

## 8. *e* ports 1

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

### 8.1 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.

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. 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 DUT or its environment. 15 20

#### 8.1.1 Creating port instances

A *port type* is defined by the following aspects: 25

- a) The kind of port: simple port, buffer port, event port, or method 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. 30
  - 4) *Method ports* call regular or time-consuming methods defined in other *e* units or written in foreign programming languages. They also allow calling of time-consuming *e* methods from foreign programming languages.
- b) Direction, either input or output (or inout for simple and event ports) 35
- c) Data element, the *e* type that can be passed through this port 35

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. 40

The generic syntax for ports is:

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

Event ports do not have a type specifier. 45

#### *Examples*

The following unit member creates a port instance: 50

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

where 55

- 1 — the port instance name is `data_in`
- the port kind is a buffer port
- the port direction is input
- 5 — the data element the port accepts is `packet`

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

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

### 8.1.2 Using ports

15 A port's behavior is influenced by port attributes, such as `hdl_path()` or `bind()`, that 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`.

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

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, as follows:

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

### 8.1.3 Using port values and attributes in constraints

40 Like units, port instances can be created only during pre-run generation. They cannot be created by using **new** or 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.

## 8.2 Using simple ports

45 *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, a SystemC field, or an internal object (another *e* unit). A simple port's direction can be either input, output, or inout.

50 Use the `$` port access operator to read or write port values. To access MVL on simple ports, either declare a port's data element to be `mvl` or `list of mvl`, or use the MVL methods. See 8.2.1 and 8.2.2 for more information.

55 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 needs to 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 8.7).

### 8.2.1 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.

This operator, e.g., `p$`, can also be used to access or update the value held in a simple port `p`. When used on the RHS, `p$` refers to the port's value. On the LHS 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.

#### Examples

#### Accessing port values

```
print p$;           Prints the value of a simple port, p. a
p$ = 0;            Assigns the value 0 to a simple port, p. b
force p$ = 0;      Forces a simple external port to 0.
print q$[1:0];     Prints the two LSBs of the value of q.
```

<sup>a</sup>Compare with `print p`, which prints information about port `p`.

<sup>b</sup>Compare `p$ = 0;` with `pref = NULL`, which modifies a port reference so it does not point to any port instance.

#### Accessing a port

```
print p;           Prints the information about port p. Port p is defined as:
                  p: simple_port of int (bits:8) is instance;
keep q == p;       q refers to the port instance p. Port reference q is defined as:
                  !q: simple_port of int (bits:8);
```

### 8.2.2 MVL on simple ports

There are two ways to read and write MVL on simple ports, as follows:

- Define a port and use the predefined MVL methods described in 8.9 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 the port values.

Ports of type `mvl` or list of `mvl` (MVL ports) allow easy transformation between exact `e` values and MVL, which is useful for communicating with objects that sometimes model bit values other than 0 or 1 during a test. Otherwise, using non-MVL ports is preferable, since they allow keeping the port values in a bit-by-bit representation, while MVL ports require having an `e` list for a MVL vector. MVL type definition and MVL functions are described in 8.9.

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.

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

Specifying 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 TEs 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 TE is evaluated, which occurs at runtime.

## 8.3 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 first-in-first-out (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 8.7.2.2 and 8.3.1 for more information.

A buffer port's direction can be either input or output. 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 also 8.7.

### 8.3.1 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.

#### 8.3.2 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;
};

```

## 8.4 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 8.4.1).

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 also 8.7.

### 8.4.1 Accessing event ports

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.

### 8.4.2 Example

This example shows how to connect event ports [using a **bind()** constraint] 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);
};

```

## 8.5 Using method ports

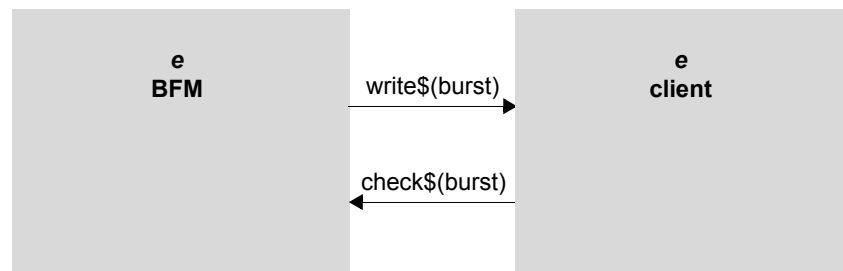
*Method ports* can be used to call regular or time-consuming methods defined in other *e* units or written in foreign programming languages. The advantages of method ports are:

- A transaction-level interface can be implemented between *e* and a high-level model described in a foreign language.
- The decision about which method to call (e.g., an *e* method or a foreign function) can be postponed from compile time to pre-run generation.

