports is:

```
port-instance-name: [direction] port-kind of [type-specifier] is instance;
```

55

1 Event ports do not have a type specifier.

### Examples

5 The following unit member creates a port instance:

```
data_in: in buffer_port of packet is instance;
```

10 where:

- The port instance name is `data_in`.
- The port kind is a buffer port.
- The port direction is input.
- The data element the port accepts is “packet”.

As another example, the following line creates a list of simple ports which each pass data of type bit:

```
ports: list of simple_port of bit is instance;
```

20

### 6.1.3 Using ports

A port’s behavior is influenced by port attributes, such as `hdl_path()` or `bind()`, which are applied to port instances using pre-run generation **keep** constraints. For example, the following lines of code create a port named “data” and connect (bind) it to an external simulator-related object whose HDL pathname is “data”.

25

```
data: inout simple_port of list of bit is instance;
keep bind(data, external);
keep data.hdl_path() == "data";
```

30

Each port kind has predefined methods that can be used to access the port values. For example, buffer ports have a predefined method `put()`, which writes a value onto an output port:

35

```
data_out: out buffer_port of cell is instance;
drive_all() @sys.any is {
var stimuli: cell;
var counter: int=0;
while counter < cells {
wait [1]*cycle;
gen stimuli;
data_out.put(stimuli);
counter+=1;
};
};
```

40

45

### 6.2 Using simple ports

Simple ports can be used to transfer one data element at a time to or from an external simulated object, such as a Verilog register, a VHDL signal, or a SystemC method, or an internal object (another *e* unit). A simple port’s direction can be either input, output, or inout.

50

Internal simple ports can transfer data elements of any type. External ports can transfer scalar types and lists of scalar types, including MVL data elements. Structs or lists of struct cannot be passed through external simple ports.

55

Use the \$ port access operator to read or write port values. To access multi-value logic (MVL) on simple ports, either declare a port's data element to be mvl or list of mvl, or use the MVL methods. See 6.2.1 and 6.2.2 for more information.

Internal and external ports shall have a bind() attribute that defines how they are connected. In addition, the delayed() attribute can be used to control whether new values are propagated immediately or at the next tick.

An external simple port must have an hdl\_path() attribute to specify the name of the object to which it is connected. In addition, an external simple port can have several additional attributes that enable continuous driving of external signals. See 6.6 for more information on attributes for simple ports.

### 6.2.1 Accessing simple ports and their values

Ports are containers and the values they hold are separate entities from the port itself. The \$ access operator distinguishes port value expressions from port reference expressions.

The \$ access operator, e.g., p\$, can be used to access or update the value held in a simple port p. When used on the right-hand side, p\$ refers to the port's value. On the left-hand side of an assignment, p\$ refers to the value's location, so an assignment to p\$ changes the value held in the port.

Without the \$ operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the \$ operator can be used for operations involving port references.

The \$ access operator cannot be applied to an item of abstract type, such as **any\_simple\_port**. This type does not have any access methods.

#### Examples

**\*\*Let's only show a few example of each 'access' here (say, 6 total, not 18);  
which ones should we keep??**

#### Accessing port values

print p\$;                      Prints the value of a simple port, p.

NOTE—Compare with “print p”, which prints information about port p.

p\$ = 0;                        Assigns the value 0 to a simple port, p.

NOTE—Compare with “pref = NULL”, which modifies a port reference so that it does not point to any port instance.

force p\$ = 0;                Forces a simple external port to 0.

print q\$[1:0];               Prints the two least-significant bits of the value of q.

print q\$[2:2];               Prints the third least-significant bit of the value of q.

print sys.pp\$;               Prints the value of port sys.pp.

```

1      print sys.plist[0]$.          Prints the value of port plist[0] from a list of ports, plist.
      print blist${0..1};          Prints the first two elements of a list value. blist is defined as:
                                   blist: in simple_port of list of bit is instance;
5      print listbl[0]${1};        Prints the second bit in a list value of the first element in a list of ports.
                                   Could be written (listbl[0])${1}. listbl is defined as:
                                   listbl: list of in simple_port of list of bit is
                                   instance;
10

```

144

### Accessing a port

```

15      print p;                    Prints the information about port p. Port p is defined as:
                                   p: simple_port of int (bits:8) is
                                   instance;
      // p = 5;                    An error, as it is an attempt to assign incompatible types.
      keep q == p;                 q refers to the port instance p. Port reference q is defined as:
                                   !q: simple_port of int (bits:8);
20      r = q;                      Port reference r refers to the port instance p too. It is defined as:
                                   var r: any_simple_port;
      keep plist.size() == 3;      plist is defined as:
                                   plist: list of in simple_port of int
                                   (bits:8) is instance;
25      keep plist[0] == p;         plist[0] refers to the port instance p.
      keep plist[1] == p2;        plist[1] refers to the port instance p2. p2 is defined as:
                                   p2: simple_port of int (bits:8) is
                                   instance;
30      keep plist[2] == q;        plist[2] refers to the port instance p (because of q).
35

```

### 6.2.2 Multi-Value Logic (MVL) on simple ports

There are two ways to read and write multi-value logic on simple ports:

- Define numeric ports (uint, int, and so on) and use the predefined MVL methods described in 6.9 to read and write values to the port.
- Define ports of type **mvl** or list of **mvl** and use the \$ access operator to read and write values to the port.

Ports of type **mvl** or list of **mvl** (MVL ports) allow easy transformation between exact *e* values and multi-value logic, which is useful for communicating with objects that sometimes model bit values other than 0 or 1 during a test. Otherwise, using numeric ports is preferable, since numeric ports allow keeping the port values in a bit-by-bit representation, while MVL ports require having an *e* list for a multi-value logic vector.

The enumerated type **mvl** is defined as:

```
type mvl: [MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N]
```

177

The Verilog comparison operators (=== or !==) cannot be used with numeric ports or MVL ports. These operators can be used only with the tick access syntax.

Some simulators do not need to support all the potential MVL values. All nine values are supported only for VHDL simulations. For Verilog simulations, only four values (MVL\_U, MVL\_X, MVL\_0, MVL\_1) are supported.

### Examples

This example shows how tick access notation translates to MVL methods, assuming the following numeric port declaration:

```
data: inout simple_port of int is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";
d: int;

d = 'data';           d = data$;

'data' = 32'bz;       data.put_mvl_list(32'bz);

Check that 'data@x' == 0;   Check that data.get_mvl_list().has(it == MVL_X) == FALSE;
                          Check that data.has_x() == FALSE;

d = 'data[31:10]@z';       d = mvl_to_int(data.get_mvl_list(), {MVL_Z})[31:0];
```

This example shows how tick access notation translates to use of an MVL port, assuming the following MVL port declaration:

```
data: inout simple_port of list of mvl is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";

Check that 'data@x' == 0;   Check that data$.has(it == MVL_X) == FALSE;
                          Check that data.has_x() == FALSE;

'data' = 32'bz;           data$ = 32'bz;
```

### 6.2.3 @sim temporal expressions with external simple ports

Specify an event port causes *e* to be sensitive to the corresponding HDL signal during the entire simulation session. This might result in some unnecessary runtime performance cost if *e* only needs to be sensitive in certain scenarios. In such cases, use an external simple port in temporal expressions with @sim instead. The syntax is:

```
[change|rise|fall](simple-port$)@sim;
```

Typically, this syntax is used in wait actions.

