2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

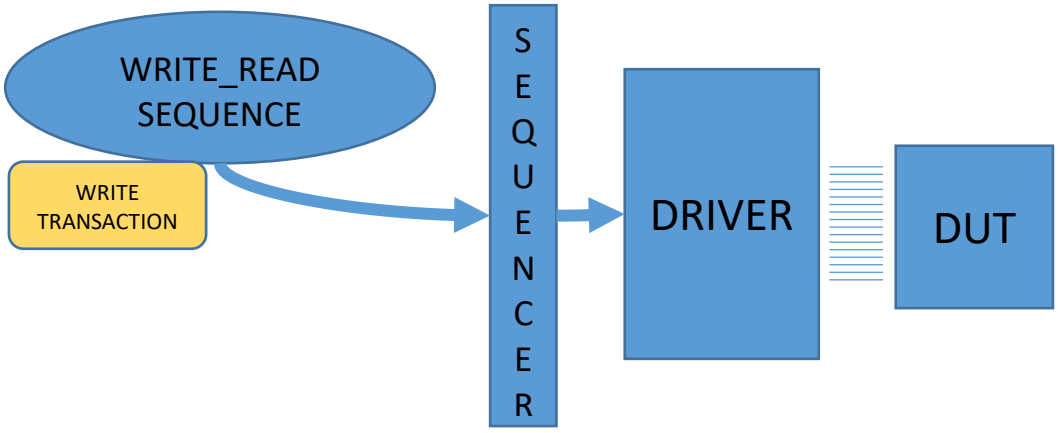
Fun with UVM Sequences – Coding and Debugging

Rich Edelman
Mentor, A Siemens Business
46871 Bayside Parkway
Fremont, CA 94538

***Abstract*- In a SystemVerilog**Error! Reference source not found. **UVM [2] test bench, most activity is generated from writing sequences. This paper will outline how to build and write basic sequences, and then extend into more advanced usage. The reader will learn about sequences that generate sequence items; sequences that cause other sequences to occur and sequences that manage sequences on other sequencers. Sequences to generate out of order transactions will be investigated. Self-checking sequences will be written.**

I. INTRODUCTION

A UVM sequence is a collection of SystemVerilog code which runs to cause "things to happen". There are many things that can happen. A sequence most normally creates a transaction, randomizes it and sends it to a sequencer, and then on to a driver. In the driver, the generated transaction will normally cause some activity on the interface pins. For example a WRITE_READ_SEQUENCE could generate a random WRITE transaction and send it to the sequencer and driver. The driver will interpret the WRITE transaction payload and cause a write with the specified address and data.



II. CREATING A SEQUENCE

A UVM sequence is just a SystemVerilog object that is constructed by calling new. It can be constructed from many different places, but normally a test might construct sequences and then run them – they embody the test. For example a test might be pseudo-coded as

```
LOAD ALL MEMORY LOCATIONS
READ ALL MEMORY LOCATIONS, CHECK THAT EXPECTED VALUES MATCH.
```

There might be a sequence to write all memory locations from A to B. And another sequence to read all memory locations from A to B. Or something simpler: A WRITE_READ_SEQUENCE that first writes all the memory locations and then reads all the memory locations.

The test below creates a sequence inside a fork/join_none. There will be four sequences running in parallel. Each sequence has a LIMIT variable set and starts to run at the end of the fork/join_none. Once all of the forks are done, the test completes.

```
class test extends uvm_test;
  `uvm_component_utils(test)

  my_sequence seq;
  ...
  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    for (int i = 0; i < 4; i++) begin
      fork
        automatic int j = i;
        seq = new($sformatf("seq%0d", j));
        seq.LIMIT = 25 * (j+1);
        seq.start(sqr);
      join_none
    end
    wait fork;
    phase.drop_objection(this);
  endtask
endclass
```

III. RUNNING A SEQUENCE - CREATING AND SENDING A SEQUENCE ITEM

The sequence below, ‘my_sequence’, is a simple sequence which creates transactions and sends them to the sequencer and driver. In the code below, the body () task is implemented. It is a simple for-loop which iterates through the loop LIMIT times. LIMIT is a variable in the sequence which can be set from outside.

Within the for-loop, a transaction object is constructed by calling new () or using the factory. Then start_item is called to begin the interaction with the sequencer. At this point the sequencer halts the execution of the sequence until the driver is ready. Once the driver is ready, the sequencer causes ‘start_item’ to return. Once start_item has returned, then this sequence has been granted permission to use the driver. Start_item should really be called “REQUEST_TO_SEND”. Now that the sequence has permission to use the driver, it randomizes the transaction, or sets the data values as needed. This is the so-called “LATE RANDOMIZATION” that is a desirable feature. The transactions should be randomized as close to executing as possible, that way they capture the most recent state information in any constraints.

