SNUG-2024
Silicon Valley, CA
Voted Best Presentation
3rd Place

World Class SystemVerilog & UVM Training

Understanding the UVM m_sequencer, p_sequencer Handles, and the `uvm_declare_p_sequencer Macro

Clifford E. Cummings
Paradigm Works, Inc.
Provo, Utah, USA
www.sunburst-design.com

ABSTRACT

There is significant confusion amongst verification engineers about the UVM m_sequencer and p_sequencer handles and the use of the `uvm_declare_p_sequencer macro. Every sequence has an m_sequencer handle but sequences only have access to a p_sequencer handle if the sequence properly declares and sets it, or if the sequence calls the `uvm_declare_p_sequencer macro.

This paper describes the m_sequencer handle and how it is defined and used by sequences. This paper also describes the p_sequencer handle, how it is optionally defined and used, and how it is typically defined using the `uvm_declare_p_sequencer macro.

This paper explains why the p_sequencer handle is typically required if engineers use the inferior uvm_config_db resources API. A commonly used virtual sequencer example is shown to demonstrate the typical usage of the p_sequencer handle by verification engineers.

This paper describes how to generally avoid usage of the confusing p_sequencer handle if engineers use the more powerful uvm_resource_db resources API.

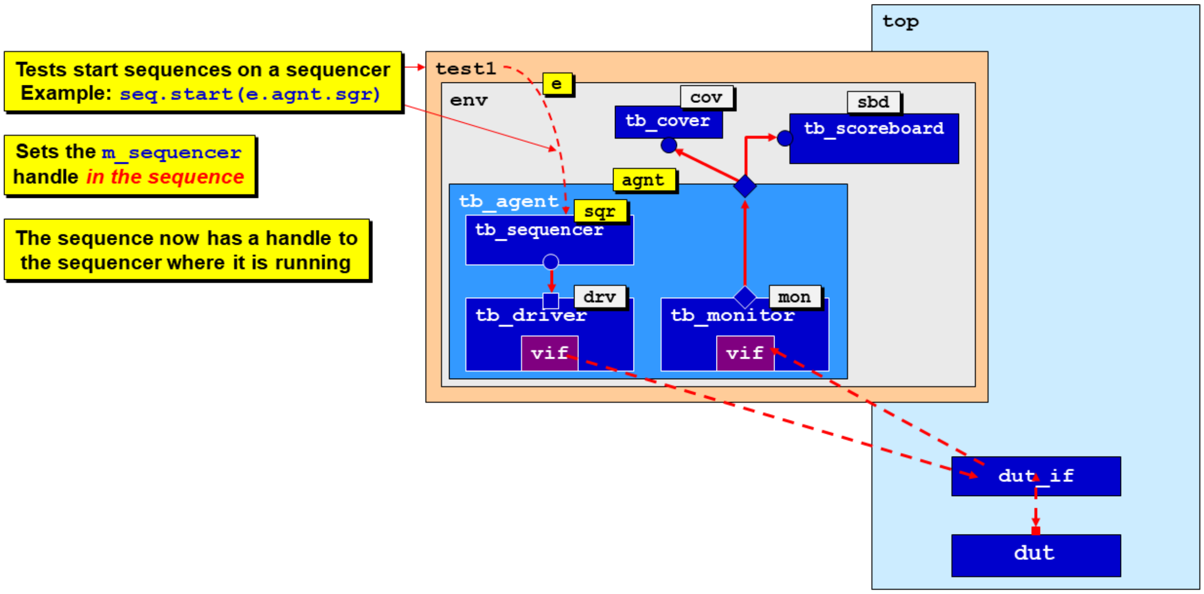
Table of Contents

Table of Figures

Table of Examples

1. Sequences are started on sequencers

Every sequence must be started on a sequencer, with one notable exception that is detailed in Section 6. The sequence must have a handle to the sequencer that is executing the sequence. There are three common methods that can be used to start the sequence.



Tests start sequences on a sequencer
Example: `seq.start(e.agnt.sgr)`

Sets the `m_sequencer`
handle ***in the sequence***

The sequence now has a handle to
the sequencer where it is running

Figure 1 - Tests start top-level sequences on sequencers

1.1 Sequence.start()

Perhaps the most common way to start a sequence is to use the `sequence.start()` method. This method can be invoked from a top-level test as well as from other sequences.

```
sequence.start(full_path_to_sequencer)
```

Example 1 - Sequence.start() on full path to sequencer

In the block level diagram shown in Figure 1, `test1` starts the sequence `seq` on the `env` (with handle name `e`), the `tb_agent` (with handle name `agnt`), and the `tb_sequencer` (with handle name `sqr`). The command that `test1` executes is shown in Example 2.

```
seq.start(e.agnt.sqr)
```

Example 2 - seq.start() to environment . agent . sequencer from top-level test

This will cause an `m_sequencer` handle in the started sequence to be set to `e.agnt.sqr`.