#### Example

```
transaction_complete: in simple_port of bit is instance;
  keep bind(transaction_complete, external);
write_transaction(data: list of byte) @clk$ is {
  //...
  data_port$ = data;
  wait rise(transaction_complete$)@sim;
};
```

Trying to apply the `@sim` operator to a bound internal port shall cause an error when the corresponding temporal expression is evaluated, which occurs at runtime.

### 6.3 Using buffer ports

Buffer ports can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write access to the port is blocked. When the queue is empty, read access to the port is blocked. The queue size is fixed during generation by the `buffer_size()` attribute and cannot be changed at runtime. The queue size can be set to 0 for rendezvous ports. See 6.6.2.2 and 6.3.1 for more information.

A buffer port's direction can be either input or output. Internal buffer ports can transfer data elements of any type. Use the buffer port's predefined `get()` and `put()` methods to read or write port values. These methods are time-consuming methods (*TCMs*). The `$` port access operator cannot be used with buffer ports.

Buffer ports shall have a `bind()` attribute that defines how they are connected. In addition, the `delayed()` attribute can be used to control whether new values are propagated immediately or at the next tick. The `pass_by_pointer()` attribute controls how data elements of composite type are passed. See 6.6 for more information on these attributes.

#### 6.3.1 Rendezvous-zero size buffer queue

In rendezvous-style handshaking protocol, access to a port is blocked after each `put()` until a subsequent `get()` is performed, and access is blocked after each `get()` until a subsequent `put()` is performed.

This style of communication is easily achieved by using buffer ports with a data queue size of 0. The following example shows how this is done.

#### 6.3.2 Example

```

unit consumer {
    in_p: in buffer_port of atm_cell is instance;
};

unit producer {
    out_p: out buffer_port of atm_cell is instance;
};

extend sys {
    consumer: consumer is instance;
    producer: producer is instance;
    keep bind(producer.out_p, consumer.in_p);
    keep producer.out_p.buffer_size() == 0;
};

```

### 6.4 Using event ports

Event ports can be used to transfer events between two *e* units or between an *e* unit and an external object. An internal event port's direction can be either input, output, or inout. Use the `$` port access operator to read or write port values. See 6.4.1 for more information.

Internal and external ports need to have a `bind()` attribute that defines how they are connected. An external port needs to have an `hdl_path()` attribute to specify the name of the object to which it is connected. The `edge()` attribute for an external input event port specifies the edge on which an event is generated.

See 6.6 for more information on these attributes. 1

### 6.4.1 Accessing event ports 5

Use the \$ access operator to access the event associated with an event port. An expression of type event\_port without the '\$' operator refers to the port itself and not to its event.

### 6.4.2 Example 10

This example shows how to connect event ports (using a **bind()** constraint) and use the \$ operator to access event ports in event contexts.

```

unit u1 {
    in_ep: in event_port is instance;
    tcml()@in_ep$ is {
        // ...
    };
};

unit u2 {
    out_ep: out event_port is instance;
    event clk is @sys.any;
    counter: uint;
    on clk {
        counter = counter + 1;
        if counter %10 == 0 {
            emit out_ep$
        };
    };
};

extend sys {
    u1: u1 is instance;
    u2: u2 is instance;
    keep bind(u1.in_ep,u2.out_ep);
};

```

## 6.5 Defining and referencing ports 146

This section covers the following topics: 40

- simple\_port
- buffer\_port
- event\_port
- any\_simple\_port, any\_buffer\_port, any\_event\_port
- port\$ 45

50

55

### 6.5.1 simple\_port

<b>Purpose</b>	Access other port instances or external simulated objects directly	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>port-instance-name</i> : [ <b>list of</b> ] [ <i>direction</i> ] <b>simple_port of element-type is instance</b> ;	
<b>Parameters</b>	<i>port-instance-name</i>	A unique identifier used to reference the port or access its value.
	<i>direction</i>	One of <b>in</b> , <b>out</b> , or <b>inout</b> . The default is <b>inout</b> , which means values can be read from and written to this port. For an <b>in</b> port, values can only be read from the port; for an <b>out</b> port, values can only be written to the port.
	<i>element-type</i>	Any predefined or user-defined <i>e</i> type, except a port type or unit type.

Simple ports can be used to transfer one data element at a time to or from an external simulated object or internal object (another *e* unit).

Internal simple ports can transfer data elements of any type. External ports can transfer scalar types and lists of scalar types, including MVL data elements. Structs or lists of struct cannot be passed through external simple ports.

The port can be configured to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. Binding is fixed during generation.

A simple port's direction can be either input, output, or inout. The direction specifier in a simple port is not a **when** subtype determinant. This means, for example, that the following type:

```
data: simple_port of byte is instance;
```

is *not* the base type of:

```
data: out simple_port of byte is instance;
```

Furthermore, the following types are fully equivalent:

```
data: simple_port of byte is instance;
data: inout simple_port of byte is instance;
```

Thus, the following constraint is an error because the two types are not equivalent:

```
data: out simple_port of byte is instance;
!data_ref: simple_port of byte; // means inout simple_port of byte
keep data_ref == data; // error
```

Syntax example:

```
data: in simple_port of byte is instance;
```

### 6.5.2 buffer\_port

<b>Purpose</b>	Implement an abstraction of queues with blocking get and put
<b>Category</b>	Unit member
<b>Syntax</b>	<i>port-instance-name</i> : <b>[list of] direction buffer_port of element-type is instance;</b>
<b>Parameters</b>	<i>port-instance-name</i> A unique identifier used to reference the port or access its value.
	<i>direction</i> One of <b>in</b> or <b>out</b> . There is no default. For an <b>in</b> port, values can only be read from the port; for an <b>out</b> port, values can only be written to the port. See 6.8 for information on how to read and write buffer ports.
	<i>element-type</i> Any predefined or user-defined <i>e</i> type, except a port type or a unit type.

Buffer ports can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write access to the port is blocked. When the queue is empty, read access to the port is blocked.

The queue size is fixed during generation by the **buffer\_size()** attribute and cannot be changed at runtime. The queue size can be set to 0 for rendezvous ports.

Use the buffer port's predefined **get()** and **put()** methods to read or write port values. These methods are time-consuming methods (*TCMs*). The \$ port access operator cannot be used with buffer ports.

A typical usage of a buffer port is in a *producer* and *consumer* protocol, where one object puts data on an output port at possibly irregular intervals and another object with the corresponding input port reads the data at its own rate.

Syntax example:

```
rq: in buffer_port of bool is instance;
```

### 6.5.3 event\_port

<b>Purpose</b>	Transfer events between units or between simulators and units
<b>Category</b>	Unit member
<b>Syntax</b>	<i>event-port-field-name</i> : <b>[list of] [direction] event_port is instance;</b>
<b>Parameters</b>	<i>event-port-field-name</i> A unique identifier used to reference the port or access its value.
	<i>direction</i> One of <b>in</b> , <b>out</b> , or <b>inout</b> . The default is <b>inout</b> , which means events can be emitted and sampled on the port. For a port with direction <b>in</b> , events can only be sampled. For a port with direction <b>out</b> , events can only be emitted.

Event ports can be used to transfer events between two *e* units or between an *e* unit and an external object. Use the \$ port access operator to read or write port values. See 6.4.1 for more information.

