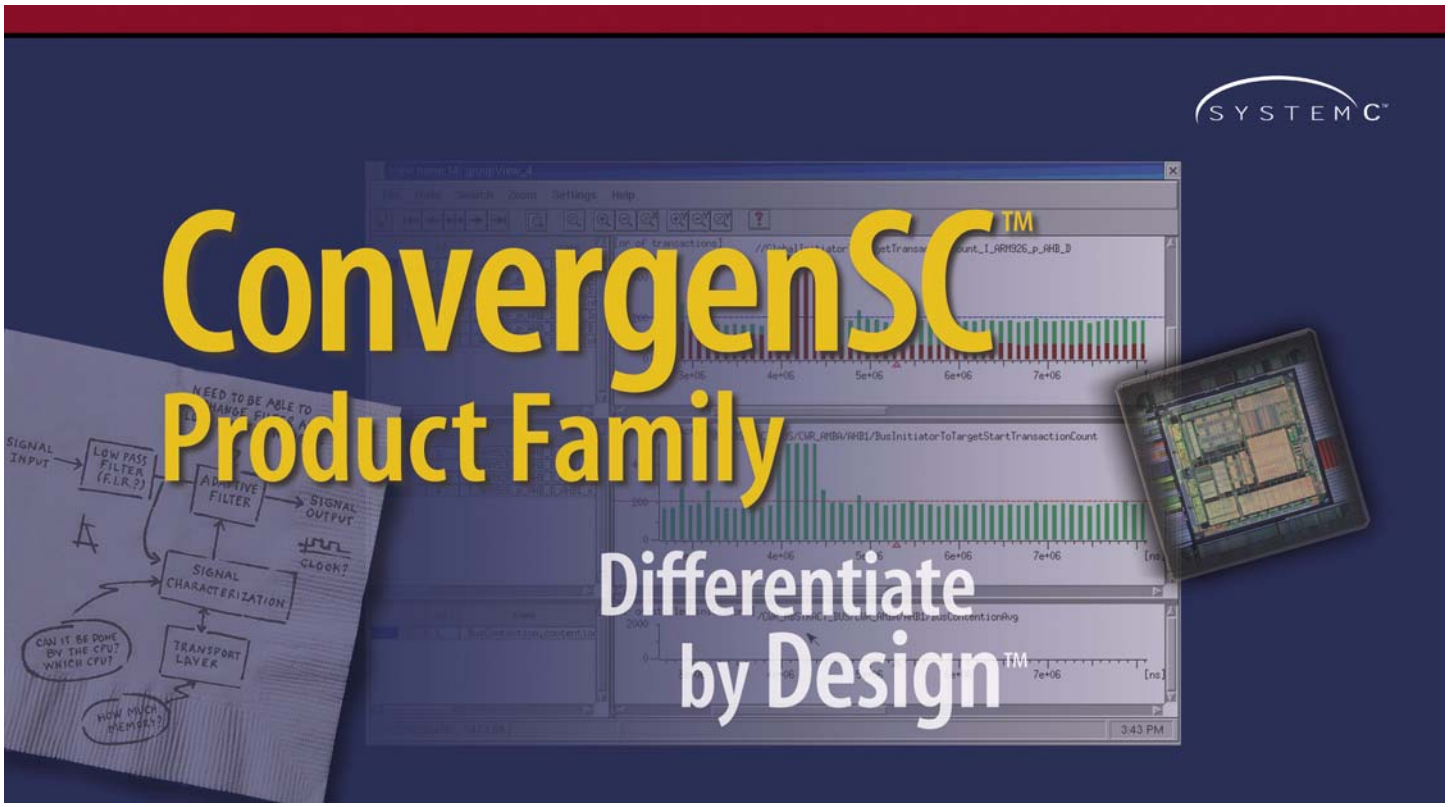
CoWare™



TLM API Manual

Version 2004.2.2 ■ January 2005

2121 North First Street
San Jose, CA 95131
USA

Main 408-436-4720
Fax 408-436-4740
www.CoWare.com

Contents

Preface

The TLM API function calls are defined in the bus library. They are specific to the protocol implemented in this library. The bus simulator described in the bus library (that is, the AMBA 2.0 Bus Library) works at the transfer level. The API function calls allow the TLM peripherals to communicate with the bus simulator at the transaction level (from an initiator port) or/and at the transfer level (from an initiator and a target port).

