Doulos Verification TechNote 1: Making Sense of Transaction Level Modeling in OVM

Welcome...

to *Doulos Verification TechNotes*, an occasional series of articles on topics that we at Doulos hope will be of interest to anyone involved in verification of digital designs. Rather than trying to duplicate the plentiful tutorial and reference material that's already available, we wanted *TechNotes* to take a thought-provoking sideways look at some of the issues we think are most interesting in the world of verification. We hope you'll agree.

Verification TechNote articles are backed up by simple working code examples on our web site, where you can also find downloadable PDF copies of the articles themselves. They are available, together with many other SystemVerilog resources including conference papers and tutorial examples, at

`www.doulos.com/knowhow/sysverilog`

You can also find information about worldwide availability of our training courses featuring SystemVerilog, OVM and VMM, along with online sales of the highly respected *Golden Reference Guide* series, at

`www.doulos.com/systemverilog`

Feedback

We welcome feedback on the content of these TechNotes. If you have any comments, or ideas for topics you would like to see in future editions of *Verification TechNotes*, please contact us by email at `info@doulos.com`.

Verification
TechNote 1

Making Sense of Transaction Level Modeling in OVM

This *Doulos Verification TechNote* demystifies the fascinating, powerful and subtle Transaction Level Modeling mechanism that is used throughout the Open Verification Methodology for conveying data from one part of a testbench to another.

It is not intended to be a basic OVM tutorial; such resources are widely available from many providers including Doulos. Rather, it is aimed at anyone who has made a start with OVM and feels that they would benefit from having a deeper understanding of the TLM mechanisms that lie at its heart.

Your primary source for OVM source code and documentation is, as always,

`www.ovmworld.org`

Working example code illustrating the ideas described here can be downloaded from the Doulos web site:

`www.doulos.com/knowhow/sysverilog`

Some background

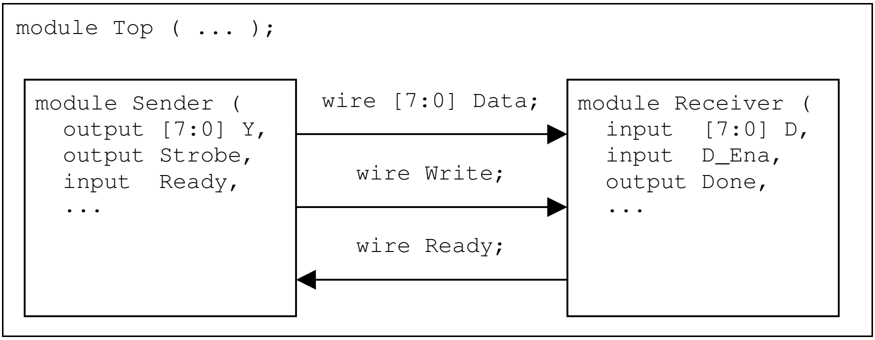
Whenever you read about OVM you are sure to find the phrase *transaction-level modeling* (TLM) or *transaction-level connection*. That innocent-sounding phrase in fact captures one of the pivotal ideas of OVM:

> **connection between verification components is standardized, so that any component can pass data to and from any other component provided both components are happy to work with the same kind of data**.

OVM's connection scheme is based on the *TLM-1* transaction-level modeling conventions that have long been popular in the world of SystemC. This kind of connection is based not on signals and events like the connections between modules in Verilog or VHDL, but on procedural connection; the fundamental unit of communication is a *function call*.

From Modules to Classes

Before we get to grips with TLM communication between objects in a class-based SystemVerilog testbench, let's review the way ordinary Verilog modules get connected together. We will find that regular Verilog port connection has some fascinating and important properties that we should aim to mimic when constructing an object-oriented SystemVerilog testbench.



```
module Top ( ... );
```

```
module Sender (
  output [7:0] Y,
  output Strobe,
  input  Ready,
  ...
```

```
module Receiver (
  input  [7:0] D,
  input  D_Ena,
  output Done,
  ...
```

Figure 1: Connecting Verilog modules

The kind of connection shown in Figure 1 is so familiar that it is hardly worth describing! If we look a little deeper, though, we can see that it has some subtle properties that make it remarkably flexible.

Our example 8-bit data interface has a simple strobe/ready handshake controlling dataflow from Sender to Receiver module. That data transfer protocol has been defined independently of any modules that use it. Consequently we can now write the Sender and Receiver modules in isolation; neither module needs to know anything about the other one. Indeed, each module does not even know what sort of module it is connected to. It simply knows that it has a set of ports corresponding to the specified protocol, and it manages those signals appropriately to follow the specification. The existence of the ports is a guarantee that the enclosing module will connect appropriate signals to those ports.

