VMM PRIMER

Writing Command-Layer Monitors

Author(s):

Janick Bergeron

Version 0.3 / 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 monitors—transactors that observe pin wiggling on one side and report the observed transactions on a transaction-level interface on the other side. Other primers will eventually cover other aspects of developing VMM-compliant verification assets, such as master and slave transactors, 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 monitor. 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 monitor is available. VMM compliance is a matter of degree. Once a certain minimum level of functionality is met, a monitor 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 monitor, 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 monitor 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 monitor interfaces directly to the DUT signals and reports all observed transactions on a transaction-level interface.

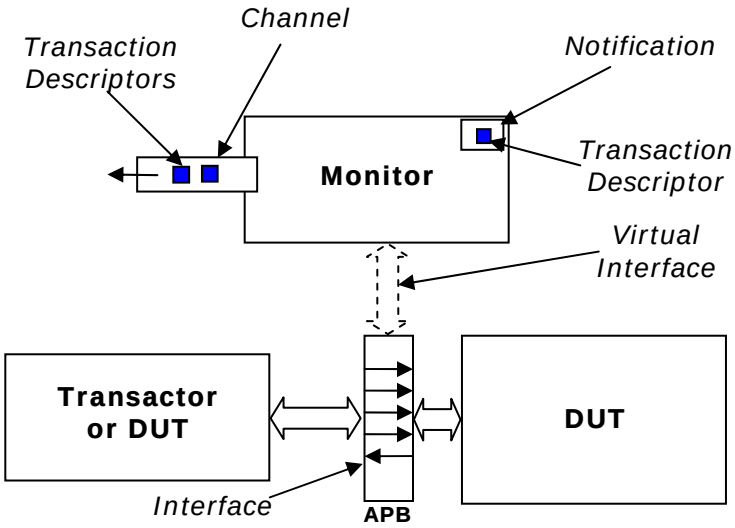

Figure 1

The complete source code for this primer can be found in the *Command_Monitor_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 pck @(posedge pclk);
      input paddr, psel, penable, pwrite, prdata, pwdata;
   endclocking: pck
   ...
endinterface: apb_if
```

```
`endif
```

The *clocking* block defining the synchronous signals is specified in the *modport* for the APB monitor (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 pck @(posedge pclk);
      input paddr, psel, penable, pwrite, prdata, pwdata;
   endclocking: pck

   modport passive(clocking pck);

endinterface: apb_if

`endif
```

The *interface* declaration is now sufficient for writing a passive APB monitor. To be fully compliant, it should eventually include a *modport* for a master and a slave 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_Monitor_Xactor/tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);
   ...
   master_ip dut_mst(...,
                     .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]),
                    ...);
   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_Monitor_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 (Rule 4-54). This descriptor is a *class* (Rule 4-53) extended from the *vmm_data* class (Rule 4-55), containing a public property enumerating the various transactions that can be observed by the monitor (Rule 4-60, 4-62) and public properties for each parameter or value in the transaction (Rule 4-59, 4-62). 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");

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

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

A single property is used for "data", despite the fact that the APB bus has separate read and write data buses. Because the APB does not support concurrent read/write transactions, there can only be one data value valid at any given time. In fact, it is possible to implement the APB bus using a single data bus (see section 5.6.4 of the AMBA™ Specification (Rev 2.0)). The "data" class property is 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, it is the data value that was read. The type for the "data" property is *logic* as it 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 monitor. 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");

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

   function new();
```

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

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

Step 4: The Monitor

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

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV
...
class apb_monitor extends vmm_xactor;
   ...
endclass: apb_monitor

`endif
```

The transactor needs a physical-level interface to observe transactions. The physical-level interface is done using a *virtual modport* passed to the transactor as a constructor argument (Rule 4-108) and is saved in a public property (Rule 4-109).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include "apb_if.sv"
`include "apb_rw.sv"
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.passive sigs;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new("APB Monitor", name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
```

```
   ...
endclass: apb_monitor

`endif
```

The transaction descriptors are filled in from observations on the physical interface in the *main()* task (Rule 4-93). The observation of READ and WRITE transactions is coded exactly as it would be if good old Verilog was used. It is a simple matter of 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. 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_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include “apb_if.sv”
`include “apb_rw.sv”
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.passive sigs;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new(“APB Monitor”, name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         ...
         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
```

```
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         ...
   end
   endtask: main
endclass: apb_monitor

`endif
```

Once a monitor recognizes the start of a transaction, it must not be stopped until the end of the transaction is observed and the entire transaction is reported. Otherwise, if the monitor is stopped in the middle of a transaction stream, it will report a transaction error at best or transactions composed of information from two different transactions. Therefore, the *vmm_xactor::wait_if_stopped()* method is called only before the beginning of transaction.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include “apb_if.sv”
`include “apb_rw.sv”
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      ...
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
```