All the TLM API function calls described in this manual are non-blocking function calls.

The concept of IP reuse with different buses using the same API is not suitable at this low level of abstraction. It would only allow real IP reuse across a small family of buses. It is better to talk about IP behavior reuse and communication refinement. The more the TLM API function calls abstract the communication, the easier it is to reuse them for different type of buses. This is not the case for the transfer level.

This manual is organized as follows:

- Chapter 1, “TLM API Syntax,” describes the TLM API syntax.
- Chapter 2, “TLM API Functions,” describes each TLM API function in detail.

The following describes:

- Terminology
- Customer Support

Terminology

- ***AHB*** stands for Advanced High-performance Bus.
- ***API*** stands for Application Programmer’s Interface.
- ***FSM*** stands for Finite-State Machine.
- ***IP*** stands for Intellectual Property.
- ***TLM*** stands for Transaction-Level Modeling.

Customer Support

If you have any problems with the software or documentation, please contact customer support via e-mail at one of the following addresses:

- ***support@CoWare.com***
- ***support.japan@CoWare.com***

Chapter 1

TLM API Syntax

The TLM API syntax follows the same pattern for all the different bus libraries.

This chapter describes:

- Sending and Receiving Transactions
- Sending and Receiving Transfers
- Accessing Attributes
- Modeling with the TLM API
- Event Sensitivity

NOTE In the rest of this manual, ***port*** indicates the name of the TLM port of the TLM peripheral.

Sending and Receiving Transactions

The following transaction API functions are available. These functions can only be called from the initiator-port side.

- ***port*.*getTransaction()***

  This function gives access to a transaction allocated in the bus simulator.

- ***port*.*canSendTransaction()***

  This function checks whether a transaction can be sent.

- ***port*.*sendTransaction()***

  This function sends the transaction to the bus simulator.

  For more information about the transaction API functions, see Chapter 2, “TLM API Functions,” on page 13.

Sending and Receiving Transfers

The following transfer API functions are available. These functions can be called from the initiator port and from the target port.

TrfName is the name of the transfer. A complete list of all available transfers is available in the documentation of the bus library in question.

- ***port*.*getTrfName()***

  This function gives access to the transfer allocated in the bus simulator.

- ***port.canSendTrfName()***

  This function checks whether a transfer can be sent.

- ***port.canReceiveTrfName()***

  This function checks whether a transfer can be received.

- ***port.sendTrfName()***

  This function sends the transfer to the bus simulator.

- ***port.sendDelayedTrfName(delay)***

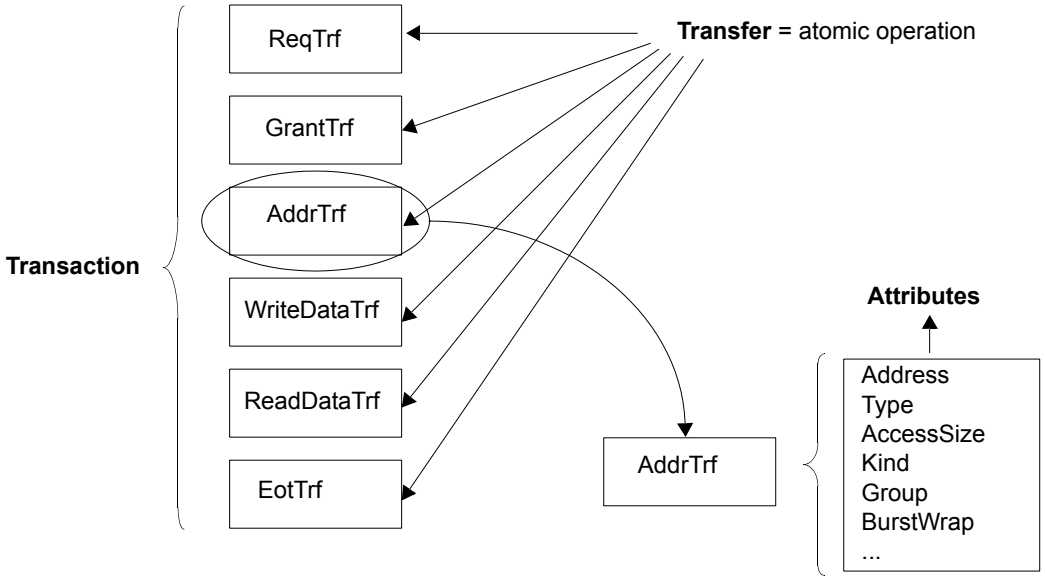
  This function sends the transfer that has previously been built to the bus simulator with a delay. The target is only allowed to access the transfer after ***delay*** bus clock cycles.

