PRIMER

Using the Synopsys Register Abstraction Layer with UVM

Author(s):

Janick Bergeron

Based on work by:

Janick Bergeron
John Choi
Brett Kobernat

Version 1.1 / Aug 17, 2010

Introduction

The Synopsys UVM Register Abstraction Layer (RAL) is an application package that can be used to automate the creation of an object-oriented abstract model of the registers and memories inside a design. It also includes pre-defined tests to verify the correct implementation of the registers and memories as specified. It may also be used to implement a functional coverage model to ensure that every bit of every register has been exercised.

This primer is designed to teach how to create a RAL model of the registers and memories in a design, how to integrate this model in a UVM verification environment and how to verify the implementation of those registers and memories using the pre-defined tests. It will also show how the RAL model can be used to model the configuration and DUT driver code so it can be reusable in a system-level environment. Finally, it shows how the RAL model is used to implement additional functional tests.

This primer assumes that you are familiar with UVM. It also assumes that the RAL model is written manually: this will seem initially odd as RAL models were designed to be automatically generated and thus the coding style was not designed to be congenial as a register modeling style. The language used to specify the registers and memories is outside the scope of this primer.

The DUT used in this primer was selected for its simplicity. As a result, it does not require the use of many elements of the RAL application package. The DUT has enough features to show the steps needed to create a RAL model to verify the design.

This document is written in the same order you would develop a RAL model, integrate it in a verification environment and verify your design. As such, you should read it in a sequential fashion. You can use the same sequence to create your own RAL model and use it to verify your design.

The source code for the primer can be found in the following directory:

```
$UVM_HOME/examples/ral_examples/primer
```

The DUT

The Design Under Test used in this primer is an AMBA™ Peripheral Bus (APB) slave device. It is a simple single-master device with a few registers and a memory, as described in Table 1. The data bus is 32-bit wide.

Table 1: Address Map

Address	Name
0x0000	CHIP_ID
0x0020	INDEX
0x0024	DATA
0x2000-0x2FFF	DMA RAM

Tables 2 through 5 define the various fields found in each registers. "RW" indicates a field that can be read and written by the firmware. "RO" indicates a field that can be read but not written by the firmware. "W1C" indicates a field that can be read and written by the firmware, but writing a '0' has no effect and writing a '1' clears the corresponding bit if it is set.

Table 2: CHIP_ID Register

Field	Reserved	PRODUCT_ID	CHIP_ID	REVISION_ID
Bits	31-28	27-16	15-8	7-0
Access	RO	RO	RO	RO
Reset	0x0	0x176	0x5A	0x03

Table 3: INDEX Register

Field	Reserved	Index
Bits	31-8	7-0
Access	RO	RW
Reset	0x0000	0x0000

Table 4: DATA Register

The DATA register is an indirect register. It accesses one of the TABLE 32-bit registers, as specified by the value in the INDEX register. For example, reading the DATA register while the INDEX register contains the value "7", the value of TABLE[7] is read, and writing the DATA register while the INDEX register contains the value "35", the value of TABLE[35] is written.

Field	Data
Bits	31-0
Access	RW

Table 5: TABLE[256] Registers

The TABLE registers are not directly visible in the DUT address space. They are accessed indirectly via the INDEX and DATA registers.

Field	TABLE[INDEX]
Bits	31-0
Access	RW

Step 1: Register Model

IMPORTANT: this step is normally automatically performed by the code generator.

The first step is to create the abstract model of the registers in the DUT. This primer details the register model of the CHIP_ID register. The model for the other registers can be similarly constructed and are left as an exercise to the reader.

A class, based on the uvm_ral_reg class, models the register type. It contains an instance of the uvm_ral_field class for each field the register contains. The name of the register model class is technically arbitrary but it is recommended that it reflect the name or type of the register.

The name of each field instance handle is technically arbitrary but it is recommended that they be named according to the name of the respective field in the register specification. Furthermore, they are specified as public and with a *rand* attribute (where relevant) so their value may be later randomized and constrained.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that arguments to the parent constructor.

