VMM PRIMER

Writing Command-Layer Master Transactors

Author(s):

Janick Bergeron

Version 1.3 / Dec 01, 2006

Introduction

The *Verification Methodology Manual for SystemVerilog* book was never written as a training book. It was designed to be a reference document that defines what is—and is not—compliant with the methodology described within it.

This primer is designed to learn how to write VMM-compliant command-lever master transactors—transactors with a transaction-level interface on one side and pin wiggling on the other, also known as bus-functional models (BFM). Other primers will eventually cover other aspects of developing VMM-compliant verification assets, such as slave transactors, monitors, functional-layer transactors, generators, assertions and verification environments.

The protocol used in this primer was selected for its simplicity. Because of its simplicity, it does not require the use of many elements of the VMM standard library. It is sufficient to achieve the goal of demonstrating, step by step, how to create a simple VMM-compliant master transactor.

This document is written in the same order you would implement a command-layer transactor. As such, you should read it in a sequential fashion. You can use the same sequence to create your own specific transactor.

A word of caution however: it may be tempting to stop reading this primer half way through, as soon as a functional transactor is available. VMM compliance is a matter of degree. Once a certain minimum level of functionality is met, a transactor may be declared VMM compliant. But additional VMM functionality—such as callbacks—will make it much easier to use in different verification environments. Therefore, you should read—and apply—this primer in its entirety.

This primer will show how to apply the various VMM guidelines, not attempt to justify them or present alternatives. If you are interested in learning more about the justifications of various techniques and approaches used in this primer, you should refer to the VMM book under the relevant quoted rules and recommendations.

The Protocol

The protocol used in this primer is the AMBA™ Peripheral Bus (APB) protocol. It is a simple single-master address-based parallel bus providing atomic individual read and write cycles. The protocol specification can be found in the AMBA™ Specification (Rev 2.0) available from ARM (http://arm.com).

When writing a reusable transactor, you have to think about all possible applications it may be used in, not just the device you are using it for the first time. Therefore, even though the device in this primer only supports 8 address bits and 16 data bits, the APB transactors should be written for the entire 32-bit of address and data information.

The Verification Components

Figure 1 illustrates the various components that will be created throughout this primer. A command-layer master transactor interfaces directly to the DUT signals and initiates transactions upon requests on a transaction-level interface.

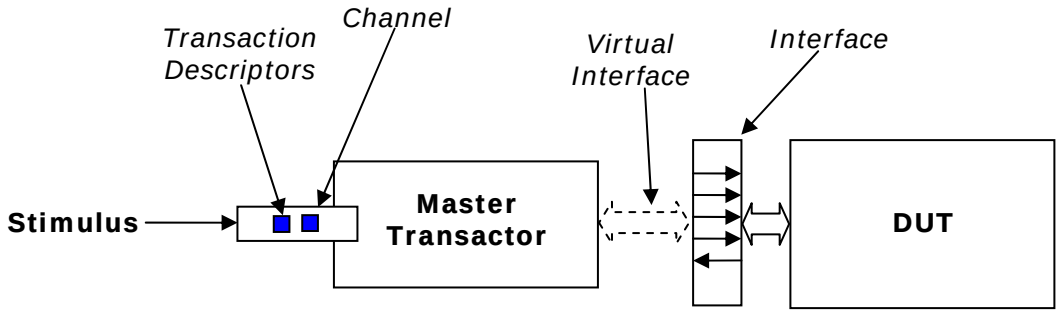

Figure 1

The complete source code for this primer can be found in the *Command_Master_Xactor* directory. To run the example, simply type “make”.

Step 1: The Interface

The first step is to define the physical signals used by the protocol to exchange information between a master and a slave. A single exchange of information (a READ or a WRITE operation) is called a *transaction*. There may be multiple slaves on an APB bus but there can only be one master. Slaves are differentiated by responding to different address ranges.