The `m_sequencer` handle is described in Section 2.

1.2 Manually Setting the Sequencer Handle

A verification engineer can manually set a sequencer handle inside of the sequence and then the sequence can be started on `null`.

Inside of a sequence base class or the sequence itself, the sequencer handle is set using the command shown in Example 3.

```
        sqr_handle.set_sequencer(full_path_to_sequencer)
```

Example 3 - set_sequencer command

The test then just starts a sequence and since the sequence already has a handle to the sequencer, the sequence can be started on `null` as shown in Example 4

```
        sequence.start(null)
```

Example 4 - Sequence started on null

Using this technique can be tricky because the `sqr_handle` in the started sequence typically cannot be the `m_sequencer` handle. Engineers who try to use this technique often make the mistake of trying to run the set of commands shown in Example 5.

```
        1  write_read seq = write_read::type_id::create("seq");
        2  …
        3  seq.set_sequencer(e.agnt.sqr);
        4  seq.start(null)
```

Example 5 - Common set_sequencer() mistake

In Example 5:

Line 1: A `write_read` sequence object is created.

Line 3: The `m_sequencer` handle in the `write_read` sequence is set to the path `"e.agnt.sqr"`

Line 4: The `m_sequencer` handle in the `write_read` sequence is overwritten and set to `null`.

Line 4 will now try to start a `write_read` sequence on a `null` sequencer handle, which will produce a runtime error as shown in Example 6.

```
 UVM_FATAL @0: reporter@@seq [SEQ] neither the item's sequencer nor dedicated
               sequencer has been supplied to start the item in seq
```

Example 6 - VCS Fatal UVM null-sequencer error message

The `seq.start(null)` starts a sequence and sets the `m_sequencer` handle to `null`. Before doing `seq.start(null)`, a user-test typically must manually set a sequencer handle that is frequently inherited from a `sequence_base` class.

An example using this `seq.start(null)` technique is shown in Section 6.

1.3 `uvm_do macros

Calling the `` `uvm_do `` macro from a user test, extended from `uvm_test` is illegal.

The `` `uvm_do `` macros can be used to start a sub-sequence from a parent sequence, but they cannot be used to start a sequence from a test.

The `` `uvm_do `` macros can determine if the item to be executed is a transaction or a sequence. If the item is a transaction, commonly referred to as a `sequence_item`, the `` `uvm_do `` macro will execute `start_item()`, `randomize()`, `finish_item()`. If the item to be executed is a sequence, the `` `uvm_do `` macro will execute `seq.start(m_sequencer)`.

2. UVM Base Classes for Sequencers and Sequences

It is useful to understand where the `m_sequencer` handle, and the `set_sequencer()` and `get_sequencer()` methods are defined in the UVM base classes.