  NOTE ***sendDelayedTrfName()*** is not available on all transfers. See the description of the actual bus protocol to see which transfers support ***sendDelayedTrfName()***.

  For more information about the transfer API functions, see Chapter 2, “TLM API Functions,” on page 13.

Accessing Attributes

A transaction is a group of transfers and a transfer is defined by a set of attributes. The following diagram shows the relationship between transaction, transfer, and attributes. It is based on the AMBA 2.0 Bus Library.



The access to the attributes is protected and can only be done through method calls. The following describes these methods in detail. The methods avoid for example that you read (***get***) an attribute that has not been set yet.

The following describes:

Accessing Attributes from a Transaction

The ***port.getTransaction()*** API function call returns ***true*** if a transaction can be sent through the port, else it returns ***false***.

After the ***port.getTransaction()*** API function call, the bus simulator copies the allocated transaction in the TLM port. It eases the syntax to access the transaction attributes. Accessing the attributes from a transaction can only be done from the initiator-port side.

- ***port.Transaction->getAttrName()***

  This method retrieves the value of an attribute.

  Usage:

  ```
  int attr_tmp = tlm_port.Transaction->getAttrName();
  ```

- ***port.Transaction->setAttrName(value)***

  This method sets an attribute. The type of this attribute is specified in the documentation of the bus library in question.

  Usage:

  ```
  tlm_port.Transaction->setAttrName(attrValue);
  ```

Accessing Attributes from a Transfer

The ***port.getTrfName()*** API function call returns ***true*** if a transfer can be received/sent through the port, else it returns ***false***.

After the ***port.getTrfName()*** API function call, the bus simulator copies the allocated transfer in the TLM port. It also eases the syntax to access the transfer attributes.

For transfers that can be received:

- ***port.TrfName->getAttrName()***

  This method retrieves the value of an attribute.

  Usage:

  ```
  int attr_tmp = tlm_port.TrfName->getAttrName();
  ```

For transfers that can be sent:

- ***port.TrfName->setAttrName(value)***

  This method sets an attribute. The type of this attribute is specified in the documentation of the bus library in question.

  Usage:

  `tlm_port.`***`TrfName`***`->set`***`AttrName`***`(`***`attrValue`***`);`

Cross-Referencing Transfers

Because from a transfer it is only possible to directly access the attributes that belong to this specific transfer, additional methods are also available to cross-reference the attributes belonging to a different transfer.

The access to a different transfer is done through method calls. Depending on the protocol, it sometimes does not make sense to use cross-references from a particular transfer. In any case the cross-referenced transfer has to be valid (parameters have to be set).

- ***port.TrfName1->getTrfName2()->getAttrName()***

  This method gives access to a different transfer, where an attribute value can be retrieved.

  Usage:

  `int attr_tmp = tlm_port.`***`TrfName1`***`->get`***`TrfName2`***`()->get`***`AttrName`***`();`

- ***port.TrfName1->getTrfName2()->setAttrName(value)***

  This method gives access to a different transfer, where an attribute value can be set.

  Usage:

  `tlm_port.`***`TrfName1`***`->get`***`TrfName2`***`()->set`***`AttrName`***`(`***`attrValue`***`);`

Modeling with the TLM API

The following code example shows the usage of the transaction API.