NOTE—Method ports can only be used with internal ports; they cannot be used with HDL simulators.

### 8.5.1 Implementing an internal method port interface

Figure 5 shows an *e* Bus Functional Model (BFM) that calls a write method in the *e* client and passes some generated input data to that method. The client manipulates the data and then passes back the modified data by calling a check method in the BFM. **\*\*Need more on check\$\*\***



**Figure 5—Internal method port interface**

To establish this type of interface, use the following process.

- a) Define a method type (see 8.5.4) that matches the prototype of the associated method, e.g.,
 

```
method_type burst_method_t (b: burst)@sys.any;
```
- b) Specify an output method port instance (see 8.5.6) in the *e* unit that makes the call, e.g.,
 

```
write: out method_port of burst_method_t is instance;
```
- c) Specify an input method port instance (see 8.5.5) in the enclosing unit of the method to be called, e.g.,
 

```
write_scoreboard: in method_port of burst_method_t is instance;
```

Since the input port instance is associated with the actual method by name, the names shall be identical.
- d) Bind the method ports (see 8.5.8), e.g.,
 

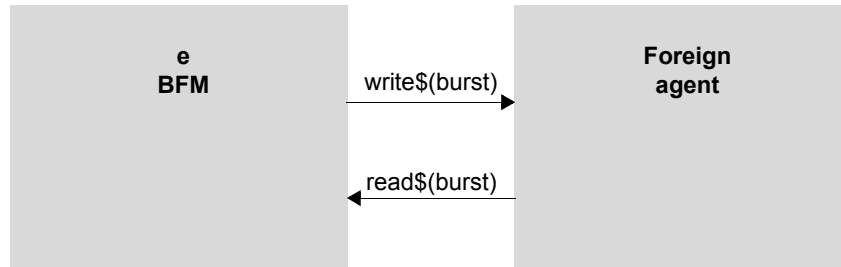
```
keep bind(e_bfm.write, e_client.write_scoreboard);
```
- e) Invoke the method by applying the  $\$( )$  operator (see 8.5.7) to the output method port, e.g.,
 

```
write$(b);
```

Only output ports can be called here. Internal ports can only be bound if they share the same method type.

### 8.5.2 Calling a function in a foreign agent from $e$

Figure 6 shows an  $e$  BFM that calls a write function in a foreign agent and passes some generated input data to that method. The foreign agent manipulates the data and then passes back the modified data by calling a read method in the BFM.



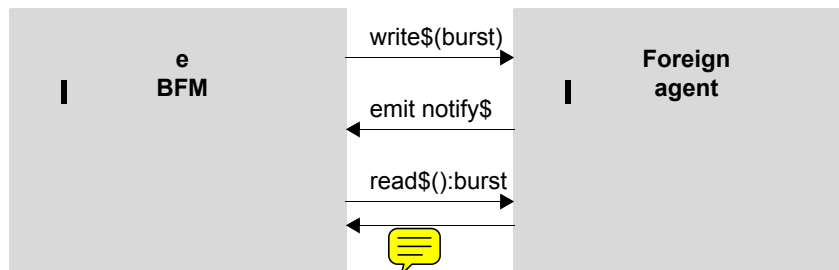
**Figure 6—External transaction level interface**

To establish this type of interface, use the following process.

- Define a method type (see 8.5.4) that matches the foreign function prototype, e.g.,  
`method_type burst_method_t (b: burst)@sys.any;`
- Create an output method port instance (see 8.5.6) in the  $e$  unit that makes the call, e.g.,  
`write: out method_port of burst_method_t is instance;`
- Bind the method port (see 8.5.8), e.g.,  
`keep bind(e_bfm.write, external);`
- Specify the corresponding path in the foreign model (**hdl\_path()**) and other language-specific attributes such as **hdl\_convertor()** (see 8.7.1 for more information).
- Invoke the method by applying the **\$()** operator (see 8.5.7) to the output method port, e.g.,  
`write$(b);`

### 8.5.3 Calling an $e$ method from a foreign agent

Figure 7 shows a variation of Figure 6, where the foreign agent emits an event (the event port `notify`). This causes the  $e$  BFM to call a read function in the foreign agent, which returns the manipulated data.



**Figure 7—External transaction level interface using an event port**


To establish this type of interface, use the following process.