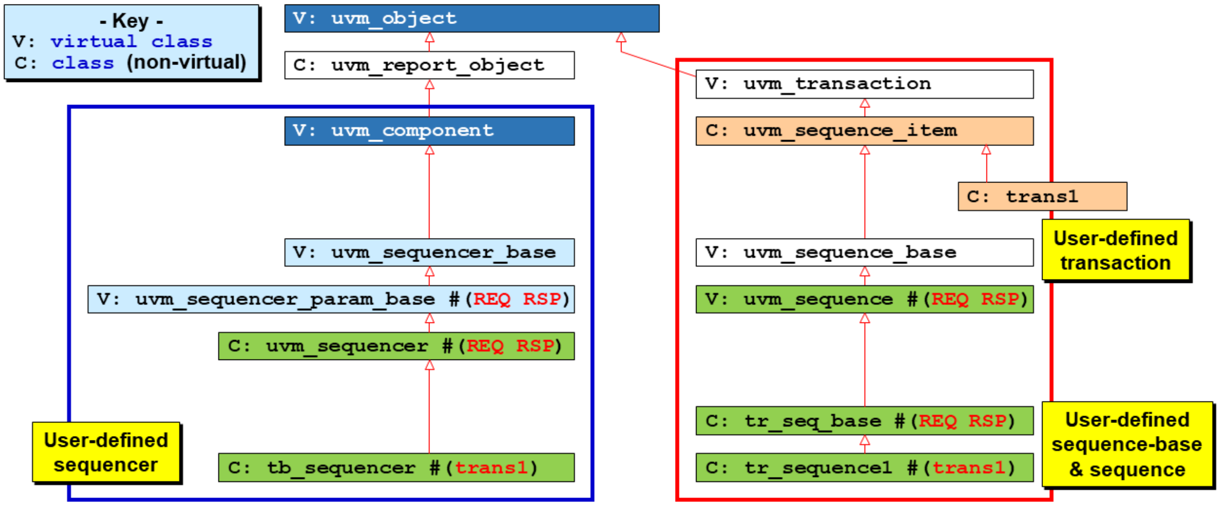


Figure 2 - Base & user-defined sequencer and sequence class hierarchies

2.1 Where is the m_sequencer variable defined?

The `m_sequencer` variable is declared in the `src/seq/uvm_sequence_item.svh` base class file, and since the `uvm_sequencer_base` and `uvm_sequence` classes are both derivatives of the `uvm_sequence_item` class, they both inherit the `m_sequencer` declaration.

Since user-defined sequences are derivatives of the `uvm_sequence` base class, user sequences also inherit an `m_sequencer` handle. Every sequence must have a properly set `m_sequencer` handle that holds the handle of the sequencer where the user-sequence is being executed. An exception to this rule is described in Section 6.

Figure 3 shows the UVM base class hierarchy associated with `uvm_sequence_items` and `uvm_sequences`. The `protected uvm_sequencer_base m_sequencer`, declared in the `uvm_sequence_item` base class, ensures that only transactions and sequences can access the `m_sequencer` handle. The `get_sequencer()` and `set_sequencer()` methods are also defined in the `uvm_sequence_item` base class. This means that all user transaction classes, the `uvm_sequence_base` and `uvm_sequence` base classes, and the user-defined sequences all inherit the `m_sequencer` handle and the `get_sequencer()/set_sequencer()` methods.

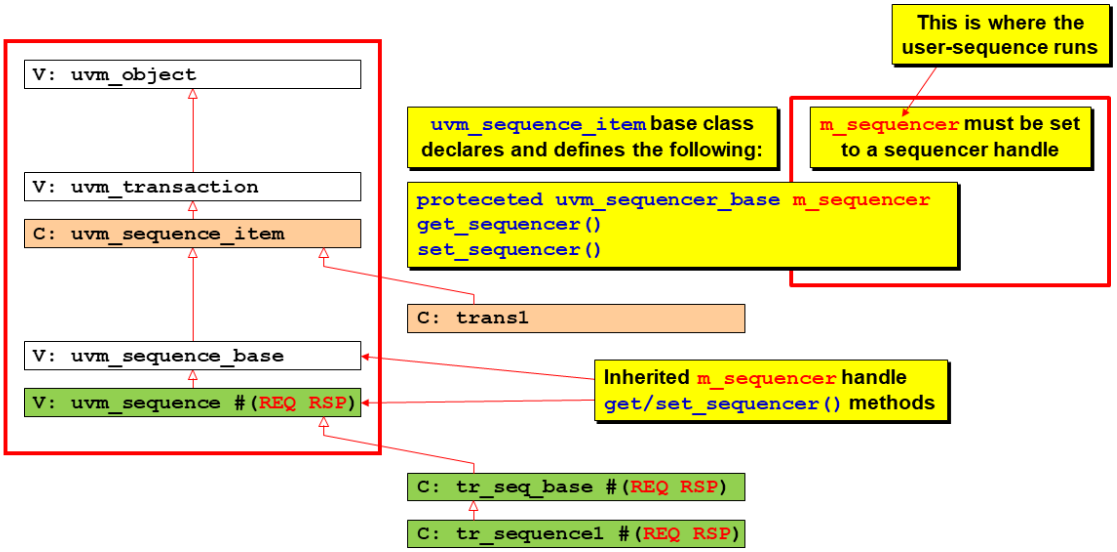


Figure 3 - m_sequencer handle declared in the uvm_sequence_item base class – inherited by transaction and sequence classes

