

---

# TLM Interface Ports for the e Language

Version 8.2 Early Adopter  
November 2008

©2008 Cadence Design Systems, Inc. All rights reserved worldwide.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Incisive<sup>®</sup> Enterprise Specman Elite<sup>®</sup> Testbench contains technology licensed from, and copyrighted by: University of California and is © 1990, The Regents of the University of California. All rights reserved. Free Software Foundation, Inc. and is © 1989, 1991 Free Software Foundation, Inc.; dwarfdump is © 1989, 1991 Free Software Foundation, Inc.; libdwarf is © 1991, 1999 Free Software Foundation, Inc., 51 Franklin St., Boston, MA 02110-1307 USA. All rights reserved. Open Source Initiative and is © 2004 by the Open Source Initiative, Law Offices of Lawrence E. Rose, 702 Marshall Ave. Ste. 301, Redwood City, CA 94063. All rights reserved. University of Colorado and is © 1995-2004, Regents of the University of Colorado, Boulder, CO 80309 USA. All rights reserved. Sun Microsystems, Inc. and is © Sun Microsystems, Inc., 4150 Network Cir., Santa Clara, CA 95054. All rights reserved. Scriptics Corporation and is © Scriptics Corporation. All rights reserved. loi Kim Lam and is © 1993-1995, loi Kim Lam. All rights reserved. Rudolf Usselmann, and is © 2000-2002 by Rudolf Usselmann. All rights reserved. AT&T and is © 1994-2004, AT&T Corp., 10999 W IH 10, San Antonio, 78230. All rights reserved. Judy, Copyright (c) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. All rights reserved.

Associated third-party license terms may be found at \$SPECMAN\_HOME/docs/thirdpartyinfo.txt.

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

**Restricted Permission:** This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer;
2. The publication may not be modified in any way;
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement;
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration

**Patents:** Cadence Product Incisive Enterprise Specman Elite Testbench, described in this document, is protected by U.S. Patents 6,920,583; 6,918,076; 6,907,599; 6,687,662; 6,684,359; 6,675,138; 6,530,054; 6,519,727; 6,502,232; 6,499,132; 6,487,704; 6,347,388; 6,219,809; 6,182,258; and 6,141,630. Other patents pending.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

# Contents

<b>TLM Interface Ports in e</b> .....	<b>1</b>
Defining TLM Interface Ports in e .....	1
Binding e TLM interface ports .....	4
Using e TLM interface ports .....	5
Limitations for Mixed Language TLM interface ports .....	6
interface_port .....	6
Supported TLM Interfaces .....	8
Supported Unidirectional TLM Interfaces .....	9
Supported Bidirectional TLM Interfaces .....	11
Supported Analysis TLM Interface .....	14



# TLM Interface Ports in e

**Note** The methodology described in this chapter is a brand new capability for Incisive. It has been thoroughly tested within Cadence, and, like most new capabilities, will be updated using customer feedback for improvements. Please use the following email alias to provide feedback: [ovm\\_ml\\_support@cadence.com](mailto:ovm_ml_support@cadence.com).

The *e* language supports interface ports for Transaction Level Modeling (TLM) standard interfaces. These ports enable TLM-based communication between verification components, taking advantage of the standardized communication mechanism provided by TLM.

This document contains the following sections:

- “Defining TLM Interface Ports in *e*” on page -1
- “Binding *e* TLM interface ports” on page -4
- “Using *e* TLM interface ports” on page -5
- “Limitations for Mixed Language TLM interface ports” on page -6
- “interface\_port” on page -6
- “Supported TLM Interfaces” on page -8

## Defining TLM Interface Ports in e

### Benefits

External *e* TLM interface ports are true mixed-language ports. An output TLM interface port can be instantiated in one language environment (*e*, SystemVerilog OVM, or SystemC) and bound to an input port (export) that is explicitly instantiated in another language environment. External *e* TLM interface ports are functionally compatible with SystemC and OVM.

TLM interface ports support passing of transactions by copy. For example, if an *e* output port is bound to a SystemVerilog export, then the transaction of a struct type in *e* is copied to a SystemVerilog object of the matching type.

Each *e* TLM interface port type is parameterized with a specific TLM interface type. For example, if you define an *e* TLM interface port with the syntax **interface\_port of tlm\_nonblocking\_put**, you have tied that port to the `tlm_nonblocking_put` interface. You can then use the set of methods predefined for that interface to exchange transactions.

