UVM SchmooVM – I Want My C Tests!

Rich Edelman
Mentor Graphics
Fremont, CA

Raghu Ardeishar
Mentor Graphics
McLean, VA

***Abstract*—The reader of this paper is interested in writing reusable tests using UVM and C. C tests, such as device drivers, are very commonly used in the industry and are written by system level designers. In many cases they have to be translated or rewritten to UVM based tests to test the design. This paper shows how the translation step can be rendered obsolete by mapping the C tests to the UVM environment. C tests include application layer software such as eHCI, xHCI for USB. These pre-written tests will be mapped to UVM sequences. The key issue of communication between SystemVerilog classes, UVM threads and C routines are shown in detail. An example is provided to be implemented in a UVM based environment where a legacy C test is also present.**

Keywords—UVM, C, sequence, sequence item, driver, agent;

I. Introduction

The reader of this paper is interested in leveraging their C programmers to write tests. They are NOT interested in rewriting the C tests in terms of SystemVerilog [3] UVM [1] based sequences. But the verification team has created a SystemVerilog UVM based test bench along with a VIP based interfaces and pre-defined UVM sequences.

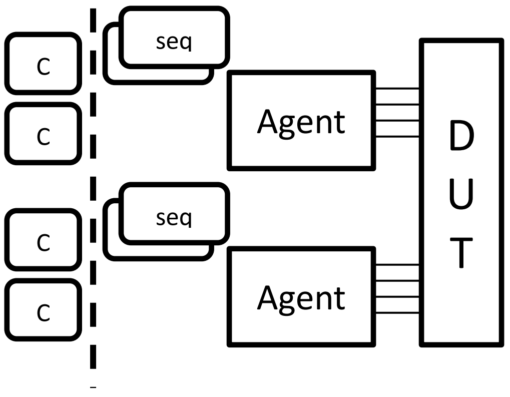

Figure 1 - C, Sequences and Agents

This paper will define a layer between the UVM based sequences and test writing in C that can provide a soft-landing for test writers new to the UVM.

SystemVerilog UVM has matured to a useful level of supported features, user guide, examples etc. The standardization of UVM has caused widespread support of SystemVerilog constructs across many simulator vendors. This standardization has allowed verification productivity to improve, as new, higher level constructs are supported and used. The UVM has become complex, and new users can have trouble getting started.

When the UVM is used as a Verification IP API, its value becomes much clearer and a new user can more easily get started. The UVM allows configuration, built-in tests (sequences), standard messaging and a simple phasing approach. Taken in context these standard features are powerful and valuable. Misused, these features cause unnecessary complexity and performance issues.

The main test writing API suggested by the UVM is UVM sequences running on agents - Figure 1. Our experience has been that sequences are hard for new users to understand and use effectively. Furthermore, sequences encourage randomization at every step, but our experience has shown that for many tests randomization is undesirable. Randomization has its place, and will be discussed in the context of this paper. This paper will propose a way to use UVM based Verification IP combined with DPI-C which allows writing tests in C while leveraging the standardized VIP. We'll show how the C tests that result are more portable and flexible than if they had been written in SystemVerilog using UVM sequences. We'll show how to create sequences that encourage being used with C. We'll demonstrate a working system which implements threaded C code using normal SystemVerilog DPI (no external thread packages). The C code has the ability to wait for time, or number of clocks as necessary enabling untimed or cycle accurate behavior. It can be as tightly coupled to simulation time as desired. We'll describe ways to work around the limitations of using DPI-C with classes, and provide an implementation of a working C interface to UVM based VIP.

The C tests will issue READS and WRITES, and can run as stand-alone tests, compiled and executed without a SystemVerilog simulator. In this mode, they provide a way to generate traffic for consumption by other tools, before simulation. When used with simulation, they provide a way to have live feedback from the traffic generation to the hardware under test. This live feedback allows modeling of at least cycle accurate effects such as cache flushes, and other resource conflicts.

II. Top Level Flow

Let's look at a possible flow from the top level. In most cases an application layer written in C. If a standard protocol like PCIe or USB is used there are standard application layers applicable for those scenarios. These application layers sit on top of the protocol layers and the main purpose of these layers is to develop an implementation agnostic test. If the underlying implementation changes the application layer test remains the same. The C tests issues READs and WRITEs, and the lower layer translates those requests into protocol specific transactions. On the implementation side there will most likely be a test written in UVM (derived from uvm_test) which will be launching many sequences.