After the transaction has been randomized, and the data values set, it is sent to the driver for processing using 'finish_item'. Finish_item should really be called "EXECUTE_ITEM". At this time, the driver gets the transaction handle and will execute it. Once the driver calls 'item_done ()', then finish_item will return and the transaction has been executed.

```
class my_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(my_sequence)

  transaction t;
  int LIMIT;
  ...
  task body();
    for (int i = 0; i < LIMIT; i++) begin
      t = new("t");
      start_item(t);
      t.data = i+1;
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      finish_item(t);
    end
  endtask
endclass
```

IV. EXECUTING A SEQUENCE ITEM – THE DRIVER

The driver code is relatively simple. It derives from a uvm_driver and contains a run_phase. The run_phase is a thread started automatically by the UVM core. The run_phase is implemented as a forever begin-end loop. In the begin-end block the driver calls seq_item_port.get_next_item (t). This is a task which will cause execution in the sequencer – essentially asking the sequencer for the next transaction that should be executed. It may be that no transaction is available, in which case this call will block. (There are other non-blocking calls that can be used, but are beyond the scope of this paper, and not a recommended usage). When the sequencer has a transaction to execute, then the get_next_item call will unblock and return the transaction handle in the task argument list (variable 't' in the example below). Now the driver can execute the transaction.

For this example, execution is simple – it prints a message using the transactions' convert2string () call, and waits for an amount of time controlled by the transactions 'duration' class member variable.

Once that 'execution' is complete, the seq_item_port.item_done () call is made to signal back to the sequencer and in turn the sequence that the transaction has been executed.

```
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  transaction t;
  ...
  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)
      #(t.duration); // Execute...
      seq_item_port.item_done();
    end
```

```
  endtask
endclass
```

V. CONTROLLING OTHER SEQUENCES

Sequences can have handles to other sequences; after all, a sequence is just a class object with data members, and a "task body()", which will run as a thread.

Virtual Sequences

The so-called "virtual sequence" – a sequence which may not generate sequence items, but rather starts sequences on other sequencers. This is a convenient way to create parallel operations from one control point.

A virtual sequence simply has handles to other sequences and sequencers. It starts them or otherwise manages them. The virtual sequence may have acquired the sequencer handles by being assigned from above, or by using a configuration database lookup, or other means. It may have constructed the sequence objects, or have acquired them by similar other means. The virtual sequence is like a puppet master, controlling other sequences.

A virtual sequence might look like

```
sequenceA_t sequenceA;
sequenceB_t sequenceB;
sequencerA sqrA;
sequencerB sqrB;

task body();
   sequenceA.start(sqrA);
   sequenceB.start(sqrB);
   …
```

Related Sequences

In the code snippet below, there are two sequences, ping and pong. They each have a handle to each other. They are designed to take turns. First one sends five transactions, then the other, and so on. See the appendix for the complete code.

First the handles are constructed and a run limit is setup.

```
ping_h = new("ping_h");
ping_h.LIMIT = 25;
pong_h = new("pong_h");
pong_h.LIMIT = 40;
```

Then the handles get their "partner" handle.

```
ping_h.pong_h = pong_h;
pong_h.ping_h = ping_h;
```

Finally the two sequences are started in parallel.

```
fork
  ping_h.start(sqr);
  pong_h.start(sqr);
join
```

VI. WRITING A SELF-CHECKING SEQUENCE

A self-checking sequence is a sequence which causes some activity and then checks the results for proper behavior. The simplest self-checking sequence issues a WRITE at an address, then a READ from the same address. Now the data read is compared to the data written. In some ways the sequence becomes the GOLDEN model.

```
class write_read_sequence extends my_sequence;
  `uvm_object_utils(write_read_sequence)
  ...

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

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

      // Check
      if (t.addr != t.data) begin
        `uvm_info(get_type_name(), $sformatf("Mismatch. Wrote %0d, Read %0d",
          t.addr, t.data), UVM_MEDIUM)
        `uvm_fatal(get_type_name(), "Compare FAILED")
      end
    end
  endtask
endclass
```