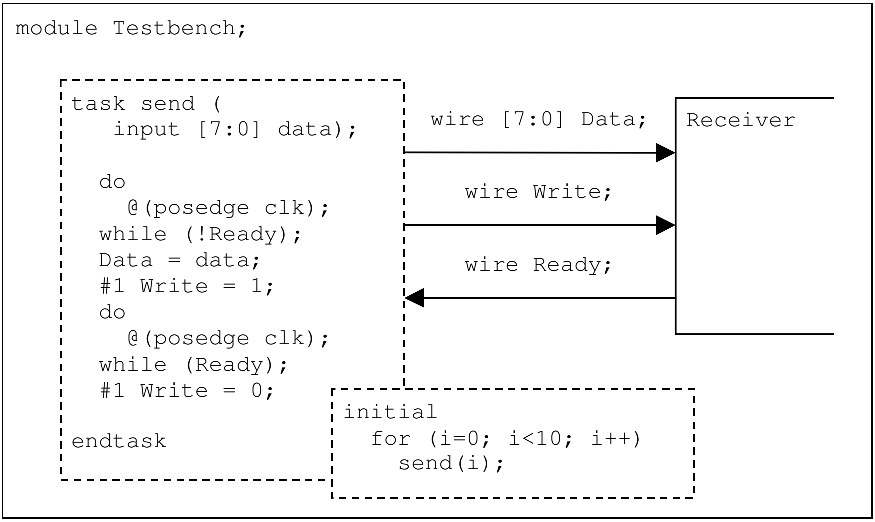
Once we have a Sender and Receiver module with the right kind of ports, supporting the right protocol, we can connect them together. This connection is achieved not by the modules themselves, but by the enclosing module that instances them. Even now, each module knows nothing about what it's connected to. The wires can have names that are quite different from the port names, and everything still works correctly.

This complete independence of a module from its environment's connections is known as *decoupling*. It allows us to write modules in isolation, knowing only the specification of an interface protocol. Thanks to this decoupling we can divide our design problem into pieces that are small and independent. From the point of view of testbench design, though, there is one big drawback: the communication operates at a very low level of detail, with every signal transition needing to be explicitly controlled or monitored by code in the modules. For a testbench we need to work at a much higher level of abstraction.

Using Task Calls for Abstract Communication

As we move away from RTL design modules and out into a testbench, we gradually leave behind us the need to do everything at the level of individual signals. To illustrate this shift, let's imagine we want to verify the operation of the `Receiver` module in Figure 1. Of course we now need some code in our testbench to control that 8-bit data interface and its handshake, but we want to hide the details and concentrate on the high-level activity that's taking place –

the transfer of data from testbench to receiver. Naturally we will write a SystemVerilog task to manage that:



```
module Testbench;

  task send (
     input [7:0] data);

    do
      @(posedge clk);
    while (!Ready);
    Data = data;
    #1 Write = 1;
    do
      @(posedge clk);
    while (Ready);
    #1 Write = 0;

  endtask
```

```
wire [7:0] Data;
wire Write;
wire Ready;
```

```
initial
  for (i=0; i<10; i++)
    send(i);
```

Figure 2: Testbench BFM task

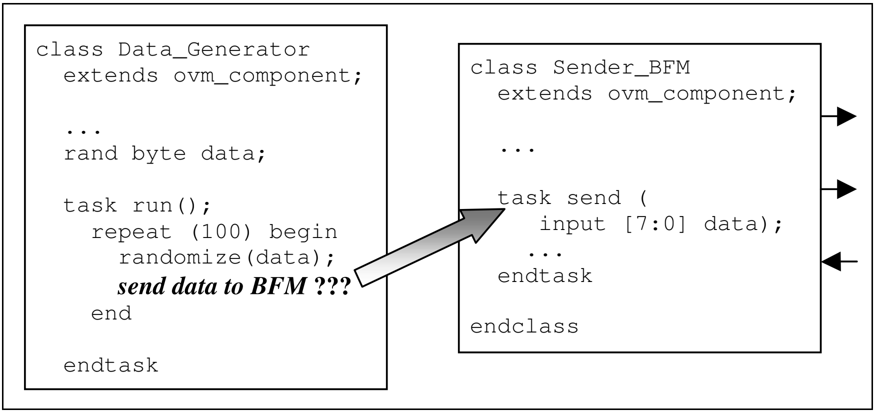
This is good: our testbench can now generate test stimulus not in terms of signal transitions, but in terms of the data that we want to transfer.

Graduating to Classes

A task of this kind, designed to generate signal-level activity conforming to a given protocol, is often known as a *BFM* task (Bus Functional Model). For small block-level testbenches it is useful in its own right, but by packaging it in a SystemVerilog `class` we can turn it into a truly re-usable component that can be incorporated into an OVM testbench.