A detailed description of the flow is provided in the sections which follow. After the build phase of the design (shown in the appendix) the test enters the run phase. This sets the context correlation between the SystemVerilog threads and classes. Then the “C” device driver is called which does a series of reads/writes or any other routine using DPI calls. Multiple sequences can be launched which can call multiple “C” routines.

The outline is shown in Figure 2 – Top Level Flow.

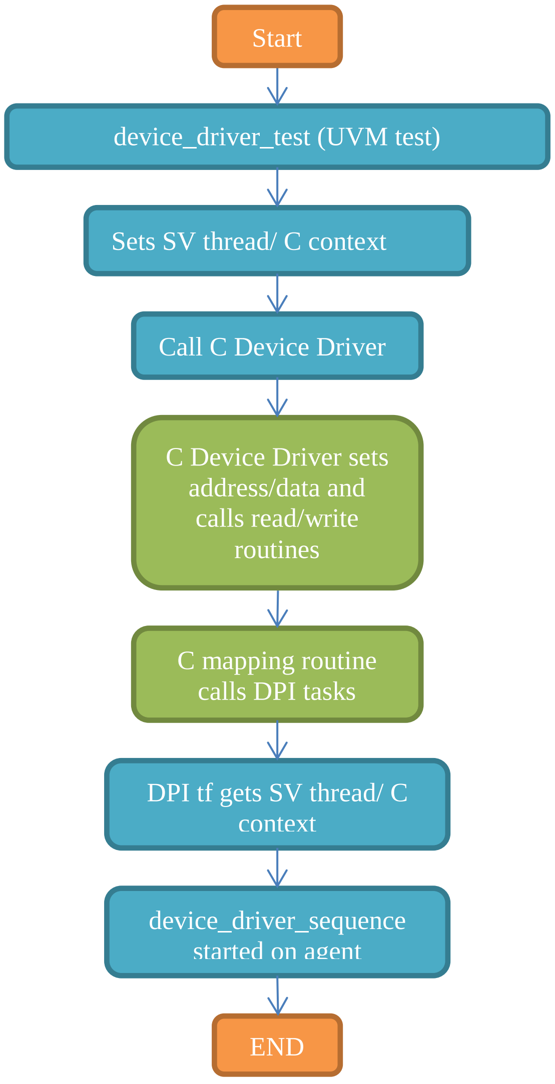
Start
device_driver_test (UVM test)
Sets SV thread/ C context
Call C Device Driver
C Device Driver sets address/data and calls read/write routines
C mapping routine calls DPI tasks
DPI tf gets SV thread/ C context
device_driver_sequence started on agent
END

Figure 2 – Top Level Flow

III. C TESTS

C tests are very common in the industry. Many are in the form of application layer device drivers such eHCI/xHCI for USB. In the initial stages of testing while the DUT is not ready a verification IP typically is substituted for the DUT and the tests are run on it as shown in Figure 3 – C/SV Test Environment.

Most verification IP these days provide a layer of UVM. However many tests are written by systems programmers in C with little knowledge of UVM. These C tests can be used to drive the UVM sequences in the VIP. The system programmer in this case does not concern himself with the UVM layer. Once the DUT is complete the VIP can be replaced with the DUT and the tests can be repeated (Figure 3 –). An example of this is ehci which is an application layer built over USB.