VII. WRITING A TRAFFIC GENERATOR SEQUENCE

A video traffic generator can be written to generate a stream of background traffic that mimic or models the amount of data a video display might require. There is a minimum bandwidth that is required for video. That calculation will change with the load on the interface or bus, but a simplistic calculation is good enough for this example. The video traffic will generate a “screen” of data 60 times a second. Each screen will have 1920 by 1024 dots. Each dot is represented by a 32 bit word. Using these numbers, the traffic generator must create 471MB per second.

```
class video extends my_sequence;
  `uvm_object_utils(video)

  int xpixels = 1920;
  int ypixels = 1024;
  int screendots;
  int rate;
  bit [31:0] addr;
```

```
    int x;
    int y;

    video_transaction t;

    task body();
      screendots = xpixels * ypixels;
      rate = 1_000_000_000 / (60 * screendots);
      forever begin
        addr = 0;
        for (x = 0; x < xpixels; x++) begin
          for (y = 0; y < ypixels; y++) begin
            t = new($sformatf("t%0d_%0d", x, y));
            start_item(t);
            if (!t.randomize())
              `uvm_fatal(get_type_name(), "Randomize FAILED")
            t.rw = WRITE;
            t.addr = addr++;
            t.duration = rate;
            finish_item(t);
          end
        end
      end
    endtask
  endclass
```

A more complete traffic generator would adjust the arrival rate based on the current conditions – as the other traffic goes up or down, the video traffic generation rate should be adjusted.

VIII. WRITING SEQUENCES THAT ARE SYNCHRONIZED WITH EACH OTHER

Sequences will be used to synchronize other sequences (so called virtual sequences). Often two sequences need to have a relationship between them formalized. For example they cannot enter their critical regions together – they must go single-file. Or they can only run after some common critical region has passed.

The code below declares two sequences which are to be synchronized (synchro_A_h and synchro_B_h). It also declares a synchronizer. There is nothing special about these classes – they have simply agreed to use some technique to be synchronized.

```
synchro synchro_A_h;
synchro synchro_B_h;
synchronizer s;

  s = new();
  synchro_A_h = new("synchroA");
  synchro_B_h = new("synchroB");
```

The synchronized sequences get a handle to the synchronizer and a starting address.

```
synchro_A_h.s = s;
synchro_A_h.start_addr = 2;
synchro_B_h.s = s;
synchro_B_h.start_addr = 2002;
```

The synchronizer control is rather simple. It just says “GO” for 20 ticks and “STOP” for 100 ticks.

```
    fork
      forever begin
        #100;
        s.state = GO;
        #20;
        s.state = STOP;
      end
    join_none
```

The synchronized sequences are started. They run to completion and then simply get restarted. They run forever.

```
    fork
      forever begin
        synchro_A_h.start(sqr);
      end
      forever begin
        synchro_B_h.start(sqr);
      end
    join_none
```

The simple synchronizer with two states – GO and STOP.

```
typedef enum bit { STOP, GO } synchro_t;

class synchronizer;
  synchro_t state;
endclass
```

The class that uses a synchronizer to NOT execute until told to do so.

```
class synchro extends my_sequence;
  `uvm_object_utils(synchro)

  bit [31:0] start_addr;
  bit [31:0] addr;
  synchronizer s;

  synchro_transaction t;

  task body();
    forever begin
      addr = start_addr;
      // Is it safe?
      while (s.state == STOP) begin
        #10;
        `uvm_info(get_type_name(), "Waiting...", UVM_MEDIUM)
      end
      t = new($sformatf("t%0d", addr));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      t.rw = WRITE;
      t.addr = addr++;
      finish_item(t);
    end
  endtask
endclass
```

In simulation, the sequence waits until the synchronizer is in the GO state. Once in the GO state, then the synchronized code generates a transaction using new, and then calls start_item/finish_item to execute it. After waiting for access to the driver and then executing, the synchronized sequence comes back to the top of the loop and checks the synchronizer state. It will either GO again, or STOP/WAIT in this example.

IX. IMPLEMENTING AN INTERRUPT SERVICE ROUTINE WITH SEQUENCES

Sequences will be used to provide an "interrupt service routine". Interrupt service routines “sleep” until needed. This is a unique kind of sequence. In this example implementation it creates an “interrupt service transaction” and does start_item and finish_item. In this way, it can send that ISR transaction handle to the driver. The driver is then going to hold that handle UNTIL there is an interrupt.