The signals are declared inside an *interface* (Rule 4-4). The name of the *interface* is prefixed with “apb_” to identify that it belongs to the APB protocol (Rule 4-5). The entire content of the file declaring the *interface* is embedded in an `ifndef/`define/`endif construct. This is an old C trick that allows the file to be included multiple time, whenever required, without causing multiple-definition errors.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if;
   ...
endinterface: apb_if

`endif
```

The signals, listed in the AMBA™ Specification in Section 2.4, are declared as *wires* (Rule 4-6) inside the *interface*.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;
   ...
endinterface: apb_if

`endif
```

Because this is a synchronous protocol, *clocking* blocks are used to define the direction and sampling of the signals (Rule 4-7, 4-11).

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking mck @(posedge pclk);
      output paddr, psel, penable, pwrite, pwdata;
      input  prdata;
   endclocking: mck
   ...
endinterface: apb_if

`endif
```

The *clocking* block defining the synchronous signals is specified in the *modport* for the APB master transactor (Rule 4-9, 4-11, 4-12). The clock signal need not be specified as it is implicit in the *clocking* block.

File: apb/apb_if.sv

```
`ifndef APB_IF__SV
`define APB_IF__SV

interface apb_if(input bit pclk);
   wire [31:0] paddr;
   wire        psel;
   wire        penable;
   wire        pwrite;
   wire [31:0] prdata;
   wire [31:0] pwdata;

   clocking mck @(posedge pclk);
      output paddr, psel, penable, pwrite, pwdata;
      input  prdata;
   endclocking: mck

   modport master(clocking mck);

endinterface: apb_if

`endif
```

The *interface* declaration is now sufficient for writing a master APB transactor. To be fully compliant, it should eventually include a *modport* for a slave and a passive monitor transactor (Rule 4-9). These can be added later, when these transactors will be written.

Step 2: Connecting to the DUT

The interface may now be connected to the DUT. It is instantiated in a top-level *module*, alongside of the DUT instantiation (Rule 4-13). The connection to the DUT pins are specified using a hierarchical reference to the *wires* in the *interface* instance.

File: Command_Master_Xactor/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);
   ...
   slave_ip dut_slv(...,
                    .apb_addr   (apb0.paddr[7:0]  ),
                    .apb_sel    (apb0.psel        ),
                    .apb_enable (apb0.penable     ),
                    .apb_write  (apb0.pwrite      ),
                    .apb_rdata  (apb0.prdata[15:0]),
                    .apb_wdata  (apb0.pwdata[15:0]),
                    ...);
```

```
endmodule: tb_top
```

This top-level module also contains the clock generators (Rule 4-15), using the *bit* type (Rule 4-17) and ensuring that no clock edges will occur at time zero (Rule 4-16).

File: Command_Master_Xactor/tb_top.sv

```
module tb_top;
   bit clk = 0;
   apb_if apb0(clk);
   ...

   my_design dut(...,
                 .apb_addr   (apb0.paddr[7:0]  ),
                 .apb_sel    (apb0.psel        ),
                 .apb_enable (apb0.penable     ),
                 .apb_write  (apb0.pwrite      ),
                 .apb_rdata  (apb0.prdata[15:0]),
                 .apb_wdata  (apb0.pwdata[15:0]),
                 ...
                 .clk       (clk));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 3: The Transaction Descriptor

The next step is to define the APB transaction descriptor. Traditionally, tasks would have been defined, one for the READ transaction and one for the WRITE transaction.

File: apb/apb_master.sv

```
task read(input  bit   [31:0] addr,
          output logic [31:0] data);

task write(input bit [31:0] addr,
           input bit [31:0] data);
```

This works well for directed tests, but not at all for random tests. A random test requires a transaction descriptor (Rule 4-54). This descriptor is a *class* (Rule 4-53) extended from the *vmm_data* class (Rule 4-55), containing a public *rand* property enumerating the directed tasks (Rule 4-60, 4-62) and public *rand* properties for each task argument (Rule 4-59, 4-62). If an argument is the same across multiple tasks, a single property can be used. It also needs a *static vmm_log* property instance used to issue messages from the transaction descriptor. This instance of the message service interface is passed to the *vmm_data* constructor (Rule 4-58).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: apb_rw
...
`endif
```