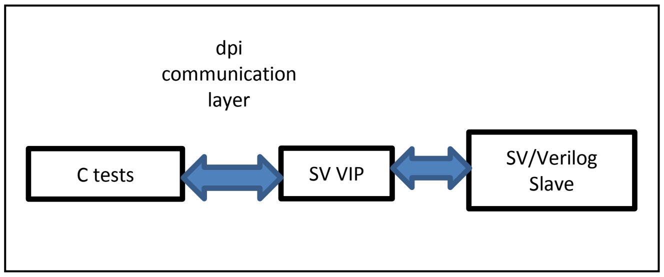

Figure 3 – C/SV Test Environment

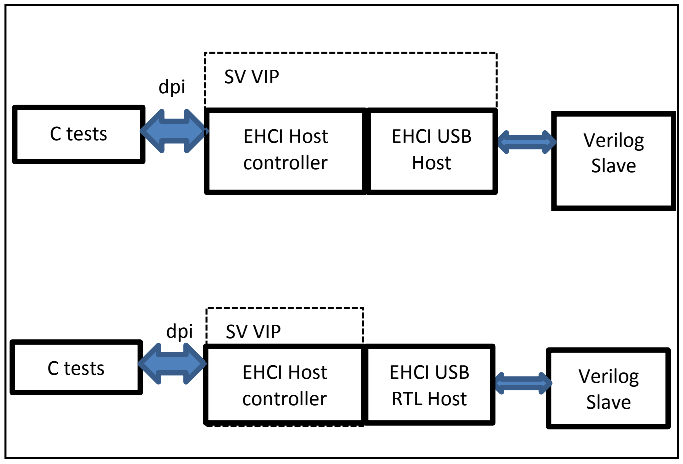

Figure 4 - EHCI C Tests & Verilog Slave

An example of a C tests is shown in Figure 5 – C Device Driver .

```
int c_device_driver() {

   int addr, data, rdata;
   printf("device_driver[[%s::%s]]\n",
          svGetNameFromScope(svGetScope()),
          sv_get_full_name());

   sv_wait_n_ticks(100);

   for (i = 16; i < 100; i+=4) {
      addr = i;
      data = addr<<8;

      my_writel(addr,   data);
      rdata = my_readl (addr);

      if (rdata != data)
       printf("ERROR: Mismatch. addr=,
       wrote data=, read data=\n",addr,data,rdata);
      else
        printf("INFO:  Match. addr=, wrote data=,
                read data=\n", addr, data, rdata);

      sv_wait_n_ticks(2);
    }
    return 0;
}
```

Figure 5 – C Device Driver

Typically a C test will set up data structures in memory and do writes or reads to certain locations in the design such as register reads and writes. In this example the address and data of the locations to be read from and written to are set and tasks "my_writel" and "my_readl" are executed. This routine will be most likely written by the engineer who has little to no knowledge of UVM or the sequences needed to translate it to a register read or write. The next step is to map these tasks to SystemVerilog tasks.

```
#include "dpiheader.h"

int my_readl(int addr) {
  int rdata;
  printf("C my_readl addr=%x\n",
      svGetNameFromScope(svGetScope()),
      sv_get_full_name(), addr);

  sv_read(&addr, &rdata);
  printf("C my_readl addr=, data=\n",
      svGetNameFromScope(svGetScope()),
      sv_get_full_name(), addr, rdata);

  return rdata;
}

my_writel(int addr, int data){
  printf("C my_writel addr=, data=",
     svGetNameFromScope(svGetScope()),
     sv_get_full_name(), addr, data);
     sv_write(&addr, &data);
}
```

Figure 6 – C/SV Mapping Routine

In Figure 6 – C/SV Mapping Routine, the device driver tasks "my_writel", and "my_readl" are mapped to SystemVerilog DPI tasks "sv_write" and "sv_read" respectively. In mapping these tasks to SystemVerilog tasks there are several housekeeping issues, which if not handled correctly can lead to improper communication between SV classes, SV threads and the C routines. In the next section we will take a look at the mechanics of how this is accomplished.

IV. C/SV CLASS LEVEL COMMMUNICATION

The C tests which include device drivers will primarily issue reads and writes to the SystemVerilog layer which will convert the high level commands to DUT register reads and writes using the DPI mechanism.

A key element of this mechanism is the correlation between SystemVerilog threads, SystemVerilog classes and C threads. The SystemVerilog thread will, at the start initialize the threads and classes so as to create a one to one mapping between the threads and classes. These classes will in many cases be UVM sequences. There needs to be a convenient mechanism to launch UVM sequences from C. The UVM user is unaware of the threading details, he just uses fork/join, for example. The C programmer is unaware of the threading details. He must be aware that his C code may become threaded under this scheme. The issues required to properly code threaded C is well documented in the industry, but is beyond the scope of this paper.

In Figure 7 – SV Read/Write, are shown the SystemVerilog DPI tasks "sv_read" and "sv_write". This task needs to be properly correlated between the C device driver task and the UVM sequence it is associated with. We use a helper class in SystemVerilog "MapPidToClassHandle" to accomplish that (Figure 8 – Class/Thread Helper Routine). The main job of this helper function is to maintain a link between the SystemVerilog thread which launched the device driver, the C device driver and the UVM sequence. This routine is explained later in this section.