A *build()* method must be implemented with a reference to the register parent block and the register address offset. A call to the register's *configure()* method defines its properties such as size, reset value, access policy, coverage model, etc. Inside the method, each instance of the field class is created using the UVM class factory. Each field is then configured by calling its *configure()* method. This defines the field's access properties, relative position in the register, etc…

ral_slave.sv

```
class ral_reg_slave_CHIP_ID extends uvm_ral_reg;
   uvm_ral_field REVISION_ID;
   uvm_ral_field CHIP_ID;
   uvm_ral_field PRODUCT_ID;

   function new(string name);
      super.new(name);
   endfunction

   function void build(uvm_ral_block parent,
                       uvm_ral_addr_t offset, …);
      super.configure(…);
      REVISION_ID = uvm_ral_field::create(“REVISION_ID”, this);
      REVISION_ID.configure(…);
      CHIP_ID     = uvm_ral_field::create(“CHIP_ID”, this);
      CHIP_ID.configure(…);
      PRODUCT_ID  = uvm_ral_field::create(“PRODUCT_ID”, this);
      PRODUCT_ID.configure(…);
   endfunction

   function void configure();|
      super.configure
   endfunction
endclass : ral_reg_slave_CHIP_ID
```

Step 2: Memory Model

IMPORTANT: this step is normally automatically performed by the code generator.

The next step is to create the abstract model of the memories in the DUT.

A class, based on the uvm_ral_mem class, models the memory type. The name of the memory model class is technically arbitrary but it is recommended that it reflect the name or type of the memory.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that arguments to the parent constructor.

A *build()* method must be implemented with a reference to the memory parent block and the memory base offset. A call to the memory’s *configure()* method defines its properties such as size, reset value, access policy, coverage model, etc.

ral_slave.sv

```
class ral_mem_slave_DMA_RAM extends uvm_ral_mem;

   function new(string name);
      super.new(name);
   endfunction

   function void build(uvm_ral_block parent,
                       uvm_ral_addr_t offset, …);
      super.configure(…);
   endfunction;
endclass: ral_block_slave
```

Step 3: Address Map Model

IMPORTANT: this step is normally automatically performed by the code generator.

The next step is to create the abstract model of the entire address map in the DUT.

A class, based on the uvm_ral_block class, models the address map type. The name of the address map model class is technically arbitrary but it is recommended that it reflect the name or type of the address map. The address map abstraction class contains a property for each register and memory that refers to an instance of the register or memory abstraction class of the appropriate type for that register. Registers arrays are modeled using arrays of abstraction classes.

The name of each register and memory instance handle is technically arbitrary but it is recommended that they be named according to the name of the respective register and memory in the address map specification. Furthermore, they are specified as public and with a *rand* attribute so their value may be later randomized and constrained. It may also optionally contain references to field instances. These allow fields to be accessed independently of the registers they are located in.

The constructor must follow the standard *uvm_object* constructor style, with an instance name. It simply passes that arguments to the parent constructor.

ral_slave.sv

```
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand ral_reg_slave_INDEX   INDEX;
   rand ral_reg_slave_DATA    DATA;
   rand ral_reg_slave_TABLES  TABLES[256];

   uvm_ral_field REVISION_ID;
   uvm_ral_field PRODUCT_ID;

   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name);
   endfunction
   ...
endclass: ral_block_slave
```

A *build()* method must be implemented with a descriptor of the coverage models to instantiate, a reference to the block parent system and the block base address offset. A call to the block's *configure()* method defines its properties such as data bus size, endianness, access policy, coverage model, etc. Inside the method, each instance of the register classes are created using the UVM class factory. Each register abstraction class is then built and configured by calling its *build()* method.

This instantiates the register abstraction classes and configures them at the proper address offset, access policy, etc…