An internal event port's direction specifier can be either input, output, or inout. The direction specifier is not a **when** subtype determinant. This means, for example, that the following type

```
clk: event_port is instance;
```

is **not** the base type of

```
clk: out event_port is instance;
```

Furthermore, the following types are fully equivalent:

```
clk: event_port is instance;
clk: inout event_port is instance;
```

In addition, the following are not allowed:

- using the **on** struct member for event ports
- coverage on event ports
- specifying a temporal formula (like `event_port is ...`) for definition of an out event port.

It is possible, however, to define an additional event and connect it to the event port, and thus be able to bypass these restrictions, e.g., **\*\*Verify this rephrasing\*\***

```
ep: in event_port is instance;
keep bind(ep, external);
event e is @ep$;
```

Syntax example:

```
clk: in event_port is instance;
```

#### 6.5.4 any\_simple\_port, any\_buffer\_port, any\_event\_port

<b>Purpose</b>	Reference a port instance
<b>Category</b>	Unit field, variable, or method parameter
<b>Syntax</b>	[!   var] <i>port-reference-name</i> : [ <i>direction</i> ] <i>port-kind</i> [of <i>element-type</i> ] [!   var] <i>port-reference-name</i> : <i>any-port-kind</i>
<b>Parameters</b>	<i>port-reference-name</i> A unique identifier.
	<i>direction</i> One of <b>in</b> or <b>out</b> ; for simple ports and event ports, this can also be <b>inout</b> .
	<i>port-kind</i> One of <b>simple_port</b> , <b>buffer_port</b> , or <b>event_port</b> .
	<i>element-type</i> Required if port-kind is <b>simple_port</b> or <b>buffer_port</b> .
	<i>any-port-kind</i> One of <b>any_simple_port</b> , <b>any_buffer_port</b> , or <b>any_event_port</b> .

Port instances can be referenced by a field, variable, or method parameter of the same port type or of an abstract type:

- any\_simple\_port

178

- `any_buffer_port` 1
- **`any_event_port`**.

Abstract port types reference only the port kind, not the port direction or data element. Thus, a method parameter of type **`any_simple_port`** accepts all simple ports, e.g. 5

```
data_length: in simple_port of uint is instance;
data: inout simple_port of list of bit is instance;
```

If a port reference is a field, then it shall be marked as non-generated or it needs to be constrained to an existing port instance. Otherwise, a generation error shall result. 10

Port binding is allowed only for port instance fields, not for port reference fields. Trying to apply a **`keep bind()`** constraint to a port reference shall result in an error. 15

The **`$`** access operator cannot be applied to an item of type **`any_simple_port`** or **`any_event_port`**. Abstract types do not have any access methods. For example, the expression “`port_arg$ == 0`” in the following code causes a syntax error. 20

```
foo_tcm ( port_arg : any_simple_port )@clk is {
  if ( port_arg$ == 0) then { -- syntax error
    out (sys.time, " Testing port logic comparison.");
  };
};
```

An abstract type cannot be used in a port instance; the element type needs to be specified instead. 25

Syntax example:

```
!last_printed_port: any_buffer_port;
!in_int_buffer_port_ref: in buffer_port of int;
```

### 6.5.5 port\$ 35

<b>Purpose</b>	Read or write a value to a simple port or event port
<b>Category</b>	Operator
<b>Syntax</b>	<i>exp</i> \$ 40
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port or event port instance. 45

The **`$`** access operator can be used to access or update the value held in a simple port or event port. When used on the right-hand side, `p$` refers to the port’s value. On the left-hand side of an assignment, `p$` refers to the value’s location, so an assignment to `p$` changes the value held in the port. 45

Without the **`$`** operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the **`$`** operator can be used for operations involving port references. 50

The **`$`** access operator cannot be applied to an item of type **`any_simple_port`** or **`any_event_port`**. Abstract types do not have any access methods. For example, the expression “`port_arg$ == 0`” in the following code causes a syntax error. 55

```

1      foo_tcm ( port_arg : any_simple_port )@clk is {
          if ( port_arg$ == 0) then { -- syntax error
              out (sys.time, " Testing port logic comparison.");
          };
5      };

```

Syntax example:

```

10     p$ = 32'bz;           // Assigns an mvl literal to the port 'p'

```

146

## 6.6 Port attributes

Ports have attributes that affect their behavior and how they can be used. Use the *attribute()* syntax to assign port attributes in pre-generation constraints, as follows:

```

15     keep [soft] port_instance.attribute() == value;

```

Use soft constraints for attributes that can be overridden.

Most port attributes are ignored unless the port is an external port, but it does no harm to specify attributes for ports that are not external ports. Attributes intended for external ports do not have to be supported for a particular simulator. A particular adapter can also define additional port attributes which are required to enhance access to simulated objects.

### 6.6.1 Generic port attributes

Port attributes that are potentially valid for all simulators are described in Table 1. However, a particular simulator adapter might not implement some of these attributes. Depending on the simulator adapter, port attributes might cause additional code to be written to the stubs file. In that case, if an attribute is added or changed, the stubs file needs to be rewritten.

**Table 1—Generic port attributes**

Attribute	Description	Applies to
bind()	Connects two internal ports or connect a port to an external object. Type: bool Default: none See also 6.6.2.1.	All kinds of internal and external ports
buffer_size()	Specifies the maximum number of elements for a buffer port queue. Type: uint Default: none See also 6.6.2.2.	Buffer ports
declared_range()	Specifies the bit width of an external multi-bit object. Type: string Default: none See also 6.6.2.3.	External output simple ports that are bound to some kinds of multi-bit objects.

Table 1—Generic port attributes (Continued)

Attribute	Description	Applies to
delayed()	Specifies whether propagation of a new port value assignment occurs immediately or is delayed to the tick boundary. Type: bool Default: TRUE See also 6.6.2.4.	Internal and external simple ports
driver()	When TRUE, an additional resolved HDL driver is created for the corresponding simulator item, and that driver is written to instead of the port. Type: bool Default: FALSE See also 6.6.2.5.	External output simple ports
driver_delay()	Specifies the delay time for all assignments from e to the port. Type: time Default: 0 See also 6.6.2.6.	External output simple ports
edge()	Specifies the edge on which an event is generated. Type: event_port_edge Default: change See also 6.6.2.8.	External input event ports
hdl_path()	Specifies a relative path of the corresponding simulated item as a string. Type: string Default: none See also 6.6.2.9.	External ports
pack_options()	Specifies how the port's data element is implicitly packed and unpacked. Type: pack_options Default: <b>global.packing.adapter</b> See also 6.6.2.10.	External simple ports whose data element is a composite type (lists and structs)
pass_by_pointer	When TRUE, composite data (structs or lists) are transferred by reference. Type: bool Default: FALSE (pass by value) See also 6.6.2.11.	Internal simple or buffer ports whose data element is a composite type (lists and structs)

### 6.6.2 Port attributes for HDL simulators

Port attributes that are potentially valid for all HDL simulators are described in Table 2. However, a particular simulator adapter might not implement some of these attributes.

The port attributes in Table 2 enable extended functionality. They cause additional information to be written into the HDL stubs file to enhance user control over the driving of HDL signals. For this reason, any attribute shown in Table 2 is added or changed, the stubs file needs to be rewritten.

Some of these attributes are similar to Verilog or VHDL unit members, such as **verilog variable** or **vhdl driver**.

