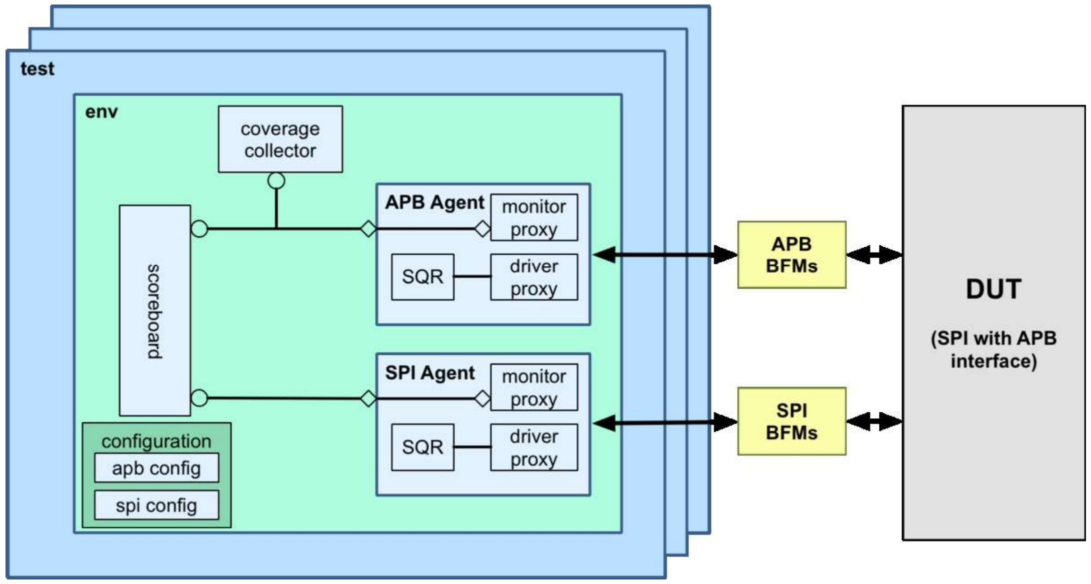
Block-Level Testbench

As an example of a block level testbench, consider a testbench built to verify a SPI Master DUT. In this case, the UVM environment has two agents - an APB agent to handle bus transfers on its APB slave port, and a SPI agent to handle SPI protocol transfers on its SPI port. The structure of the overall UVM verification environment is illustrated in the block diagram. Let us go through each layer of the testbench and describe how it is put together from the top down.



The Testbench Modules

Two top level testbench modules are used in the SPI block level testbench. The hdl_top module contains the SPI Master DUT, the APB and SPI BFMs and the apb_if, spi_if and intr_if pin interfaces. The SPI Master DUT is connected to the apb_if, spi_if and intr_if which in turn are connected to the APB and SPI master and slave BFMs, respectively. Two initial blocks are also encapsulated within the hdl_top module. The first initial block places the virtual interface handles for the BFM interfaces into the UVM configuration space using uvm_config_db::set. The second initial block generates a clock and a reset signal for the APB interface.

```
module hdl_top;

`include "timescale.v"

// PCLK and PRESETn
//
logic PCLK;
logic PRESETn;

//
// Instantiate the pin interfaces:
//
apb_if APB(PCLK, PRESETn);
spi_if SPI();
intr_if INTR();
```

```
//
// Instantiate the BFM interfaces:
//
  apb_monitor_bfm APB_mon_bfm(
     .PCLK    (APB.PCLK),
     .PRESETn (APB.PRESETn),
     .PADDR   (APB.PADDR),
     .PRDATA  (APB.PRDATA),
     .PWDATA  (APB.PWDATA),
     .PSEL    (APB.PSEL),
     .PENABLE (APB.PENABLE),
     .PWRITE  (APB.PWRITE),
     .PREADY  (APB.PREADY)
  );
  apb_driver_bfm APB_drv_bfm(
     .PCLK    (APB.PCLK),
     .PRESETn (APB.PRESETn),
     .PADDR   (APB.PADDR),
     .PRDATA  (APB.PRDATA),
     .PWDATA  (APB.PWDATA),
     .PSEL    (APB.PSEL),
     .PENABLE (APB.PENABLE),
     .PWRITE  (APB.PWRITE),
     .PREADY  (APB.PREADY)
  );
  spi_monitor_bfm SPI_mon_bfm(
     .clk  (SPI.clk),
     .cs   (SPI.cs),
     .miso (SPI.miso),
     .mosi (SPI.mosi)
  );
  spi_driver_bfm SPI_drv_bfm(
     .clk  (SPI.clk),
     .cs   (SPI.cs),
     .miso (SPI.miso),
     .mosi (SPI.mosi)
  );
  intr_bfm INTR_bfm(
     .IRQ  (INTR.IRQ),
     .IREQ (INTR.IREQ)
  );
```

```
// DUT
spi_top DUT(
 // APB Interface:
 .PCLK(PCLK),
 .PRESETN(PRESETn),
 .PSEL(APB.PSEL[0]),
 .PADDR(APB.PADDR[4:0]),
 .PWDATA(APB.PWDATA),
 .PRDATA(APB.PRDATA),
 .PENABLE(APB.PENABLE),
 .PREADY(APB.PREADY),
 .PSLVERR(),
 .PWRITE(APB.PWRITE),
 // Interrupt output
 .IRQ(INTR.IRQ),
 // SPI signals
 .ss_pad_o(SPI.cs),
 .sclk_pad_o(SPI.clk),
 .mosi_pad_o(SPI.mosi),
 .miso_pad_i(SPI.miso)
);

// Initial block for virtual interface wrapping:
initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual apb_monitor_bfm)::set(null, "uvm_test_top", "APB_mon_bfm", APB_mon_bfm);
  uvm_config_db #(virtual apb_driver_bfm) ::set(null, "uvm_test_top", "APB_drv_bfm", APB_drv_bfm);
  uvm_config_db #(virtual spi_monitor_bfm)::set(null, "uvm_test_top", "SPI_mon_bfm", SPI_mon_bfm);
  uvm_config_db #(virtual spi_driver_bfm) ::set(null, "uvm_test_top", "SPI_drv_bfm", SPI_drv_bfm);
  uvm_config_db #(virtual intr_bfm)       ::set(null, "uvm_test_top", "INTR_bfm", INTR_bfm);
end

//
// Initial blocks for clock and reset generation:
//
initial begin
  PCLK = 0;
  forever #10ns PCLK = ~PCLK;
end
initial begin
  PRESETn = 0;
  repeat(4) @(posedge PCLK);
  PRESETn = 1;
end
endmodule: hdl_top
```