Only from the initiator port

```
1  if (P.getTransaction()) {
2     P.Transaction->setAddress(0x1000);
      P.Transaction->setType(tlmwriteAtAddress);
      P.Transaction->setWriteData(0x2000);
3     P.sendTransaction();
   }
```

1 Test if the transaction can be accessed in order to get or set the attributes
2 Set some attributes of the transaction
(no need to set all of them if they are not used or if they use the default value)
3 Send the transaction to the bus simulator

The following code example shows the usage of the transfer API.

Initiator port

```
1  if (P.getAddrTrf()) {
2     P.AddrTrf->setAddress(0x1000);
      P.AddrTrf->setType(tlmwriteAtAddress);
3     P.sendAddrTrf();
   }
```

Target port

```
1  if (P.getWriteDataTrf()) {
2     int a=P.WriteDataTrf->getAddrTrf()->getAddress();
      int d=P.WriteDataTrf->getData();
   }
```

1 Test if the transfer can be accessed in order to get or set the attributes
2 Set some attributes of the transaction
(no need to set all of them if they are not used or if they use the default value)
3 Send the transfer to the bus simulator

Event Sensitivity

In SystemC, the event notification mechanism from a channel is usually a way to trigger a peripheral when something new happened in the channel (for example, a default event in ***sc_prim_channel***). The bus simulator can generate a specific event for each transfer. The event occurs whenever it is either possible to send a transfer or to receive a transfer. By having a TLM peripheral sensitive to these events, the bus simulator can fully control the communication. Thus it is not always necessary to model the FSM of the bus interface in the TLM peripheral.

The events are generated according to the protocol implemented in the bus library. A detailed explanation of the event generation mechanism can be found in the documentation of the bus library in question.

However, the coding style is flexible enough and it is still possible to connect a TLM peripheral with an FSM in its bus interface. The synchronization between this FSM and the bus simulator is done by testing the access to the transaction or the transfer:

- ***if (port.getTrfName()) {...}***
- ***if (port.getTransaction()) {...}***

The following describes:

Static Sensitivity

A set of two event finders is available for each event generated by the bus simulator.

- ***port*.*getSendTrfNameEventFinder()***

  This event finder allows the most recently declared process to be sensitive to ***getSendTrfNameEvent*** when the port binding takes place.

  Usage:

  ```
  sensitive << port.getSendTrfNameEventFinder();
  ```

- ***port*.*getReceiveTrfNameEventFinder()***

  This event finder allows the most recently declared process to be sensitive to the ***getReceiveTrfNameEvent*** when the port binding takes place.

  Usage:

  ```
  sensitive << port.getReceiveTrfNameEventFinder();
  ```

The following code example shows the usage of the static sensitivity.

```
SC_MODULE (MyModule)
  TLMInitiatorPort port;

  void send_address() {
    port.getAddrTrf();
    port.AddrTrf->setAddress(0x100);
    port.sendAddrTrf();
  }

  SC_CTOR(MyModule) {
    SC_METHOD(send_address);
    sensitive << port.getSendAddrTrfEventFinder();
  }
}
```

Method sensitive to the SendAddrTrf event

For a comparison to the dynamic sensitivity, see “Static Sensitivity Versus Dynamic Sensitivity” on page 9.

Dynamic Sensitivity

A TLM peripheral can be dynamically sensitive to the bus simulator events. Two methods returning an event are available for each bus simulator event.

- ***port*.*getSendTrfNameEvent()***

  This method returns the ***SendTrfName*** event whenever ***TrfName*** can be sent.

  Usage:

  `wait(`***port***`.getSend`***TrfName***`Event());`

- ***port*.*getReceiveTrfNameEvent()***

  This method returns the ***ReceiveTrfName*** event whenever ***TrfName*** can be received.

  Usage:

  `wait(`***port***`.getReceive`***TrfName***`Event());`

The following code example shows the usage of the dynamic sensitivity.

```
SC_MODULE (MyModule)
  TLMTargetPort P;
  SC_HAS_PROCESS(MyModule);
  MyModule(sc_module_name name):sc_module(name) {
    SC_THREAD(do_something);
  }

  void do_something() {
    while (true) {

      wait(P.getReceiveAddrTrfEvent());
      //...
      P.getAddressTrf();
      myVar = P.AddressTrf->getAddress();
    }
  }
}
```

Wait until the ReceiveAddrTrf event occurs

For a comparison to the static sensitivity, see “Static Sensitivity Versus Dynamic Sensitivity” on page 9.