Putting the task into a class means that we need to use virtual interface connection to reach the wires, but that's another topic for another TechNote; let's assume that has been done. So now we have a `Sender_BFM` class that knows how to send bytes of data on the appropriate interface.

The next step in constructing our testbench is to create a stimulus data generator. Because we already have a BFM to do all the low-level work of wiggling Verilog signals, our data generator can be just that – something that creates a stream of interesting data words for the driver BFM to send. But what should it do with those words? Let's sketch the testbench we now have:



```
class Data_Generator
  extends ovm_component;

  ...
  rand byte data;

  task run();
    repeat (100) begin
      randomize(data);
      send data to BFM ???
    end

  endtask
```

```
class Sender_BFM
  extends ovm_component;

  ...

  task send (
     input [7:0] data);
    ...
  endtask

endclass
```

Figure 3: BFM and data generator classes

Here, in a nutshell, is the problem that TLM aims to solve. The data generator object wishes to call the `send` method in the BFM object, but both classes must be written in a way that is completely re-usable: neither is permitted to know anything about the other one. It all works so beautifully for modules, with ports isolating each module from its immediate environment. What is the equivalent for classes?

Ports and Exports in OVM

TLM Connection Decouples Components

In our example we have a source component (the data generator) and a consumer component (the BFM). Code in the source component should call a task in the consumer component, but we wish to keep the two components isolated from each other. Figure 4 indicates the overall approach we will use to reconcile these apparently conflicting requirements.

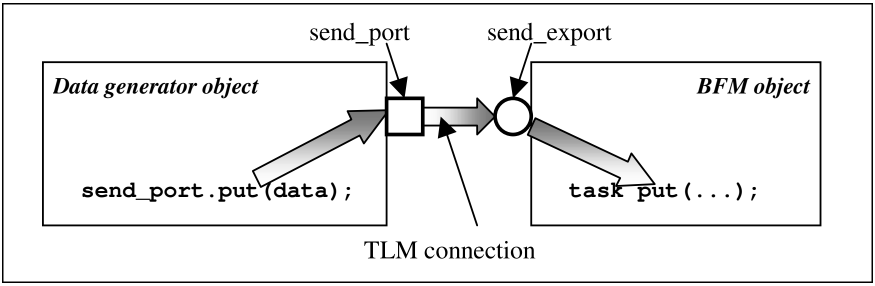


Figure 4: Port, connection and export

The source object does not attempt to call directly into the consumer. Instead, it calls into a *port*. That port is connected to an *export* on the consumer, and this export makes a call to the method's implementation within the consumer. In this way, the source in fact calls a method in the consumer, but neither source nor consumer needs to know about the other.

Unfortunately, Figure 4 is just a high-level picture and does not give us the details we need to use this idea. We need to understand how those three mechanisms – ports, connections and exports – really work.

A Disclaimer

Typical real OVM testbenches use a specific pattern of component-to-component connection that we will describe in more detail later. At this stage we will focus on getting a clear understanding of how the TLM communication mechanism works. If you have already seen a complete OVM testbench, please don't be surprised if what we show you here does not exactly match that model. Towards the end of this TechNote we will review the conventional idioms for using TLM communication in an OVM testbench.

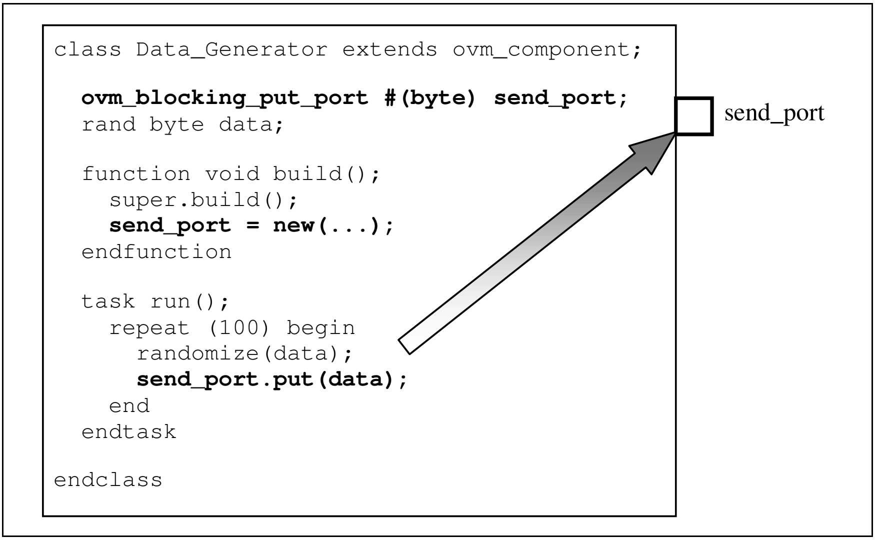
Isolating the Caller with a Port

Our data generator class creates some data and then must send it on to a data consumer. It is very important to us that we should **not** be concerned with exactly what or where this data consumer might be; we want to write the data generator completely independently. A regular module does this by having one or more *ports*, and an OVM component class can do much the same by equipping itself with a TLM port.