The hvl_top module simply imports the uvm_pkg and the spi_test_lib_pkg which contains definitions for the tests that can be run. It also contains the initial block that calls the run_test() method to construct and launch the specified test and thus UVM phasing.

```
module hvl_top;

`include "timescale.v"

import uvm_pkg::*;
import spi_test_lib_pkg::*;

// UVM initial block:
initial begin
  run_test();
end

endmodule: hvl_top
```

The Test

The next phase in the UVM construction process is the build phase. For the SPI block level example this means building the spi_env component, after first creating and preparing all pertinent configuration objects to be used by the environment. The configuration and build process is largely common to most test cases, so it is generally good practice to devise a test base class that can be extended to create specific tests.

In the SPI example, the configuration object for the spi_env contains handles for the SPI and APB configuration objects. This allows the env configuration object to be used to pass all required sub-configuration objects to the env, as part of the build method of the spi_env. This "Russian Doll" approach to nesting configurations is scalable for many levels of hierarchy.

Before the configuration objects for the agents are assigned to their handles in the env configuration block, they are themselves constructed, and their virtual interfaces assigned using the uvm_config_db::get method, and then configured. The virtual interface assignments are to the virtual BFM interface handles that were set in the hdl_top. The APB agent may well be configured differently for different test cases and so its configuration process has been dedicated to a specific virtual method in the base class. This lets derived test classes overload this method and custom configure the APB agent as required.

