6 CALLBACKS

This chapter addresses the use of registered callbacks that provide an aid in plugging into the code "unforeseen" functionality by identifying "strategic" insertion/callback points. Callbacks provide controllability to the user, and allow transactors to adapt to the needs of an environment or a testcase. We use two examples to demonstrate the strengths of callbacks. The first example emulates the error injection addressed in Chapter 5 with the factory pattern. The second example copies class properties from the command transactor to a debug SystemVerilog interface. This allows you to write within the debug interface assertions derived from both the DUT interface and copies of the variables defined in the class object. In addition, copies of the class variables posted on the debug interface can be viewed with a waveform viewer.[1]

[1] At the time this book was written, current tools do not display variables of class instances because those instances are dynamically allocated and destroyed. However, tool vendors may provide this capability in the near future.

6.1 CALLBACK OVERVIEW

Figure 6.1 represents an overview as to where callbacks fit in the verification environment.

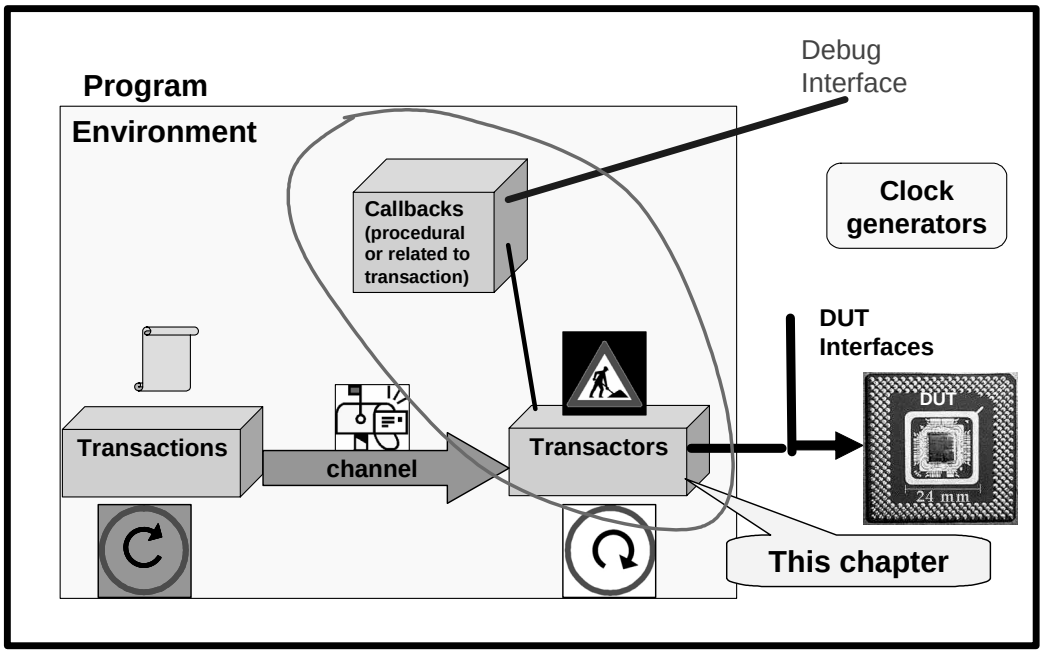

Figure 6.1 Overview of Callbacks in the Verification Environment

6.1.1 What is a callback?

A callback is a design pattern with an access mechanism to registered methods that can be called by an instance in which the callback is registered. A callback is implemented in a class (called a façade pattern) that defines methods (i.e., functions and tasks). It must be extended then registered for a specific class instance in the environment (e.g., the command transactor instance), and is accessed by the application class through the callback macro call (*`vmm_callback* macro). Callbacks make transactors truly reusable, and allow the creation of tasks or tests with fewer lines of code without modifying existing code known to work. For example, callbacks allow transactors to easily add testcases, or inject exceptions (e.g., errors), or link to monitors, scoreboards, or debug interfaces to provide information to those design units. Where are callbacks used? Callbacks can be used to perform quite a few tasks such as:

1. Inject an error (e.g., invert bits in the CRC)
2. Add a test that was not initially planned
   e.g., simultaneous *load* and *enable*
3. Stretch a gap in the protocol
4. Drop or delay a transaction
5. Synchronize multiple generators
6. Monitor coverage points in the design.
7. Test the behavior of a hard to reach condition
8. Add or save channel transactions.
   a. For later reuse, or
   b. To fill time with specific transactions
9. Provide information to scoreboards and/or debug interface
   a. This information includes the transaction, forced error condition, original error-free data, etc.

Callbacks are particularly important when building your testbench or a library because they define a set of methods that can be extended by the user to add functionality initially not defined by the designer. This is done through the definition of facades (or templates) for the methods, but without the implementation (or body) of those methods. The implementation of the callback methods is postponed until they are needed by the application. For example, the *vmm_log_callbacks* class provides a façade for the callback methods provided by the message service. The callback task *vmm_log::pre_stop(vmm_log log)* is invoked by the message service before the simulation is stopped because of a *STOP* simulation handler.