2.2 set_sequencer() details

When the `set_sequencer()` method is called, it first sets the `m_sequencer` handle to the value that was passed to this method as an input argument. Then it calls UVM's internal `m_set_p_sequencer()` method, as shown in Example 7.

```
    // Sets the default sequencer for the sequence to sequencer. It will
    // take effect immediately, so it should not be called while the
    // sequence is actively communicating with the sequencer.

    virtual function void set_sequencer(uvm_sequencer_base sequencer);
      m_sequencer = sequencer;
      m_set_p_sequencer();
    endfunction
```

Example 7 - Default activity of set_sequencer() method

By default, the `m_set_p_sequencer()` method is empty and does nothing. The default `m_set_p_sequencer()` method is shown in Example 8.

```
    // Internal method
    virtual function void m_set_p_sequencer();
      return;
    endfunction
```

Example 8 - Default m_set_p_sequencer() method definition – empty - returns nothing

The `get_sequencer()` method is an accessor method that is used to retrieve the `m_sequencer` handle of the current sequence. In theory, UVM users should not reference the `m_sequencer` handle directly but are encouraged to use the `get_sequencer()` method to access the `m_sequencer` handle; however, it is not uncommon for engineers to access the `m_sequencer` handle directly from user-defined sequences.

```
      // Returns a reference to the default sequencer used by this sequence.
      function uvm_sequencer_base get_sequencer();
        return m_sequencer;
      endfunction
```

Example 9 - Default activity of the get_sequencer() method

The reason behind the invisible internal-variable theory (which is a common software coding practice) is that the `m_sequencer` handle is an internal variable that should be unknown to the user, and as an internal variable, UVM library developers might decide to change the name. A quick search for any reference to the `m_sequencer` handle in the UVM 1.2 Class Reference manual turns up empty.

The empty `m_set_p_sequencer()` is defined in the `src/seq/uvm_sequence_item.svh` base class file. This method is overridden by the `` `uvm_declare_p_sequencer() `` macro, and since the `uvm_sequence_base` and `uvm_sequence` classes are both derivatives of the `uvm_sequence_item` class, they both inherit the `m_set_p_sequencer` empty method.

When a user-sequence base class or user-sequence class calls the `` `uvm_declare_p_sequencer `` macro, it overrides the `m_set_p_sequencer` method and populates it as described in Section 5. and is shown in Example 13.

3. What does the sequence start() method do?

Sequences are started on a sequencer using the `start()` method shown in Example 10. The `start()` method can take up to four arguments, but in typical usage the verification engineer only passes in the first argument, which is the full path to the sequencer. The other three arguments have default values that are rarely modified.

The `start()` method definition can be found in the `src/seq/uvm_sequence_base.svh` base class file and is shown in Example 10.

```
      virtual task start (uvm_sequencer_base sequencer,
                          uvm_sequence_base  parent_sequence = null,
                          int this_priority = -1,
                          bit call_pre_post = 1);
```

Example 10 - virtual start task prototype defined in the uvm_sequence_base class

Three of the arguments on the `start()` method have default values and are not discussed in this paper. The full path to the sequencer in the testbench is typically passed as the first argument to the `start()` method. The first argument is sometimes called with the value `null` if the started-sequence already has subsequencer handles declared and set inside of the calling sequence, and if the subsequences are started on the defined subsequencer handles and not on the `m_sequencer` handle. More about starting sequences on `null` is described in Section 6.

The `start()` task calls the `set_item_context()` method defined in the `src/seq/uvm_sequence_item.svh` base class file. The prototype for the `set_item_context()` method is shown in Example 11.

```
      function void set_item_context(uvm_sequence_base  parent_seq,
                                     uvm_sequencer_base sequencer = null);
```

Example 11 - set_item_context method prototype defined in the uvm_sequence_item base class

When the test or another sequence makes a call to the `start()` method, (for example: `sequence.start(e.agnt.sqr)`), the `start()` method will call `set_item_context(e.agnt.sqr)`, which will set the `m_sequencer` handle in the started sequence to the value of `e.agnt.sqr`. The sequence will then execute its code on the sequencer whose handle is stored in the `m_sequencer` variable.

4. Starting a sequence on null

One must execute caution when attempting to start a sequence on `null`. Starting a sequence on null (for example: `sequence.start(null)`), will set the `m_sequencer` handle in the started sequence to the value of `null`.