Just as there are input, output and inout ports on a module, there are various different kinds of port available for TLM connection. In this case we want to send (*put*) some data, and we want to wait (*block*) until the consumer component has finished with the data before we continue; so we use an `ovm_blocking_put_port`. There are several other kinds of port that we will discuss later in this note.

In addition to their direction, ports on a module must be characterized for the type of data that will flow through them – it makes no sense to put 32-bit data through a `[7:0]` port, for example. In the same way, our `ovm_blocking_put_port` must be characterized for data type. Since the port is in fact an object of class type, we can parameterize it.

The `ovm_blocking_put_port` has a `put` method. The data generator class can call this `put` method whenever it has new data to pass on, *and its responsibility is then fulfilled*. The generator has no need to concern itself with what is connected to its port. Figure 5 sketches the pattern of code you need to write when implementing this arrangement. Later we will look at the constructor arguments, and other details, of ports in OVM.



```
class Data_Generator extends ovm_component;

  ovm_blocking_put_port #(byte) send_port;
  rand byte data;

  function void build();
    super.build();
    send_port = new(...);
  endfunction

  task run();
    repeat (100) begin
      randomize(data);
      send_port.put(data);
    end
  endtask

endclass
```

Figure 5: Data generator class with blocking put port

Ports, Exports and Imps

To understand the export mechanism, we must first look at the various possible relationships among ports, exports and the objects that use them. Figure 6 shows how we could use ports and exports to connect a method call to its implementation through multiple levels of component hierarchy:

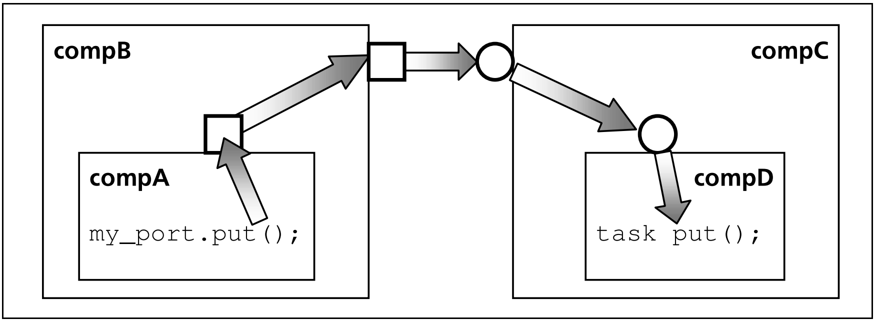


Figure 6: Hierarchical ports and exports

The overall flow of activity in Figure 6 is:

- Component **compA** has a port (square bubble). To call a method in another component, it places a call to its own port's `put` method.
- **compA**'s port is connected to a port of **compB.** Thanks to this connection, **compA**'s method call is passed on to **compB**'s port.
- The port on **compB** is connected to an *export* (round bubble) on its sibling component **compC**. As far as **compB** is concerned, the `put` method is provided by that export on **compC**.
- In fact, though, **compC** does not have an implementation of the `put` method. Instead, it passes on the call by connecting its export to the corresponding export on **compD**.
- The real, concrete implementation of the `put` method is in **compD**.

Looking more closely at Figure 6 we can see that the export on **compD** has a special status. All the other ports and exports simply pass on a method call from one place to another. The export on **compD**, however, is intimately associated with an *implementation* of the method. As a result, this must be a special kind of export known as an *imp*.

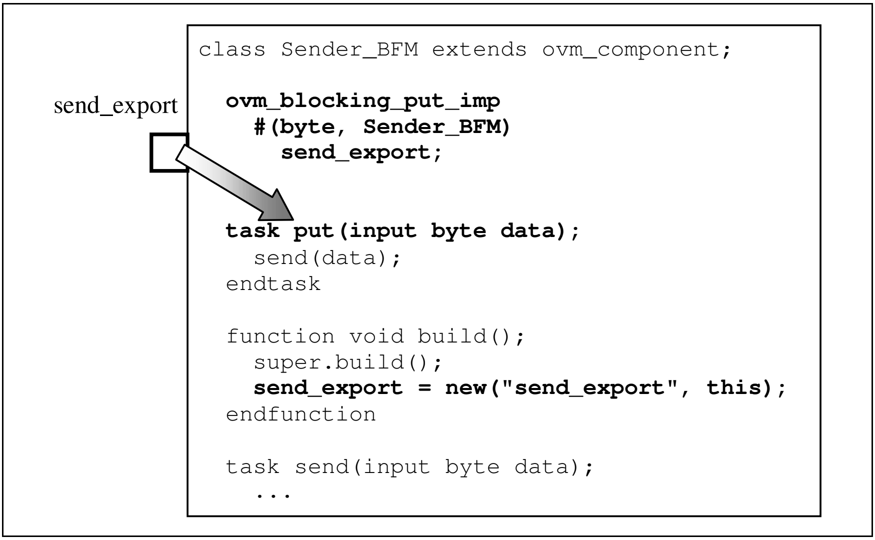
An *imp*?