## Procedure

1. If the port is an input *e* TLM interface port, ensure that the methods predefined for the TLM interface are implemented.

Methods must be implemented before the port is defined, or in the same module. The methods for each supported TLM interface are defined in [“Supported TLM Interfaces” on page -8](#).

Example:

```
unit server {
  // The following four lines define the four methods required
  // by the TLM interface tlm_put. The port can be defined
  // after these methods are defined.
  put(value:packet)@sys.any is {...};
  try_put(value:packet) : bool is {...};
  can_put() : bool is {...};
  ok_to_put() : tlm_event is {...};
  ...
}
```

2. Instantiate the *e* TLM interface port with **interface\_port is instance**.

You must identify:

- The port instance name
- The port direction (in or out)
- The port interface type
- The type of transaction to be transferred

See [interface\\_port on page -6](#) for the full syntax.

Example:

```
e_packet : in interface_port of tlm_put of packet is instance;
```

You can also optionally define a string to be prefixed or suffixed to the TLM methods used with this port. This step lets multiple input TLM interface ports use the same TLM method with different parameters.

Example:

```
in interface_port of tlm_put of data using prefix=data_ is instance;
```

Following is a full example with two ports. One of the ports, `data_put_export`, defines the prefix “`data_`” to be attached to TLM method names. This allows the two ports to both implement the same TLM methods, one with parameter of type `data` and one with parameter of type `packet`, in the same unit, which would be illegal in *e* if the methods had the same name:

```
unit server {
  packet_put_export :
    in interface_port of tlm_put of packet is instance;
  data_put_export :
    in interface_port of tlm_put of data using prefix=data_ is instance;
  put(p: packet)@sys.any is { ...};
  try_put(p: packet): bool is { ...}; ...
  data_put(d: data)@sys.any is { ...};
  data_try_put(d: data): bool is { ...}; ...
};
```

3. If this is an external TLM interface port, define the **external\_ovm\_path()** attribute.

The **external\_ovm\_path()** attribute is used to identify the external OVM path for this port.

**external\_ovm\_path** is similar to **hdl\_path()** for other ports. However, **external\_ovm\_path()** can be defined for TLM interface ports only and thus must contain the full OVM path (the path is not concatenated to the parent unit **hdl\_path()** ).

Example:

```
keep client.nb_put.external_ovm_path == \
    "ovm_test_top.top_env.monitor.nb_in";
```

## Notes

- Using **hdl\_path()** for TLM interface ports is illegal and results in a load time error.
- The predefined method **remap\_hdl\_path()** is not applicable for TLM interface ports. Instead, use the **remap\_external\_ovm\_path()** method.

Like **remap\_hdl\_path()**, **remap\_external\_ovm\_path()** can be called only during the **connect\_ports** phase.

- There are no other TLM interface port attributes.

4. Bind the port if required.

- Internal (*e2e*) TLM interface ports can be bound in the same way as any other *e* port, declaratively with a **keep bind()** constraint or procedurally with a **do\_bind()** or **do\_bind\_unit()** pseudo-routine.

- External *e* TLM interface ports that have a non-empty **external\_ovm\_path()** are implicitly bound to external.
- Empty and undefined bindings are supported for *e* TLM interface ports.
- You do not need to bind input *e* TLM interface ports that are unused (unlike other *e* ports).

## See Also

- [“Binding e TLM interface ports” on page -4](#)
- [“Using e TLM interface ports” on page -5](#)
- [“Limitations for Mixed Language TLM interface ports” on page -6](#)
- [“interface\\_port” on page -6](#)
- [“Supported TLM Interfaces” on page -8](#)

# Binding e TLM interface ports

## Binding Rules for TLM interface ports

A TLM output port can be bound to a TLM input port if the interface type of the output port is either the same as the interface type of the input port or subset of it (with exactly the same element type in the template parameter).

For example, you can bind an output port of `tlm_nonblocking_put` to an input port of `tlm_put`, because the `tlm_nonblocking_put` interface is a subset of the `tlm_put` interface.

Additionally:

- Empty and undefined bindings are supported for *e* TLM interface ports.
- Multiple binding is not supported for *e* TLM interface ports.
- Unification of ports bound to the same external port is not supported for *e* TLM interface ports.
- External *e* TLM interface ports are implicitly bound to external if they have a non-empty **external\_ovm\_path()**.

## Declarative and Procedural Binding

You can bind *e* TLM interface ports as you would any other port, declaratively with **keep bind()** constraints or procedurally with a **do\_bind()** or **do\_bind\_unit()** pseudo-routines.