Note how the same property is used for "data". It will be interpreted differently depending on the transaction kind (Rule 4-71). In a WRITE transaction, it is interpreted as the data to be written. In a READ transaction, the random content is initially ignored and it will be replaced by the data value that was read. The type for the "data" property is *logic* as it is a superset of the *bit* type and will allow the description of READ cycles to reflect unknown results.

Although the transaction descriptor is not yet VMM-compliant, it is has the minimum functionality to be used by a transactor. A transaction-level interface will be required to transfer transaction descriptors to a transactor to be executed. This is done using the `*vmm_channel* macro (Rule 4-56).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
```

```
   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Step 4: The Master Transactor

The master transactor can now be started. It is a class (Rule 4-91) derived from the *vmm_xactor* base class (4-92).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV
...
class apb_master extends vmm_xactor;
   ...
endclass: apb_master

`endif
```

The task implementing the READ and WRITE cycles are implemented in this class. They are declared *virtual* so the transactor may be extended to modify the behavior of these tasks if so required. They are also declared *protected* to prevent them from being called from outside the class and create concurrent bus access problems.

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include “apb_if.sv”
`include “apb_rw.sv”
...
class apb_master extends vmm_xactor;
   ...
   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write
   ...
endclass: apb_master
```

```
`endif
```

The transactor needs a transaction-level interface to receive transactions to be executed and a physical-level interface to wiggle pins. The former is done using a *vmm_channel* instance and the latter is done using a *virtual modport*. Both are passed to the transactor as constructor arguments (Rule 4-108, 4-113) and both are saved in public properties (Rule 4-109, 4-112).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.master  sigs,
                apb_rw_channel         in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in_chan;
      ...
   endfunction: new
   ...
   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write
   ...
endclass: apb_master

`endif
```

When the transactor is reset, the input channel must be flushed and the critical output signals must be driven to their idle state. This is accomplished in the extension of the *vmm_xactor::reset_xactor()* method (Table A-8). This method may

be called in the transactor constructor to initialize the output signals to their idle state, or explicit signal assignments may be used in the constructor.

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.master  sigs,
                apb_rw_channel         in_chan = null);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor
   ...
   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write
   ...
endclass: apb_master

`endif
```

The transaction descriptors are pulled from the input channel and translated into method calls in the *main()* task (Rule 4-93). The most flexible transaction execution mechanism uses the active slot as it supports block, nonblocking and out-of-order

execution models (Rules 4-121, 4-122, 4-122, 4-124 and 4-129). Because the protocol supports being suspended between transactions, the *vmm_xactor::wait_if_stopped_or_empty()* method is used to suspend the execution of the transactor if it is stopped.

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include “apb_if.sv”
`include “apb_rw.sv”
...
class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new(“APB Master Input Channel”, name);
      this.in_chan = in chan;
      this.sigs.mck.psel    <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
         ...
         this.in_chan.start();
         case (tr.kind)
         apb_rw::READ : this.read(tr.addr, tr.data);
         apb_rw::WRITE: this.write(tr.addr, tr.data);
         endcase
         this.in_chan.complete();
         ...
```

```
         this.in_chan.remove();
      end
   endtask: main

   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write

endclass: apb_master

`endif
```

The READ and WRITE tasks are coded exactly as they would be if good old Verilog was used. It is a simple matter of assigning output signals to the proper value then sampling input signals at the right point in time. The only difference is that the physical signals are accessed through the *clocking* block of the *virtual modport* instead of pins on a module and they can only be assigned using nonblocking assignments. Similarly, the active clock edge is defined by waiting on the *clocking* block itself, not an edge of an input signal (Rule 4-7, 4-12).

File: apb/apb_master.sv