ral_slave.sv

```
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand ral_reg_slave_INDEX   INDEX;
   rand ral_reg_slave_DATA    DATA;
   rand ral_reg_slave_TABLES  TABLES[256];

   uvm_ral_field REVISION_ID;
   uvm_ral_field PRODUCT_ID;

   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name);
   endfunction

   function build(int cover_on :: uvm_ral::NO_COVERAGE,
                  uvm_ral_sys parent = null,
                  uvm_ral_addr_t base_addr = 0);
      super.configure(…);
      this.CHIP_ID =
                 reg_ral_slave_CHIP_ID::type_id::create(“CHIP_ID”);
      this.CHIP_ID.build(this, ‘h0, …);
      this.REVISION_ID = this.CHIP_ID.REVISION_ID;
      this.PRODUCT_ID  = this.CHIP_ID.PRODUCT_ID;

      this.INDEX = reg_ral_slave_INDEX::type_id::create(“INDEX”);
      this.INDEX.build(this, ‘h20, …);
      this.DATA = reg_ral_slave_INDEX::type_id::create(“DATA”);
      this.DATA.build(this, ‘h20, …);

      foreach (this.TABLES[i]) begin
         string name = $psprintf("TABLES[%0d]", i);
         this.TABLES[i] =
                       ral_reg_slave_TABLES::type_id::create(name);
         this.TABLES[i].build(this, 'h0, …);
         ...
      end

       this.DMA_RAM =
         ral_mem_slave_DMA_RAM::type_id::create("DMA_RAM");
      this.DMA_RAM.build(this, 'h2000, …);
      ...
   endfunction
endclass: ral_block_slave
```

Step 4: Quirky Registers Model

IMPORTANT: this step could be automatically performed by the code generator if the quirkiness is built-in and understood by the generator. Otherwise, this step is implemented by the integrator.

The next step is to define the functionality of registers that is not available directly in the base class. These are called “quirky registers”. Their behavior is specified by extending the base register models to model the non-standard behavior.

The *TABLES[256]* registers are quirky because they are not directly visible in the address map. Instead, they are accessed indirectly via the *INDEX* and *DATA* register. Because the quirkiness involves how they are accessed via the physical interface, that behavior is modeled by providing a user-defined front-door access.

A generic indexed access front-door class is defined by extending the *uvm_ral_reg_frontdoor* sequence class and implementing the *body()* method.

File: ral_slave.sv

```
class indexed_reg extends uvm_ral_reg_frontdoor;
   uvm_ral_reg INDEX;
   int         addr;
   uvm_ral_reg DATA;

   function new(string name = "indexed_reg");
      super.new(name, null);
   endfunction: new

   virtual task body();
      INDEX.write(status, addr);
      if (is_write) DATA.write(status, data);
      else DATA.read(status, data);
   endtask
endclass
```

An instance of the user-defined front-door sequence is then attached to each indirect register with the appropriate configuration:

ral_slave.sv

```
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand ral_reg_slave_INDEX   INDEX;
   rand ral_reg_slave_DATA    DATA;
   rand ral_reg_slave_TABLES  TABLES[256];

   uvm_ral_field REVISION_ID;
   uvm_ral_field PRODUCT_ID;
   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name);
   endfunction

   function build(int cover_on :: uvm_ral::NO_COVERAGE,
                  uvm_ral_sys parent = null,
                  uvm_ral_addr_t base_addr = 0);
      super.configure(…);
      this.CHIP_ID =
                  reg_ral_slave_CHIP_ID::type_id::create(“CHIP_ID”);
      this.CHIP_ID.build(this, ‘h0, …);
      this.REVISION_ID = this.CHIP_ID.REVISION_ID;
      this.PRODUCT_ID  = this.CHIP_ID.PRODUCT_ID;

      this.INDEX = reg_ral_slave_INDEX::type_id::create(“INDEX”);
      this.INDEX.build(this, ‘h20, …);
      this.DATA = reg_ral_slave_INDEX::type_id::create(“DATA”);
      this.DATA.build(this, ‘h20, …);

      foreach (this.TABLES[i]) begin
         string name = $psprintf("TABLES[%0d]", i);
         this.TABLES[i] =
                        ral_reg_slave_TABLES::type_id::create(name);
         this.TABLES[i].build(this, 'h0, …);
         begin
            indexed_reg fd = new({name, " FD"});
            fd.INDEX = this.INDEX;
            fd.addr  = i;
            fd.DATA  = this.DATA;
            this.TABLES[i].set_frontdoor(fd);
         end
      end

       this.DMA_RAM =
         ral_mem_slave_DMA_RAM::type_id::create("DMA_RAM");
      this.DMA_RAM.build(this, 'h2000, …);
   endfunction
endclass: ral_block_slave
```

Step 5:   RAL Sequencers

IMPORTANT: this step is normally automatically performed by the code generator.

The next step is to instantiate and associate a virtual sequencer with the model of the block. This virtual sequencer will serve two purposes. It will be the container of all and any block-level sequences defined for that block and, if the block is the top-most register container in the RAL model, provide access to the block’s physical interfaces.