In the tasks "sv_read" and "sv_write" which were called from the C device driver (Figure 5 – C Device Driver) we are attempting to call a sequence task.

```
task sv_read( addr_t addr, output data_t data);
  DEVICE_A_SEQ_base c;
  $cast(c, MapPidToClassHandle::get());
  c.read(addr, data);
endtask

task sv_write( addr_t addr, data_t data);
  SEQ_base c;
  $cast(c, MapPidToClassHandle::get());
  c.write(addr, data);
endtask
```

Figure 7 – SV Read/Write

To get the handle of that sequence we are retrieving the handle which was stored correlating the SystemVerilog class (UVM sequence device_driver_sequence) to the thread from which it was called. Figure 8 – Class/Thread Helper Routine shows how that mapping was created by initializing the class by creating a mapping to the thread in which the class was created. This initialization is done using the “MapPidToClassHandle::set” function when this sequence is run in the test (Figure 10 - Device Driver Test and Figure 8 – Class/Thread Helper Routine).

```
class MapPidToClassHandle;
  static base_class class_handle_table[process];

  static function void set(base_class
                           class_handle);
    process pid;
    pid                     = process::self();
    class_handle_table[pid] = class_handle;
  endfunction

  static function uvm_object get();
    uvm_object class_handle;
    process pid;
    pid          = process::self();
    class_handle = class_handle_table[pid];
    return class_handle;
  endfunction
endclass
```

Figure 8 – Class/Thread Helper Routine

MapPidToClassHandle is used to create a map between the SEQ class and the thread from which it was started. It has a table “class_handle_table” which stores the class handle based on the PID (process ID) of the process from which it was started. This is done in the “set” function.

Once the initialization is complete we can pass control to the C side of the test using a DPI call where the device driver will be called. The C thread will then launch reads and writes.

Now that control has been passed back to the SV thread from the C device driver we need to associate the proper sequence with the test. Now we will retrieve the class handle so that the SV sequence can be executed.

The helper class MapPidToClassHandle is again used to retrieve the class handle using the thread mapping. Once the class handle is acquired the associated class based tasks are run.

In the function called “MapPidToClassHandle::get” first the process ID is retrieved. Then the “class_handle_table” array which stores PID based class handles is used. The code snippet is shown in Figure 8 – Class/Thread Helper Routine.

Notice that the “device_driver_sequence” is the sequence which calls the “c_device_driver” which was illustrated in Figure 9 - Device Driver Sequence. This class is registering its class handle to the process thread and passing control to the C routine. This is needed when control is returned from C to the read/write routines of “device_driver_sequence”.

Please refer to Figure 7 – SV Read/Write for the tasks sv_read and sv_write. These tasks in turn call “read” and “write” routines from class “c”. The class “c” is “device_driver_sequence” as shown in Figure 9 - Device Driver Sequence.

```
class device_driver_sequence extends
      DEVICE_A_SEQ_BASE;

    data_t mem[addr_t];
    task read(  addr_t addr,
         output data_t data);
      mem_item t;
      t = new("item");
      t.rw = 1;
      t.addr = addr;
      start_item(t);
      finish_item(t);
      data = t.data;
      `uvm_info(get_type_name(),
        $sformatf(" read(%0x, %0x)",
        addr, data), UVM_HIGH)
    endtask

    task write( addr_t addr,
                data_t data);
      mem_item t;
      t = new("item");
      t.rw = 0;
      t.addr = addr;
      t.data = data;
      start_item(t);
      finish_item(t);
      `uvm_info(get_type_name(),
        $sformatf("write(%0x, %0x)",
        addr, data), UVM_HIGH)
    endtask
endclass
```

Figure 9 - Device Driver Sequence

In Figure 10 - Device Driver Test, the UVM test (“device_driver_test”) which starts everything is shown. The UVM test “test” is shown in the appendix. It sets up the connections needed for the test.

In the run_phase the device_driver_sequence is instantiated. One or more sequences can be forked as shown in the code snippet. The agent shown in the figure is shown in detail in the appendix.

```
class device_driver_test extends test;

  task run_phase(uvm_phase phase);
    device_driver_sequence
      device_driver_seq1,device_driver_seq2;

    uvm_top.print();
    `uvm_info(get_type_name(), "Starting");
    phase.raise_objection(this);
    for(int i = 0; i < 10; i++) begin
      device_driver_seq1=
        new($sformatf("device_driver_seq1"));
      device_driver_seq2=
        new($sformatf("device_driver_seq2"));

      fork
  device_driver_seq1.start(i1_agentA.sequencer);
  device_driver_seq2.start(i2_agentA.sequencer);
      join
    end
    phase.drop_objection(this);
    `uvm_info(get_type_name(),"Finished")
  endtask
endclass
```