```
...
   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      this.sigs.mck.paddr   <= addr;
      this.sigs.mck.pwrite  <= '0;
      this.sigs.mck.psel    <= '1;
      @ (this.sigs.mck);
      this.sigs.mck.penable <= '1;
      @ (this.sigs.mck);
      data = this.sigs.mck.prdata;
      this.sigs.mck.psel    <= '0;
      this.sigs.mck.penable <= '0;
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      this.sigs.mck.paddr   <= addr;
      this.sigs.mck.pwdata  <= data;
      this.sigs.mck.pwrite  <= '1;
      this.sigs.mck.psel    <= '1;
      @ (this.sigs.mck);
      this.sigs.mck.penable <= '1;
      @ (this.sigs.mck);
      this.sigs.mck.psel    <= '0;
```

```
      this.sigs.mck.penable <= '0;
   endtask: write
```

Step 5: The First Test

Although not completely VMM-compliant, the transactor can now be used to exercise the DUT. It is instantiated in a verification environment class, extended from the *vmm_env* base class. The transactor is constructed in the extension of the *vmm_env::build()* method (Rule 4-34, 4-35) and started in the extension of the *vmm_env::start()* method (Rule 4-41). Note that if the transactor is required to configure the DUT in the extension of the *vmm_env::cfg_dut()* method, it needs to be started in that latter method. The reference to the *interface* encapsulating the APB physical signals is made using a hierarchical reference in the extension of the *vmm_env::build()* method.

File: Command_Master_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_master.sv"

class tb_env extends vmm_env;
   apb_master mst;

   virtual function void build();
      super.build();
      this.mst = new("0", 0, tb_top.apb0);
   endfunction: build

   virtual task start();
      super.start();
      this.mst.start_xactor();
   endtask: start
   ...
endclass: tb_env

`endif
```

A simple test to perform a write followed by a read of the same address can now be written and executed to verify the correct operation of the transactor and the DUT interface. The test is written in a *program* (Rule 4-27) that instantiates the verification environment (Rule 4-28). The directed stimulus is created by instantiating transaction descriptors appropriately filled. It is a good idea to randomize these descriptors, only constraining those properties that are needed for the directed test. This way, any additional property will be randomized instead of defaulting to always the same value.

File: Command_Master_Xactor/test_simple.sv

```
`include "tb_env.sv"

program simple_test;

vmm_log log = new("Test", "Simple");
tb_env env = new;
initial begin
   apb_rw rd, wr;
   bit ok;

   env.start();

   wr = new;
   ok = wr.randomize() with {
      kind == WRITE;
   };
   if (!ok) begin
      `vmm_fatal(log, "Unable to randomize WRITE cycle");
   end
   env.mst.in_chan.put(wr);

   rd = new;
   ok = rd.randomize() with {
      kind == READ;
      addr == wr.addr;
   };
   if (!ok) begin
      `vmm_fatal(log, "Unable to randomize READ cycle");
   end
   env.mst.in_chan.put(rd);

   if (rd.data[15:0] !== wr.data[15:0]) begin
      `vmm_error(log, "Readback value != write value");
   end

   log.report();
   $finish();
end

endprogram
```

This test can be run many times, each time with a different seed, to verify the transactor and the DUT using different addresses.

Step 6: Extension Points

The transactor, as presently coded, will provide basic functionality. You may be tempted to stop here because you now have a transactor that can perform READ and

WRITE cycles with identical capabilities to one you would have written using the old Verilog language.

The problem is that the transactor, as coded, is not very reusable. It will not be possible to modify the behavior of this transactor—for example to introduce delays between transactions, to synchronize the start of a transaction with some other external event, or modify a transaction to inject errors—without modifying the transactor itself or constantly rewrite the *apb_master::read()* and *apb_master::write()* virtual methods.