```
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                     this.sigs.pck.prdata :
                     this.sigs.pck.pwdata;
         ...
   end
   endtask: main
endclass: apb_monitor

`endif
```

Step 5: Reporting Transactions

The monitor must have the ability to inform the testbench that a specific transaction has been observed. Simply displaying the observed transactions is not very useful as it will require that the results be manually checked every time. Observed transactions can be reported by indicating a notification in the *vmm_xactor::notify* property. The observed transaction, being derived from *vmm_data*, is attached to a newly defined “OBSERVED” notification and can be recovered by all interested parties waiting on the notification.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

`include “apb_if.sv”
`include “apb_rw.sv”
...
class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   ...
   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                  name,
                int unsigned            stream_id,
                virtual apb_if.passive sigs,
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new
   ...
```

```
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                     apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                     this.sigs.pck.prdata :
                     this.sigs.pck.pwdata;
         ...
         this.notify.indicate(OBSERVED, tr);
         ...
   end
   endtask: main
endclass: apb_monitor

`endif
```

Step 6: The First Test

Although not completely VMM-compliant, the monitor can now be used to monitor the activity on an APB bus. It is instantiated in a verification environment class, extended from the *vmm_env* base class. The monitor 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). 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_Monitor_Xactor/tb_env.sv

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include “vmm.sv”
`include “apb_monitor.sv”
```

```
class tb_env extends vmm_env;
   ...
   apb_monitor mon;

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

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

`endif
```

As the simulation progress, the monitor can report the observed transactions. The monitor can also be used to determine when it is time to end the test by reporting when enough APB transactions have been observed.

```
`ifndef TB_ENV__SV
`define TB_ENV__SV

`include "vmm.sv"
`include "apb_monitor.sv"

class tb_env extends vmm_env;
   ...
   apb_monitor mon;

   int stop_after = 10;

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

   virtual task start();
      super.start();
      ...
      this.mon.start_xactor();
      ...
      fork
         ...
         forever begin
            apb_rw tr;
```

```
            this.mon.notify.wait_for(apb_monitor::OBSERVED);
            this.stop_after--;
            if (this.stop_after <= 0) -> this.end_test;
            $cast(tr,
                  this.mon.notify.status(apb_monitor::OBSERVED));
            tr.display("Notified: ");
         end
      join_none
   endtask: start

   virtual task wait_for_end();
      super.wait_for_end();
      @ (this.end_test);
   endtask: wait_for_end
   ...
endclass: tb_env

`endif
```

A test can control how long the simulation should run by simply setting the value of the *tb_env::stop_after* property. The test is written in a *program* (Rule 4-27) that instantiates the verification environment (Rule 4-28).

File: Command_Monitor_Xactor/test_simple.sv

```
`include “tb_env.sv”

program simple_test;

vmm_log log = new(“Test”, “Simple”);
tb_env env = new;
initial begin
   env.stop_after = 5;
   env.run();

   $finish();
end

endprogram
```

Step 7: Standard Methods

The transactions reported by the monitor are not very useful as they do not show the content of the observed 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");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   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 now yields useful and meaningful transaction monitoring. 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");

   enum {READ, WRITE} kind;
   bit   [31:0] addr;
   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 8: More on Transactions

The *vmm_notify* notification service interface can notify an arbitrary number of threads or transactors but it can store only one transaction at a time. Should a higher-level thread or transactor have the potential to block, it is possible that transactions will be missed. The *vmm_channel* transaction-level interface is used to provide a buffering reporting mechanism, where the transaction descriptors are kept until they are explicitly consumed. Transactions reported via a *vmm_notify* can be consumed by an arbitrary number of consumers but must be consumed in zero-time. Transactions reported via a *vmm_channel* can only be consumed by a single consumer, but that consumer can only consume transactions at its own pace. Should there be no consumers, a *vmm_notify* mechanism will not accumulate transaction descriptors. However, a channel with no consumer will accumulate transaction descriptors, unless it is sunk using the *vmm_channel::sink()* method. The output channel, passed to the transactor as constructor arguments (Rule 4-113), is saved in a public property (Rule 4-112). It is sunk by default to avoid accidental memory leakage. Transaction descriptors are added to the output channel using the *vmm_channel::sneak()* method (Rule 4-140) to avoid the monitor from blocking on a full channel and missing transactions.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new
   ...
   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
```

```
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            ...
            this.notify.indicate(OBSERVED, tr);
            this.out_chan.sneak(tr);
      end
      endtask: main
endclass: apb_monitor

`endif
```

When the transactor is reset, the output channel must be flushed. This is accomplished in the extension of the *vmm_xactor::reset_xactor()* method (Table A-8).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
```

```
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         ...
         tr.kind = (this.sigs.pck.pwrite) ?
                     apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                     this.sigs.pck.prdata :
                     this.sigs.pck.pwdata;
         ...
         this.notify.indicate(OBSERVED, tr);
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

It may be useful for the higher-level functions using the transactions reported by the monitor to know when the transaction was started and when it ended. These transaction endpoints are recorded in the transaction descriptor itself by indicating the *vmm_data::STARTED* and *vmm_data::ENDED* notifications (Rule 4-142).