Table 2—Port attributes for Verilog or VHDL agents

Attribute	Description	Applies to
driver_initial_value()	Applies an initial <b>mvl</b> value to the port. Type: list of mvl Default: {} (empty list) See also 6.6.2.7.	External output simple ports
verilog_drive()	Specifies the event on which the data is driven to the Verilog object. Type: string Default: none See also 6.6.2.12.	External output simple ports
verilog_drive_hold()	Specifies an event after which the port data is set to Z. Type: string Default: none See also 6.6.2.13.	External output simple ports
verilog_forcible()	Allows forcing of Verilog wires. Type: bool Default: FALSE See also 6.6.2.14.	External output simple ports
verilog_strobe()	Specifies the sampling event for the Verilog signal that is bound to the port. Type: string Default: none See also 6.6.2.15.	External output simple ports
verilog_wire()	Binds an external out port to a Verilog wire. Type: bool Default: FALSE See also 6.6.2.16.	External output simple ports
vhdl_delay_mode()	Specifies whether pulses whose period is shorter than the delay are propagated through the driver. Type: <code>sn_vhdl_delay_mode</code> Default: TRANSPORT (all pulses, regardless of length, are propagated) See also 6.6.2.17.	External output simple ports
vhdl_driver()	This is an alias for the <b>driver()</b> attribute. Type: bool Default: FALSE See also 6.6.2.5.	External output simple ports

*Example*

The following **verilog variable** declaration

```
verilog variable 'sig[7:0]' using strobe="#1", drive="#5" ;
```

is equivalent to the following port attributes:

```
data : inout simple_port of uint(bits: 8) is instance;
keep bind(data, external);
keep data.hdl_path() == "sig";
keep data.declared_range() == "[7:0]";
keep data.verilog_strobe() == "#1";
```

```
keep data.verilog_drive() == "#5";
```

1

### 6.6.2.1 bind()

<b>Purpose</b>	Connect two internal ports or connect a port to an external object		5
<b>Category</b>	Generic port attribute		
<b>Syntax</b>	<b>bind(<i>exp1</i>, <i>exp2</i>);</b> <b>bind(<i>exp1</i>, <b>external</b>);</b> <b>bind(<i>exp1</i>, <b>empty</b>   <b>undefined</b>);</b>		10
<b>Parameters</b>	<i>exp1</i> , <i>exp2</i>	One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected.	15
	<b>external</b>	Defines a port as connected to a simulated object, such as a Verilog register, VHDL signal, or SystemC object.	
	<b>empty</b>	Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed.	20
	<b>undefined</b>	Defines a disconnected port. Runtime accessing of a port with an undefined binding shall cause an error.	

Ports are connected to other *e* ports or to external simulated objects, such as Verilog registers, VHDL signals, or SystemC methods, using a pre-run generation constraint on the **bind()** attribute. Ports can also be left explicitly disconnected with **empty** or **undefined**.

25

#### 6.6.2.1.1 Rules

30

- a) All ports must be bound in one of the following ways:
  - 1) Bound in pairs, that is, one in or inout port bound to one out or inout port. It is illegal to bind together two input ports, two output ports, or two inout ports.
  - 2) Only ports of the same kind can be bound together. A simple port cannot be bound to a buffer port or an event port and a buffer port cannot be bound to an event port.
  - 3) Bound to an external simulated item.
  - 4) Explicitly disconnected (**empty** or **undefined**).
- b) Dangling ports (ports without **bind()** attributes) shall cause an error during elaboration. See 6.6.2.1.2 for more information.
- c) No port can be connected to more than one other port, i.e., port A can be connected to port B or port C, but not to both.
- d) A port can be explicitly disconnected and then overridden with a binding to an internal or external object. No other multiple bindings are allowed, i.e., a port cannot be bound to an internal object and also to an external object. Similarly, a port's binding cannot be simultaneously empty and undefined.
- e) Ports connected in a pair shall have the exact same element type.

35

40

45

#### 6.6.2.1.2 Checking of ports

Binding and checking of ports takes place automatically at the end of the predefined **generate\_test()** test method. This process, called *elaboration of ports*, includes checking for dangling ports and binding consistency (directions, buffer sizes, and so on).

150

204

50

A port that has no **bind()** constraint is a dangling port. Since all ports need to be bound, a dangling port shall cause an elaboration-time error.

55

### 6.6.2.1.3 Disconnected ports

A port that is bound using the **empty** or **undefined** keyword is called a disconnected port. The **empty** or **undefined** keyword can only appear as the second argument of the **bind()** constraint, in place of a second port instance name.

The same port cannot be both empty and undefined. Attempting to apply such contradicting constraints to one port shall cause an elaboration-time error.

Empty binding can be used to define a port that is connected to nothing. Runtime accessing of an empty-bound port is allowed. Its effect depends on the operation and type of the port:

- Reading from an empty-bound simple port returns the last written value or the default of the port element type, if no value has been written so far.
- Writing to an empty-bound out or inout simple port stores the new value internally.
- Reading from an empty-bound buffer port causes the thread to halt.
- Writing to an empty-bound buffer port causes the thread to halt if the buffer is full.
- Waiting for an empty-bound event port causes the thread to halt. If the port direction is inout, then emitting the port resumes the thread.
- An empty-bound event port can be emitted.

A subsequent constraint can be used to overwrite the empty binding constraint.

Like empty binding, undefined binding can define a port that is connected to nothing. The difference is that runtime accessing of a port with an undefined binding shall cause an error.

A subsequent constraint can be used to overwrite the undefined binding constraint.

Syntax example:

```
buf_in1: in buffer_port of int(bits:16) is instance;
buf_out1: out buffer_port of int(bits:16) is instance;
keep bind(buf_in1, buf_out1); // Valid
```

*Examples of invalid bindings*

```
buf_in2: in buffer_port of int(bits:32) is instance;
buf_out2: out buffer_port of int(bits:16) is instance;
keep bind(buf_in2, buf_out2); // Invalid; different bit size
```

```
buf_in3: in buffer_port of packet is instance;
buf_out3: out buffer_port of small packet is instance;
keep bind(buf_in3, buf_out3); // Invalid; different subtypes
```

```
simple_in2: in simple_port of int(bits:16) is instance;
simple_out2: out simple_port of int(bits:16) is instance;
keep bind(simple_in2, simple_out2);
keep bind(simple_in2, external); // Invalid; multiple binding
```

### 6.6.2.2 `buffer_size()`

<b>Purpose</b>	Specify the size of a buffer port queue
<b>Category</b>	Buffer port attribute
<b>Syntax</b>	<code>exp.buffer_size() == num</code>
<b>Parameters</b>	<i>exp</i> An expression of type <b>[in   out] buffer_port of type</b> .
	<i>num</i> An integer specifying the maximum number of elements for the queue.

This attribute determines the number of **put()** actions that can be performed before a **get()**. A **get()** action is required to remove data and make more room in the queue. Specifying a buffer size of 0 means rendezvous-style synchronization.

No default buffer size is provided. If a buffer size is not specified in a constraint, an error shall occur. It is only necessary to specify a buffer size for one of the two ports in a pair of connected ports. That size applies to both ports. If the two ports have different buffer sizes specified, then both of them get the larger of the two sizes.