A callback method allows a user to extend the behavior of a transactor without having to modify the transactor itself. Callback methods should be provided before and after a transaction executes (Rule 4-155). The “pre-transaction” callback method allows errors to be injected and delays to be inserted. The “post-transaction” callback method allows delays to be inserted and the result of the transaction to be recorded in a functional coverage model or checked against an expected response.

The callback methods are first defined as *virtual tasks* or *virtual void functions* (Rule 4-160) in a callback façade class extended from the *vmm_xactor_callbacks* base class (Rule 4-159). It is a good idea to create a mechanism in the “pre-transaction” callback method to allow an entire transaction to be skipped or dropped (Rule 4-160).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include “apb_if.sv”
`include “apb_rw.sv”

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
```

```
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new("APB Master Input Channel", name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= '0;
      this.sigs.mck.penable <= '0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
         ...
         this.in_chan.start();
         case (tr.kind)
         apb_rw::READ : this.read(tr.addr, tr.data);
         apb_rw::WRITE: this.write(tr.addr, tr.data);
         endcase
         this.in_chan.complete();
         ...
         this.in_chan.remove();
      end
   endtask: main

   virtual protected task read(input  bit   [31:0] addr,
                               output logic [31:0] data);
      ...
   endtask: read

   virtual protected task write(input bit [31:0] addr,
                                input bit [31:0] data);
      ...
   endtask: write

endclass: apb_master

`endif
```

Next, the appropriate callback method needs to be invoked at the appropriate point in the execution of the transaction, using the *`vmm_callback()* macro (Rule 4-163).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include “apb_if.sv”
`include “apb_rw.sv”

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new(“APB Master Input Channel”, name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         bit drop;

         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
```

```
            drop = 0;
            `vmm_callback(apb_master_cbs, pre_cycle(this, tr, drop));
            if (drop) begin
               this.in_chan.remove();
               continue;
            end

            this.in_chan.start();
            case (tr.kind)
            apb_rw::READ : this.read(tr.addr, tr.data);
            apb_rw::WRITE: this.write(tr.addr, tr.data);
            endcase
            this.in_chan.complete();

            `vmm_callback(apb_master_cbs, post_cycle(this, tr));

            this.in_chan.remove();
         end
      endtask: main

      virtual protected task read(input  bit   [31:0] addr,
                                  output logic [31:0] data);
         ...
      endtask: read

      virtual protected task write(input bit [31:0] addr,
                                   input bit [31:0] data);
         ...
      endtask: write

   endclass: apb_master

   `endif
```

Step 7: Debug Messages

To be truly reusable, it should be possible to understand what the transactor does and debug its operation without having to inspect the source code. This capability may even be a basic requirement if you plan on shipping encrypted or compiled code.

Debug messages should be added at judicious points to indicate what the transactor is about to do, is doing or has done. These debug messages are inserted using the *`vmm_trace()*, *`vmm_debug()* or *`vmm_verbose()* macros (Rule 4-101).