Figure 10 - Device Driver Test

V. A FEW MINOR POINTS

There are a few points which we need to elaborate to complete the picture. Up to now we showed how to run the sequences from SystemVerilog and connect it to the C side. But there is also a need to accurately elapse time in between the operations. We have used a routine called "sv_wait_n_ticks" and "sv_wait_n_clocks" to achieve that. Note the same issues as seen before remain. If you are running several sequences and would like to elapse different times in them as you most likely would you cannot afford to get the times and threads mixed up. The time elapse functions are shown in Figure 11. Using the MapPidToClassHandle helper function we once again "get" the handle to the class (device_driver_sequence) which started the sequence of events and pass control to the sequence routine shown in Figure 12 – Sequence Time Elapse Routine.

```
task sv_wait_n_clocks(int n = 1);
  DEVICE_A_SEQ_BASE c;
  $cast(c, mapper_pkg::MapPidToClassHandle::get());
  c.wait_n_clocks(n);
endtask

task sv_wait_n_ticks(int n = 1);
  DEVICE_A_SEQ_BASE c;
  $cast(c, mapper_pkg::MapPidToClassHandle::get());
  c.wait_n_ticks(n);
endtask
```

Figure 11 – Time Elapse Routine

The sequence time elapse routines are tasks which are run once the control has been passed to the appropriate sequence.

```
task wait_n_clocks(int n = 1);
  `uvm_info(get_type_name(), $sformatf("
       wait_n_clocks(%0d)", n), UVM_HIGH)
  if ( n <= 0 ) n = 1;
  while(n-- > 0) begin
    ...
  end
  `uvm_info(get_type_name(), $sformatf("
        wait_n_clocks(%0d) DONE", n), UVM_HIGH)
endtask

task wait_n_ticks(int n = 1);
  `uvm_info(get_type_name(), $sformatf("
        wait_n_ticks(%0d)", n), UVM_HIGH)
  if (n <= 0) n = 1;
  #n;
  `uvm_info(get_type_name(), $sformatf("
        wait_n_ticks(%0d) DONE", n), UVM_HIGH)
endtask
```

Figure 12 – Sequence Time Elapse Routine

VI. SUMMARY

This paper shows how legacy C tests can be reused in a UVM based environment where the user does not have to recode the C routines. The original C code can be compiled and linked into a SystemVerilog compatible shared library. The C code can run as threaded code, and can wait for time, or clocks.

Using a mapping function from C to UVM and a helper class which tracks SystemVerilog threads and classes the user can easily move between SystemVerilog sequences and C tests. This solution provides a transparent way to reuse device driver C code with a UVM based agent verification environment.

A detailed example is provided in the appendix which shows the mechanics of how this is accomplished. Contact the authors for the complete source code.

VII. REFERENCES

[1] UVM 1.1d, http://www.accellera.org/downloads/standards/uvm/uvm-1.1d.tar.gz

[2] Furber, Stephen B. ARM System-on-chip Architecture. Harlow, England: Addison-Wesley, 2000.

[3] "IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2012, 2012. http://standards.ieee.org/getieee/1800/download/1800-2012.pdf

[4] TLM-2.0 Standard." SystemC TLM (Transaction-level Modeling). http://www.accellera.org/downloads/standards/systemc/tlm

[5] Peryer, Mark. C Based Stimulus for UVM. Mentor Graphics, n.d. Web. 06 Feb. 2013. http://www.mentor.com/products/fv/events/c-based-stimulus-for-uvm

[6] Spear, Chris. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. New York, NY: Springer, 2008. 229+

[7] Edelman, Rich. "Using SystemVerilog Now with DPI", Proc. of DVCon 2005, San Jose.

[8] Intel,. Full System Simulation with Wind River Simics. Web. 06 Feb. 2013. <http://www.windriver.com/products/simics/>.