6.1.2 What is the structure of a callback?

A callback has the following characteristics:

1. **Declaration**: A callback is implemented in a class that consists of a declaration or façade. The façade includes virtual methods with no code, thus they do not provide any functionality unless the façade class is extended (where the body of the facade is defined).
2. **Application:** To use a callback the macro *`vmm_callback* must be used. For example, if the callback class has a function named *flip_data*, then in the command transactor we can write: `vmm_callback(Data_callback, flip_data(ldata));
3. **Registration**: Callbacks must be registered in the environment to be used. For example, if class *Data2_callback* extends class *Data_callback* facade, then the following needs to be written in the *build* step of the environment so that the callback can be used:
   a. Declaration of an instance of the class extension.
   `Data2_callback d2_cbk;` // e.g., the body with the *flip_data()* function definition.
   b. Instantiation of the class
   ```
   this.d2_cbk = new(counter_cmd_xactor_0,
                     vir_if, dbug_if);
   ```
   c. Registration of the callback
   Register the callback (e.g., *d2_cbk*) to the object that will use the callback (e.g., *fifo_cmd_xactor_0* command transactor object). Registration can be accomplished with the *vmm_xactor*::*append_callback* function For example,
   `this.fifo_cmd_xactor_0.append_callback(d2_cbk);`

6.2 BUILDING A CALLBACK

Building a callback is four-step process involving potentially two distinct phases. The first phase is to "design the hooks for callback", and the second phase is to "use those hooks and integrate the application through callback".

First Phase: Callback design
This phase is performed during the design of the testbench. Here you strategically identify the relevant points in the transaction flow where potential candidates for callbacks can be used. Examples of such points where potential callbacks can be used are:

- After the transaction gets generated but before pushing it to a channel
- Before applying the transaction to the DUT
- After sending a transaction to DUT

In the FIFO model we chose one such point inside command transactor - just before applying the transaction. Once the strategic point is identified the next step is to build a façade, as explained in Section 6.2.1. The utilization of the callback using the façade is shown in Section 6.2.2.

Second Phase: Callback Integration and hookup
This phase uses the façade built in First Phase to integrate the application. It consists of two steps. First is the build of the implementation, and second is the registration of the callback. These steps are explained in Section 6.2.3 and 6.2.4. The simulation results are shown in section 6.2.5.

6.2.1 Callback Design: the Façade

For the FIFO example, we add a function that flips some bits in a data stream to simulate an error injection. We also add a task that performs a *push()* with data equal to what is passed as an argument. This is then followed by idle cycles (as defined by the task argument), and then a *pop()*.

The first step in building a callback is to build a class that has templates for the callback methods, but not the implementation of those methods. This is called a façade, from the definition of the terms that means "The face of a building, especially the principal face." For our example, the façade is: shown in Figure 6.2.1.

```
virtual class Data_callback extends vmm_xactor_callbacks;
  import fifo_pkg::*;
  virtual function void flip_data(ref word_t data);
  endfunction : flip_data

  virtual task push_then_pop_task (
      Fifo_cmd_xactor xactor,
      word_t data,
      int  num_idle_cycles
             );
  endtask : push_then_pop_task
endclass : Data_callback
```

NO implementation (body) in the façade. Just the template.

Figure 6.2.1 Callback Façade (*/ch6_callback/ data_callback.sv*)

Notes:

1. When writing a callback, do not assume a specific purpose. The callback should be named after their location in the transaction execution sequence, not the intended purpose; for example, "pre execution". Using it to flip some bits is your extension of the callback.[2]
2. The façade class and all methods in it must be declared *virtual*
3. The callback has tasks and functions
4. All callback functions must be void[3]
5. Can use *ref* to return data (i.e., pass by reference)
6. The façade should not have a body

[2] In the example we called the callback "Data_callback". As an afterthought, a better name would have been "Post_peek_pre_execute". The extension should have been called "Flip_PushPop".

[3] VMM Rule 4-160 Callbacks shall be declared as tasks or void functions.

6.2.2 Adding the Hook (Using the Callback)

A callback is typically used by a transactor. To use a registered callback you just use the macro [4]

```
`vmm_callback(base_name_of_callback_class,
              method_name(arguments));
```

Note 1: When using the callback, it is the façade class name that is referenced in the code (i.e., *Data_callback)* and not the extended class name (addressed in Section 6.2.3). This allows you to write the application that uses the callback without concern about the callback implementations, as there can be many implementations. You can modify the implementation of the façade, with an extension of that class, and register the needed implementations in the environment. If multiple façade extensions are registered, then a method call to a façade will execute all the methods in the extensions in the order in which they were registered (more on that further down).

Note 2: At their root, callbacks are an access mechanism. If the callbacks are not extended, there is no need to invoke the callback methods. If no callback is registered, then the list of registered callbacks is empty hence there is nothing to call. But if a façade has two callbacks, but only the first one is extended and has 10 registrations, then a transactor will still call 10 instances of the 2nd callback method (the empty one) because there are 10 registrations.