- a) Define a method type (see 8.5.4) that matches the *e* method prototype, e.g.,  

```
method_type burst_method_t (b: burst)@sys.any;
```
- b) Specify an input method port instance (see 8.5.5) in the enclosing unit of the method to be called, e.g.,  

```
read: in method_port of burst_method_t is instance;
```

Since the input port instance is associated with the actual method by name, the names ~~need to~~ match.
- c) Bind the method ports (see 8.5.8), e.g.,  

```
keep bind(e_bfm.read, external);
```
- d) Specify the corresponding path in the foreign model (**hdl\_path()**) and other language-specific attributes such as **hdl\_convertor**  **sync\_mode()**. See 8.7.1 for more information.

### 8.5.4 Method types

A method port shall be parameterized by a type of a special kind — a method type. The *method type* specifies the prototype (signature) of the method. For example, the following declares a method type for a method that accepts two integer arguments and returns an integer:

```
method_type adder_method_t (arg1: int, arg2: int): int;
```

A method type that is associated with a TCM shall be defined with the **@sys.any** sampling event, e.g.,

```
method_type send_packet_method_t (p : packet)@sys.any;
```

Method types shall be defined with a unique name; this name shall be explicitly specified in the instance declaration of the method port (see 8.6.4). For example, the following associates the add method port with the `adder_method_t` method type:

```
add: out method_port of adder_method_t is instance;
```

The method type has **semantic implications** for a port beyond the simple matching of parameters and result types; it is also used to clarify runtime messages related to a particular method port. Thus, two method ports cannot be bound just because they have the same signature; they also ~~need to~~ be associated with the same method type.

Parameter mapping in *e* is positional, so the names of the formal parameters do not have to match the actual parameter list. However, using meaningful formal parameter names can improve code readability.

### 8.5.5 Input method ports

An *input method port* declares an *e* method as callable from another *e* unit or from a foreign agent. The method port instance shall:

- a) Reside in the same unit as its associated method;
- b) Have an instance name that matches the name of the associated method;
- c) Have a method type that matches the prototype of the associated method.

The method type and its prototype match if:

- 1) They have the same number of parameters.
  - 2) For any parameter types, the names ~~shall be~~ identical.
  - 3) For any return value, the types ~~shall be~~ the same.
- d) Include the **@sys.any** sampling event (in the method type declaration) if the method type is associated with a TCM.

Inline methods cannot be associated with method ports. Input method ports can be externally bound. 1

### 8.5.6 Output method ports

*Output method ports* can be used to call regular or time-consuming methods defined in other *e* units or written in foreign programming languages. 5

### 8.5.7 Invoking method ports

The  $\$()$  access operator can be used to call the method port (see also 8.6.9). The rules for parameter type checking, time-consuming method call requirements, etc. are the same as those for invoking an *e* method directly. In particular, TCM method ports can only be called from inside a TCM scope. 10

For input TCM method ports, the same instance of a port shall not be activated multiple times in parallel. The same restriction applies for output TCM method ports. A runtime error is issued if an attempt to invoke an input TCM method port occurs before the previous call has completed. 15

The parameter passing semantics are the same as in direct calls to *e* methods. Scalar parameters are passed by value, while composite parameters (*struct* or list types) are passed by reference. 20

Other considerations:

- Do not rely on the ability to modify separate fields or list elements of the incoming parameter in the actual method. Instead, use the return value (or explicit passing of parameters by reference). 25
- All ports are elaborated after the end of `post_generate()` (thus, method ports cannot be invoked before generation or from constraints).
- Calling an empty-bound method port is equivalent to calling an empty *e* method.

### 8.5.8 Binding method ports

If a set of input and output ports are bound, all the port are connected (no matter how the binding pairs were specified) and a change on any output port affects all input ports. While this makes sense for simple ports, which are used to emulate wires, it does not for method ports. For example, if there are two output method ports,  $A_o$  and  $B_o$ , three input method ports,  $A_i$ ,  $B_i$ , and  $AB_i$ , and the binding looks like: 30

```
bind(Ao,Ai);
bind(Bo,Bi);
bind(Ao,ABi);
bind(Bo,ABi);
```

the intention probably is that a call to  $A_o$  causes a call of  $A_i$  and  $AB_i$ , while a call to  $B_o$  causes a call of  $B_i$  and  $AB_i$ . This intention is implemented; however, a call to  $A_o$  also causes a call of  $B_i$  and a call to  $B_o$  also causes a call of  $A_i$ . 35