VIII. Appendix

```
// ====================================================
// FILE: agentA-device-driver.sv
// ====================================================
package deviceA_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  import agentA_pkg::*;

  // ---------------------------------------------

  typedef bit[31:0]addr_t;
  typedef bit[31:0]data_t;

  virtual class DEVICE_A_SEQ_BASE
      extends uvm_sequence #(mem_item);

    pure virtual task read(  addr_t addr, output data_t data);
    pure virtual task write( addr_t addr,        data_t data);
    pure virtual task wait_n_clocks(int n = 1);
    pure virtual task wait_n_ticks(int n = 1);

    function new(string name = "DEVICE_A_SEQ_BASE");
      super.new(name);
    endfunction
  endclass

  // ---------------------------------------------

  export "DPI-C"         task     sv_read;
  export "DPI-C"         task     sv_write;
  export "DPI-C"         task     sv_wait_n_clocks;
  export "DPI-C"         task     sv_wait_n_ticks;
  export "DPI-C"         function sv_get_full_name;

  import "DPI-C" context task     c_device_driver();

  function string sv_get_full_name();
    return mapper_pkg::MapPidToClassHandle::get_full_name();
  endfunction

  task sv_read( addr_t addr, output data_t data);
    DEVICE_A_SEQ_BASE c;
    $cast(c, mapper_pkg::MapPidToClassHandle::get());
    c.read(addr, data);
  endtask

  task sv_write( addr_t addr, data_t data);
    DEVICE_A_SEQ_BASE c;
    $cast(c, mapper_pkg::MapPidToClassHandle::get());
    c.write(addr, data);
  endtask

  task sv_wait_n_clocks(int n = 1);
    DEVICE_A_SEQ_BASE c;
    $cast(c, mapper_pkg::MapPidToClassHandle::get());
    c.wait_n_clocks(n);
  endtask

  task sv_wait_n_ticks(int n = 1);
    DEVICE_A_SEQ_BASE c;
    $cast(c, mapper_pkg::MapPidToClassHandle::get());
    c.wait_n_ticks(n);
  endtask

  // ---------------------------------------------

  class device_driver_sequence extends DEVICE_A_SEQ_BASE;
    `uvm_object_utils(device_driver_sequence)

    virtual busA vif;

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

    task read(  addr_t addr, output data_t data);
      mem_item t;
      t = new("item");
      t.rw = 1;
      t.addr = addr;
      start_item(t);
      finish_item(t);
      data = t.data;
      `uvm_info(get_type_name(),
        $sformatf(" read(%0x, %0x)", addr, data), UVM_HIGH)
    endtask

    task write( addr_t addr,        data_t data);
      mem_item t;
      t = new("item");
      t.rw = 0;
      t.addr = addr;
      t.data = data;
      start_item(t);
      finish_item(t);
      `uvm_info(get_type_name(),
        $sformatf("write(%0x, %0x)", addr, data), UVM_HIGH)
    endtask

    task wait_n_clocks(int n = 1);
      `uvm_info(get_type_name(),
        $sformatf(" wait_n_clocks(%0d)", n), UVM_HIGH)
      if ( n <= 0 ) n = 1;
        while(n-- > 0) begin
          vif.wait_for_posedge();
        end
      `uvm_info(get_type_name(),
        $sformatf(" wait_n_clocks(%0d) DONE", n), UVM_HIGH)
    endtask

    task wait_n_ticks(int n = 1);
      `uvm_info(get_type_name(),
        $sformatf(" wait_n_ticks(%0d)", n), UVM_HIGH)
      if (n <= 0) n = 1;
        #n;
      `uvm_info(get_type_name(),
        $sformatf(" wait_n_ticks(%0d) DONE", n), UVM_HIGH)
    endtask
```

```
    task body();
      sequencerA sequencer;

      `uvm_info(get_type_name(), "Starting... ", UVM_MEDIUM)
      $cast(sequencer, m_sequencer);
      vif = sequencer.vif;
      // Start two instances of the "device driver"
      fork
        begin
          mapper_pkg::MapPidToClassHandle::set(this);
          c_device_driver();
        end
        begin
          mapper_pkg::MapPidToClassHandle::set(this);
          c_device_driver();
        end
      join
      `uvm_info(get_type_name(), "Finishing...", UVM_MEDIUM)
    endtask
  endclass

  class device_driver_test extends test;
    `uvm_component_utils(device_driver_test)

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

    task run_phase(uvm_phase phase);
      device_driver_sequence
        device_driver_seq1, device_driver_seq2;
      uvm_top.print();
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      phase.raise_objection(this);
      for(int i = 0; i < 10; i++) begin
        device_driver_seq1 =
          new($sformatf("device_driver_seq1"));
        device_driver_seq2 =
          new($sformatf("device_driver_seq2"));
        fork
          device_driver_seq1.start(i1_agentA.sequencer);
          device_driver_seq2.start(i2_agentA.sequencer);
        join
      end
      phase.drop_objection(this);
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    endtask
  endclass
endpackage
```

```
// ====================================================
// FILE: agentA-interface.sv
// ====================================================
interface busA (input clk, input reset);