For more information, see “Declarative and Procedural Binding” in the *e Language Reference*.

## Language-Neutral Binding for External Language Ports

The global method `ml_ovm.connect_names()` lets you bind TLM interface ports in a language-neutral way. This method is intended to be used when you want to bind two ports which both are not defined in your language. For example, you can use this method to bind a SystemC port to a SystemVerilog port from *e*.

The syntax for `ml_ovm.connect_names()` is as follows:

```
ml_ovm.connect_names(external_path1: string, external_path2: string)
```

- For SystemVerilog or SystemC, the external path must be quasi-static (an OVM full name).
- For *e*, the external path is an e-path, beginning with “sys”.

**Note** `connect_names()` can be called only in the `connect_ports()` phase. The effect of this method is immediate--it issues an error in case of any mismatch (wrong external path, mismatching interface types, unsupported multiple binding, and so on).

## Using e TLM interface ports

**To call a TLM method with an output *e* TLM interface port:**

- Use the \$ operator.

The syntax for using the \$ operator is as follows:

```
tlm-out-port-exp$.tlm-method
```

For example:

```
struct packet {
...
};
unit client {
    p : out interface_port of tlm_put of packet;
    verify()@sys.any is {
        var my_packet : packet;
        gen my_packet;
        p$.put(my_packet);
    };
};
```

**Note** You can indicate that an *e2e* TLM input port is ready to receive the next transaction by calling the predefined method `tlm_event.notify()`. The predefined struct `tlm_event` is defined as follows:

```
struct tlm_event {
    event trigger;
```

```
    notify() is {  
        emit trigger;  
    };  
};
```

The TLM functions return this struct. The user code for an input port calls `tlm_event.notify()` when it is ready to accept a next transaction.

## Limitations for Mixed Language TLM interface ports

Mixed language TLM interface ports are supported with the following list of limitations:

- External *e* TLM interface ports are supported for IES only, not for third-party simulators.
- Transactions are passed as a copy. Passing of transactions by reference is not supported.
- Only transactions of a struct (or class) type are supported. This means that the element type in *e* must be a legal *e* type that inherits from `any_struct`.
- There is no automated type mapping for transactions between languages and no automated checking of the mixed language types' conformance.

This limitation impacts the set of the supported transaction data types:

- Fields of *e* structs need to be physical (%)
- **when** inheritance is not supported.

Cadence intends to gradually remove these limitations in subsequent releases. Meanwhile, it is the user's responsibility to ensure that the transaction type fields for connected mixed language ports match on binary basis. The table of the binary matching types is located in the Mixed Language OVM Functional Specification.

---

---

## interface\_port

### Purpose

Transfer transactions between *e* units or with an external client

### Category

Unit member

## Syntax

*port-instance-name* : [list of] **direction interface\_port of tlm-intf-type**  
[using prefix=*prefix* | using suffix=*suffix*] [is instance]

## Syntax Examples

```
e_packet : in interface_port of tlm_put of packet is instance;

p1 : out interface_port of tlm_nonblocking_transport of (packet, msg) \
    is instance;
```

## Parameters

<i>port-instance-name</i>	A unique <i>e</i> identifier used to refer to the port or access its value.
<i>direction</i>	<b>in</b> or <b>out</b> . There is no default.
<i>tlm-intf-type</i>	The full syntax for one of the TLM interfaces listed in “Supported TLM Interfaces” on page -8 <ul style="list-style-type: none"> <li>• For internal <i>e</i> TLM interface ports, the type (or types) you specify for the interface can be any legal <i>e</i> type.</li> <li>• External <i>e</i> TLM interface ports support transactions of a struct (or class) type only. Thus, for externally bound <i>e</i> TLM interface ports, the type (or types) you specify for the interface must be legal <i>e</i> types that inherit from <b>any_struct</b>.</li> </ul>
<b>using prefix=<i>prefix</i></b> <b>using suffix=<i>suffix</i></b>	Applies for <i>e</i> TLM input ports only. Specifies a prefix or suffix string to be attached to the predefined TLM methods for the given port.  Using a prefix or suffix ensures that there are no method name collisions if a port contains more than instance of an <i>e</i> TLM interface port tied to the same TLM interface.  (This syntax can be used only for the port instance members. It cannot be used in other declarations, such as declarations for parameters or variables.)