File: apb/apb_master.sv

```
`ifndef APB_MASTER__SV
`define APB_MASTER__SV

`include “apb_if.sv”
`include “apb_rw.sv”

typedef class apb_master;
class apb_master_cbs extends vmm_xactor_callbacks;
   virtual task pre_cycle(apb_master xactor,
                          apb_rw     cycle,
                          ref bit    drop);
   endtask: pre_cycle

   virtual task post_cycle(apb_master xactor,
                           apb_rw     cycle);
   endtask: post_cycle
endclass: apb_master_cbs

class apb_master extends vmm_xactor;
   apb_rw_channel        in_chan;
   virtual apb_if.master sigs;

   function new(string                name,
                int unsigned          stream_id,
                virtual apb_if.master sigs,
                apb_rw_channel        in_chan = null);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      if (in_chan == null)
         in_chan = new(“APB Master Input Channel”, name);
      this.in_chan = in chan;
      this.sigs.mck.psel <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset(rst_typ);
      this.in_chan.flush();
      this.sigs.mck.psel <= ’0;
      this.sigs.mck.penable <= ’0;
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      @ (this.sigs.mck);
      forever begin
         apb_rw tr;
         bit drop;

         this.wait_if_stopped_or_empty(this.in_chan);
         this.in_chan.activate(tr);
```

```
            drop = 0;
            `vmm_callback(apb_master_cbs, pre_cycle(this, tr, drop));
            if (drop) begin
               `vmm_debug(log, {"Dropping transaction...\n",
                                tr.psdisplay("   ")});
               this.in_chan.remove();
               continue;
            end

            `vmm_trace(log, {"Starting transaction...\n",
                             tr.psdisplay("   ")});

            this.in_chan.start();
            case (tr.kind)
            apb_rw::READ : this.read(tr.addr, tr.data);
            apb_rw::WRITE: this.write(tr.addr, tr.data);
            endcase
            this.in_chan.complete();

            `vmm_trace(log, {"Completed transaction...\n",
                             tr.psdisplay("   ")});

            `vmm_callback(apb_master_cbs, post_cycle(this, tr));

            this.in_chan.remove();
         end
      endtask: main

      virtual protected task read(input  bit   [31:0] addr,
                                  output logic [31:0] data);
         ...
      endtask: read

      virtual protected task write(input bit [31:0] addr,
                                   input bit [31:0] data);
         ...
      endtask: write

   endclass: apb_master

   `endif
```

Step 8: Standard Methods

You can now run the "simple test" with the latest version of the transactor and increase the message verbosity to see debug messages displayed as the test executes the various transactions.

File: Command_Master_Xactor/Makefile

```
% vcs –sverilog –ntb_opts vmm +vmm_log_default=trace ...
```

But the messages will be not very useful as they do not display the content of the executed transactions. That's because the *psdisplay()* method, defined in the *vmm_data* base class does not know about the content of the APB transaction descriptor. For that method to display the information that is relevant for the APB transaction, it is necessary to overload this method in the transaction descriptor class (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new
   ...
   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
               this.kind.name(), this.addr, this.data);
   endfunction: psdisplay
   ...
endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Re-running the test with increased verbosity now yields useful and meaningful debug messages. The *vmm_data::psdisplay()* method is one of the pre-defined methods in the *vmm_data* base class that users expect to be provided to simplify and abstract common operations on transaction descriptors. These common operations include creating transaction descriptors (*vmm_data::allocate()*), copying transaction descriptors (*vmm_data::copy()*), comparing transaction descriptors (*vmm_data::compare()*) and checking that the content of transaction descriptors is valid (*vmm_data::is_valid()*) (Rule 4-76).

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw instance");
         return null;
      end

      super.copy_data(tr);
      tr.kind = this.kind;
      tr.addr = this.addr;
      tr.data = this.data;

      return tr;
   endfunction: copy

   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
               this.kind.name(), this.addr, this.data);
   endfunction: psdisplay

   virtual function bit is_valid(bit silent = 1,
                                 int kind   = -1);
      return 1;
   endfunction: is_valid

   virtual function bit compare(input  vmm_data to,
                                output string   diff,
                                input  int      kind = -1);
```

```
      apb_rw tr;

      if (to == null) begin
         `vmm_fatal(log, "Cannot compare to NULL reference");
         return 0;
      end
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot compare against non-apb_rw
instance");
         return 0;
      end

      if (this.kind != tr.kind) begin
         $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
         return 0;
      end

      if (this.addr !== tr.addr) begin
         $sformat(diff, "Addr 0x%h !== 0x%h", this.addr, tr.addr);
         return 0;
      end

      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data, tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
...
`endif
```

Three other standard methods, *vmm_data:byte_size()*, *vmm_data::byte_pack()* and *vmm_data::byte_unpack()* should also be overloaded for packet-oriented transactions, where the content of the transaction is transmitted over a serial interface (Recommendation 4-77).

Step 9: Transaction Generator

To promote the use of random stimulus, it is a good idea to pre-define random transaction generators whenever transaction descriptors are defined. It is a simple matter of using the *`vmm_atomic_gen()* and *`vmm_scenario_gen()* macros (Recommendation 5-23, 5-24) in the transaction descriptor file.

File: apb/apb_rw.sv