As part of the drivers’ job of handling the SystemVerilog Interface, it will handle the interrupts. In this case, handling the interrupt means that some data is put into the “held handle” and then the handle is marked done. Meanwhile, the interrupt service sequence has been waiting for the transaction to be marked DONE. In some parlance this is known as ITEM REALLY DONE. In the UVM, there are other mechanisms for handling this kind of thing, but they are unreliable and more complicated than this solution.

```
class interrupt_transaction extends transaction;
  `uvm_object_utils(transaction)
  int VALUE;
  bit DONE;
endclass

class interrupt_sequence extends my_sequence;
  `uvm_object_utils(interrupt_sequence)

  interrupt_transaction t;

  task body();
    forever begin
      t = new("isr_transaction");
      start_item(t);
      finish_item(t);
      wait(t.DONE == 1);
      `uvm_info(get_type_name(), $sformatf("Serviced %0d", t.VALUE), UVM_MEDIUM)
    end
  endtask
endclass

class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  transaction t;
  interrupt_transaction isr;
  bit done;
```

```
    int value;

    bit [31:0] mem[1920*1024];

    task interrupt_service_routine(interrupt_transaction isr_h);
      `uvm_info(get_type_name(), "Setting ISR", UVM_MEDIUM)
      done = 0;
      isr_h.DONE = 0;
      wait(done == 1);
      isr_h.VALUE = value;
      isr_h.DONE = 1;
    endtask

    task run_phase(uvm_phase phase);
      forever begin
        seq_item_port.get_next_item(t);

        if ($cast(isr, t)) begin
          fork
            interrupt_service_routine(isr);
          join_none
        end
        else begin
          ...
          // REGULAR driver processing
          ...
          if (AN INTERRUPT OCCURS) begin
            done = 1;
            value = mem[t.addr];
          end
        end
        seq_item_port.item_done();
      end
    endtask
  endclass
```

X. SEQUENCES WITH "UTILITY LIBRARIES"

Sequence "utility libraries" will be created and used. Utility libraries are simple bits of code that are useful for the sequence writer – helper functions or other abstractions of the verification process.

The open_door sequence below does just as its name implies. It opens the door to the sequencer and driver. Outside calls can now be made at will using the sequence object handle (seq.read () and seq.write () for example).

```
  class open_door extends my_sequence;
    `uvm_object_utils(open_door)

    read_transaction r;
    write_transaction w;

    task read(input bit[31:0]addr, output bit[31:0]data);
      r = new("r");
      start_item(r);
      if (!r.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      r.rw = READ;
      r.addr = addr;
```

```
    finish_item(r);
    data = r.data;
  endtask

  task write(input bit[31:0]addr, input bit[31:0]data);
    w = new("w");
    start_item(w);
    if (!w.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    w.rw = WRITE;
    w.addr = addr;
    w.data = data;
    finish_item(w);
  endtask

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    wait(0);
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass
```

The open_door is constructed and then started using normal means. Then a test program can issue reads and writes simply as in the RED lines below.

```
open_door open_door_h;

  open_door_h = new("open_door");
  fork
    open_door_h.start(sqr);
    begin
      bit [31:0] rdata;
      for (int i = 0; i < 100; i++) begin
        open_door_h.write(i, i+1);
        open_door_h.read(i, rdata);
        if ( rdata != i+1 ) begin
          `uvm_info(get_type_name(), $sformatf("Error: Wrote '%0d', Read '%0d'",
            i+1, rdata), UVM_MEDIUM)
          //`uvm_fatal(get_type_name(), "MISMATCH");
        end
      end
    end
  join_none
```

XI. CALLING C CODE FROM SEQUENCES.

Calling C code (using DPI-C) from sequences is easy, but there are a few limitations. DPI import and export statements cannot be placed inside a class – so they must be outside the class in file, global or package scope. As such, they have no design or class object scope.

```
import "DPI-C" function void c_code_add(output int z, input int a, input int b);
export "DPI-C" function sv_code;
```

A DPI-C export function or task is just a SystemVerilog function or task that has been “exported” using the export command.

```
function void sv_code(int z);
  $display("sv_code(z=%0d)", z);
endfunction
```

A DPI-C import function or task is a C function with a return value. For a task, the return value is an int (See the SystemVerilog LRM for details). For a function, the return value, is whatever the return value should be.

The simple void function c_code_add () is defined below. It has two inputs and "returns" a value in the pointer *z. This C function calls the exported SystemVerilog function 'sv_code ()'.

```
t.c:
  #include "stdio.h"
  #include "dpiheader.h"

  void
  c_code_add(int *z, int a, int b)
  {
      *z = a + b;
      sv_code(*z);
  }
```

The dpiheader.h is a handy way to check the API for DPI-C. In this example, the dpiheader.h (below) is very simple.

```
void c_code_add( int* z, int a, int b);
void sv_code( int z);
```