Static Sensitivity Versus Dynamic Sensitivity

The dynamic sensitivity is very useful during a high-level architecture exploration phase. Indeed, it is possible to reuse a C algorithm and easily insert the communication part of an IP without loosing the initial code structure. There is no need to code an explicit state machine for the communication that would change the structure of the code. The synchronization between behavior and communication is naturally done and does not require any special attention.

However, this approach has a few drawbacks. Because it is based on ***SC_THREAD***, the simulation speed is of course slower than the static sensitivity based on ***SC_METHOD***.
A good IP reuse strategy in TLM comes with a clear separation of the behavior and communication. The dynamic sensitivity interleaves the two together. For example, it would not be possible to keep behavior and communication in two different entities (a behavior module and a communication module).

A bus simulator user should prefer the static sensitivity above the dynamic sensitivity whenever it is possible. Speed and TLM are usually tightly coupled and you should always go for the fastest coding style in the IP modeling. Here also it is not necessary to write an explicit state machine to describe the communication. You do not need a deep knowledge of the bus protocol since the bus simulator can fully control the sequencing of the different phases of the communication. For each transfer of the TLM protocol, two event finders are available. They are triggered by the bus simulator whenever the transfer can be sent or received. Processes in the initiator port or target port can be statically sensitive to these event finders and contain the functionality associated to the particular protocol phase.

Modeling Multiple Methods with Static Sensitivity

When writing a peripheral, it is possible to define several methods, each called using static sensitivity on different transfers. This can raise some synchronization issues when information retrieved or generated in one method is used in another method, while the methods can be called on the same clock edge. The reason for this is that you do not always know the order in which the methods will be called.

In that case, you should be aware of the following facts:

- Within the method with sensitivity to ***Trf X***, it can be possible to send/receive ***Trf Y*** if the protocol allows it at that time. You can check if it is possible to send or receive ***Trf Y*** by calling the ***getTransferName()*** method of the port interface.
- Within the method sensitive to ***Trf X***, you can do more than only send or receive the transfer ***X***. It is also possible to retrieve information related to other tranfers which can be received at that moment or which could be sent or received before that moment. This can be done directly by calling ***getTransferName()*** and then access the transfer, or through cross-referencing if the cross-reference is available. This is illustrated in “Example 2” on page 11.
- Using methods with static sensitivity does not imply that you should have one thread for each transfer.

Having good knowledge of the protocol helps, as illustrated in the following examples. The examples are based on the AMBA AHB protocol and model part of a target peripheral where the sending of one transfer, ***eotTrf***, depends on a value received in another transfer, ***writeDataTrf***. This can be done by writing two methods, one sensitive to ***EotTrf***, and one sensitive to ***WriteDataTrf***. This will, however, cause synchronization issues, which will have to be resolved somehow. It is much easier to model this behavior using only one method sensitive to one of the two transfers.

Example 1

From the AHB protocol, you know that when you receive ***writeDataTrf***, you can also send ***eotTrf*** if you have not sent it for that transaction yet. If you then send ***eotTrf*** at the same moment as you receive ***writeDataTrf***, you will have inserted one wait cycle.

If you compare with a pin-accurate interface, sending ***eotTrf*** corresponds with asserting ***HREADY***. Basically, you will assert ***HREADY*** when you have seen ***HWDATA***.

```
//method with static sensitivity to the writeDataTrf.
receiveWriteDataTrf () {

     data = p.getWriteDataTrf->getWriteData();
      if p.getEotTrf(){  //to be safe you should always
                         //check the result of this.

      if (data == 0xF000) then {
       //send ok response
       p.EotTrf->setStatus(tlmOk);
      }
      else {
       //send error response
       p.EotTrf->setStatus(tlmError);
      }
      p.sendEotTrf();
    }
}
```

Example 2

In this example, the ***receiveEotTrf*** method is called each time it is possible to send an ***EotTrf***. This does not mean ***EotTrf*** must be sent; you could try to send it again at a later time, when the method is triggered again.