  logic        rw;
  logic        valid;
  logic [31:0] addr;
  logic [31:0] rdata;
  logic [31:0] wdata;

  task wait_for_posedge();
    @(posedge clk);
  endtask

  task read(logic [31:0] l_addr, output logic [31:0] l_data);
    rw     = 1;
    addr   = l_addr;
    valid  = 1;
    @(posedge clk);
    #1;
    l_data = rdata;
    valid  = 0;
  endtask

  task write(logic [31:0] l_addr, logic [31:0] l_data);
    rw    = 0;
    addr  = l_addr;
    wdata = l_data;
    valid = 1;
    @(posedge clk);
    valid = 0;
  endtask
endinterface
```

```
// ====================================================
// FILE: agentA.sv
// ====================================================
package agentA_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"

  class mem_item extends uvm_sequence_item;
    `uvm_object_utils(mem_item)

    int id;
    rand int delay = 5;
    static int g_id = 0;

    rand bit      rw;
    rand bit[31:0]addr;
    rand bit[31:0]data;

    constraint addr_value { addr < 256; addr >= 0; }
    constraint data_value { data == 32'hdeadbeef; }

    constraint delay_value { delay < 10; delay > 0; }

    function new(string name = "mem_item");
      super.new(name);
      id = g_id++;
    endfunction

    function string convert2string();
       return $sformatf("id=%0d, %s(%0x, %0x)",
         id, (rw==1)?" READ":"WRITE", addr, data);
    endfunction

    function void do_record(uvm_recorder recorder);
      super.do_record(recorder);

      if ((id & 'h01) && (recorder.tr_handle != 0 ))
        $add_color(recorder.tr_handle, "pink");
      else
        $add_color(recorder.tr_handle, "purple");

      `uvm_record_field("name",  get_name());
      `uvm_record_field("id",    id);

      `uvm_record_field("rw",    rw);
      `uvm_record_field("addr",  addr);
      `uvm_record_field("data",  data);
      `uvm_record_field("delay", delay);
    endfunction
  endclass

  class mem_sequence extends uvm_sequence #(mem_item);
    `uvm_object_utils(mem_sequence)

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

    task body();
      mem_item t;
      `uvm_info(get_type_name(), "Starting... ", UVM_MEDIUM)

      for(int i = 0; i < 3; i++) begin
        t = new($sformatf("t%0d", i));
        if( !t.randomize() ) begin
          `uvm_fatal(get_type_name(), "Randomize Failed")
        end
        t.rw = 0;
        start_item(t);
        finish_item(t);

        t.rw = 1;
        start_item(t);
        finish_item(t);
      end
      `uvm_info(get_type_name(), "Finishing...", UVM_MEDIUM)
    endtask
  endclass

  class driverA extends uvm_driver #(mem_item);
    `uvm_component_utils(driverA)

    virtual busA vif;

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

    mem_item t;

    task run_phase(uvm_phase phase);
      `uvm_info(get_type_name(), "Starting... ", UVM_MEDIUM)
      forever begin
        seq_item_port.get_next_item(t);
        if (t.rw)
          vif.read(t.addr, t.data);
        else
          vif.write(t.addr, t.data);

        `uvm_info("DRVR",
          $sformatf("Got t=%s", t.convert2string()), UVM_MEDIUM)
        seq_item_port.item_done();
        #1;
      end
      `uvm_info(get_type_name(), "Finishing...", UVM_MEDIUM)
    endtask
  endclass

  class sequencerA extends uvm_sequencer #(mem_item);
    `uvm_component_utils(sequencerA)

    virtual busA vif; // For use by sequences.

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

  class agentA extends uvm_agent;
    `uvm_component_utils(agentA)

    driverA      driver;
    sequencerA   sequencer;

    virtual busA vif;

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

    function void build_phase(uvm_phase phase);
      driver    = driverA::type_id::create("driver", this);
      sequencer = sequencerA::type_id::create(
                                    "sequencer", this);
      driver.vif = vif;
      sequencer.vif = vif;
    endfunction

    function void connect_phase(uvm_phase phase);
      driver.seq_item_port.connect(sequencer.seq_item_export);
           driver.rsp_port.connect(sequencer.rsp_export);
    endfunction

  endclass

  class test extends uvm_test;
    `uvm_component_utils(test)

    agentA i1_agentA, i2_agentA;

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

    function void build_phase(uvm_phase phase);
      virtual busA vif1;
      virtual busA vif2;

      if (!uvm_config_db#(virtual busA)::
          get(null, "*", "1", vif1))
        `uvm_fatal(get_type_name(), "Can't find vif for 'busA'")

      if (!uvm_config_db#(virtual busA)::
          get(null, "*", "2", vif2))
        `uvm_fatal(get_type_name(), "Can't find vif for 'busA'")

      i1_agentA = agentA::type_id::create("i1_agentA", this);
      i2_agentA = agentA::type_id::create("i2_agentA", this);

      i1_agentA.vif = vif1;
      i2_agentA.vif = vif2;
    endfunction