This sequence does nothing particularly special, it generates transactions, but it does call a C function. (c_code_add RED line below). In terms of writing sequences that call C code, there is really nothing special to do in terms of sequences. The DPI-C code must be properly written and must be declared in a proper scope.

```
class use_c_code_sequence extends my_sequence;
  `uvm_object_utils(use_c_code_sequence)

  int z;

  c_code_transaction t;

  task body();
    forever begin
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      for (int i = 0; i < 10; i++) begin
        for (int j = 0; j < 10; j++) begin
          c_code_add(z, i, j);
          t = new($sformatf("t%0d", i));
          start_item(t);
          if (!t.randomize())
            `uvm_fatal(get_type_name(), "Randomize FAILED")
          t.duration = z;
          t.rw = WRITE;
          finish_item(t);
        end
      end
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    end
```

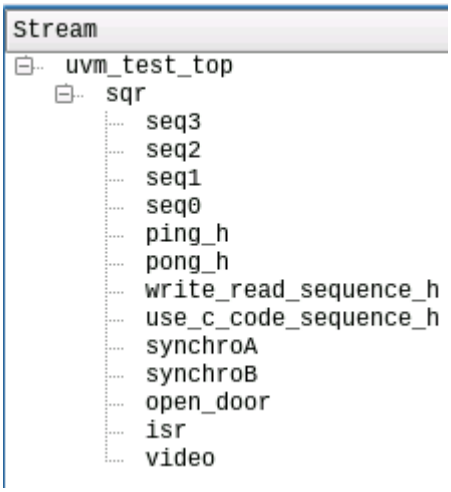
```
    endtask
endclass
```

XII. CALLING SEQUENCES FROM C CODE

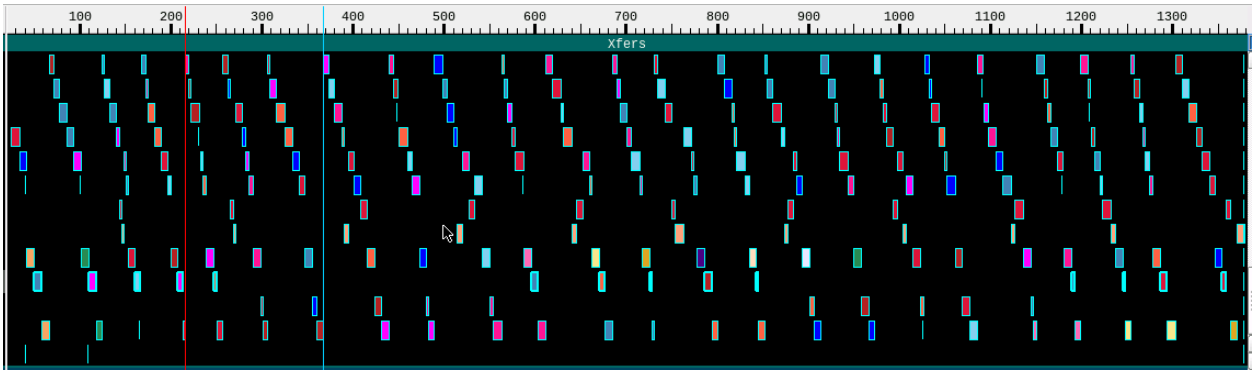
Calling sequences from C code is harder than calling C code from sequences. This is because sequences are class objects. Class objects do not have "DPI-C Scope", so in order to call into a sequence (or start a sequence), other means must be used. There are many other references on techniques to do this.

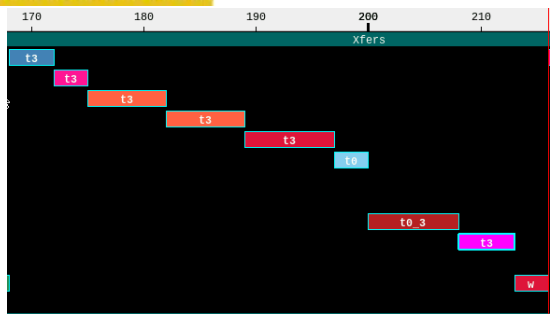
XIII. SEQUENCES AND TRANSACTIONS RECORDING

In the example code discussed throughout this paper, each of the sequences is running in parallel – at the same time on the single sequencer. The sequences (and recorded streams are listed below as children of that sequencer).



Each row is a sequence executing. It is easy to see in the two screenshots below how the sequences each take turns sending and executing a transaction on the driver.





XIV. CONCLUSION

The reader of this paper now knows that sequences are not mysterious or things to be afraid of, but rather that sequences are simply "code" – usually stimulus or test code. That code can be written to do many different things, from the original “random transaction generation” to synchronization to interrupt service routines.

Sequences are just code – important code that causes stimulus generation and results checking.

All source code is available from the author.