A block-specific sequencer, extended from the *uvm_ral_sequencer* base class, must be used. The short-hand macro `` ` ``*uvm_ral_sequencer()* facilitates its implementation. The name of the sequencer type is arbitrary but it is recommended that it be the name of the block abstraction class appended with "_sequencer".

File: ral_slave.sv

```
...
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand ral_reg_slave_INDEX   INDEX;
   rand ral_reg_slave_DATA    DATA;
   rand ral_reg_slave_TABLES  TABLES[256];

   uvm_ral_field REVISION_ID;
   uvm_ral_field PRODUCT_ID;

   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name);
   endfunction

   function build(int cover_on :: uvm_ral::NO_COVERAGE,
                  uvm_ral_sys parent = null,
                  uvm_ral_addr_t base_addr = 0);
      super.configure(…);
      this.CHIP_ID =
                 reg_ral_slave_CHIP_ID::type_id::create(“CHIP_ID”);
      this.CHIP_ID.build(this, ‘h0, …);
      this.REVISION_ID = this.CHIP_ID.REVISION_ID;
      this.PRODUCT_ID  = this.CHIP_ID.PRODUCT_ID;

      this.INDEX = reg_ral_slave_INDEX::type_id::create(“INDEX”);
      this.INDEX.build(this, ‘h20, …);
      this.DATA = reg_ral_slave_INDEX::type_id::create(“DATA”);
      this.DATA.build(this, ‘h20, …);

      foreach (this.TABLES[i]) begin
         string name = $psprintf("TABLES[%0d]", i);
         this.TABLES[i] =
                      ral_reg_slave_TABLES::type_id::create(name);
         this.TABLES[i].build(this, 'h0, …);
         ...
      end

       this.DMA_RAM =
         ral_mem_slave_DMA_RAM::type_id::create("DMA_RAM");
      this.DMA_RAM.build(this, 'h2000, …);
      ...
   endfunction
endclass: ral_block_slave
```

```
`uvm_ral_sequencer(ral_block_slave_sequencer, ral_block_slave)
```

The RAL sequencer is instantiated in the *build()* method of the block's abstraction class, then associated together using the *set_model()* method in the RAL sequencer.

File: tb_env.sv

```
typedef class ral_block_slave_sequencer;
class ral_block_slave extends vmm_ral_block;
   rand ral_reg_slave_CHIP_ID CHIP_ID;
   rand ral_reg_slave_INDEX   INDEX;
   rand ral_reg_slave_DATA    DATA;
   rand ral_reg_slave_TABLES  TABLES[256];

   uvm_ral_field REVISION_ID;
   uvm_ral_field PRODUCT_ID;

   rand ral_mem_slave_DMA_RAM DMA_RAM;

   function new(string name);
      super.new(name);
   endfunction

   function build(int cover_on :: uvm_ral::NO_COVERAGE,
                  uvm_ral_sys parent = null,
                  uvm_ral_addr_t base_addr = 0);
      super.configure(…);
      this.CHIP_ID =
                 reg_ral_slave_CHIP_ID::type_id::create(“CHIP_ID”);
      this.CHIP_ID.build(this, ‘h0, …);
      this.REVISION_ID = this.CHIP_ID.REVISION_ID;
      this.PRODUCT_ID  = this.CHIP_ID.PRODUCT_ID;

      this.INDEX = reg_ral_slave_INDEX::type_id::create(“INDEX”);
      this.INDEX.build(this, ‘h20, …);
      this.DATA = reg_ral_slave_INDEX::type_id::create(“DATA”);
      this.DATA.build(this, ‘h20, …);

      foreach (this.TABLES[i]) begin
         string name = $psprintf("TABLES[%0d]", i);
         this.TABLES[i] =
                        ral_reg_slave_TABLES::type_id::create(name);
         this.TABLES[i].build(this, 'h0, …);
         ...
      end

       this.DMA_RAM =
         ral_mem_slave_DMA_RAM::type_id::create("DMA_RAM");
      this.DMA_RAM.build(this, 'h2000, …);

      begin
```

```
         ral_block_slave_sequencer ral_sqr;
         ral_sqr =
ral_block_slave_sequencer::type_id::create(this.get_name(), (parent
== null) ? null : parent.get_sequencer());
         ral_sqr.set_model(this);
      end
   endfunction
endclass: ral_block_slave

`uvm_ral_sequencer(ral_block_slave_sequencer, ral_block_slave)
```

.