To bind multiple output ports to a common input, define the common input as a list of in method ports (see 8.6.4). Then, each of the input method ports is associated with the method via the list name. 40

*Example*

The list of in method ports is 45

```
type src_t : [ A, B ];
method_type p_t (s: src_t);
```

55

```

1      extend sys {
        Ao : out method_port of p_t is instance;
        Bo : out method_port of p_t is instance;

5      ABi : list of in method_port of p_t is instance;
          keep ABi.size() == 2;
          ABi(src: src_t) is { out("AB(", src, ")") };

```

and the binding is:

```

10     // each output also invokes the common input
        keep bind(Ao, ABi[0]);
        keep bind(Bo, ABi[1]);
15     run() is also {
        Ao$(A);
        Bo$(B);
        };
20     };

```

## 8.6 Defining and referencing ports

This subclause details how to define or reference a port.

### 8.6.1 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). 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 in, out, or inout. Omitting the direction is the same as writing inout. Port's types with different direction are not equivalent. Hence, the following types are fully equivalent:

```

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

```

Syntax example:

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

### 8.6.2 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</b> ] <i>direction</i> <b>buffer_port</b> of <i>element-type</i> <b>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 8.8 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 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;
```

### 8.6.3 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> : [list of] [ <i>direction</i> ] <b>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 8.4.1).

An event port's direction can be either in, out or inout. Omitting the direction is the same as writing inout. Port's types with different direction are not equivalent. Hence, 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, e.g.,

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

Syntax example:

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

### 8.6.4 method\_port

<b>Purpose</b>	Enable invocation of abstract functions	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>port-instance-name</i> : <b>[list of] direction method_port of method-type is instance;</b>	
<b>Parameters</b>	<i>port-instance-name</i>	A unique identifier used to reference the method port or invoke the actual method. For input method ports, this name shall be the same as that of the associated method.
	<i>direction</i>	One of <b>in</b> or <b>out</b> . There is no default. For an <b>in</b> port, only the method to activate can be specified; for an <b>out</b> port, the method can be invoked.
	<i>method-type</i>	A <i>method type</i> that specifies the port semantics (see also 8.6.5).

Method ports implement an abstraction of the calling methods (time-consuming or not) in other units or external agents, while delaying the binding from compile time to pre-run generation time.

Syntax example:

```
convert_string: out method_port of str2uint_method_t is instance;
```

### 8.6.5 method\_type *method-type-name*

<b>Purpose</b>	Associate method prototype with type name and enable notification	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>method_type method-type-name ([param-list]) [:return-type] [@sys.any];</b>	
<b>Parameters</b>	<i>method-type-name</i>	A unique identifier used to reference the method type.
	<i>param-list</i>	This needs to match the parameter list of the <i>e</i> method or external function.
	<i>return-type</i>	This needs to match the return type of the <i>e</i> method or external function.
	<b>@sys.any</b>	If the method type declaration includes the <b>@sys.any</b> sampling event, this method type can only be used for method ports associated with a TCM.

A method port (see 8.6.4) shall be parameterized by a *method type* which specifies the prototype (signature) of the method. The method type name can also be included in any runtime messages related to a specific method port.

Syntax example:

```
method_type str2uint_method_t (s: string):uint;
```

## 8.6.6 Port reference

<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> ]	
<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> .

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.

Syntax example:

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

## 8.6.7 Port: \$

<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> \$	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port or event port instance.

The \$ access operator can be used to access or update the value held in a simple port or event port. When used on the RHS, p\$ refers to the port's value. On the LHS 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 type any\_simple\_port or any\_event\_port. Abstract types do not have any access methods.~~

Syntax example:

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

### 8.6.8 Method port reference

<b>Purpose</b>	Reference a method port instance	
<b>Category</b>	Unit field, variable, or method parameter	
<b>Syntax</b>	[!   var] <i>port-reference-name</i> : <i>direction</i> <b>method_port</b> of <i>method-type</i>	
<b>Parameters</b>	<i>port-reference-name</i>	A unique identifier used to reference the method port.
	<i>direction</i>	One of <b>in</b> or <b>out</b> .
	<i>method-type</i>	A <i>method type</i> that specifies the port semantics (see also 8.6.5).

Method port instances may be referenced by a field, variable, or method parameter of the same port type.

If a port reference is a field, it shall be marked as non-generated or it ~~needs to~~ be constrained to an existing port instance. Otherwise, a generation error shall result. Also, port binding is allowed only for port instance fields, not for port reference fields (see also 8.5.8).

Syntax example:

```
!in_method_port_ref: in method_port of burst_method_t;
```

### 8.6.9 Method port: \$

<b>Purpose</b>	Call an out method port	
<b>Category</b>	Operator	
<b>Syntax</b>	<i>port-exp</i> \$( <i>out-method-port-param-list</i> )	
<b>Parameters</b>	<i>port-exp</i>	An expression that returns an output method port instance.
	<i>out-method-port-param-list</i>	A list of actual parameters to the output method port. The number and type of the parameters, if any, shall match the <i>method type</i> (see also 8.6.5).

The \$ access operator can be used to call an output method port. An attempt to call a method via the port without using the \$ operator shall result in a syntax error. 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.

Syntax example:

```
u = convert_string$("32"); //calls the convert_string out method port
```

## 8.7 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:

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

Use soft constraints for attributes that can be overridden.

5 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.

### 10 8.7.1 Generic port attributes

Port attributes that are potentially valid for all simulators are described in Table 21. 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 (see Clause 23). In that case, if an attribute is added or changed, the stubs file ~~needs to be rewritten~~.

20 **Table 21—Generic port attributes**

Attribute	Description	Applies to
<b>bind()</b>	Connects two internal ports or connect a port to an external object. Type: <i>bool</i> Default: none See also 8.7.2.1.	All kinds of internal and external ports
<b>buffer_size()</b>	Specifies the maximum number of elements for a buffer port queue. Type: <b>uint</b> Default: none See also 8.7.2.2.	Buffer ports
<b>declared_range()</b>	Specifies the bit width of an external multi-bit object. Type: <i>string</i> Default: none See also 8.7.2.3.	External output simple ports that are bound to some kinds of multi-bit objects
<b>delayed()</b>	Specifies whether propagation of a new port value assignment occurs immediately or is delayed to the tick boundary. Type: <i>bool</i> Default: TRUE See also 8.7.2.4.	Internal and external simple ports
<b>driver()</b>	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: <i>bool</i> Default: FALSE See also 8.7.2.5.	External output simple ports
<b>driver_delay()</b>	Specifies the delay time for all assignments from <i>e</i> to the port. Type: <b>time</b> Default: 0 See also 8.7.2.6.	External output simple ports
<b>edge()</b>	Specifies the edge on which an event is generated. Type: <b>event_port_edge</b> Default: <b>change</b> See also 8.7.2.8.	External input event ports

Table 21—Generic port attributes (Continued)

Attribute	Description	Applies to
<b>hdl_convertor()</b>	Specifies the rules for converting method port arguments between <i>e</i> and a foreign language, such as SystemVerilog or VHDL. The syntax of the string value associated with <b>hdl_convertor()</b> is defined by the language adapter itself. Type: <i>string</i> Default: none See also 8.5.2 and 8.5.3.	Method ports
<b>hdl_path()</b>	Specifies a relative path of the corresponding simulated item as a string. Type: <i>string</i> Default: none See also 8.7.2.9.	External ports
<b>pack_options()</b>	Specifies how the port's data element is implicitly packed and unpacked. Type: <b>pack_options</b> Default: NULL See also 8.7.2.10.	External simple ports
<b>pass_by_pointer</b>	When TRUE, composite data (structs or lists) are passed by reference. Type: <i>bool</i> Default: FALSE (pass by value) See also 8.7.2.11.	Internal simple or buffer ports whose data element is a composite type (lists and structs)

### 8.7.2 Port attributes for HDL simulators

Port attributes that are potentially valid for all HDL simulators are described in Table 22. However, a particular simulator adapter might implement some of these attributes. The port attributes in Table 22 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 22 is added or changed, the stubs file ~~needs to be rewritten~~.

#### Example

The following attributes define a port that is 8 bits wide; read operations occur with one-unit delay; drive operations have a five-unit delay:

```
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";
```

Table 22—Port attributes for Verilog or VHDL agents


Attribute	Description	Applies to
<b>driver_initial_value()</b>	Applies an initial mvl value to the port. Type: <i>list of mvl</i> Default: {} (empty list) See also 8.7.2.7.	External output simple ports
<b>verilog_drive()</b>	Specifies the event on which the data is driven to the Verilog object. Type: <i>string</i> Default: none See also 8.7.2.12.	External output simple ports
<b>verilog_drive_hold()</b>	Specifies an event after which the port data is set to Z. Type: <i>string</i> Default: none See also 8.7.2.13.	External output simple ports
<b>verilog_forcible()</b>	Allows forcing of Verilog wires. Type: <i>bool</i> Default: FALSE See also 8.7.2.14.	External output simple ports
<b>verilog_strobe()</b>	Specifies the sampling event for the Verilog signal that is bound to the port. Type: <i>string</i> Default: none See also 8.7.2.15.	External output simple ports
<b>verilog_wire()</b>	Binds an external out port to a Verilog wire. Type: <i>bool</i> Default: FALSE See also 8.7.2.16.	External output simple ports
<b>vhdl_delay_mode()</b>	Specifies whether pulses whose period is shorter than the delay are propagated through the driver. Type: <b>vhdl_delay_mode</b> Default: <b>TRANSPORT</b> (all pulses, regardless of length, are propagated) See also 8.7.2.17.	External output simple ports
<b>vhdl_driver()</b>	This is an alias for the <b>driver()</b> attribute. Type: <i>bool</i> Default: FALSE See also 8.7.2.5.	External output simple ports

**8.7.2.1 bind()**

<b>Purpose</b>	Connect two internal ports or connect a port to an external object	
<b>Category</b>	Generic port attribute	
<b>Syntax</b>	<b>bind</b> ( <i>exp1</i> , <i>exp2</i> ); <b>bind</b> ( <i>exp1</i> , <b>external</b> ); <b>bind</b> ( <i>exp1</i> , <b>empty</b>   <b>undefined</b> );	
<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.
	<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.
	<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 by using **empty** or **undefined**.

**8.7.2.1.1 Rules**

- a) All ports shall be bound in one of the following ways:
  - 1) Bound in pairs, that is, the parameters to **bind()** shall include one input or inout port bound to one output 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]  cause an error during elaboration (see 8.7.2.1.2).
- c) For pair-wise connections: 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) All ports connected together shall have the exact same element type.

**8.7.2.1.2 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).

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.

### 8.7.2.1.3 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.

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 output 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 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;
keep bind(buf_in1, empty);
```

### 8.7.2.2 buffer\_size()

<b>Purpose</b>	Specify the size of a buffer port queue
<b>Category</b>	Buffer port attribute
<b>Syntax</b>	<i>exp.buffer_size() == num</i>
<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;
```

### 8.7.2.3 declared\_range()

<b>Purpose</b>	Specify the bit width of a multi-bit external object
<b>Category</b>	External port attribute
<b>Syntax</b>	<i>exp.declared_range() == string</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</i> A string that is a valid range expression, e.g., " [msb:lsb] "

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 (see Clause 23), this attribute is required when using the `verilog_wire()`, `verilog_drive()`, `verilog_strobe()`, or `driver()` attributes.

The interpretation of the string is simulator-specific.

Syntax example:

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

### 8.7.2.4 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;
```

### 8.7.2.5 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;
```

### 8.7.2.6 driver\_delay()

<b>Purpose</b>	Specify the delay for assignments to a port
<b>Category</b>	External out simple port attribute
<b>Syntax</b>	<i>exp.driver_delay() == time</i>
<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 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;
```

**8.7.2.7 driver\_initial\_value()**

<b>Purpose</b>	Specify an initial value for an HDL object
<b>Category</b>	HDL port attribute
<b>Syntax</b>	<i>exp.driver_initial_value() == mvl-list</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a port instance.
	<i>mvl-list</i> A lists of mvl values. The default is {} (an empty list).

This *mvl-list* 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.

The default value of this attribute is MVL\_X.

Syntax example:

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

**8.7.2.8 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() == edge-option</i>
<b>Parameters</b>	<i>exp</i> An expression of a <b>buffer_port</b> type.
	<i>edge-option</i> A value of type <b>event_port_edge</b> .

This attribute of type **event\_port\_edge** (for an external event port) specifies the edge on which an event is generated. The possible values are:

- a) **change, rise, fall**—equivalent to the behavior of **@sim** TEs. This means that transitions between *x* and 0, *z*, and 1 are not detected; *x* to 1 is considered a rise; *z* to 0 a fall, and so on.
- b) **any\_change**—any change within the supported MVL values is detected, including transitions from *x* to 0 and 1 to *z*.
- c) **MVL\_0\_to\_1**—transitions from 0 to 1 only.
- d) **MVL\_1\_to\_0**—transitions from 1 to 0 only.
- e) **MVL\_X\_to\_0**—transitions from *X* to 0 only.
- f) **MVL\_0\_to\_X**—transitions from 0 to *X* only.
- g) **MVL\_Z\_to\_1**—transitions from *Z* to 1 only.
- h) **MVL\_1\_to\_Z**—transitions from 1 to *Z* only.