## Description

An *e* TLM interface port type is parameterized with a specific TLM interface type. For example, if you define an *e* TLM interface port with the syntax **interface\_port of tlm\_nonblocking\_put**, you have tied that port to the `tlm_nonblocking_put` interface. You can then use the set of methods (functions) predefined for that interface to exchange transactions.

An external TLM input port in *e* is functionally equivalent to a SystemVerilog or SystemC export.

An external TLM output port in *e* is functionally equivalent to a SystemVerilog or SystemC TLM interface port.

## Defining Input *e* TLM interface ports

When a unit contains an instance member of an input TLM interface port, the unit must implement all methods required by the TLM interface type of that input port. The list of methods is predefined according to the standard TLM specification.

These methods must be defined before the port is defined. (If the methods and port are defined in the same module, however, the order does not matter.) If any of the required methods is missing, a compile time error is issued.

Example:

```
struct packet {
    ...
};
unit server {
// The following four lines define the four methods required
// by the TLM interface tlm_put.
    put(value : packet)@sys.any is {...};
    try_put(value: packet) : bool is {...};
    can_put() : bool is {...};
    ok_to_put() : tlm_event is {...};
    packet_in : in interface_port of tlm_put of packet is instance;
};
```

In this example, the unit server implements the four methods/tasks which are required by the interface `tlm_put` of `packet`.

## See Also

- [“Supported TLM Interfaces” on page -8](#)
- [“Binding \*e\* TLM interface ports” on page -4](#)
- [“Using \*e\* TLM interface ports” on page -5](#)
- [“Limitations for Mixed Language TLM interface ports” on page -6](#)

## Supported TLM Interfaces

The following sections list the supported TLM interfaces and their related methods:

- [“Supported Unidirectional TLM Interfaces” on page -9](#)

- “Supported Bidirectional TLM Interfaces” on page -11
- “Supported Analysis TLM Interface” on page -14

## Supported Unidirectional TLM Interfaces

**Note** Nonblocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to *e* TCM methods, consume an additional Specman tick and one cycle of simulation time.

**Table 1 Supported TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
<b>Blocking Unidirectional Interfaces</b>	
<code>tlm_blocking_put</code> of <i>type</i>	<code>put(value: <i>type</i>)@sys.any</code>
<code>tlm_blocking_get</code> of <i>type</i>	<code>get(value: *<i>type</i>)@sys.any</code>
<code>tlm_blocking_peek</code> of <i>type</i>	<code>peek(value: *<i>type</i>)@sys.any</code>
<code>tlm_blocking_get_peek</code> of <i>type</i>	<code>get(value: *<i>type</i>)@sys.any</code> <code>peek(value: *<i>type</i>)@sys.any</code>
<b>Nonblocking Unidirectional Interfaces</b>	
<code>tlm_nonblocking_put</code> of <i>type</i>	<code>try_put(value: <i>type</i>) : bool</code> <code>can_put() : bool</code> <code>ok_to_put() : tlm_event</code>
<code>tlm_nonblocking_get</code> of <i>type</i>	<code>try_get(value: *<i>type</i>) : bool</code> <code>can_get() : bool</code> <code>ok_to_get() : tlm_event</code>
<code>tlm_nonblocking_peek</code> of <i>type</i>	<code>try_peek(value: *<i>type</i>) : bool</code> <code>can_peek() : bool</code> <code>ok_to_peek() : tlm_event</code>

**Table 1 Supported TLM Interfaces and Related Methods**

<b>TLM Interface</b>	<b>Interface Methods</b>
tlm_nonblocking_get_peek of <i>type</i>	try_get(value:* <i>type</i> ) : bool can_get() : bool ok_to_get() : tlm::tlm_event try_peek(value:* <i>type</i> ) : bool can_peek() : bool ok_to_peek() : tlm::tlm_event
<b>Combined Unidirectional Interfaces (Blocking plus Nonblocking)</b>	
tlm_put of <i>type</i>	put(value: <i>type</i> )@sys.any try_put(value: <i>type</i> ) : bool can_put() : bool ok_to_put() : tlm::tlm_event
tlm_get of <i>type</i>	get(value:* <i>type</i> )@sys.any try_get(value:* <i>type</i> ) : bool can_get() : bool ok_to_get() : tlm::tlm_event
tlm_peek of <i>type</i>	peek(value:* <i>type</i> )@sys.any try_peek(value:* <i>type</i> ) : bool can_peek() : bool ok_to_peek() : tlm::tlm_event

**Table 1 Supported TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
tlm_get_peek of <i>type</i>	get(value: <i>*type</i> )@sys.any try_get(value: <i>*type</i> ) : bool can_get() : bool ok_to_get() : tlm::tlm_event peek(value: <i>*type</i> )@sys.any try_peek(value: <i>*type</i> ) : bool can_peek() : bool ok_to_peek() : tlm::tlm_event

## Supported Bidirectional TLM Interfaces

**Note** Nonblocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to *e* TCM methods, consume an additional Specman tick and one cycle of simulation time.