The following code is for the spi_test_base class:

```
//
// Class Description:
//
//
class spi_test_base extends uvm_test;

// UVM Factory Registration Macro
//
`uvm_component_utils(spi_test_base)
```

```
//------------------------------------------
// Data Members
//------------------------------------------

//------------------------------------------
// Component Members
//------------------------------------------
// The environment class
spi_env m_env;
// Configuration objects
spi_env_config m_env_cfg;
apb_agent_config m_apb_cfg;
spi_agent_config m_spi_cfg;
// Register map
spi_register_map spi_rm;
//Interrupt Utility
intr_util INTR;

//------------------------------------------
// Methods
//------------------------------------------
extern virtual function void configure_apb_agent(apb_agent_config cfg);
// Standard UVM Methods:
extern function new(string name = "spi_test_base", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );

endclass: spi_test_base

function spi_test_base::new(string name = "spi_test_base", uvm_component parent = null);
 super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
 virtual intr_bfm temp_intr_bfm;
 // env configuration
 m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
 // Register map - Keep reg_map a generic name for vertical reuse reasons
 spi_rm = new("reg_map", null);
 m_env_cfg.spi_rm = spi_rm;
 m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
 configure_apb_agent(m_apb_cfg);
 if (!uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_mon_bfm",
 m_apb_cfg.mon_bfm)) `uvm_fatal(...)
 if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "", "APB_drv_bfm",
 m_apb_cfg.drv_bfm)) `uvm_fatal(...)
 m_spi_cfg.has_functional_coverage = 0;
 m_env_cfg.m_spi_agent_cfg = m_spi_cfg;
 // Insert the interrupt virtual interface into the env_config:
 INTR = intr_util::type_id::create("INTR");
 if (!uvm_config_db #(virtual intr_bfm)::get(this, "", "INTR_bfm", temp_intr_bfm) )
 `uvm_fatal(...)
 INTR.set_bfm(temp_intr_bfm);
 m_env_cfg.INTR = INTR;
 uvm_config_db #( spi_env_config )::set( this ,"*", "spi_env_config", m_env_cfg);
 m_env = spi_env::type_id::create("m_env", this);
 // Override for register adapter:

 register_adapter_base::type_id::set_inst_override(apb_register_adapter::get_type(),
 "spi_bus.adapter");
 endfunction: build_phase

 //
 // Convenience function to configure the apb agent
 //
 // This can be overloaded by extensions to this base class
 function void spi_test_base::configure_apb_agent(apb_agent_config cfg);
  cfg.active = UVM_ACTIVE;
  cfg.has_functional_coverage = 0;
  cfg.has_scoreboard = 0;
  // SPI is on select line 0 for address range 0-18h
  cfg.no_select_lines = 1;
  cfg.start_address[0] = 32'h0;
  cfg.range[0] = 32'h18;
endfunction: configure_apb_agent
```

To create a specific test case, the spi_test_base class is extended, and this allows the test writer to take advantage of the configuration and build process defined in the parent class. As a result, the test writer only needs to add a run_phase method. In the following (simplistic and to be updated) example, the run_phase method instantiates sequences and starts them on appropriate sequencers in the env. All of the configuration process is carried out by the super.build_phase() method call in the build_phase method.

```
//
// Class Description:
//
//
class spi_poll_test extends spi_test_base;
  // UVM Factory Registration Macro
  //
  `uvm_component_utils(spi_poll_test)

  //------------------------------------------
  // Methods
  //------------------------------------------

  // Standard UVM Methods:
  extern function new(string name = "spi_poll_test", uvm_component parent = null);
  extern function void build_phase(uvm_phase phase);
  extern task run_phase(uvm_phase phase);

  endclass: spi_poll_test

  function spi_poll_test::new(string name = "spi_poll_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  // Build the env, create the env configuration
  // including any sub configurations and assigning virtual interfaces
  function void spi_poll_test::build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction: build_phase

  task spi_poll_test::run_phase(uvm_phase phase);

    config_polling_test t_seq = config_polling_test::type_id::create("t_seq");
    set_seqs(t_seq);

    phase.raise_objection(this, "Test Started");
    t_seq.start(null);
    #100;
    phase.drop_objection(this, "Test Finished");

 endtask: run_phase
```

The Environment

The next level in the SPI UVM environment is the spi_env. This class contains a number of sub-components, namely the SPI and APB agents, a scoreboard and a functional coverage collector. Which of these sub-components gets built is determined by variables in the spi_env configuration object.

In this case, the spi_env configuration object also contains a utility which contains a method for detecting an interrupt. This will be used by ***sequences***. The contents of the spi_env_config class are as follows:

```
//
// Class Description:
//
//
class spi_env_config extends uvm_object;

const string s_my_config_id = "spi_env_config";
const string s_no_config_id = "no config";
const string s_my_config_type_error_id = "config type error";

// UVM Factory Registration Macro
//
`uvm_object_utils(spi_env_config)

// Interrupt Utility - used in the wait for interrupt task
//
intr_util INTR;

//------------------------------------------
// Data Members
//------------------------------------------
// Whether env analysis components are used:
bit has_functional_coverage = 0;
bit has_spi_functional_coverage = 1;
bit has_reg_scoreboard = 0;
bit has_spi_scoreboard = 1;
// Whether the various agents are used:
bit has_apb_agent = 1;
bit has_spi_agent = 1;
// Configurations for the sub_components
apb_agent_config m_apb_agent_cfg;
spi_agent_config m_spi_agent_cfg;
// SPI Register model
uvm_register_map spi_rm;

//------------------------------------------
// Methods
//------------------------------------------
extern task wait_for_interrupt;
extern function bit is_interrupt_cleared;
```

```
// Standard UVM Methods:
extern function new(string name = "spi_env_config");

endclass: spi_env_config

function spi_env_config::new(string name = "spi_env_config");
 super.new(name);
endfunction

// This task is a convenience method for sequences waiting for the interrupt
// signal
task spi_env_config::wait_for_interrupt;
 INTR.wait_for_interrupt();
endtask: wait_for_interrupt

// Check that interrupt has cleared:
function bit spi_env_config::is_interrupt_cleared;
 return INTR.is_interrupt_cleared();
endfunction: is_interrupt_cleared
```

In this example, there are build configuration field bits for each sub-component. This gives the env the ultimate flexibility for reuse.