For example, in the FIFO command transactor we call the callback as shown in Figure 6.2.2.

```
task Fifo_cmd_xactor::main();
  ldata = (fifo_xactn.data);
    case (fifo_xactn.kind)
      PUSH : begin : case_push
        if (err_inject.randomize()) begin : good_randomization
          if (err_inject.inject_err==INJECT)
          begin : inject_the_error
            `vmm_trace(log,  "MUST DO THE FLIP_DATA THEN push_then_pop_task");
             `vmm_callback(Data_callback,flip_data(ldata));
              `vmm_trace(log, $psprintf("xaction.data= %h, flipped ldata= %h ",
                                fifo_xactn.data, ldata));
             `vmm_note(log, "About to do a push_then_pop_task thru callback");
               `vmm_callback(Data_callback,
                                push_then_pop_task(this, ldata, 3));

           end : inject_the_error
         else this.push_task(ldata); // no error injection
        end : good_randomization


…
endtask : main
```

Refer to base name

Figure 6.2.2 Using the Callback (*ch6_callback/ fifo_cmd_xactor.sv)*

[4] VMM Rule 4-163 —Transactors shall use the *vmm_callback()* macro to invoke the registered callbacks.

6.2.3 Building the Implementation

This phase uses the façade built in First Phase to integrate the application. Implementations of the callback methods are performed in classes extended from the façade class. Figure 6.2.3 represents two implementations using class extensions.

```
class Data2_callback extends Data_callback;
   vmm_log log;
  function void flip_data(ref word_t data);
    data={{data[WIDTH-1:2], !data[1], data[0]}};
    log=new("Data_callback", "0");
      `vmm_trace(log, $psprintf("data2_callback %b", data));
  endfunction : flip_data

  virtual task push_then_pop_task (Fifo_cmd_xactor xactor,
      word_t data,
      int  num_idle_cycles
             );
    begin
      log=new("Data_callback", "0");
      `vmm_trace(log, $psprintf ("Push callback: data %0h ", data));
        xactor.f_if.driver_cb.data_in <= data;
        xactor.f_if.driver_cb.push <= 1'b1;
        xactor.f_if.driver_cb.pop  <= 1'b0;
        @ ( xactor.f_if.driver_cb);
       xactor.f_if.driver_cb.push <= 1'b0;
      repeat (num_idle_cycles) @ ( xactor.f_if.driver_cb);
      // pop
      `vmm_trace(log, "Pop callback");
      xactor.f_if.driver_cb.pop  <= 1'b1;
      xactor.f_if.driver_cb.push <= 1'b0;
      @ ( xactor.f_if.driver_cb);
      xactor.f_if.driver_cb.pop  <= 1'b0;
    end
  endtask : push_then_pop_task
endclass : Data2_callback

class Data3_callback extends Data_callback;
  function void flip_data(ref word_t data);
     log=new("Data_callback", "0");
     data={{~data[WIDTH-1:2], data[1], data[0]}};
     `vmm_trace(log, $psprintf("data3_callback %b", data));
  endfunction : flip_data
endclass : Data3_callback
```

Figure 6.2.3 Class Extensions of the Façade (*ch6_callback/ data_callback.sv*)

In the above example we have two extensions of the façade class. If one of the two extensions is registered, then a call to the method within that extended class will be used. However, if both extensions are registered, then a single call to the method (e.g., *flip_data()*) will result in a call to each of the methods that are in the registered extensions in the order in which those callbacks are registered. This is demonstrated further down.

6.2.4 Registering

The registration of the callbacks can be done in the build step of the environment or in the *program* as a testcase. Both techniques are shown here for demonstration purposes. Use either technique, but not both as every registration appends the callback. Registration involves the following steps:

In the environment:

1. Declare an instance of the transactor that will use the callback, e.g.,
   `fifo_cmd_xactor fifo_cmd_xactor_0;`
2. Instantiate and allocate the callbacks, e.g.,
   `data2_callback d2_cbk =new;`
   `data3_callback d3_cbk =new;`
3. In the *build* function add the registration of the callbacks, e.g.,
   `this.fifo_cmd_xactor_0.append_callback(d2_cbk);`
   `this.fifo_cmd_xactor_0.append_callback(d3_cbk);`

Figure 6.2.4-1 is a snippet of this code for the FIFO environment.

```
class Fifo_env extends vmm_env;
  import fifo_pkg::*;
  Fifo_cmd_xactor fifo_cmd_xactor_0;
  Data2_callback d2_cbk =new;
  Data3_callback d3_cbk =new;
…
 function void build();
    …
    this.fifo_cmd_xactor_0 = new("PUSH_XACTOR",
                         0,
                         vir_if,
                         fifo_xactn_channel
                         );


  // registering the callback
   this.fifo_cmd_xactor_0.append_callback(d2_cbk);
   this.fifo_cmd_xactor_0.append_callback(d3_cbk);

  endfunction : build

…

endclass : Fifo_env
```

Both versions are now registered for the fifo_cmd_xactor_0

Figure 6.2.4-1 FIFO Environment for the Registration of the Callbacks
ch6_callback/fifo_env.sv

In the *program* block:

1. Instantiate and allocate the callbacks, e.g.,
   ```
   data2_callback d2_cbk =new;
   data3_callback d3_cbk =new;
   ```
2. Following the ***fifo_env_0.build*** function add the registration of the callbacks, e.g.,
   ```
   fifo_env_0.fifo_cmd_xactor_0.append_callback(d2_cbk);
   fifo_env_0.fifo_cmd_xactor_0.append_callback(d3_cbk);
   ```

Figure 6.2.4-2 is a snippet of this code for the FIFO environment.

```
program automatic fifo_test_pgm;
  …
  data2_callback d2_cbk =new;
  data3_callback d3_cbk =new;
  initial begin : test
     fifo_env_0.build();
     fifo_env_0.fifo_cmd_xactor_0.append_callback(d2_cbk);
     fifo_env_0.fifo_cmd_xactor_0.append_callback(d3_cbk);
     fifo_env_0.run();
    `vmm_note(log, "End of Test");
  end :test
endprogram : fifo_test_pgm
```

Figure 6.2.4-2 FIFO program for the Registration of the Callbacks

Figure 6.2.4-3 represents the UML view for the design of the callback using registration in the environment.