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
```

```
                apb_rw_channel         out_chan = null
                ...);
      super.new("APB Master", name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);
         tr = new;
         tr.notify.indicate(vmm_data::STARTED);

         tr.kind = (this.sigs.pck.pwrite) ?
                     apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                     this.sigs.pck.prdata :
                     this.sigs.pck.pwdata;
         tr.notify.indicate(vmm_data::ENDED);
         ...
         this.notify.indicate(OBSERVED, tr);
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

Step 9: Debug Messages

To be truly reusable, it should be possible to understand what the monitor 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 monitor 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_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan = null
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;

      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
         this.wait_if_stopped();
```

```
            `vmm_trace(log, "Waiting for start of transaction...");

            // Wait for a SETUP cycle
            do @ (this.sigs.pck);
            while (this.sigs.pck.psel !== 1'b1 ||
                   this.sigs.pck.penable !== 1'b0);
            tr = new;
            tr.notify.indicate(vmm_data::STARTED);

            tr.kind = (this.sigs.pck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.pck.paddr;

            @ (this.sigs.pck);
            if (this.sigs.pck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
            end
            tr.data = (tr.kind == apb_rw::READ) ?
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            tr.notify.indicate(vmm_data::ENDED);
            ...
            `vmm_trace(log, {"Observed transaction...\n",
                             tr.psdisplay("   ")});
            this.notify.indicate(OBSERVED, tr);
            this.out_chan.sneak(tr);
      end
      endtask: main
endclass: apb_monitor

`endif
```

You can now run the “simple test” with the latest version of the monitor and increase the message verbosity to see debug messages displayed as the monitor observes the various transactions.

File: Command_Monitor_Xactor/Makefile

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

Step 10: Extension Points

A callback method is a third mechanism for reporting observed transactions. A callback method should be provided after a transaction has been observed (Rule 4-155). This callback method allows the observed transaction to be recorded in a functional coverage model or checked against an expected response.

The callback method is first defined as a *virtual void function* (Rule 4-160) in a callback façade class extended from the *vmm_xactor_callbacks* base class (Rule 4-159).

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

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

typedef class apb_monitor;
class apb_monitor_cbs extends vmm_xactor_callbacks;
   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw      cycle);
   endfunction: post_cycle
endclass: apb_monitor_cbs

class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel         out_chan;

   typedef enum {OBSERVED} notifications_e;
   ...
   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan   = null,
                ...);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;
      ...
      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
      forever begin
         apb_rw tr;
```

```
            this.wait_if_stopped();

            `vmm_trace(log, "Waiting for start of transaction...");

            // Wait for a SETUP cycle
            do @ (this.sigs.pck);
            while (this.sigs.pck.psel !== 1'b1 ||
                   this.sigs.pck.penable !== 1'b0);

            tr = new;
            tr.notify.indicate(vmm_data::STARTED);

            tr.kind = (this.sigs.pck.pwrite) ?
                        apb_rw::WRITE : apb_rw::READ;
            tr.addr = this.sigs.pck.paddr;

            @ (this.sigs.pck);
            if (this.sigs.pck.penable !== 1'b1) begin
               `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
            end
            tr.data = (tr.kind == apb_rw::READ) ?
                        this.sigs.pck.prdata :
                        this.sigs.pck.pwdata;
            tr.notify.indicate(vmm_data::ENDED);

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

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

            this.notify.indicate(OBSERVED, tr);
            this.out_chan.sneak(tr);
      end
      endtask: main
endclass: apb_monitor

`endif
```

The transaction descriptor created by the monitor is always of type *apb_rw* because of the following statement:

```
            tr = new;
```

This allocates a new instance of an object of the same type as the “tr” variable. However, a user may wish to annotate the transaction descriptor with additional information inside an extension of the “post_cycle” callback method. Unfortunately, the *apw_rw* transaction descriptor cannot be written to meet the unpredictable needs of users for annotating it with arbitrary information.

Using a factory pattern, a user can cause the monitor to instantiate an extension of the *apb_rw* class (Rule 4-115) that will then be filled in by the monitor but can also provide additional properties and methods for user-specified annotations.