No, that's not a mischievous little demon! It's an abbreviation for *implementation*.

Isolating the Target with an Imp

The ultimate target of the method call – for example, **compD** in Figure 6, or our BFM – must have an appropriate `imp` object to expose its method to the outside world. Just as we needed to choose and parameterize the data source's port correctly, so we must choose the correct kind of imp to do the job. In this case we need an `ovm_blocking_put_imp` to match the original caller's `ovm_blocking_put_port`. Unlike a port, which is simply parameterized for the type of data flowing through it, an imp needs two parameters: the type of data, and the class that contains the method implementation.

Bearing all this in mind we can now sketch a version of the BFM that exposes its `send` method through an imp.



```
class Sender_BFM extends ovm_component;

  ovm_blocking_put_imp
    #(byte, Sender_BFM)
      send_export;


  task put(input byte data);
    send(data);
  endtask

  function void build();
    super.build();
    send_export = new("send_export", this);
  endfunction

  task send(input byte data);
    ...
```

Figure 7: BFM class with blocking put imp

The key to the whole mechanism can be found in the constructor call:

```
send_export = new("send_export", this);
```

The string name `"send_export"` is merely a label, used for debugging and reporting. Much more important, the newly constructed `send_export` object is given a reference (`this`) to its enclosing (parent) component, the BFM. That allows the imp to reach back into its parent and call the `put` method that it finds there.

Unfortunately we have no choice about the name of the target method. For an `ovm_blocking_put_imp` the method *must* be named `put`; but the method in our original BFM was named `send`. So we must either rename the method in our BFM or, as we have done here, pass on the call by writing a simple wrapper method named `put`.

Finally, note that our `ovm_blocking_put_imp` object is named `send_export` rather than `send_imp`. This makes sense because, from outside the component, exports and imps look the same and connect in the same way.

Requiring and Providing

Looking again at the diagram we can see that each connection arrow represents a relationship. At one end is a method call, or a port, that *requires* an implementation to exist (but does not care where or how that implementation is provided). At the other end is a port, export or method body that *provides* the required functionality. In general:

- a method call *requires* there to be an implementation;
- an implementation *provides* the required functionality;
- TLM connection links call to implementation through a series of ports and exports.

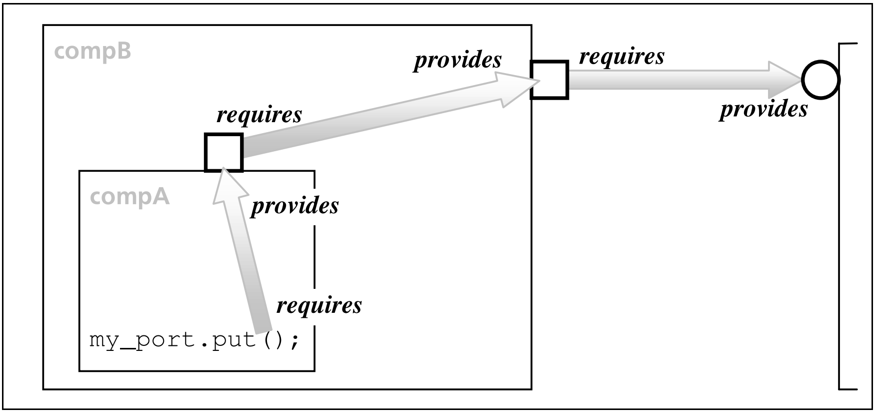


Figure 8: The requires/provides relationship

Connecting the Ports

We have now created both ends of the TLM connection: a task call made through a port, and a task implementation called from an imp. Our next job is to link them together, just as we might use wires to connect the ports of two Verilog module instances.

We can see from Figure 6 that there are three possible kinds of connection among TLM ports and exports. Working through that diagram from left to right:

- **Port-to-port** connections link a port on an inner (child) object to a port on its parent object. In this way, the child object's function call is redirected to the parent's port. From there, of course, it will be connected to an export on some other object.
- **Port-to-export** connections link sibling objects within a given level of hierarchy.
- **Export-to-export** connections link an export on a parent object to an export or imp on one of its children. In this way, the parent appears to implement a function or task, but in fact its child object provides the implementation.