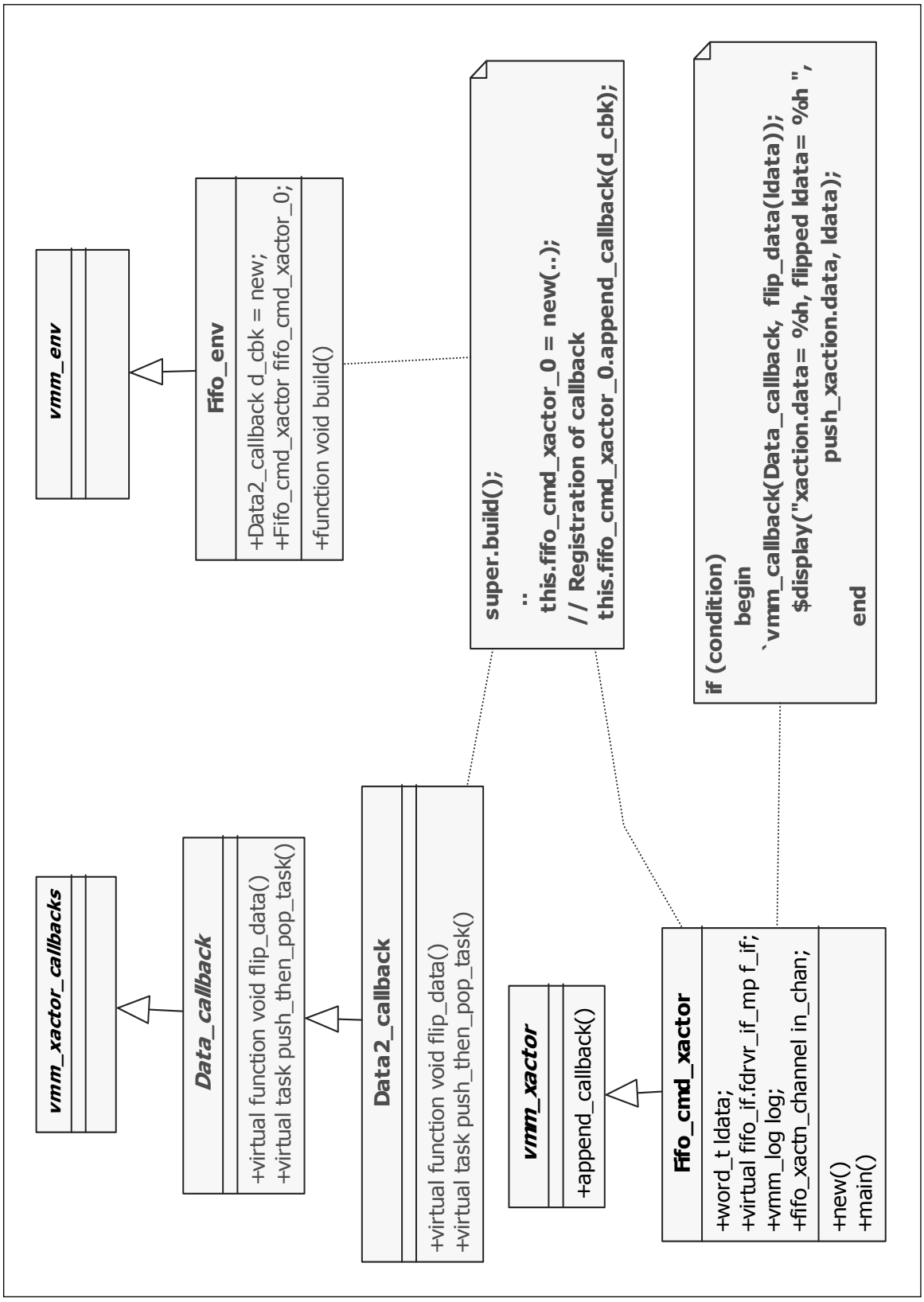

Figure 6.2.4-3 UML View for Design and Application of a Callback

6.2.5 Simulation Results

NOTE 3: As previously addressed, because both versions the callback instance are registered, both versions of the *flip_data()* function will be called in the order in which they were registered. Simulation demonstrated this concept as shown in Figure 6.2.5.