File: apb/apb_monitor.sv

```
`ifndef APB_MONITOR__SV
`define APB_MONITOR__SV

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

typedef class apb_monitor;
class apb_monitor_cbs extends vmm_xactor_callbacks;
   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw     cycle);
   endfunction: post_cycle
endclass: apb_monitor_cbs

class apb_monitor extends vmm_xactor;
   virtual apb_if.master sigs;
   apb_rw_channel        out_chan;

   typedef enum {OBSERVED} notifications_e;

   apb_rw tr_factory;

   function new(string                 name,
                int unsigned           stream_id,
                virtual apb_if.passive sigs,
                apb_rw_channel         out_chan   = null,
                apb_rw                 tr_factory = null);
      super.new(“APB Master”, name, stream_id);
      this.sigs = sigs;
      if (out_chan == null) begin
         out_chan = new("APB Monitor Output Channel", name);
         out_chan.sink();
      end
      this.out_chan = out_chan;

      if (tr_factory == null) tr_factory = new;
      this.tr_factory = tr_factory;

      this.notify.configure(OBSERVED);
   endfunction: new

   virtual function void reset_xactor(reset_e rst_typ = SOFT_RST);
      super.reset_xactor(rst_typ);
      this.out_chan.flush();
   endfunction: reset_xactor

   virtual protected task main();
      super.main();
```

```
      forever begin
         apb_rw tr;
         this.wait_if_stopped();

         `vmm_trace(log, "Waiting for start of transaction...");

         // Wait for a SETUP cycle
         do @ (this.sigs.pck);
         while (this.sigs.pck.psel !== 1'b1 ||
                this.sigs.pck.penable !== 1'b0);

         $cast(tr, this.tr_factory.allocate());
         tr.notify.indicate(vmm_data::STARTED);

         tr.kind = (this.sigs.pck.pwrite) ?
                      apb_rw::WRITE : apb_rw::READ;
         tr.addr = this.sigs.pck.paddr;

         @ (this.sigs.pck);
         if (this.sigs.pck.penable !== 1'b1) begin
            `vmm_error(this.log, "APB protocol violation: SETUP
cycle not followed by ENABLE cycle");
         end
         tr.data = (tr.kind == apb_rw::READ) ?
                      this.sigs.pck.prdata :
                      this.sigs.pck.pwdata;
         tr.notify.indicate(vmm_data::ENDED);

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

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

         this.notify.indicate(OBSERVED, tr);
         this.out_chan.sneak(tr);
   end
   endtask: main
endclass: apb_monitor

`endif
```

With the transaction descriptors allocated using a factory pattern and the transaction descriptor accessible in a callback method before it is reported to other transactors, a test may now add user-defined information to a transaction descriptor.

File: Command_Monitor_Xactor/test_annotate.sv

```
`include “tb_env.sv”

program annotate_test;

class annotated_apb_rw extends apb_rw;
   string note;
```

```
   virtual function vmm_data allocate();
      annotated_apb_rw tr = new;
      return tr;
   endfunction

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

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

      super.copy(tr);
      tr.note = this.note;

      return tr;
   endfunction

   virtual function string psdisplay(string prefix = "");
      psdisplay = {super.psdisplay(prefix), " (", this.note, ")"};
   endfunction
endclass

class annotate_tr extends apb_monitor_cbs;

   local int    seq = 0;
   local string format = "[-%0d-]";

   function new(string format = "");
      if (format != "") this.format = format;
   endfunction

   virtual function void post_cycle(apb_monitor xactor,
                                    apb_rw     cycle);
      annotated_apb_rw tr;
      if (!$cast(tr, cycle)) begin
         `vmm_error(xactor.log, "Transaction descriptor is not a
annotated_apb_rw");
         return;
      end

      $sformat(tr.note, this.format, this.seq++);
   endfunction: post_cycle
endclass


vmm_log log = new("Test", "Annotate");
tb_env env = new;
initial begin
```

```
   env.stop_after = 5;

   env.build();
   begin
      annotated_apb_rw tr = new;
      annotate_tr      cb = new;

      env.mon.tr_factory = tr;
      env.mon.append_callback(cb);
   end

   env.run();

   $finish();
end

endprogram
```

Step 11: Random Transactions

To promote the use of random stimulus and make the same transaction descriptor useable in transaction generators, all public properties in a transaction descriptor should be declared as *rand* (Rules 4-59, 4-60, 4-62). It is also 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 12: 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_monitor.sv"

`endif
```

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

Step 13: Congratulations!

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

Upon reading this primer, you probably realized that there is much code that is similar across different monitors. Wouldn't be nice if you could simply cut-and-paste from an existing VMM-compliant monitor 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 monitor.

Command

```
% rvmgen -l sv
```

The relevant templates for writing command-layer monitors are:

1) Physical interface declaration

2) Transaction Descriptor

3) Monitor, 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 master transactors, command-layer slave transactors, functional-layer transactors or verification environments.