During the spi_env's build phase, a handle to the spi_env_config is retrieved from the configuration space using uvm_config_db get(). Then the build process tests the various has_<sub_component> fields in the configuration object to determine whether to build a sub-component. In the case of the APB and SPI agents, there is an additional step which is to unpack the configuration objects for each of the agents from the envs configuration object and then to set the agent configuration objects in the envs configuration table after any local modification.

In the connect phase, the spi_env configuration object is again used to determine which TLM connections to make.

```
//
// Class Description:
//
//
class spi_env extends uvm_env;

  // UVM Factory Registration Macro
  //
  `uvm_component_utils(spi_env)
  //------------------------------------------
  // Data Members
  //------------------------------------------
  apb_agent m_apb_agent;
  spi_agent m_spi_agent;
  spi_env_config m_cfg;
  spi_scoreboard m_scoreboard;

  // Register layer adapter
  reg2apb_adapter m_reg2apb;
  // Register predictor
  uvm_reg_predictor#(apb_seq_item) m_apb2reg_predictor;

  //------------------------------------------
  // Constraints
  //------------------------------------------

  //------------------------------------------
  // Methods
  //------------------------------------------

  // Standard UVM Methods:
  extern function new(string name = "spi_env", uvm_component parent = null);
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);
endclass:spi_env

function spi_env::new(string name = "spi_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void spi_env::build_phase(uvm_phase phase);
  if (!uvm_config_db #(spi_env_config)::get(this, "", "spi_env_config", m_cfg))
    `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration spi_env_config from
    uvm_config_db. Have you set() it?")
  uvm_config_db #(apb_agent_config)::set(this, "m_apb_agent*",
    "apb_agent_config",
    m_cfg.m_apb_agent_cfg);
  m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);

  // Build the register model predictor
  m_apb2reg_predictor =
  uvm_reg_predictor#(apb_seq_item)::type_id::create("m_apb2reg_predictor", this);
  m_reg2apb = reg2apb_adapter::type_id::create("m_reg2apb");
  uvm_config_db #(spi_agent_config)::set(this, "m_spi_agent*",
    "spi_agent_config",
    m_cfg.m_spi_agent_cfg);
  m_spi_agent = spi_agent::type_id::create("m_spi_agent", this);

  if(m_cfg.has_spi_scoreboard) begin
    m_scoreboard = spi_scoreboard::type_id::create("m_scoreboard", this);
  end
endfunction:build_phase
```

```
function void spi_env::connect_phase(uvm_phase phase);

  // Only set up register sequencer layering if the spi_rb is the top block
  // If it isn't, then the top level environment will set up the correct sequencer
  // and predictor
  if(m_cfg.spi_rb.get_parent() == null) begin
   if(m_cfg.m_apb_agent_cfg.active == UVM_ACTIVE) begin
   m_cfg.spi_rb.spi_reg_block_map.set_sequencer(m_apb_agent.m_sequencer, m_reg2apb);
    end

    //
    // Register prediction part:
    //
    // Replacing implicit register model prediction with explicit prediction
    // based on APB bus activity observed by the APB agent monitor
    // Set the predictor map:
    m_apb2reg_predictor.map = m_cfg.spi_rb.spi_reg_block_map;
    // Set the predictor adapter:
    m_apb2reg_predictor.adapter = m_reg2apb;
    // Disable the register models auto-prediction
    m_cfg.spi_rb.spi_reg_block_map.set_auto_predict(0);
    // Connect the predictor to the bus agent monitor analysis port
    m_apb_agent.ap.connect(m_apb2reg_predictor.bus_in);
  end

  if(m_cfg.has_spi_scoreboard) begin
    m_spi_agent.ap.connect(m_scoreboard.spi.analysis_export);
    m_scoreboard.spi_rb = m_cfg.spi_rb;
  end
endfunction: connect_phase
```

The Agents

Since the UVM build process is top down, the SPI and APB agents are constructed next. The article on the ***agent build process*** describes how the APB agent is configured and built, and the SPI agent follows the same process.

The components within the agents are at the bottom of the testbench hierarchy, so the build process terminates there.