13350.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | About to Get a new fifo xaction from in_channel
13350.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | Got a new fifo xaction from in_channel #0.0.84 Fifo Xaction PUSH
13350.00 ns Fifo COMMAND Layer Xactor [Trace:DEBUG] **| MUST DO THE FLIP_DATA THEN push_then_pop_task**
13350.00 ns Data_callback [Trace:DEBUG] | data2_callback 00000000000000000000000100000110
13350.00 ns Data_callback [Trace:DEBUG] | data3_callback 11111111111111111111111011111010
13350.00 ns Fifo COMMAND Layer Xactor [Trace:DEBUG] | xaction.data= 00000104, flipped ldata= fffffefa
13350.00 ns Fifo COMMAND Layer Xactor [Normal:NOTE] | About to do a push_then_pop_task thru callback
13350.00 ns Data_callback [Trace:DEBUG] | 13350.00 ns fifo_tb.utest_pgm.\Data2_callback::push_then_pop_task Push callback: data fffffefa
13450.00 ns FIFO_MON [Normal:NOTE] | Found a PUSH Xactn at time 13450.00 ns data fffffefa

An analysis of the callbacks is summarized below:

```
Non-flipped:  0000..0001_0000_0100 // Original data 00000104
#1  flip:     0000..0001_0000_0110 // 1st callback   00000106
                                ^  // Bit inverted by callback
#2  flip:     1111..1110_1111_1010 // 2nd callback   FFFFFEFA
              ^^^^..^^^^ ^^^^ ^^   // Bits inverted by callback
```

Figure 6.2.5 Simulation Results and Analysis of Callbacks

6.3 VERIFICATION WITH DEBUG INTERFACE

Another interesting application of callbacks is the transfer of class properties into a SystemVerilog interface, which we call a debug interface. Transferring such information into a SystemVerilog interface provides the following benefits:

1. The display of the transactor variables onto the interface. This allows the waveform visualization of those variables, thus facilitating the debug and documentation of the design. Typically, simulation tools do not display class variables because classes are dynamically allocated and destroyed.

2. The use of assertions. Assertions are not allowed in classes, but they are allowed in interfaces. Thus, by copying class variables into interfaces, you can use those copies in the assertions. If you only use classes for verification, then instead of assertions you would need to define your own FSM to verify design compliance to specifications. An example of an assertion that makes use of the transaction information is:

```
property p_PushThenPopOnEmpty;
  word_t v_data_in;
  @(posedge clk)
   (kind==PUSH_POP && f_if.empty, v_data_in=data_in) |=>
        push
        ##[1:5] pop
        ##0 f_if.data_out==v_data_in;
endproperty : p_PushThenPopOnEmpty
a_PushThenPop: assert property(_PushThenPopOnEmpty);
```

3. The verification of the transactions. The debug interfaces can also include as inputs the DUT interfaces. This allows you to write assertions that not deal with the verification of the DUT, but also with the verification of the transactors to insure that they abide to the bus protocols. For example,

```
property p_PushThenPOP_protocol;
  @(posedge clk)
   (kind==PUSH_POP) |=> push ##[1:5] pop;
endproperty : _PushThenPOP_protocol
a_PushThenPOP: assert property(_PushThenPOP_protocol);
```

Warning**:** Many activities in classes may occur in zero time, and when transferring values onto an interface you must insure that final value of the variables is the one that remains in that interface. For example, it is also possible that another callback modifies that value in the class, but not the debug interface.

6.3.1 Concept

The transfer of class properties onto an interface is conceptually very simple, and is shown in Figure 6.3.1. Basically, at specific points in the transactor code, a callback to a task is made to copy values of the needed class properties onto a debug SystemVerilog interface. These properties can include objects such as transactions that are about to be emitted (e.g., *kind*), modes (e.g., error injection), data to be send, etc. The task can also trigger within that interface an event that can be used in assertions or in other verification tasks within the interface. These tasks include code for the assertions.