Step 6: Top-Level Module

IMPORTANT: this step is performed by the integrator.

The DUT must be instantiated in a top-level module and connected to protocol-specific interfaces corresponding to the agent that drives and monitors the DUT's signals. The DUT and the relevant interfaces are instantiated in a top-level *module*.

File: tb_top.sv

```
module tb_top;
   ...
   apb_if apb0(...);

   slave dut(.paddr   (apb0.paddr[15:0] ),
             .psel    (apb0.psel        ),
             .penable (apb0.penable     ),
             .pwrite  (apb0.pwrite      ),
             .prdata  (apb0.prdata[31:0]),
             .pwdata  (apb0.pwdata[31:0]),
              ...);
   ...
endmodule: tb_top
```

This top-level module also contains the clock generators and reset signal.

File: tb_top.sv

```
module tb_top;
   bit clk = 0;
   bit rst = 0;

   apb_if apb0(clk);
   ...

   slave dut(.paddr   (apb0.paddr[15:0] ),
             .psel    (apb0.psel        ),
             .penable (apb0.penable     ),
             .pwrite  (apb0.pwrite      ),
             .prdata  (apb0.prdata[31:0]),
             .pwdata  (apb0.pwdata[31:0]),
             .clk     (clk),
             .rst     (rst));

   always #10 clk = ~clk;
endmodule: tb_top
```

Step 7: Physical Interface

IMPORTANT: this step is performed by the integrator.

A RAL model is not aware of the physical interface used to access the registers and memories. It issues generic read and write transaction requests at specific addresses but these generic transactions need to be executed on whatever physical interface is provided by the DUT.

The translation must be accomplished in a user-defined extension of the *uvm_ral_gp_seq* sequence. The implementation of the *body()* method translates and executes the generic transaction specified by the *uvm_tlm_gp* instance in the *gp* property into suitable sequences on the appropriate sequencer type.

File: gp_to_apb_seq.sv

```
class gp_to_apb_rw_seq extends uvm_ral_gp_seq;
   apb_sequencer apb_seqr;

   function new(string        name    = "gp_to_apb_rw_seq",
                apb_sequencer on_seqr = null);
      super.new(name);
      this.apb_seqr = on_seqr;
   endfunction: new

   virtual task body();
      apb_rw cyc;
      bit      ok;
      cyc = apb_rw::type_id::create("cyc", this.apb_seqr);
      cyc.addr = gp.m_address;
      if (gp.m_command == uvm_tlm_gp::TLM_WRITE_COMMAND) begin
         // Write cycle
         cyc.kind = apb_rw::WRITE;
         cyc.data = {gp.m_data[3], gp.m_data[2],
                     gp.m_data[1], gp.m_data[0]};
      end
      else begin
         // Read cycle
         cyc.kind = apb_rw::READ;
      end

      this.apb_seqr.execute_item(cyc);

      if (gp.m_command == uvm_tlm_gp::TLM_READ_COMMAND) begin
          gp.m_data = new [4];
         {gp.m_data[3], gp.m_data[2],
          gp.m_data[1], gp.m_data[0]} = cyc.data;
      end
      gp.set_response_status(uvm_tlm_gp::TLM_OK_RESPONSE);
   endtask : body

endclass : gp_to_apb_rw_seq
```

Step 8: Verification Environment

IMPORTANT: this step is performed by the integrator.

The RAL model is instantiated in the environment's *build()* method using the UVM class factory. It's own build() method must then be explicitly called, Optionally specifying coverage models to be included and the base address of the DUT.

File: tb_env.sv

```
class tb_env extends uvm_component;
   ral_block_slave     ral_model;
   apb_agent           apb;

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

   virtual function void build();
      super.build();

      ral_model = ral_block_slave::type_id::create("slave");
      ral_model.build();
      apb       = apb_agent::type_id::create("apb", this);
   endfunction: build
   ...
endclass tb_env;
```

In the environment's *connect()* method, an instance of the generic transaction translation sequence, appropriately connected to the sequencer of the physical bus, must then be registered with the RAL sequencer using the *add_subsequence()* method.