```
`ifndef APB_RW__SV
`define APB_RW__SV

`include "vmm.sv"

class apb_rw extends vmm_data;
   static vmm_log log = new("apb_rw", "class");

   rand enum {READ, WRITE} kind;
   rand bit   [31:0] addr;
   rand logic [31:0] data;

   function new();
      super.new(this.log);
   endfunction: new

   virtual function vmm_data allocate();
      apb_rw tr = new;
      return tr;
   endfunction: allocate

   virtual function vmm_data copy(vmm_data to = null);
      apb_rw tr;

      if (to == null) tr = new;
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot copy into non-apb_rw instance");
         return null;
      end

      super.copy_data(tr);
      tr.kind = this.kind;
      tr.addr = this.addr;
      tr.data = this.data;

      return tr;
   endfunction: copy

   virtual function string psdisplay(string prefix = "");
      $sformat(psdisplay, "%sAPB %s @ 0x%h = 0x%h", prefix,
               this.kind.name(), this.addr, this.data);
   endfunction: psdisplay

   virtual function bit is_valid(bit silent = 1,
                                 int kind   = -1);
      return 1;
   endfunction: is_valid

   virtual function bit compare(input  vmm_data to,
                                output string   diff,
                                input  int      kind = -1);
```

```
      apb_rw tr;

      if (to == null) begin
         `vmm_fatal(log, "Cannot compare to NULL reference");
         return 0;
      end
      else if (!$cast(tr, to)) begin
         `vmm_fatal(log, "Cannot compare against non-apb_rw
instance");
         return 0;
      end

      if (this.kind != tr.kind) begin
         $sformat(diff, "Kind %s !== %s", this.kind, tr.kind);
         return 0;
      end

      if (this.addr !== tr.addr) begin
         $sformat(diff, "Addr 0x%h !== 0x%h", this.addr, tr.addr);
         return 0;
      end

      if (this.data !== tr.data) begin
         $sformat(diff, "Data 0x%h !== 0x%h", this.data, tr.data);
         return 0;
      end

      return 1;
   endfunction: compare

endclass: apb_rw

`vmm_channel(apb_rw)
`vmm_atomic_gen(apb_rw, "APB Bus Cycle")
`vmm_scenario_gen(apb_rw, "APB Bus Cycle")

`endif
```

Step 10: Top-Level File

To help users include all necessary files, without having to know the detailed filenames and file structure of the transactor, interface and transaction descriptor, it is a good idea to create a top-level file that will automatically include all source files that make up the verification IP for a protocol.

File: apb/apb.sv

```
`ifndef APB__SV
`define APB__SV
```

```
`include "vmm.sv"
`include "apb_if.sv"
`include "apb_rw.sv"
`include "apb_master.sv"

`endif
```

In this example, we implemented only a master transactor; but a complete VIP for a protocol would also include a slave transactor and a passive monitor transactor. All of these transactors would be included in the top-level file.

Step 11: Congratulations!

You have now completed the creation of a VMM-compliant command-layer master transactor!

Upon reading this primer, you probably realized that there is much code that is similar across different master transactors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant master transactor and only modify what is unique or different for your protocol? That can easily be done using the "vmmgen" tool provided with VCS 2006.06-6. Based on a few simple question and answers, it will create a template for various components of a VMM-compliant master transactor.

Command

```
% rvmgen -l sv
```

The relevant templates for writing command-layer master transactors are:

1) Physical interface declaration

2) Transaction Descriptor

3) Driver, Physical-level, Half-duplex

Note that the menu number used to select the appropriate template may differ from the number in the above list.

You may consider reading other publications in this series to learn how to write VMM-compliant command-layer slave transactors, command-layer passive monitor transactors, functional-layer transactors or verification environments.