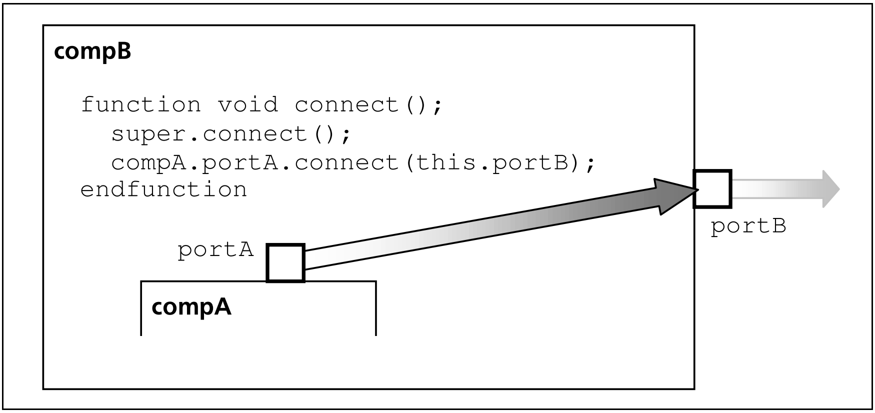
The internal machinery of these connections in OVM is quite complicated, but using them is extremely simple. Every port and export has a `connect` method that can be used to link it to another port or export. As a user, you must simply remember to call the `connect` method of the *requiring* port, and you must supply the *providing* port or export as its argument:

```
requirer.connect(provider);
```

As we have already discussed, the two ends of this chain of connections are handled in a slightly different way. At the originating end (**compA** in Figure 6) the code simply calls a method in its port. At the destination end (**compD** in Figure 6) there is an implementation of the method, associated with – and called from – the special kind of export known as an imp. All other connections are made using the ports' or exports' `connect` methods.

Who Makes the Connections?

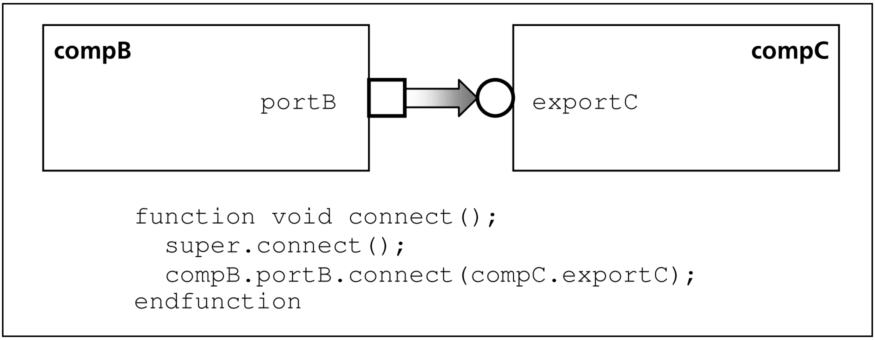
All OVM components have a built-in `connect` phase method that is automatically invoked at the correct point in the life of an OVM testbench – just after the components have been created using `build`, but before any simulation begins. Obviously, the components' `connect` methods provide the right place to make TLM port connections. But which component should do it? Ideally we want the connections between sub-components (children) to be managed by their parent because it is the parent, and *only* the parent, that knows about the organization of its children. Consequently, the three different forms of port connection should be managed like this:



```
function void connect();
  super.connect();
  compA.portA.connect(this.portB);
endfunction
```

Figure 9: Connecting a child's port to its parent's port

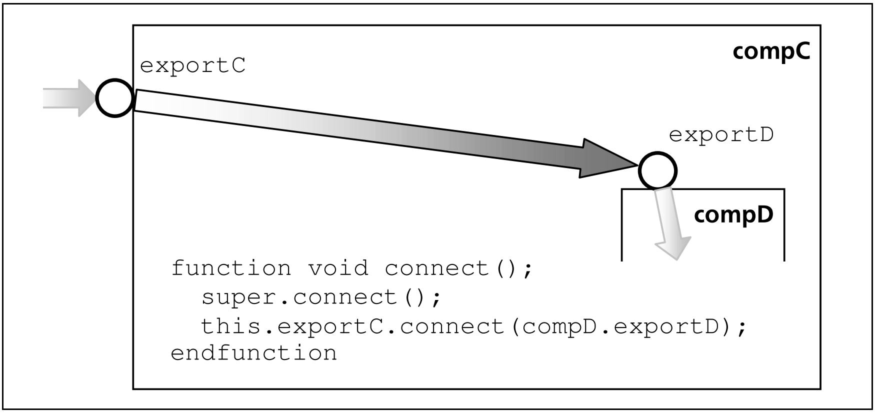
In Figure 9 the enclosing block **compB** wishes to pass on a method call from its child component **compA**. It does so by connecting its child's port to its own. Note that, because it is the child port that *requires* the function, it is the child port's `connect` method that must be called.



```
function void connect();
  super.connect();
  compB.portB.connect(compC.exportC);
endfunction
```