Syntax example:

```
keep u.p.buffer_size() == 20;
```

### 6.6.2.3 `declared_range()`

<b>Purpose</b>	Specify the bit width of a multi-bit external object
<b>Category</b>	External port attribute
<b>Syntax</b>	<code>exp.declared_range() == string</code>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</i> An expression in the form: " [ <i>msb</i> : <i>lsb</i> ] "

This string attribute is meaningful for external simple ports that are bound to multi-bit objects. Because it is legal to bind a port to an HDL object with a different size, the range information is not extracted from the port declaration. In order to implement access to multi-bit signals correctly in the stubs file, this attribute is required when using the **verilog\_wire()**, **verilog\_drive()**, **verilog\_strobe()** or **driver()** attributes.

The interpretation of the string is adapter-specific.

Syntax example:

```
keep u.p.declared_range() == "[31:0]";
```

150

204

1

5

10

15

20

25

30

35

40

45

50

55

#### 6.6.2.4 delayed()

<b>Purpose</b>	Specify immediate or delayed propagation of new values
<b>Category</b>	Simple port attribute
<b>Syntax</b>	<i>exp.delayed() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is TRUE.

This Boolean attribute specifies whether propagation of a new port value assignment occurs immediately or is delayed.

When the **delayed()** attribute is TRUE (the default), propagation of external ports is delayed until the next tick. Propagation of internal ports is delayed until the next tick when the **sys.time** value changes. This behavior is consistent with the definition of delayed assignments in *e* and matches temporal *e* semantics with regard to the multiple ticks occurring at the same simulator time.

To make assigned values on ports visible immediately, constrain this attribute to be FALSE.

Syntax example:

```
keep u.p.delayed() == FALSE;
```

#### 6.6.2.5 driver()

<b>Purpose</b>	Create a resolved driver for an external object
<b>Category</b>	External out port attribute
<b>Syntax</b>	<i>exp.driver() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

This Boolean attribute is meaningful only for external out ports. When this attribute is set to TRUE, an additional resolved HDL driver is created for the corresponding simulator item and that driver is written to instead of the port.

Every port instance associated with the same simulator can create a separate driver, thus allowing HDL resolution to be applied for multiple *e* resources.

Syntax example:

```
keep u.p.driver() == TRUE;
```

150

204

6.6.2.6 `driver_delay()`

<b>Purpose</b>	Specify the delay for assignments to a port
<b>Category</b>	External out simple port attribute
<b>Syntax</b>	<code>exp.driver_delay() == time</code>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>time</i> A value of type <b>time</b> (64 bits). The default is 0.

This attribute of type **time** is meaningful only for external out ports. It specifies the delay time for all assignments from *e* to the port. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl\_driver()** attribute is set to TRUE.

Syntax example:

```
keep u.p.driver_delay() == 2;
```

6.6.2.7 `driver_initial_value()`

<b>Purpose</b>	Specify an initial value for an HDL object
<b>Category</b>	HDL port attribute
<b>Syntax</b>	<code>exp.driver_initial_value() == mvl-list</code>
<b>Parameters</b>	<i>exp</i> An expression that returns a port instance.
	<i>mvl-list</i> A lists of mvl values. Possible values are MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N. The default is {} (an empty list).

This **mvl** list type attribute applies an initial **mvl** value to an external Verilog or VHDL object. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl\_driver()** attribute is set to TRUE.

When an *e* program is driving a `std_logic` signal that is also driven from VHDL, the adapter creates a VHDL driver that is initialized by MVL\_X, unless an initial value is specified.

Syntax example:

```
keep u.p.driver_initial_value() == {MVL_X;MVL_X;MVL_1;MVL_1};
```

180

204

### 6.6.2.8 edge()

<b>Purpose</b>	Specify the edge on which an event is generated
<b>Category</b>	Event port attribute
<b>Syntax</b>	<i>exp.edge()</i> == <i>edge-option</i>
<b>Parameters</b>	<i>exp</i> An expression of an event port type.
	<i>edge-option</i> Possible values are of type <code>event_port_edge</code> : <ul style="list-style-type: none"> <li>a) <b>change, rise, fall</b> — equivalent to the behavior of <code>@sim</code> temporal expressions. This means that transitions between <code>x</code> and <code>0</code>, <code>z</code>, and <code>1</code> are not detected, <code>x</code> to <code>1</code> is considered a rise, <code>z</code> to <code>0</code> a fall, and so on.</li> <li>b) <b>any_change</b> — any change within the supported MVL values is detected, including transitions from <code>x</code> to <code>0</code> and <code>1</code> to <code>z</code>.</li> <li>c) <b>MVL_0_to_1</b> — transitions from <code>0</code> to <code>1</code> only.</li> <li>d) <b>MVL_1_to_0</b> — transitions from <code>1</code> to <code>0</code> only.</li> <li>e) <b>MVL_X_to_0</b> — transitions from <code>X</code> to <code>0</code> only.</li> <li>f) <b>MVL_0_to_X</b> — transitions from <code>0</code> to <code>X</code> only.</li> <li>g) <b>MVL_Z_to_1</b> — transitions from <code>Z</code> to <code>1</code> only.</li> <li>h) <b>MVL_1_to_Z</b> — transitions from <code>1</code> to <code>Z</code> only.</li> </ul> The default is <b>change</b> . <p style="text-align: right; color: red;">**xref the change defs w/in @sim**</p>

This attribute of type `event_port_edge` for an external event port specifies the edge on which an event is generated.

Syntax example:

```
keep e.edge() == any_change;
```

### 6.6.2.9 hdl\_path()

<b>Purpose</b>	Map port instance to an external object
<b>Category</b>	Generic port attribute
<b>Syntax</b>	<i>exp.hdl_path()</i> == <i>string</i>
<b>Parameters</b>	<i>exp</i> An expression of a port type.
	<i>string</i> The path to the external object, enclosed in double quotes. The default is an empty string.

This attribute specifies a path for accessing an external, simulated object. The path is a concatenation of the partial paths for the port itself and its enclosing units. The partial paths can use any supported separator. To allow portability between simulators, use the `e` canonical path notation.

Syntax example:

150

204

```

clk: in event_port is instance;
keep clk.hdl_path() == "clk";

```

1

### 6.6.2.10 pack\_options()

5

<b>Purpose</b>	Specify how an external port's data element is implicitly packed and unpacked
<b>Category</b>	External simple port attribute
<b>Syntax</b>	<i>exp.pack_options() == pack-option</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple or buffer port type.
	<i>pack-option</i> A predefined or user-defined pack option. The default is <b>global.packing.adapter</b> .

10

15

This attribute of type **pack\_options** is meaningful only for external ports whose data element is a composite type (lists and structs). It affects the way a port's data element is implicitly packed and unpacked. This attribute exists both for units and ports, and can be propagated downwards from an enclosing unit instance to its ports and other unit instances.

20

179

Syntax example:

```
keep u.p.pack_options() == packing.low_big_endian;
```

25

### 6.6.2.11 pass\_by\_pointer()

<b>Purpose</b>	Specify how composite data is transferred by internal ports
<b>Category</b>	Internal port attribute
<b>Syntax</b>	<i>exp.pass_by_pointer() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple or buffer port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

30

35

This Boolean attribute specifies how composite data (structs or lists) is transferred by internal simple ports or buffer ports.

40

By default, this attribute is FALSE, and complex objects are deep-copied upon an internal port access operation. To pass data by reference and speed up the test, set this attribute to TRUE (and verify no test correctness violations exist).