The default is **change**.

Syntax example:

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

### 8.7.2.9 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() == string</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a port type.
	<i>string</i>	A string specifying the path to the external object. 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:

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

### 8.7.2.10 pack\_options()

<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 NULL.

This attribute can be used to specify the way that data element of external ports 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.

Syntax example:

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

### 8.7.2.11 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.

This Boolean attribute specifies how composite data (*struct* lists) is transferred by internal simple ports or buffer ports. By default, this attribute is *FALSE*; 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).

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

Syntax example:

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

### 8.7.2.12 verilog\_drive()

<b>Purpose</b>	Specify timing control for data driven to a 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 TE, such as `@(posedge top.clk)`, or a simple unit delay, e.g., `#1`.

Syntax example:

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

### 8.7.2.13 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() == string</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</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 8.7.2.12) ~~needs to~~ be specified before using this attribute.

Syntax example:

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

### 8.7.2.14 verilog\_forcible()

<b>Purpose</b>	Specify 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 8.7.2.16) ~~needs to~~ be specified before using this attribute.

Syntax example:

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

**8.7.2.15 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() == string</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</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";
```

**8.7.2.16 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. The main difference between this attribute and the **driver()** attribute is 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.

Syntax example:

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

### 8.7.2.17 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 attribute specifies whether pulses whose period is shorter than the delay specified by the **driver\_delay()** attribute are propagated through the driver. INERTIAL specifies such pulses are not propagated, TRANSPORT 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;
```

## 8.8 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.

### 8.8.1 get()

<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.get()</i> : port element type
<b>Parameters</b>	<i>in-port-instance-name</i> An expression that returns an input buffer port instance.

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.

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

Syntax example:

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

### 8.8.2 put()

<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.put()</i> ( <i>data</i> : port element type)
<b>Parameters</b>	<i>out-port-instance-name</i> An expression that returns an output buffer port instance.
	<i>data</i> A data item of the port element type.

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.

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.

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

Syntax example:

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

### 8.8.3 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> An expression that returns an input buffer port instance.

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

Syntax example:

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

### 8.8.4 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> An expression that returns an output buffer port instance.

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

Syntax example:

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

### 8.9 MVL methods for simple ports

The predefined port methods in this subclause 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]
```

The enumeration literals are the same as those of VHDL, except for `MVL_N`, which corresponds to the VHDL - (“don’t care”) literal.

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

Accessing a port with MVL methods and accessing it through the `$` operator is allowed (*mixed access*).

#### 8.9.1 MVL four-value logic

Some MVL methods operate on a subset of the enumeration in 8.9, `MVL_X`, `MVL_Z`, `MVL_0`, and `MVL_1`, which corresponds to the four-value logic of Verilog. To convert from nine-value logic to four-value logic, the mapping shown in Table 23 is used.

**Table 23—MVL logic mapping**

Nine value	Four value
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

### 8.9.2 MVL string

Several functions allow specifying the MVL value or returning an MVL value expressed as string. A format of MVL string is the number of bits followed by the ' sign, the radix, and then the MVL literals. When an MVL list is converted into a string, the mapping shown in Table 24 is used.

The mapping is done in the following way:

**Table 24—MVL string conversion**

MVL value	String
MVL_U	u
MVL_X	x
MVL_0	0
MVL_1	1
MVL_Z	z
MVL_W	w
MVL_L	L
MVL_H	h
MVL_N	n

When a string is converted to a list of mvl, the mapping is case-insensitive.

### 8.9.3 put\_mvl()

<b>Purpose</b>	Put an mvl data on a port of a non-mvl 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 wider than one bit places the value in the LSB of the element.

Syntax example:

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

#### 8.9.4 get\_mvl()

<b>Purpose</b>	Read mvl data from a port of a non-mvl type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.get_mvl()</i> : mvl
<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 on a port whose element type is wider than one bit returns the value in the LSB of the element.

Syntax example:

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

#### 8.9.5 put\_mvl\_list()

<b>Purpose</b>	Put a list of mvl values on a port of a non-mvl 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 mvl 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});
```

#### 8.9.6 get\_mvl\_list()

<b>Purpose</b>	Get a list of mvl values from a port of a non-mvl type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.get_mvl_list()</i> : list of mvl
<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");
```

### 8.9.7 put\_mvl\_string()

<b>Purpose</b>	Put an mvl value on a port of a non-mvl 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 mvl value in the form of a base and one or more characters, entered as a string. The mvl values in the string shall 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. See also 8.9.2.

Syntax example:

```
pbol.put_mvl_string("32'hxxxxl111");
```

### 8.9.8 get\_mvl\_string()

<b>Purpose</b>	Get a value in form of a string from a port of a non-mvl 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. See also 8.9.2.

Syntax example:

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

### 8.9.9 get\_mvl4()

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

Reads a nine-value mvl value from an input simple port and converts it to four-value subset mvl. See also 8.9.1.

Syntax example:

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

### 8.9.10 get\_mvl4\_list()

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

Reads a list of nine-value mvl values from an input simple port and converts them to four-value ~~MVL~~. See also 8.9.1.

Syntax example:

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

### 8.9.11 get\_mvl4\_string()

<b>Purpose</b>	Get a four-state value in form of a string from a port of a non-mvl type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp.get_mvl4_string(radix: radix)</i> : string
<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 representing a four-value logic value. The ~~MVL~~ are first converted into four-value logic (see 8.9.1) and then converted to a string (see 8.9.2).

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);
```

### 8.9.12 has\_x()

<b>Purpose</b>	Determine if a 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:

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

### 8.9.13 has\_z()

<b>Purpose</b>	Determine if a 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.

Returns `TRUE` if at least one bit of the port is `MVL_Z`.

Syntax example:

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

### 8.9.14 has\_unknown()

<b>Purpose</b>	Determine if a port has an unknown value
<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.

Returns TRUE if at least one bit of the port is one of the following: MVL\_U, MVL\_X, MVL\_Z, MVL\_W, or MVL\_N.

Syntax example:

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

## 8.10 Global MVL routines

The subclass describes the global routines for manipulating MVL values.

### 8.10.1 string\_to\_mvl()

<b>Purpose</b>	Convert a string to a list of mvl values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>string_to_mvl</b> ( <i>value-string</i> : string): list of mvl
<b>Parameters</b>	<i>value-string</i> A string representing mvl values.

Converts a string into a list of mvl (see 8.9.2).

Syntax example:

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

### 8.10.2 mvl\_to\_string()

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

Converts a list of mvl values to a string (see 8.9.2). A sized number shall always be returned as a string.

Syntax example:

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

### 8.10.3 mvl\_to\_int()

<b>Purpose</b>	Convert a list of mvl 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 mvl values to convert to an integer value.
	<i>mask</i> A list of mvl values that are to be converted to 1.

Converts each value in a list of mvl values into a bit (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});
```

### 8.10.4 int\_to\_mvl()

<b>Purpose</b>	Convert an integer value to a list of mvl 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 mvl values.
	<i>mask</i> An mvl 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)
```

### 8.10.5 mvl\_to\_bits()

<b>Purpose</b>	Convert a list of mvl 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 mvl values to convert to bits.
	<i>mask</i> A list of mvl values that specifies which mvl 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})
```

### 8.10.6 bits\_to\_mvl()

<b>Purpose</b>	Convert a list of bits to a list of mvl 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 mvl values.
	<i>mask</i> An mvl 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)
```

### 8.10.7 mvl\_to\_mvl4()

<b>Purpose</b>	Convert an mvl value to a four-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 mvl value to convert to a four-value logic value.

Converts an mvl value to a subset of four-value logic (see 8.9.1).

Syntax example:

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

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

<b>Purpose</b>	Convert a list of mvl values to a list of four-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 mvl values to convert to a list of four-value logic subset values.

Converts a list of mvl values to a list of the four-value logic subset (see 8.9.1).

Syntax example:

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

### 8.10.9 string\_to\_mvl4()

<b>Purpose</b>	Convert a string to a list of four-value logic mvl 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 nine-value logic values.

Converts a string into a list of mvl, using the four-value logic subset. Logically, the string is converted to a list of mvl (see 8.9.2), then converted into the four-value logic subset (see 8.9.1).

Syntax example:

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

## 8.11 Comparative analysis of ports and tick access

The *e* language supports both tick access (see 23.3) and ports in order to access external simulated objects. Ports have the following advantages:

- They support modularity and encapsulation by explicitly declaring interfaces to *e* units.
- They are typed.
- They improve the 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 that can be used to specify information needed for enhanced access to DUT objects.

1 *Example 1*

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

```

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

        d = 'data';                d = data$;

        'data' = 32'bz;            data.put_mvl_list(32'bz);
15     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(),
20     {MVL_Z})[31:0];

```

*Example 2*

25 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";
30     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;
35

```

40

45

50

55