    task run_phase(uvm_phase phase);
      mem_sequence seq1, seq2;
      uvm_top.print();
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      phase.raise_objection(this);
      for(int i = 0; i < 10; i++) begin
        seq1 = new($sformatf("mem_seq1"));
        seq2 = new($sformatf("mem_seq2"));
        fork
          seq1.start(i1_agentA.sequencer);
          seq2.start(i2_agentA.sequencer);
        join
      end
      phase.drop_objection(this);
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    endtask
  endclass
endpackage
```

```
// ====================================================
// FILE: dut.sv
// ====================================================
module dut(
  input  wire      clk,
  input  wire      reset,
  input  wire      valid,
  input  wire      rw,
  input  wire[31:0]addr,
  output reg [31:0]rdata,
  input  wire[31:0]wdata);

  logic [31:0]mem[256];

  always @(posedge clk) begin
    if (valid == 1) begin
      if (rw == 1) begin // READ
        rdata = mem[addr];
        $display("DUT %m  READ(%x, %x)", addr, rdata);
      end
      else if (rw == 0) begin // WRITE
        mem[addr] = wdata;
        $display("DUT %m WRITE(%x, %x)", addr, wdata);
      end
      else begin // IDLE
        $display("DUT %m IDLE");
      end
    end
  end
endmodule
```

```
// ====================================================
// FILE: mapper.sv
// ====================================================
package mapper_pkg;
  import uvm_pkg::*;

  class MapPidToClassHandle;
    static uvm_object class_handle_table[process];

    static function uvm_object get();
      uvm_object class_handle;
      process pid;
      pid          = process::self();
      class_handle = class_handle_table[pid];
      return class_handle;
    endfunction

    static function void set(uvm_object class_handle);
      process pid;
      pid                     = process::self();
      class_handle_table[pid] = class_handle;
    endfunction

    // Just like get(), but call class_handle.get_full_name()
    static function string get_full_name();
      uvm_object class_handle;
      process pid;
      pid          = process::self();
      class_handle = class_handle_table[pid];
      return $sformatf("%s::%0d",
        class_handle.get_full_name(), pid);
    endfunction
  endclass
endpackage
```

```
// ====================================================
// FILE: top.sv
// ====================================================
import uvm_pkg::*;

import deviceA_pkg::*;
import agentA_pkg::*;

module top();
  reg clk;
  reg reset;

  busA busA_1(clk, reset);
  busA busA_2(clk, reset);

  dut   dut_1(clk,
              reset,
              busA_1.valid,
              busA_1.rw,
              busA_1.addr,
              busA_1.rdata,
              busA_1.wdata);

  dut   dut_2(clk,
              reset,
              busA_2.valid,
              busA_2.rw,
              busA_2.addr,
              busA_2.rdata,
              busA_2.wdata);

  initial begin
    uvm_config_db#(int)::set(null, "*", "recording_detail", 1);

    uvm_config_db#(virtual busA)::set(null, "*", "1", busA_1);
    uvm_config_db#(virtual busA)::set(null, "*", "2", busA_2);

    run_test();
  end

  always begin
    clk = 1; #10;
    clk = 0; #10;
  end
endmodule
```

```
// ====================================================
// FILE: device_driver.c
// ====================================================
#include <stdio.h>
///
/// Device Driver Layer.
/// This code is not really a device driver, but writes
/// values to addresses using writel() and readl() on
/// memory mapped data structures (like a device driver might).
///

int
c_device_driver() {
    int addr, data, rdata;

    for (addr = 16; addr < 100; addr+=4) {
      data = addr<<8;

      my_writel(addr,   data);
      rdata = my_readl (addr);

      if (rdata != data)
        printf(
"C ERROR: Mismatch. addr=%0x, wrote data=%0x, read data=%0x\n",
          addr, data, rdata);
      else
        printf(
"C INFO:  Match.    addr=%0x, wrote data=%0x, read data=%0x\n",
          addr, data, rdata);

    }
    return 0;
}
```

```
// ====================================================
// FILE: os_layer.c
// ====================================================
#include <stdio.h>
#include "dpiheader.h"

int
my_readl(int addr) {
  int rdata;
  printf("C my_readl[[%s::%s]] addr=%x\n",
          svGetNameFromScope(svGetScope()),
          sv_get_full_name(), addr);
  sv_read(&addr, &rdata);
  printf("C my_readl[[%s::%s]] addr=%x, data=%x\n",
          svGetNameFromScope(svGetScope()),
          sv_get_full_name(), addr, rdata);
  return rdata;
}

my_writel(int addr, int data){
  printf("C my_writel[[%s::%s]] addr=%x, data=%x\n",
          svGetNameFromScope(svGetScope()),
          sv_get_full_name(), addr, data);
  sv_write(&addr, &data);
}
```