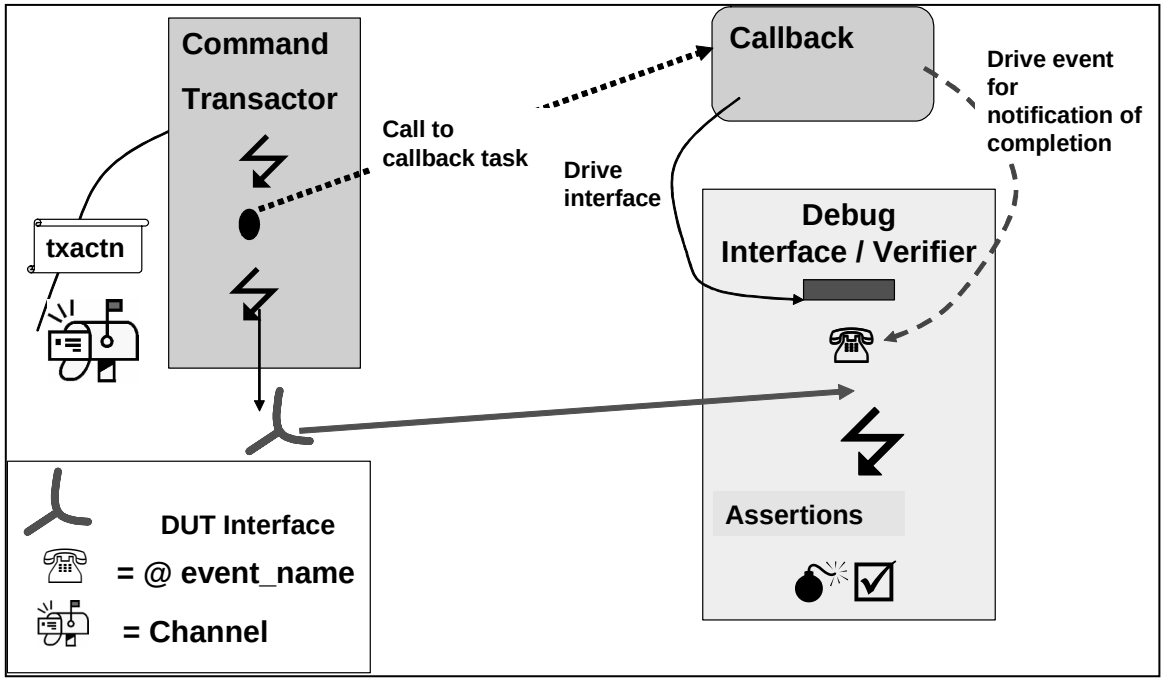

Figure 6.3.1 Transfer of class properties onto an interface

6.3.2 Model

To demonstrate this concept we will use the FIFO model and transfer into a debug interface (*debug_if*) the transactor property *kind*, random data originally defined in the transaction, and modified data with potential error injection that was modified within the transactor with a callback. The debug interface also includes as inputs the FIFO interface used by the DUT. Thus, in combination with the effective transaction information and FIFO interface used by the DUT, we have enough information to write in the debug interface assertions related to the protocol used by the transactor and to the operation of the DUT. For demonstration purposes, we wrote two assertions related to the protocol of the transactor. These assertions detected an error in the transfer of the transaction into the debug interface (i.e., the assertion paid off). We see this use of transaction and data transfer into a debug interface as a great benefit in the verification process because we can write additional assertions that have access to all needed information (from the source of the transaction to the DUT interface). The other benefit that we also experienced in using this approach was the greater understanding of the model, and the documentation provided by viewing the transaction information onto the waveform viewer.

6.3.2.1 Debug interface

Figure 6.3.2.1 represents the code for the debug interface. Note that this interface includes the DUT FIFO interface used by the assertion properties within this debug interface.

```
interface debug_if(input clk,
                   input rst_n,
                   fifo_if f_if);
  timeunit 1ns; timeprecision 100ps;
  import fifo_pkg::*;
  word_t  data_in;    // in : data to load
  word_t vgdata_in;  // from original transaction generator
  fifo_scen_t kind;  // kind of transaction
  inject_err_t err_inj;  // error Injection

  parameter hold_time=3;
  parameter setup_time = 5;

  clocking driver_cb @(posedge clk);
    default input #setup_time output #hold_time;
    output data_in;
    output vgdata_in; // from original transaction generator
    output kind;
    output err_inj;
  endclocking : driver_cb
  // ignoring the resets
    property p_PUSH;
       @(posedge clk)
         kind==PUSH |-> f_if.push && !f_if.pop;
    endproperty : p_PUSH
    ap_PUSH: assert property(p_PUSH);

    property p_POP;
       @(posedge clk)
         kind ==POP |-> f_if.pop && !f_if.push;
    endproperty : p_POP
    ap_POP : assert property(p_POP);
  modport driver_dbug_mp(clocking driver_cb);
endinterface : debug_if
```

Figure 6.3.2.1 Debug Interface (*ch6_debug_cb/debug_if.sv*)

6.3.2.2 Callback Façade

Figure 6.3.2.2 represents a view of the callback façade. Note that this façade class has no variable declarations, and the task *displayToDebug_if* has no body.

```
class Debug_callback extends vmm_xactor_callbacks;
 virtual task displayToDebug_if();
 endtask : displayToDebug_if
endclass : Debug_callback
```

Figure 6.3.2.2 Callback Façade for the Debug Callback *(ch6_debug_cb/debug_callback.sv)*

6.3.2.3 Using the callback

Figure 6.3.2.3-1 represents the FIFO transactor that calls the callbacks to inject an error and to transfer class information to the debug interface.

```
task Fifo_cmd_xactor::main();
    Fifo_response  fifo_response; // response to generator
    fork
      super.main();
    join_none

    fifo_xactn_0 = factory_fifo_xactn.allocate();  // transaction to get
    inject_error0 = factory_inject_err.allocate();
    forever
    begin : main_loop
      this.in_chan.peek(fifo_xactn_0);
      fifo_xactn_0.notify.indicate(vmm_data::STARTED);
      `vmm_note(log, $psprintf("Got a new fifo xaction from in_channel %s ",
                fifo_xactn_0.psdisplay()));
      ldata = (fifo_xactn_0.data);
     `vmm_callback(Debug_callback, displayToDebug_if());
    case (fifo_xactn_0.kind)
      PUSH :
        begin  : push1
          if (inject_error0.randomize()) begin : randOK
              if (inject_error0.inject_err==INJECT) begin  : if2
               `vmm_callback(Data_callback,flip_data(ldata));
               `vmm_callback(Debug_callback, displayToDebug_if());
            `vmm_trace(log, $psprintf("xaction.data= %h, flipped ldata= %h ",
                fifo_xactn_0.data, ldata));
             end :if2
             this.push_task(ldata);
            end : randOK
        end : push1
       POP  :  this.pop_task();
       PUSH_POP : this.push_pop_task(ldata);
       IDLE :  this.idle_task(fifo_xactn_0.idle_cycles);
       RESET :  this.reset_task(5);
      endcase
    fifo_response=new();
    fifo_response.kind = fifo_xactn_0.kind;
    fifo_response.status= PASSED;
    fifo_xactn_0.notify.indicate(vmm_data::ENDED, fifo_response);
    // Send the response to generator thru the response class object.
    // in nonblocking manner.
    this.resp_chan.sneak(fifo_response);
    // Rule 4-121
    this.in_chan.get(fifo_xactn_0);
   end : main_loop
 endtask : main
```