**Table 2 Supported Bidirectional TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
<b>Blocking Bidirectional Interfaces</b>	
tlm_blocking_master of ( <i>req-type</i> , <i>rsp-type</i> )	put(value: <i>req-type</i> )@sys.any get(value: <i>*rsp-type</i> )@sys.any peek(value: <i>*rsp-type</i> )@sys.any
tlm_blocking_slave of ( <i>req-type</i> , <i>rsp-type</i> )	put(value: <i>rsp-type</i> )@sys.any get(value: <i>*req-type</i> )@sys.any peek(value: <i>*req-type</i> )@sys.any
tlm_blocking_transport of ( <i>req-type</i> , <i>rsp-type</i> )	transport(request: <i>req-type</i> , response: <i>*rsp-type</i> )@sys.any

**Table 2 Supported Bidirectional TLM Interfaces and Related Methods**

<b>TLM Interface</b>	<b>Interface Methods</b>
<b>Nonblocking Bidirectional Interfaces</b>	
tlm_nonblocking_master of ( <i>req-type</i> , <i>rsp-type</i> )	try_put(value: <i>req-type</i> ) : bool can_put() : bool ok_to_put() : tlm::tlm_event try_get(value: <i>*rsp-type</i> ): bool can_get(): bool ok_to_get(): tlm::tlm_event try_peek(value: <i>*rsp-type</i> ): bool can_peek(): bool ok_to_peek(): tlm::tlm_event
tlm_nonblocking_slave of ( <i>req-type</i> , <i>rsp-type</i> )	try_put(value: <i>rsp-type</i> ) : bool can_put() : bool ok_to_put() : tlm::tlm_event try_get(value: <i>*req-type</i> ): bool can_get(): bool ok_to_get(): tlm::tlm_event try_peek(value: <i>*req-type</i> ): bool can_peek(): bool ok_to_peek(): tlm::tlm_event
tlm_nonblocking_transport of ( <i>req-type</i> , <i>rsp-type</i> )	nb_transport(request: <i>req-type</i> , response: <i>*rsp-type</i> ): bool

**Table 2 Supported Bidirectional TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
<b>Combined Bidirectional Interfaces (Blocking plus Nonblocking)</b>	
tlm_master of ( <i>req-type</i> , <i>rsp-type</i> )	put(value: <i>req-type</i> )@sys.any get(value: * <i>rsp-type</i> )@sys.any peek(value: * <i>rsp-type</i> )@sys.any try_put(value: <i>req-type</i> ): bool can_put(): bool ok_to_put(): tlm::tlm_event try_get(value: * <i>rsp-type</i> ): bool can_get(): bool ok_to_get(): tlm::tlm_event try_peek(value: * <i>rsp-type</i> ): bool can_peek(): bool ok_to_peek(): tlm::tlm_event
tlm_slave of ( <i>req-type</i> , <i>rsp-type</i> )	put(value: <i>rsp-type</i> )@sys.any get(value: * <i>req-type</i> )@sys.any peek(value: * <i>req-type</i> )@sys.any try_put(value: <i>rsp-type</i> ): bool can_put(): bool ok_to_put(): tlm::tlm_event try_get(value: * <i>req-type</i> ): bool can_get(): bool ok_to_get(): tlm::tlm_event

**Table 2 Supported Bidirectional TLM Interfaces and Related Methods**

<b>TLM Interface</b>	<b>Interface Methods</b>
tlm_transport of ( <i>req-type</i> , <i>rsp-type</i> )	transport(request: <i>req-type</i> , response: <i>*rsp-type</i> )@sys.any  nb_transport(request: <i>req-type</i> , response: <i>*rsp-type</i> ): bool

## Supported Analysis TLM Interface

**Table 2-1 Supported Analysis TLM Interface and Related Methods**

<b>TLM Interface</b>	<b>Interface Methods</b>
tlm_analysis of <i>type</i>	write( <i>value</i> : <i>type</i> )