Executing `sequence.start(null)` requires the sequence to have declared subsequencer handles in the sequence, and a component, such as the test, must hierarchically set the handles during the `run_phase` after the sequence and subsequences have been created.

```
class test_base extends uvm_test;
  `uvm_component_utils(test_base)
  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = env::type_id::create("e", this);
  endfunction

  // Initialize the vseq_base handle
  function void init_seq(seq_base seq);
    seq.SQR = e.agnt.sqr;
  endfunction
endclass

class test1 extends test_base;
  `uvm_component_utils(test1)

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

  task run_phase(uvm_phase phase);
    write_read seq = write_read::type_id::create("seq");

    phase.raise_objection(this);
    if (!seq.randomize())
        `uvm_error("RAND", "Failed randomization");
    seq.set_sequencer(e.agnt.sqr);      // No compilation error
    seq.start(null);                    // but does not work
    init_seq(seq);               // Works!
    seq.start(null);             // Works!
    phase.drop_objection(this);
  endtask
endclass
```

Example 12 - Failing set_sequencer() / seq.start(null)

5. What does the `uvm_declare_p_sequencer macro do?

The UVM Class Reference Manual, Section 21.3 – with title Sequence-Related Macros, includes the following definition for the **`` `uvm_declare_p_sequencer ``** macro under the sub-heading, “Sequencer Subtypes:”

> This macro is used to declare a variable *p_sequencer* whose type is specified by *SEQUENCER*.

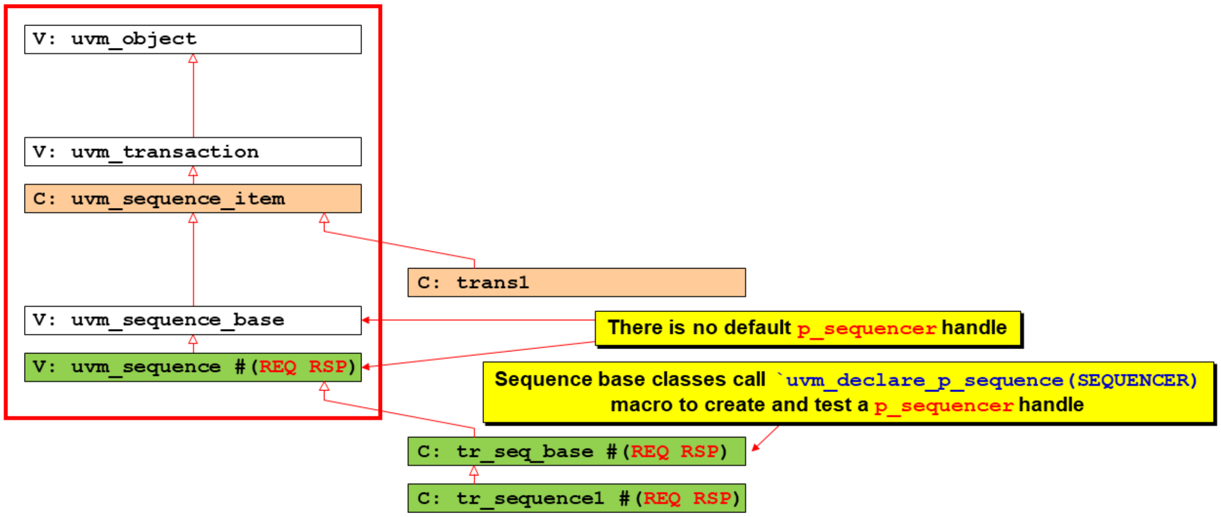


Figure 4 - p_sequencer handle in user sequence classes by calling the `uvm_declare_p_sequencer macro

The UVM base classes do not define a **`p_sequencer` handle**. A user could declare a **`p_sequencer`** handle manually, but the **`p_sequencer`** handle is declared most often by calling the **`` `uvm_declare_p_sequencer ``** macro.

If a **`p_sequencer`** handle is not created by the user, any attempt to start a sequence on the **`p_sequencer`** handle, or to reference the **`p_sequencer`** handle will fail with a null pointer error message.

The **`` `uvm_declare_p_sequencer(SEQUENCER) ``** macro is defined as shown in Example 13. This **`` `uvm_declare_p_sequencer ``** macro definition can be found in the **`src/macros/uvm_sequence_defines.svh`** base class file:

```
`define uvm_declare_p_sequencer(SEQUENCER)  \
  SEQUENCER p_sequencer;\
  virtual function void m_set_p_sequencer();\
    super.m_set_p_sequencer(); \
    if( !$cast(p_sequencer, m_sequencer)) \
        `uvm_fatal("DCLPSQ", \
        $sformatf("%m %s Error casting p_sequencer, …",
                   get_full_name())) \
  endfunction
```