***data_in* modified, must do callback again**

Figure 6.3.2.3-1 Application of Callback *(ch6_debug_cb/fifo_cmd_xactor.sv)*

Figure 6.3.2.3-2 represents the simulation view of the with the debug interface displaying the actual transaction kind, original data from randomized transaction (*vgdat_in*), and actual data imposed onto the DUT (*data_in*). See Section 6.3.2.4 for implementation, and Section 6.3.2.5 for registration of the callbacks.

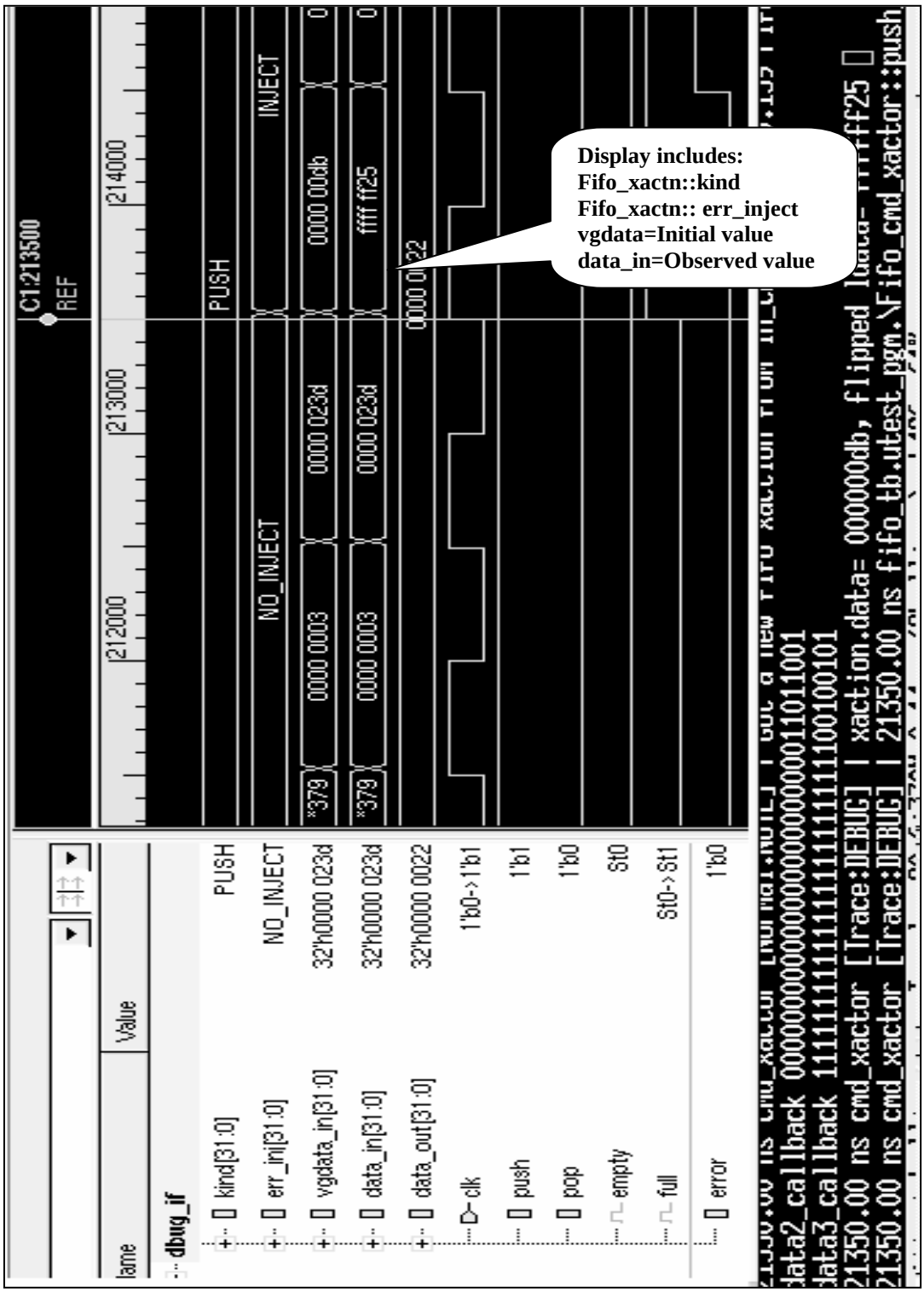

Figure 6.3.2.3-2 Simulation Results with Debug Callback

6.3.2.4 Callback implementation

Figure 6.2.2.4 represents the callback implementation. Note that the class includes properties for the transactor, the debug interface, and the FIFO interface (which is not currently used in this model, but was included in case this task is expanded and needs to access the *fifo_if*).

```
class Debug2_callback extends Debug_callback;
  local Fifo_cmd_xactor             xactor;
  virtual fifo_if.fdrvr_if_mp       f_if; // not used
  virtual debug_if.driver_dbug_mp   dbug_if;

 function new(Fifo_cmd_xactor                    xactor,
              virtual fifo_if.fdrvr_if_mp        new_vir_if,
              virtual debug_if.driver_dbug_mp    dbug_if);
   this.xactor = xactor;
   this.f_if=new_vir_if;  // not used
   this.dbug_if = dbug_if;
 endfunction : new

  virtual task displayToDebug_if ();

  this.dbug_if.driver_cb.kind =
             this.xactor.begotten_fifo_xactn.kind;
   this.dbug_if.driver_cb.err_inj =
             this.xactor.err_inject.inject_err;

      this.dbug_if.driver_cb.vgdata_in <=
                 this.xactor.begotten_fifo_xactn.data;
      this.dbug_if.driver_cb.data_in <= this.xactor.ldata;
  endtask : displayToDebug_if
endclass : Debug2_callback
```

Figure 6.2.2.4 Callback Implementation *(ch6_debug_cb/debug_callback.sv)*

6.3.2.5 Registering the callbacks

In this example, the registration is done in the environment. Figure 6.3.2.5 represents snippets of code for this registration.

```
class Fifo_env extends vmm_env;

…
  Data2_callback d2_cbk =new;
  Data3_callback d3_cbk =new;
  Debug2_callback dbug_cbk new( fifo_cmd_xactor_0,
                          `TOP.f_if,
                          `TOP.dbug_if);

  …

  function void build();
  string msg;
    super.build();
…
   this.fifo_cmd_xactor_0.append_callback(d2_cbk);
   this.fifo_cmd_xactor_0.append_callback(d3_cbk);


   this.fifo_cmd_xactor_0.append_callback(dbug_cbk);
  …
   endfunction : build

enclass : Fifo_env
```

Instantiation and allocation of callbacks

Instantiation of debug callbacks

Registering of the callbacks

Figure 6.3.2.5 FIFO Environment *(ch6_debug_cb/fifo_env.sv)*

6.3.3 Guidelines in using callbacks

1. Use callbacks wherever you want to provide future entry points to your transactor.
2. Place a callback call in your transactor wherever you think you might want access something significant or something that is about to be done or is done (e.g., about to execute DUT protocol). This requires some planning. However, placing callback calls into the transactors provides a placeholder for a potential enhancement to the testbench. Note that the callback implementation does not need to be defined until it is used.