Figure 10: Connecting a child's port to its sibling's export or imp

Figure 10 shows a parent component taking responsibility for connecting a port on one of its children (**compB**) to an export on another of its children (**compC**). Again note that it is the *requiring* port's `connect` method that is called.



```
function void connect();
  super.connect();
  this.exportC.connect(compD.exportD);
endfunction
```

Figure 11: Connecting a parent's export to its child's export or imp

Finally, in Figure 11 we see a parent component **compC** that provides a method through its export. However, it chooses to delegate the implementation of that method to its child component **compD**. Consequently the parent component must connect its own export to the export on its child. Once again note carefully that it is the *parent's* export that *requires* the method, and so it is the parent's `connect` method that must be called.

Note also that at this level we do not know whether the export on **compD** is in fact an export or an imp. Either is acceptable and will satisfy the requirement of the parent's export. If **compD** has an export then it in turn presumably delegates the implementation to its own child component, but at the level shown in Figure 11 we do not need to concern ourselves with that. As far as **compC**'s connection is concerned, **compD** provides the implementation.

Kinds of TLM Connection

Control Flow vs. Data Flow

Up to now we have consistently used the `blocking_put` form of TLM connection. The data source calls a `put` method to give away some data; the data recipient (our BFM) provides a `put` method that accepts the data. It is the data source that is in control and the data recipient that is the target.

It is also possible to reverse the roles, so that the data recipient is in control. In this case the recipient (BFM, in our example) would call a `get` method to pull data from the source, which would provide a target `get` method.

Consequently it is important to be clear about the relationship between control flow and data flow in a TLM connection.

Put transfer: control and data flow are in the same direction

When using `put` transfer, control and data flow in the same direction: the data source is also in control of the timing by calling a `put` method. The data recipient passively accepts a call to its `put` method. Both data and control flow from source to recipient. TLM connections link a port on the source to an export on the recipient.

Get transfer: control and data flow are in opposite directions

When using `get` transfer, control and data flow in opposite directions: the data source passively accepts a call to its `get` method, but the data recipient chooses when this should happen by calling a `get` method. Control flows from recipient to source; data flows the other way. TLM connections link a port on the recipient to an export on the source. The TLM port/export arrangements are relative to *control* flow.

Transport: bidirectional data flow

It is also possible to have a *transport* connection in which a single connection has both `put` and `get` methods, allowing tight coupling between a request and its response. As before, though, the TLM connections link a port on the *caller* to an export on the *target*; the target provides implementations of both `put` and `get`, and these methods are called under control of the caller through the TLM connection.

Blocking vs. Nonblocking

Our `put` example was a *blocking* call. When the data source calls its port's `put` method, that call executes the data recipient's `put` method, with the data source stalled (blocked) until that method returns. A similar description would apply to the `get` method called by a data recipient that initiates a TLM transaction. We can visualize the blocking `get` or `put` activity rather like this:

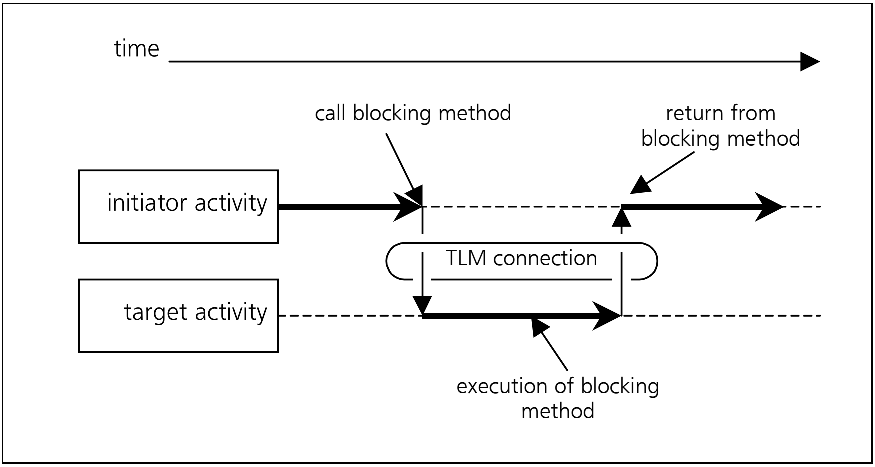


Figure 12: Control flow in a blocking call

Note that the target does nothing unless the initiator explicitly calls it, and the initiator does nothing while waiting for the target to finish its work. The two components work in lockstep, with activity alternating between them under control of the initiator's method calls. TLM connection makes it possible for the initiator and target to remain decoupled – each of them can be written without detailed knowledge of the other, or of the environment in which both are enclosed.