XV. REFERENCES

[1] SystemVerilog, 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language https://ieeexplore.ieee.org/document/8299595/citations#citations

[2] UVM LRM, https://standards.ieee.org/standard/1800_2-2017.html

XVI. APPENDIX – SIMPLE MEMORY TESTBENCH

```
// ======================================================
//
//   File: t.sv
//
// ======================================================
import uvm_pkg::*;
`include "uvm_macros.svh"

typedef enum bit[1:0] { WRITE, READ, IDLE} rw_t;

import "DPI-C" context function void c_code_add(output int z, input int a, int b);
export "DPI-C" function sv_code;

function void sv_code(int z);
  $display("sv_code(z=%0d)", z);
endfunction

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

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

  rand int duration;

  constraint rw_value {
    rw != IDLE;
  }
```

```
  constraint addr_value {
    addr > 1;
    addr < 10;
  };

  constraint value {
    duration > 1;
    duration < 10;
  };

  function void post_randomize();
    data = addr;
  endfunction

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

  function string convert2string();
    return $sformatf("[%s] %s: addr=%0d, data=%0d, duration=%0d",
      get_type_name(), rw.name(), addr, data, duration);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("rw", rw.name())
    `uvm_record_field("addr", addr)
    `uvm_record_field("data", data)
    `uvm_record_field("duration", duration)
  endfunction
endclass

class interrupt_transaction extends transaction;
  `uvm_object_utils(transaction)

  int VALUE;
  bit DONE;

  function new(string name = "transaction");
    super.new(name);
    DONE = 0;
  endfunction

  function string convert2string();
    return $sformatf("[%s] VALUE=%0d", get_type_name(), VALUE);
  endfunction

  function void do_record(uvm_recorder recorder);
    `uvm_record_field("VALUE", VALUE)
  endfunction
endclass

// Extended classes for self-documentation

class video_transaction extends transaction;
  `uvm_object_utils(video_transaction)
  function new(string name = "video_transaction");
    super.new(name);
```

```
  endfunction
endclass

class synchro_transaction extends transaction;
  `uvm_object_utils(synchro_transaction)
  function new(string name = "synchro_transaction");
    super.new(name);
  endfunction
endclass

class write_transaction extends transaction;
  `uvm_object_utils(write_transaction)
  function new(string name = "write_transaction");
    super.new(name);
  endfunction
endclass

class read_transaction extends transaction;
  `uvm_object_utils(read_transaction)
  function new(string name = "read_transaction");
    super.new(name);
  endfunction
endclass

class c_code_transaction extends transaction;
  `uvm_object_utils(c_code_transaction)
  function new(string name = "c_code_transaction");
    super.new(name);
  endfunction
endclass

class ping_transaction extends transaction;
  `uvm_object_utils(ping_transaction)
  function new(string name = "ping_transaction");
    super.new(name);
  endfunction
endclass

class pong_transaction extends transaction;
  `uvm_object_utils(pong_transaction)
  function new(string name = "pong_transaction");
    super.new(name);
  endfunction
endclass

////////////////////////////////////////////////////

class my_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(my_sequence)

  transaction t;
  int LIMIT;

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

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
```

```
    `uvm_record_field("LIMIT", LIMIT)
  endfunction

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    for (int i = 0; i < LIMIT; i++) begin
      t = new($sformatf("t%0d", i));
      start_item(t);
      t.data = i+1 ; if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      finish_item(t);
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

class video extends my_sequence;
  `uvm_object_utils(video)

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

  int xpixels = 1920;
  int ypixels = 1024;
  int screendots;
  int rate;
  int screens;
  bit [31:0] addr;

  int x;
  int y;

  video_transaction t;

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    screendots = xpixels * ypixels;
    rate = 1_000_000_000 / (60 * screendots);
    $display("rate = %0d", rate);
    forever begin
      addr = 0;
      screens++;
      for (x = 0; x < xpixels; x++) begin
        for (y = 0; y < ypixels; y++) begin
          t = new($sformatf("t%0d_%0d", x, y));
          start_item(t);
          if (!t.randomize())
            `uvm_fatal(get_type_name(), "Randomize FAILED")
          t.rw = WRITE;
          t.addr = addr++;
          t.duration = rate;
          finish_item(t);
        end
      end
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass
```

```
typedef enum bit { STOP, GO } synchro_t;

class synchronizer;
  synchro_t state;
endclass

class synchro extends my_sequence;