3. If one version of the callback is needed, then register only the needed callback in the environment.
4. You can dynamically remove a callback registration with the function:
   virtual function void vmm_xactor::unregister_callback(vmm_log_callbacks cb);
   Note that callback façade instances can later be re-registered with the same or another transactor.
5. Since the callback is a method in a class, you can implement a "turn me off" control mechanism in a callback that will behaviorally skip any optional behavior. This can be simply done with a variable or task argument that identifies the turn-off flag, and a condition in the task to skip the processing of the task.

6.4 FILE STRUCTURE

Table 6.4 demonstrates the file Structure and the purpose of each file.

Table 6.4. File Structure and Functions

/ch6/ch6_callback and /ch5*/ch5_fct_inject_err* directories

File	Function	Used by
fifo_pkg.sv	Defines types and initialized variables	ALL
fifo_if.sv	Defines the FIFO interface	RTL and by program, testbench, transaction and transactors
fifo_csr_if.sv	Defines the FIFO configuration interface	RTL, property models, and by environment, and possibly transactors
fifo_xactn.sv	Defines the transaction class with the constraints Also used for the channel generation with: *`vmm_channel* (Fifo_xactn)	*`vmm_channel* macro for generation of channel, *`vmm_atomic_gen* macro for generation of atomic generator, monitor transactor for creation of transaction from observed values on bus interface
fifo_rtl.sv	Represents the FIFO RTL DUT	Top level
fifo_props.sv	Defines the properties for assertions	Top level for bind
fifo_log_fmt.sv	Defines formatting information for display	FIFO environment
fifo_pgm.sv	Creates the modeling for simulation and initiates the *run* in the environment	Top level
fifo_env.sv	Creates the build and start for simulation	program
fifo_mon_xactor.sv	Creates a copy of the observed transaction onto a transaction channel.	Scoreboard, top level
top_tb.sv	Represents the top level and instantiates the RTL, the bind, the monitor, etc	none
fifo_gen_xactor.sv	Uses the macro *`vmm_atomic_gen* for generation of atomic generator, defines the constraints for the number of transactions	Environment for creation of the build model
inject_err.sv	Error injection classes	Command transactor
test.svh	Common include files	Program block
fifo_response.sv	Class derived from vmm_data to provide a response to a transactor (e.g., generator) through a channel	command transactor and environment
data_callback.sv		

Ch6/ch6_debug_cb
Above files +

debug_callback.sv	Callback for debug interface	Command transactor and environment
debug_if.sv	Interface to put values of class objects	Command transactor and environment

Chapter 6 Questions and LAB

Q1. What are good user applications of callbacks?

Q2. Do the VMM classes have callbacks? Which ones? And why?

Q3. What are the advantages of transferring information from a transactor via callback to an interface?

Lab06

Use a callback to inject an error in the data to be loaded into the counter. See instructions in subdirectory lab/lab06/todo/readme.txt.