45

There is also a global **config misc** option, **ports\_data\_pass\_by\_pointer**. Setting this option influences all internal ports.

Syntax example:

50

```
keep u.p.pass_by_pointer() == TRUE;
```

55

### 6.6.2.12 verilog\_drive()

<b>Purpose</b>	Specify timing control for data driven to <b>**a??</b> Verilog object
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_drive() == timing-control</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>timing-control</i> A string specifying any legal Verilog timing control (event or delay).

This string attribute tells an external output port to drive its data to **\*\*a??** Verilog signal when the specified timing occurs. This can be a Verilog temporal expression, such as “@(posedge top.clk)”, or a simple **\*\*delay of kind??**, e.g., “#1”. This attribute is functionally equivalent to a **verilog variable using drive** declaration.

150

204

Syntax example:

```
keep u.p.verilog_drive() == "@posedge clk2";
```

### 6.6.2.13 verilog\_drive\_hold()

<b>Purpose</b>	Specify when to set the port to Z
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_drive_hold() == event</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>event</i> A string specifying any legal Verilog timing control.

On the first occurrence of the specified event after the port data is driven, the value of the corresponding Verilog signal is set to Z. The event is a string specifying any legal Verilog timing control. The **verilog\_drive()** attribute (see 6.6.2.12) [needs to be specified before using this attribute](#).

Syntax example:

```
keep u.p.verilog_drive_hold() == "@negedge clk2";
```

**6.6.2.14 verilog\_forcible()**

<b>Purpose</b>	Specifies that a Verilog object can be forced
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_forcible() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

By default, Verilog wires are not forcible. This Boolean attribute allows forcing of Verilog wires. The **verilog\_wire()** attribute (see 6.6.2.16) [needs to be specified before using this attribute](#).

Syntax example:

```
keep u.p.verilog_forcible() == TRUE;
```

**6.6.2.15 verilog\_strobe()**

<b>Purpose</b>	Specify the sampling event for a Verilog object
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_strobe() == event</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>event</i> A string specifying any legal Verilog timing control.

This string attribute specifies the sampling event for the Verilog signal that is bound to an external input port. This attribute is equivalent to the **verilog variable ... using strobe** declaration.

Syntax example:

```
keep u.p.verilog_strobe() == "@posedge clk1";
```

**6.6.2.16 verilog\_wire()**

<b>Purpose</b>	Create a single driver for a port (or multiple ports)
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_wire() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

This Boolean attribute allows an external out port to be bound to a Verilog wire, in a manner similar to a **verilog variable using wire** declaration.

150 35

204

40

45

50

55

The main difference between this attribute and the **driver()** attribute is that, being backward compatible, the **verilog\_wire()** attribute merges all of the ports containing this attribute into a single Verilog driver, while the **driver()** attribute creates a separate driver for each port.

181

204

Syntax example:

```
keep u.p.verilog_wire() == TRUE;
```

### 6.6.2.17 vhdl\_delay\_mode()

<b>Purpose</b>	Specify whether short pulses are propagated through driver
<b>Category</b>	HDL port attribute
<b>Syntax</b>	<i>exp.vhdl_delay_mode() == mode-option</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>mode-option</i> Either TRANSPORT (the default) or INERTIAL.

~~This **vhdl\_delay\_mode** type attribute applies a VHDL delay mode value to an external out port.~~ This attribute specifies whether pulses whose period is shorter than the delay specified by the **driver\_delay()** attribute are propagated through the driver. INERTIAL specifies that such pulses are not propagated. TRANSPORT specifies that all pulses, regardless of length, are propagated.

This attribute also influences what happens if another driver (either VHDL or another unit) schedules a signal change and, before that change occurs, this driver schedules a different change. With INERTIAL, the first change never occurs.

This attribute is silently ignored, unless the **driver\_delay()** attribute is also specified.

Syntax example:

```
keep u.p.vhdl_delay_mode() == INERTIAL;
```

146

## 6.7 Using port values and attributes in constraints

Like units, port instances can be created only during pre-run generation. They cannot be created using **new** nor generated at runtime. Consequently, a port value cannot be initialized or sampled in pre-run generation constraints. Port values can be used in on-the-fly generation constraints, in accordance with the basic constraint principles, such as the bidirectional nature of constraints.

Another methodological requirement is attribute values shall be explicitly specified in hard constraints if the attributes are used anywhere in bidirectional constraints, including implication constraints.

146

## 6.8 Buffer port methods

The following methods are used to read from or write to buffer ports and to check whether a buffer port queue is empty or full:

- `get()`
- `put()`

- `is_empty()`
- `is_full()`

1

### 6.8.1 `get()`

5

<b>Purpose</b>	Read and remove data from an input buffer port queue
<b>Category</b>	Predefined TCM for buffer ports
<b>Syntax</b>	<i>in-port-instance-name</i> . <b>get()</b> : port element type
<b>Parameters</b>	<i>in-port-instance-name</i> <span style="color: red;">**Definition??</span>

10

15

Reads a data item from the buffer port queue and removes the item from the queue. Since buffer ports use a FIFO queue, **get()** returns the first item that was written to the port.

The thread blocks upon **get()** when there are no more items in the queue. If the queue is empty, or if it has a buffer size of 0 and no **put()** has been done on the port since the last **get()**, then the **get()** is blocked until a **put()** is done on the port.

20

The number of consecutive **get()** actions that is possible is limited to the number of items inserted by **put()**.

25

Syntax example:

```
rec_cell = in_port.get();
```

### 6.8.2 `put()`

30

<b>Purpose</b>	Write data to an output buffer port queue
<b>Category</b>	Predefined TCM for buffer ports
<b>Syntax</b>	<i>out-port-instance-name</i> . <b>put()</b> ( <i>data</i> : port element type)
<b>Parameters</b>	<i>out-port-instance-name</i> <span style="color: red;">**Definition??</span>
	<i>data</i> A data item of the port element type.

35

40

Writes a data item to the output buffer port queue. The sampling event of this TCM is **sys.any**. The new data item is placed in a FIFO queue in the output buffer port.

45

The thread blocks upon **put()** when there is no more room in the queue, i.e., when the number of consequent **put()** operations exceeds the **buffer\_size()** of the port instance. If the queue is full, or if it has a buffer size of 0 and no **get()** has been done on the port since the last **put()**, then the **put()** is blocked until a **get()** is done on the port.

50

The number of consecutive **put()** actions that is possible is limited to the buffer size.

Syntax example:

55

```
1 out_port.put(trans_cell);
```

### 6.8.3 is\_empty()

<b>Purpose</b>	Check if an output buffer port queue is empty
<b>Category</b>	Pseudo-method for buffer ports
<b>Syntax</b>	<i>in-port-instance-name.is_empty()</i> : bool
<b>Parameters</b>	<i>in-port-instance-name</i> <b>**Definition??</b>

15 Returns TRUE if the input port queue is empty. Returns FALSE if the input port queue is not empty.

Syntax example:

```
20 var readable: bool;
   readable = not cell_in.is_empty();
```

### 6.8.4 is\_full()

<b>Purpose</b>	Check if an output buffer port queue is full
<b>Category</b>	Pseudo-method for buffer ports
<b>Syntax</b>	<i>out-port-instance-name.is_full()</i> : bool
<b>Parameters</b>	<i>out-port-instance-name</i> <b>**Definition??</b>

25 Returns TRUE if the output port queue is full. Returns FALSE if the output port queue is not full.

Syntax example:

```
40 var overflow: bool;
   overflow = cell_out.is_full();
```

146

## 6.9 Multi-Value Logic (MVL) methods for simple ports

45 The predefined port methods in this section are for reading and writing MVL data between ports, to facilitate communication with objects where MVL values occur. These methods operate on data of type **mvl**, which is defined as follows:

```
type mvl: [MVL_U, MVL_X, MVL_0, MVL_1, MVL_Z, MVL_W, MVL_L, MVL_H, MVL_N]
```

50 The enumeration literals are the same as those of VHDL, except for MVL\_N, which corresponds to the VHDL '-' ("don't care") literal.