  `uvm_object_utils(synchro)

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

  bit [31:0] start_addr;
  bit [31:0] addr;
  synchronizer s;

  synchro_transaction t;

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    forever begin
      addr = start_addr;
      while (s.state == STOP) begin
        #10;
        `uvm_info(get_type_name(), "Waiting...", UVM_MEDIUM)
      end
      t = new($sformatf("t%0d", addr));
      start_item(t);
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      t.rw = WRITE;
      t.addr = addr++;
      finish_item(t);
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass


class interrupt_sequence extends my_sequence;
  `uvm_object_utils(interrupt_sequence)

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

  interrupt_transaction t;

  task body();
    forever begin
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      t = new("isr_transaction");
      start_item(t);
      finish_item(t);
      wait(t.DONE == 1);
      `uvm_info(get_type_name(), $sformatf("Serviced %0d", t.VALUE), UVM_MEDIUM)
    end
  endtask
endclass
```

```
class open_door extends my_sequence;
  `uvm_object_utils(open_door)

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

  read_transaction r;
  write_transaction w;

  task read(input bit[31:0]addr, output bit[31:0]data);
    r = new("r");
    start_item(r);
    if (!r.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    r.rw = READ;
    r.addr = addr;
    finish_item(r);
    data = r.data;
  endtask

  task write(input bit[31:0]addr, input bit[31:0]data);
    w = new("w");
    start_item(w);
    if (!w.randomize())
      `uvm_fatal(get_type_name(), "Randomize FAILED")
    w.rw = WRITE;
    w.addr = addr;
    w.data = data;
    finish_item(w);
  endtask

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    wait(0);
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

class write_read_sequence extends my_sequence;
  `uvm_object_utils(write_read_sequence)

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

  read_transaction r;
  write_transaction w;

  task body();
    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    for (int i = 0; i < LIMIT; i++) begin
      w = new($sformatf("t%0d", i));
      start_item(w);
      if (!w.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      w.rw = WRITE;
      finish_item(w);
    end
```

```
    for (int i = 0; i < LIMIT; i++) begin
      r = new($sformatf("t%0d", i));
      start_item(r);
      if (!r.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      r.rw = READ;
      r.data = 0;
      finish_item(r);
      if (w.addr != r.data) begin
        `uvm_info(get_type_name(), $sformatf("Mismatch. Wrote %0d, Read %0d",
            w.addr, r.data), UVM_MEDIUM)
      end
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

class use_c_code_sequence extends my_sequence;
  `uvm_object_utils(use_c_code_sequence)

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

  int z;

  c_code_transaction t;

  task body();
    forever begin
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      for (int i = 0; i < 10; i++) begin
        for (int j = 0; j < 10; j++) begin
          c_code_add(z, i, j);
          t = new($sformatf("t%0d", i));
          start_item(t);
          if (!t.randomize())
            `uvm_fatal(get_type_name(), "Randomize FAILED")
          t.duration = z;
          t.rw = WRITE;
          finish_item(t);
        end
      end
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    end
  endtask
endclass

typedef class pong;

class ping extends my_sequence;
  `uvm_object_utils(ping)

  pong pong_h;

  ping_transaction t;
  int LIMIT;

  int waiting;
```

```
    int done;

    function new(string name = "ping");
      super.new(name);
      waiting = 1;
      done = 0;
    endfunction

    task body();
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      waiting = 0;
      for (int i = 0; i < LIMIT; i++) begin
        if ((i % 5) == 0) begin
          if (!pong_h.done) waiting = 1;
          pong_h.waiting = 0;
        end
        wait(waiting == 0);
        t = new($sformatf("t%0d", i));
        start_item(t);
        t.data = i+1 ;
        if (!t.randomize())
          `uvm_fatal(get_type_name(), "Randomize FAILED")
        `uvm_info(get_type_name(),   $sformatf("Executing   %s",   t.convert2string()),
UVM_MEDIUM)
        finish_item(t);
      end
      pong_h.waiting = 0;
      done = 1;
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    endtask
  endclass

  class pong extends my_sequence;
    `uvm_object_utils(pong)

    ping ping_h;

    pong_transaction t;
    int LIMIT;

    int waiting;
    int done;

    function new(string name = "pong");
      super.new(name);
      waiting = 1;
      done = 0;
    endfunction

    task body();
      `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
      for (int i = 0; i < LIMIT; i++) begin
        if ((i % 5) == 0) begin