As an alternative to the blocking control flow of Figure 12, TLM offers a *nonblocking* control flow.

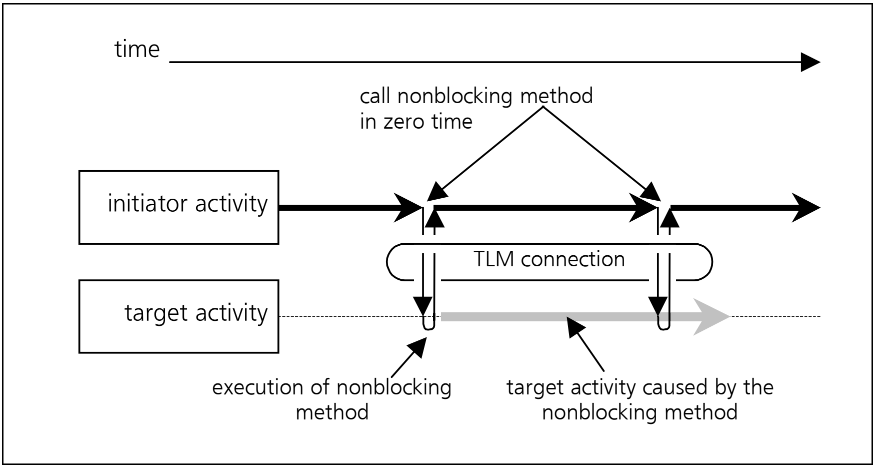

time
call nonblocking method in zero time
initiator activity
TLM connection
target activity
execution of nonblocking method
target activity caused by the nonblocking method

Figure 13: Control flow using nonblocking calls

Using a nonblocking call such as `try_put` or `try_get`, the initiator tries to do a data transfer *and returns immediately*. Consequently the initiator's activity is continuous and uninterrupted, and is not directly synchronized to the target's activity. That's why this style of communication is sometimes known as *asynchronous* in the software world – it has nothing to do with asynchronous hardware! Of course, we can expect that the initiator's action is likely to start some time-consuming activity in the target – indicated by the gray arrow. This proceeds in parallel with the initiator's continuing activity.

There may, though, be an interesting problem when the initiator's second method call occurs. If the target is still busy dealing with the first call from the initiator, it may be unable to accept the initiator's request. That is the reason why these nonblocking methods are named `try_put` and `try_get`. Unlike the blocking `get` or `put`, which simply waits for its target to finish before returning, `try_put` and `try_get` return a status code to indicate success or failure. In both cases they return immediately, but if the function call returns "false" then the initiator knows that the target could not accept the call and it should try again later.

Modeling Idioms in OVM

That concludes our overview of the TLM communication mechanisms as implemented in SystemVerilog for OVM. Naturally you will want to explore the details more fully; the code examples on our web site, and the source code of the OVM kit itself, will be helpful as your experience grows.

However, it would be misleading to finish here without first taking a look at the most common ways in which this powerful technology is used in OVM. Although the full repertoire of TLM methods and connections is available to you, and can be useful when creating custom verification components, there are two special forms of TLM connection that you will encounter more often than any other in typical OVM testbenches. They are *analysis ports* and *sequence pull ports*.

Analysis ports

Another Doulos TechNote in this series, *Observing Activity in VMM and OVM Testbenches*, surveys the special needs of observation in testbenches and highlights the tools provided to facilitate it in both OVM and VMM methodologies. OVM uses a specialized form of TLM connection known as an analysis port, which offers a modified nonblocking put interface that is especially appropriate to the needs of observation.

Sequence pull ports

OVM strongly encourages the construction of verification components using an *agent* architecture in which stimulus is created by a data generator known as a *sequencer* and passed to a pin-level BFM known as a *driver*. As you might expect, the communication between them uses TLM. The driver is responsible for timing, and therefore acts as the initiator; the connection is known as a *sequence pull* connection because the driver "pulls" data items from the sequencer as it needs them. Of course this could be done using a regular TLM `blocking_get` connection, but the full power of the sequences mechanism requires a somewhat more complex interaction between driver and sequencer. To support this flexibility, the sequence pull connection is extended with a number of additional methods to provide enhanced communication between driver and sequencer. Comprehensive training and support material on this topic is available from Doulos and other sources.