Basically, you will send ***EotTrf*** when you can get ***WriteDataTrf***.

```
//method with static sensitivity to the eotTrf.
receiveEotTrf () {

   //is writedata already available?
   if (p.getWriteDataTrf()){
     p.getEotTrf(); //do not need to check the result because
                    //the method is sensitive to this transfer.

     //writeData is available
     if (data == 0xF000) then {
       //send ok response
       p.EotTrf->setStatus(tlmOk);
     }
     else {
       //send error response
       p.EotTrf->setStatus(tlmError);
     }
     p.sendEotTrf();

   }
   else {
     //WriteDataTrf is not available yet, do not send EotTrf.
     //You are inserting wait states, and the thread will be
     //triggered again the next cycle.
   }
}
```

Chapter 2

TLM API Functions

This chapter describes each TLM API function in detail.

Summary of TLM API Functions

The following table summarizes the TLM API functions and gives a short description of each function.

Function	Description
canReceiveTrfName()	Checks whether a transfer can be received.
canSendTransaction()	Checks whether a transaction can be sent.
canSendTrfName()	Checks whether a transfer can be sent.
getAttrName()	Retrieves the value of an attribute.
getReceiveTrfNameEventFinder()	Allows the most recently declared process to be sensitive to the ***getReceiveTrfNameEvent*** when the port binding takes place.
getReceiveTrfNameEvent()	Returns the ***ReceiveTrfName*** event whenever ***TrfName*** can be received.
getSendTrfNameEvent()	Returns the ***SendTrfName*** event whenever ***TrfName*** can be sent.
getSendTrfNameEventFinder()	Allows the most recently declared process to be sensitive to ***getSendTrfNameEvent*** when the port binding takes place.
getTransaction()	Gives access to a transaction allocated in the bus simulator.
getTrfName()	Gives access to the transfer allocated in the bus simulator.
sendDelayedTrfName()	Sends the transfer that has previously been built to the bus simulator with a delay.
sendTransaction()	Sends the transaction to the bus simulator.
sendTrfName()	Sends the transfer to the bus simulator.
setAttrName()	Sets an attribute.

The following describes each TLM API function in detail.

NOTE Within the TLM API function names:

- ***TrfName*** is the name of a transfer. A complete list of all available transfers is available in the documentation of the bus library in question.
- ***AttrName*** is the name of an attribute. A complete list of all available transfers is available in the documentation of the bus library in question. The type of this attribute depends on the bus library in question.

canReceiveTrfName()

canReceiveTrfName() checks whether a transfer can be received.

Syntax

port.canReceive***TrfName***()

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether or not a transfer can be received.

Example

```
if (P.canReceiveAddrTrf()) {
...
}
```

canSendTransaction()

canSendTransaction() checks whether a transaction can be sent.

Syntax

port`.canSendTransaction()`

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether or not a transaction can be sent.

Example

```
if (P.canSendTransaction()) {
...
}
```

canSendTrfName()

canSendTrfName() checks whether a transfer can be sent.

Syntax

port.canSend***TrfName***()

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** when the transfer can or cannot be sent, respectively.

Example

```
if (P.canSendAddrTrf()) {
...
}
```

getAttrName()

getAttrName() retrieves the value of an attribute.

Syntax

```
get***AttrName***()
```

Arguments

None.

Value Returned

Returns the attribute type.

attrType `get`***`AttrName`***`();.`

Example

```
attrType attr_tmp = port.TrfName->getAttrName();
```

getReceiveTrfNameEventFinder()

getReceiveTrfNameEventFinder() allows the most recently declared process to be sensitive to the ***getReceiveTrfNameEvent*** when the port binding takes place.

Syntax

```
port.getReceiveTrfNameEventFinder()
```

Arguments

None.