File: tb_env.sv

```
class tb_env extends uvm_component;
   ral_block_slave     ral_model;
   apb_agent           apb;

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

   virtual function void build();
      super.build();

      ral_model = ral_block_slave::type_id::create("slave");
      ral_model.build();
      apb       = apb_agent::type_id::create("apb", this);
   endfunction: build

   virtual function void connect();
      super.connect();

      begin
         uvm_ral_sequencer sqr = ral_model.get_sequencer();
         gp_to_apb_rw_seq gp2apb;
         gp2apb = new("gp2apb", apb.sqr);
         sqr.add_subsequence(gp2apb);
         ...
      end
      ...
   endfunction: connect
endclass tb_env;
```

The pre-defined RAL test sequences need to reset the DUT periodically. To that end, a hardware reset sequence must be defined then registered with the RAL sequencer using the by setting its *hw_reset_seq_name* string option to the name of the reset sequence:

File: tb_env.sv

```
class hw_reset_seq extends uvm_sequence;
   `uvm_sequence_utils(hw_reset_seq, slave_ral_sequencer)
   function new(string name = "hw_reset_seq");
      super.new(name);
   endfunction: new

   virtual task body();
      $root.tb_top.rst <= 1;
      repeat (5) @ (posedge $root.tb_top.clk);
      $root.tb_top.rst <= 0;
   endtask
endclass
```

```
class tb_env extends uvm_component;
   ral_block_slave     ral_model;
   apb_agent           apb;

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

   virtual function void build();
      super.build();

      ral_model = ral_block_slave::type_id::create("slave");
      ral_model.build();
      apb       = apb_agent::type_id::create("apb", this);
   endfunction: build

   virtual function void connect();
      super.connect();

      begin
         uvm_ral_sequencer sqr = ral_model.get_sequencer();
         gp_to_apb_rw_seq gp2apb;
         gp2apb = new("gp2apb", apb.sqr);
         sqr.add_subsequence(gp2apb);
         sqr.set_hw_reset_seq(“hw_reset_seq”);
      end
      ...
   endfunction: connect
endclass tb_env;
```

Step 9:   The Pre-Defined Tests

IMPORTANT: this step is performed by the integrator.

You are now ready to execute any of the pre-defined tests! It is best to start with the simplest test: applying hardware reset then reading all of the registers to verify their reset values. Many of the problems with the DUT, the RAL model, or the integration of the two will be identified by this simple test.

Command:

```
% vcs –f compile_vcs.f
% simv +UVM_TESTNAME=uvm_ral_hw_reset_test
```

Step 10: User-Defined Tests

IMPORTANT: this step is performed by the verification engineer(s).

Congratulations! You are now ready to execute your actual tests! These tests may be implemented as sequences executed on the RAL sequencer, or as directed calls to the *read()* and *write()* methods in the RAL model.

user_test.sv

```
class user_test_seq extends uvm_sequence;

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

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

   `uvm_sequence_utils(user_test_seq , ral_block_slave_sequencer);

   virtual task body();
      // Randomize the content of 10 random indexed registers
      repeat (10) begin
         bit [7:0]         idx = $urandom;
         uvm_ral_data_t    data = $urandom;
         uvm_ral::status_e status;
         p_sequencer.ral.TABLES[idx].write(status, data);
      end
   endtask : body
endclass : user_test_seq

class ral_user_test extends uvm_test;
   `uvm_component_utils(ral_user_test);

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual task run();
      ral_block_slave_sequencer sqr;
      uvm_sequence_base seq;

      $cast(sqr, uvm_top.lookup("rals.slave"));
      seq = sqr.get_sequence(sqr.get_seq_kind("user_test_seq"));
      seq.start(sqr);
      seq.wait_for_sequence_state(FINISHED);

      // Find which indexed registers are non-zero
      foreach (sqr.ral.TABLES[i]) begin
         uvm_ral_data_t    data;
         uvm_ral::status_e status;

         sqr.ral.TABLES[i].read(status, data);
         if (data != 0)
            $write("TABLES[%0d] is 0x%h...\n", i, data);
      end
      global_stop_request();
   endtask : run
endclass : ral_user_test
```