55 The MVL methods are applicable according to the port direction. Methods that write a value to a port are accessible for out and inout simple ports, while methods that read a value from a port are accessible for in

and inout simple ports. *Mixed access* - accessing a port with MVL methods and accessing it through the \$ operator is allowed.

The predefined methods for simple ports are:

- put\_mvl()
- get\_mvl()
- put\_mvl\_list()
- get\_mvl\_list()
- put\_mvl\_string()
- get\_mvl\_string()
- get\_mvl4()
- get\_mvl4\_list()
- get\_mvl4\_string()

### 6.9.1 put\_mvl()

<b>Purpose</b>	Put an <b>mvl</b> data on a port of a non- <b>mvl</b> type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.put_mvl(value: mvl)</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.
	<i>value</i> A multi-value logic value.

Places an **mvl** value on an output or inout simple port, e.g., to initialize an object to a “disconnected” value. Placing an **mvl** value on a port whose element type is list places the value in the LSB of the list.

Syntax example:

```
p.put_mvl(MVL_Z)
```

### 6.9.2 get\_mvl()

<b>Purpose</b>	Read <b>mvl</b> data from a port of a non- <b>mvl</b> type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.get_mvl(): mvl</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads an **mvl** value from an input or inout simple port, e.g., to check that there are no undefined “x” bits. Getting an **mvl** value from a port whose element type is list reads the LSB of the list.

Syntax example:

```
check that pbi.get_mvl() != MVL_X else dut_error("Bad value");
```

### 6.9.3 put\_mvl\_list()

<b>Purpose</b>	Put a list of <b>mvl</b> values on a port of a non- <b>mvl</b> type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.put_mvl_list(values: list of mvl)</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.
	<i>values</i> A list of <b>mvl</b> values

Writes a list of **mvl** values to an output or inout simple port. Putting a list of **mvl** values on a port whose element type is a single bit writes only the LSB of the list.

Syntax example:

```
pbo.put_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0});
```

### 6.9.4 get\_mvl\_list()

<b>Purpose</b>	Get a list of <b>mvl</b> values from a port of a non- <b>mvl</b> type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.get_mvl_list(): list of mvl</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a list of **mvl** values from an input or inout simple port.

Syntax example:

```
check that pbil.get_mvl_list().has(it == MVL_U) == FALSE else
  dut_error("Bad list");
```

### 6.9.5 put\_mvl\_string()

<b>Purpose</b>	Put an <b>mvl</b> value on a port of a non- <b>mvl</b> type when a value is represented as a string	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp.put_mvl_string(value: string)</i>	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.
	<i>value</i>	An <b>mvl</b> value in the form of a base and one or more characters, entered as a string. The <b>mvl</b> values in the string must be lowercase. Use 1 for MVL_1, 0 for MVL_0, z for MVL_Z, and so on.

Writes a string representing a list of **mvl** values to a simple output or inout port. The **mvl** value consists of any legal base, e.g., 32'b, followed by one or more characters, e.g., xxxzzzz. The string representation follows the same rules as Verilog literals. The difference is that Verilog literals support only 4-value logic digits (1,0,x, and z), while *e* allows also the characters u, l, h, w, and n.

Syntax example:

```
pbol.put_mvl_string("32'hxxxxl1111");
```

### 6.9.6 get\_mvl\_string()

<b>Purpose</b>	Get a value in form of a string from a port of a non- <b>mvl</b> type	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp.get_mvl_string(radix: radix): string</i>	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.
	<i>radix</i>	One of BIN, OCT, or HEX.

Returns a string in which each character represents an **mvl** value. The characters are lowercase. HDL value '1' is represented by the character l, 'Z' by z, and '-' by character n.

The returned string always includes all the bits, with no implicit extensions. For example, a port of type **\*\*uint??** returns a string of 32 characters, since an int is a 32-bit data type.

Syntax example:

```
print pbis.get_mvl_string(BIN);
```

### 6.9.7 get\_mvl4()

<b>Purpose</b>	Get an <b>mvl</b> value from a port, converting 9-value logic to 4-value logic
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .get_mvl4(): mvl
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a 9-value **mvl** value from an input simple port and converts it to 4-value subset **mvl**.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z
```

Syntax example:

```
check that pbi.get_mvl4() != MVL_Z else dut_error("Bad value");
```

### 6.9.8 get\_mvl4\_list()

<b>Purpose</b>	Get a list of <b>mvl</b> value from a port, converting 9-value logic to 4-value logic
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .get_mvl4(): list of mvl
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a list of 9-value **mvl** values from an input simple port and converts them to 4-value MVL.

The predefined mapping from 9-value logic to 4-value logic is:

```
MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z
```

Syntax example:

```
check that pbi4l.get_mvl4_list().has(it == MVL_X) == FALSE else
  dut_error("Bad list");
```

**6.9.9 get\_mvl4\_string()**

<b>Purpose</b>	Get a 4-state value in form of a string from a port of a non- <b>mvl</b> type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .get_mvl4_string( <i>radix</i> : radix): string
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.
	<i>radix</i> One of BIN, OCT, or HEX.

Reads a string in which each character represents a 4-value logic digit from a subset of **mvl**, converted from 9-value logic. The characters are lowercase.

The predefined mapping from 9-value logic to 4-value logic is the same as it is commonly used when converting from VHDL std\_logic to Verilog:

```
U, W, X, N -> x
L, 0 -> 0
H, 1 -> 1
Z -> z
```

The returned string always includes all the bits, with no implicit extensions. For example, a port of type int returns a string of 32 characters, since an int is a 32-bit data type.

Syntax example:

```
print pbi4s.get_mvl4_string(BIN);
```

**6.10 Methods for simple ports**

146

These methods are defined for all simple ports, regardless of the type of data element:

- has\_x()
- has\_z()
- has\_unknown()

**6.10.1 has\_x()**

<b>Purpose</b>	Determine if port has X
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .has_x(): bool
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.

Returns TRUE if at least one bit of the port is MVL\_X.

Syntax example:

```
1      print pbi4s.has_x();
```

### 6.10.2 has\_z()

<b>Purpose</b>	Determine if port has Z
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .has_z(): bool
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.

15 Returns TRUE if at least one bit of the port is MVL\_Z.

Syntax example:

```
20      print pbi4s.has_z();
```

### 6.10.3 has\_unknown()

<b>Purpose</b>	Determine if port has U
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> .has_unknown(): bool
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.

25 Returns TRUE if at least one bit of the port is one of the following:

- 35 — MVL\_U
- MVL\_X
- MVL\_Z
- MVL\_W
- MVL\_N

40 Syntax example:

```
      print pbi4s.has_unknown();
```

45 **146**

## 6.11 Global MVL routines

The global routines for manipulating MVL values are:

- 50 — string\_to\_mvl()
- mvl\_to\_string()
- mvl\_to\_int()
- int\_to\_mvl()
- mvl\_to\_bits()
- bits\_to\_mvl()
- 55 — mvl\_to\_mvl4()

- `mvl_list_to_mvl4_list()`
- `string_to_mvl4()`

1

### 6.11.1 `string_to_mvl()`

5

<b>Purpose</b>	Convert a string to a list of <b>mvl</b> values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<code>string_to_mvl(value-string: string): list of mvl</code>
<b>Parameters</b>	<i>value-string</i> A string representing <b>mvl</b> values, consisting of a width and base followed by a series of characters corresponding to <b>mvl</b> values. The format of the input string is the same as in Verilog literals, except there are additional 9-value logic digits: u, l, h, w, and n.

10

15

Converts each character in the input string to an **mvl** value.

20

Syntax example:

```
mlist = string_to_mvl("8'bxz1");
```

### 6.11.2 `mvl_to_string()`

25

<b>Purpose</b>	Convert a list of <b>mvl</b> values to a string
<b>Category</b>	Predefined routine
<b>Syntax</b>	<code>mvl_to_string(mvl-list: list of mvl, radix: radix): string</code>
<b>Parameters</b>	<i>mvl-list</i> A list of <b>mvl</b> values.
	<i>radix</i> One of BIN, OCT, or HEX.

30

35

Converts a list of **mvl** values to a string. A sized number shall always be returned as a string.

The mapping is done in the following way:

40

```
MVL_U is converted to character "u" (lowercase)
MVL_X - "x"
MVL_0 - "0"
MVL_1 - "1"
MVL_Z - "z"
MVL_W - "w"
MVL_L - "L" **Is this the only use of uppercase in these mappings??
MVL_H - "h"
MVL_N - "n"
```

45

149

Syntax example:

50

```
mstring = mvl_to_string({MVL_Z; MVL_Z; MVL_Z; MVL_Z; MVL_X; MVL_X; MVL_X;
    MVL_X}, BIN);
```

55

### 6.11.3 mvl\_to\_int()

<b>Purpose</b>	Convert an <b>mvl</b> value to an integer
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_int</b> ( <i>mvl-list</i> : list of mvl, <i>mask</i> : list of mvl): uint
<b>Parameters</b>	<i>mvl-list</i> A list of <b>mvl</b> values to convert to an integer value.
	<i>mask</i> A list of <b>mvl</b> values that are to be converted to 1.

Converts each value in a list of **mvl** values into a binary integer (1 or 0), using a list of **mvl** mask values to determine which **mvl** values are converted to 1.

When the list is less than 32 bits, it is padded with 0's. When it is greater than 32 bits, it is truncated, leaving the 32 least-significant bits.

Syntax example:

```
var ma: uint = mvl_to_int(1, {MVL_X});
```

### 6.11.4 int\_to\_mvl()

<b>Purpose</b>	Convert an integer value to a list of <b>mvl</b> values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>int_to_mvl</b> ( <i>value</i> : uint, <i>mask</i> : mvl): list of mvl
<b>Parameters</b>	<i>value</i> An integer value to convert to a list of <b>mvl</b> values.
	<i>mask</i> An <b>mvl</b> value that replaces each bit in the integer that has the value 1.

Maps each bit that has the value 1 to the mask **mvl** value, retains the 0 bits as MVL\_0, and returns a list of 32 **mvl** values. The returned list always has a size of 32.

Syntax example:

```
var mlist: list of mvl = int_to_mvl(12, MVL_X)
```

**6.11.5 mvl\_to\_bits()**

<b>Purpose</b>	Convert a list of <b>mvl</b> values to a list of bits
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_bits</b> ( <i>mvl-list</i> : list of mvl, <i>mask</i> : list of mvl): list of bit
<b>Parameters</b>	<i>mvl-list</i> A list of <b>mvl</b> values to convert to bits.
	<i>mask</i> A list of <b>mvl</b> values that specifies which <b>mvl</b> values are to be converted to 1.

Converts a list of **mvl** values to a list of bits, using a mask of **mvl** values to indicate which **mvl** values are converted to 1 in the list of bits.

Syntax example:

```
var bl: list of bit = mvl_to_bits({MVL_Z; MVL_Z; MVL_X; MVL_L}, {MVL_Z; MVL_X})
```

**6.11.6 bits\_to\_mvl()**

<b>Purpose</b>	Convert a list of bits to a list of <b>mvl</b> values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>bits_to_mvl</b> ( <i>bit-list</i> : list of bit, <i>mask</i> : mvl): list of mvl
<b>Parameters</b>	<i>bit-list</i> A list of bits to convert to <b>mvl</b> values.
	<i>mask</i> An <b>mvl</b> value that replaces each bit in the list that has the value 1.

Maps each bit with the value 1 to the mask **mvl** value, retains the 0 bits as MVL\_0, and returns an **mvl** list that has a size of *bit-list*.

Syntax example:

```
var ml: list of mvl = bits_to_mvl({1; 0; 1; 0}, MVL_Z)
```

**6.11.7 mvl\_to\_mvl4()**

<b>Purpose</b>	Convert an <b>mvl</b> value to a 4-value logic value
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_mvl4</b> ( <i>value</i> : mvl): mvl
<b>Parameters</b>	<i>value</i> An <b>mvl</b> value to convert to a 4-value logic value.

Converts an **mvl** value to the appropriate 4-value logic subset value.

The predefined mapping from 9-value logic to 4-value logic is:

```

MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> 0
MVL_H, MVL_1 -> 1
MVL_Z -> MVL_Z

```

Syntax example:

```
var m4: mvl = mvl_to_mvl4(MVL_U)
```

### 6.11.8 mvl\_list\_to\_mvl4\_list()

<b>Purpose</b>	Convert a list of <b>mvl</b> values to a list of 4-value logic subset values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_list_to_mvl4_list</b> ( <i>mvl-list</i> : list of mvl): list of mvl
<b>Parameters</b>	<i>mvl-list</i> : A list of <b>mvl</b> values to convert to a list of 4-value logic subset values.

Converts each value in a list of **mvl** values to the corresponding 4-value logic value.

The predefined mapping from 9-value logic to 4-value logic is:

```

MVL_U, MVL_W, MVL_X, MVL_N -> MVL_X
MVL_L, MVL_0 -> MVL_0
MVL_H, MVL_1 -> MVL_1
MVL_Z -> MVL_Z

```

Syntax example:

```
var m4l: list of mvl = mvl_list_to_mvl4_list({MVL_N; MVL_L; MVL_H; MVL_1})
```

### 6.11.9 string\_to\_mvl4()

<b>Purpose</b>	Convert a string to a list of 4-value logic <b>mvl</b> subset values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>string_to_mvl4</b> ( <i>value-string</i> : string): list of mvl
<b>Parameters</b>	<i>value-string</i> : A string representing MVL values, consisting of a width and base followed by a series of characters corresponding to 9-value logic values.

Converts each character in the string to the corresponding 4-value logic value. If the string contains characters other than '0', '1', 'x', 'z', 'h', 'l', 'u', 'w', or 'n' a runtime error shall be issued.

Syntax example:

```
m1ist = string_to_mvl4("8'bxz");
```