          if (!ping_h.done) waiting = 1;
          ping_h.waiting = 0;
        end
        wait(waiting == 0);
        t = new($sformatf("t%0d", i));
        start_item(t);
        t.data = i+1 ;
```

```
        if (!t.randomize())
          `uvm_fatal(get_type_name(), "Randomize FAILED")
        `uvm_info(get_type_name(),   $sformatf("Executing   %s",   t.convert2string()),
UVM_MEDIUM)
        finish_item(t);
      end
      ping_h.waiting = 0;
      `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
    endtask
  endclass

  class driver extends uvm_driver#(transaction);
    `uvm_component_utils(driver)

    transaction t;
    interrupt_transaction isr;
    bit done;
    int value;

    bit [31:0] mem[1920*1024];

    function new(string name = "driver", uvm_component parent = null);
      super.new(name, parent);
      done = 0;
    endfunction

    task interrupt_service_routine(interrupt_transaction isr_h);
      `uvm_info(get_type_name(), "Setting ISR", UVM_MEDIUM)
      done = 0;
      isr_h.DONE = 0;
      wait(done == 1);
      isr_h.VALUE = value;
      isr_h.DONE = 1;
    endtask

    task run_phase(uvm_phase phase);
      forever begin
        seq_item_port.get_next_item(t);
        `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)

        if ($cast(isr, t)) begin
          fork
            interrupt_service_routine(isr);
          join_none
        end
        else begin
          #(t.duration);

          if (t.addr >= 1920*1024)
            `uvm_fatal(get_type_name(), "ADDRESS FAILED")

          if (t.rw == WRITE)
            mem[t.addr] = t.data;
          else if (t.rw == READ)
            t.data = mem[t.addr];

          if ((t.rw == WRITE) && ((t.addr%42) == 0)) begin
            done = 1;
            value = mem[t.addr];
          end
```

```
      end
      seq_item_port.item_done();
    end
  endtask
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  uvm_sequencer#(transaction) sqr;
  driver d;

  my_sequence seq;
  ping ping_h;
  pong pong_h;

  open_door open_door_h;

  synchro synchro_A_h;
  synchro synchro_B_h;
  synchronizer s;

  video video_h;
  interrupt_sequence isr_h;

  write_read_sequence write_read_sequence_h;
  use_c_code_sequence use_c_code_sequence_h;

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

  function void build_phase(uvm_phase phase);
    sqr = new("sqr", this);
    d = new("d", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
      begin
        #100_000;
        phase.drop_objection(this);
      end
    join_none

    fork
      forever begin
        video_h = new("video");
        video_h.start(sqr);
      end
    join_none

    fork
```

```
    forever begin
      isr_h = new("isr");
      isr_h.start(sqr);
    end
  join_none

  open_door_h = new("open_door");
  fork
    open_door_h.start(sqr);
    begin
      bit [31:0] rdata;
      for (int i = 0; i < 100; i++) begin
        open_door_h.write(i, i+1);
        open_door_h.read(i, rdata);
        if ( rdata != i+1 ) begin
          `uvm_info(get_type_name(), $sformatf("Error: Wrote '%0d', Read '%0d'",
            i+1, rdata), UVM_MEDIUM)
        end
      end
    end
  join_none

  s = new();
  synchro_A_h = new("synchroA");
  synchro_B_h = new("synchroB");

  synchro_A_h.s = s;
  synchro_A_h.start_addr = 2;
  synchro_B_h.s = s;
  synchro_B_h.start_addr = 2002;

  fork
    forever begin
      #100;
      s.state = GO;
      #20;
      s.state = STOP;
    end
  join_none


  fork
    forever begin
      synchro_A_h.start(sqr);
    end
    forever begin
      synchro_B_h.start(sqr);
    end
  join_none

  fork
    forever begin
      use_c_code_sequence_h = new("use_c_code_sequence_h");
      use_c_code_sequence_h.start(sqr);
    end
  join_none

  fork
    forever begin
      write_read_sequence_h = new("write_read_sequence_h");
```

```
        write_read_sequence_h.LIMIT = 25;
        write_read_sequence_h.start(sqr);
      end
    join_none

    ping_h = new("ping_h");
    ping_h.LIMIT = 25;
    pong_h = new("pong_h");
    pong_h.LIMIT = 40;

    ping_h.pong_h = pong_h;
    pong_h.ping_h = ping_h;

    fork
      forever begin
        fork
          ping_h.start(sqr);
          pong_h.start(sqr);
        join
      end
    join_none

    fork
      begin
        for (int i = 0; i < 4; i++) begin
          fork
            automatic int j = i;
            seq = new($sformatf("seq%0d", j));
            seq.LIMIT = 25 * (j+1);
            seq.start(sqr);
          join_none
        end
        wait fork;
      end
    join_none

    #2468619; // Safety Valve. Never reached.
    phase.drop_objection(this);
  endtask
endclass

module top();
  initial
    run_test();
endmodule



// ====================================================
//
//   File: t.c
//
// ====================================================
#include "stdio.h"
#include "dpiheader.h"

void
c_code_add(int *z, int a, int b)
{
        *z = a + b;
```

```
            sv_code(*z);
  }
```