Value Returned

```
sc_event_finder &getReceiveTrfNameEventFinder();
```

Example

```
SC_METHOD(receive_address);
sensitive << P.getReceiveAddrTrfEventFinder();
```

getReceiveTrfNameEvent()

getReceiveTrfNameEvent() returns the ***ReceiveTrfName*** event whenever ***TrfName*** can be received.

Syntax

```
port.getReceiveTrfNameEvent()
```

Arguments

None.

Value Returned

```
const sc_event& getReceiveTrfNameEvent() const;
```

Example

```
wait(P.getReceiveAddrTrfEvent());
```

getSendTrfNameEvent()

getSendTrfNameEvent() returns the ***SendTrfName*** event whenever ***TrfName*** can be sent.

Syntax

port.getSend***TrfName***Event()

Arguments

None.

Value Returned

const sc_event& getSend***TrfName***Event() const;

Example

wait(P.getSendAddrTrfEvent());

getSendTrfNameEventFinder()

getSendTrfNameEventFinder() allows the most recently declared process to be sensitive to ***getSendTrfNameEvent*** when the port binding takes place.

Syntax

```
port.getSendTrfNameEventFinder()
```

Arguments

None.

Value Returned

```
sc_event_finder &getSendTrfNameEventFinder();
```

Example

```
SC_METHOD(send_address);
sensitive << port.getSendAddrTrfEventFinder();
```

getTransaction()

getTransaction() gives access to a transaction allocated in the bus simulator. Any communication at the transaction level must always start with this API function call.

Syntax

```
port.getTransaction()
```

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether or not a transaction can be allocated by the bus simulator.

Example

```
if (P.getTransaction()) {
...
}
```

getTrfName()

getTrfName() gives access to the transfer allocated in the bus simulator. Any communication at the transfer level must always start with this API function call.

In a peripheral, you should not call ***getTrfName()*** more than once for the same transaction on the same clock edge. So you should not check the result of ***getTrfName()*** in two different functions that can be triggered at the same time. The second call will always return ***false*** in that case, even if it is located in a method which is sensitive to that particular transfer.

Syntax

port.get***TrfName***()

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether or not the operation is allowed.

Example

```
P.getAddressTrf();
```

sendDelayedTrfName()

sendDelayedTrfName() sends the transfer that has previously been built to the bus simulator with a delay. Once the bus simulator receives the transfer, it processes the number of wait states defined by ***delay*** in bus clock cycles. When used in conjunction with ***getTrfName()***, this function should be sent within the same bus clock cycle in order to have an accurate behavior of the bus simulator.

NOTE ***sendDelayedTrfName()*** is not available on all transfers. See the description of the actual bus protocol to see which transfers support ***sendDelayedTrfName()***.

Syntax

port`.sendDelayed`***TrfName***`(`***delayed***`)`

Arguments

- ***delay*** specifies the number of wait states.

Value Returned

This function returns ***true*** or ***false*** depending on whether the delayed transfer has been sent successfully.

Example

```
P.sendDelayedAddressTrf(2);
```

sendTransaction()

sendTransaction() sends the transaction to the bus simulator. When used with ***`getTransaction()`***, this function must be sent within the same bus clock cycle.

Syntax

`port``.sendTransaction()`

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether the transaction has been sent successfully.

Example

```
P.sendTransaction();
```

sendTrfName()

sendTrfName() sends the transfer to the bus simulator. When used with ***getTrfName()***, this function must be sent within the same bus clock cycle in order to have an accurate behavior of the bus simulator.

Syntax

port.send***TrfName***()

Arguments

None.

Value Returned

This function returns ***true*** or ***false*** depending on whether the transfer has been sent successfully.

Example

```
P.sendAddrTrf();
```

setAttrName()

setAttrName() sets an attribute.

Syntax

```
set<b><i>AttrName</i></b>(<b><i>attrValue</i></b>);
```

Arguments

- ***attrvalue*** specifies the value of the attribute. It can be an enumerated type or a numeric value.

Value Returned

```
void set<b><i>AttrName</i></b>(<b><i>attrType attrValue</i></b>);
```

Examples

```
P.Transaction->setAddress(0x1000);
P.Transaction->setType(tlmwriteAtAddress);
P.Transaction->setWriteData(0x2000);
```