Example 13 - UVM source code for the uvm_declare_p_sequencer macro definition

Example 13 shows the **`` `uvm_declare_p_sequencer ``** macro definition with an abbreviated error message. The fully defined and formatted error message is shown in Example 14.

```
      $sformatf("%m %s Error casting p_sequencer, please verify that this
                 sequence/sequence item is intended to execute on this
                 type of sequencer", get_full_name()))
```

Example 14 - Full uvm_declare_p_sequencer error message

The `` `uvm_declare_p_sequencer `` macro performs the following actions:

1. Declares a `p_sequencer` handle of the `SEQUENCER` type passed in as a parameter. The `SEQUENCER` handle should be a sequencer-***type***, **NOT** a sequencer-***handle***.
2. Overrides the empty virtual `m_set_p_sequencer()` method to do the following:
   a. Calls the `super.m_set_p_sequencer()` method (typically empty).
   b. Casts the sequence's `m_sequencer` handle to the locally declared `p_sequencer` handle.
   c. The `$cast` operation checks to ensure the sequence's `m_sequencer` handle matches the type of the locally declared `p_sequencer` handle.
   d. If the `m_sequencer` and `p_sequencer` handle types match, the `m_set_p_sequencer()` completes successfully.
   e. If the `m_sequencer` and `p_sequencer` handle types do NOT match, then the declared `p_sequencer` type is wrong and the `m_set_p_sequencer()` method executes a `` `uvm_fatal ``-command, which will abort the simulation and report the consistent error message shown in Example 14.

The purpose of this macro is to declare and qualify a consistent `p_sequencer` handle of the user-specified `SEQUENCER` *TYPE* that is passed to the macro as an input argument:

```
      SEQUENCER p_sequencer;
```

Example 15 - p-sequencer handle declaration made by the `uvm_declare_p_sequencer

After declaring the `p_sequencer` handle (initially `null`), the macro calls the `m_set_p_sequencer()` method to cast the current sequence's `m_sequencer` handle to the newly declared `p_sequencer` handle to ensure that the `p_sequencer` handle is properly vetted and set to exactly match the active `m_sequencer` handle.

Why cast `m_sequencer` to `p_sequencer`? Why not just use the `m_sequencer` handle directly? Technically, there is no need to create the `p_sequencer` handle if the verification engineer properly uses the `m_sequencer` handle and never makes a mistake. Unfortunately, erroneous handle usage leads to runtime null-handle reference errors that abort the simulation and can be difficult to debug. Part of the action performed by the `m_set_p_sequencer()` cast operation is to trap any illegal `m_sequencer/p_sequencer` assignments and report a consistent error message to help identify the problematic `p_sequencer` type declaration.

6. sequence.start(null) example

In multiple sections, we mentioned that a sequence could be started on `null` under the right conditions. In this section, we show how to properly setup sequences so that a top-level test can start a top-level sequence on `null`.

The verification academy description of virtual sequences [3] uses this basic technique. This technique is not recommended because it requires a `test_base` class to create an `init_seq()` method that uses fixed, constant paths to sequencers, which is not portable, and often missed when modifying the testbench environment.

6.1.1 seq_base

The `seq_base` class shown in Example 16 has declared a sequencer handle name `SQR`. All user sequences will extend from this `seq_base` class and inherit the `SQR` handle, which initially is not set and is `null`.

```
class seq_base extends uvm_sequence #(trans1);
  `uvm_object_utils(seq_base)

  tb_sequencer SQR;

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

Example 16 - seq.start(null) – seq_base example

6.1.2 sequence

The `write_read` sequence shown in Example 17 will start the write (`wseq`) and read (`rseq`) sequences on the inherited `SQR` handle.

```
class write_read extends seq_base;
  `uvm_object_utils(write_read)

  rand int cnt;
  constraint loop_cnt {cnt inside {[3:5]};}

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

  task body;
    write_sequence wseq = write_sequence::type_id::create("wseq");
    read_sequence  rseq =  read_sequence::type_id::create("rseq");
    //---------------------------------------------
    repeat (cnt) begin
      wseq.start(SQR);
      rseq.start(SQR);
    end
  endtask
endclass
```

Example 17 - seq.start(null) – sequence extended from seq_base example

Each sequence that extends the `seq_base` class shown in Example 16 will inherit the `tb_sequencer SQR` handle. But where is this `SQR` handle being set?

6.1.3 test_base

The `test_base` class shown in Example 18 defines an `init_seq()` method that takes as an input, a sequencer handle. The `init_seq()` method then sets the `SQR` handle to `e.agnt.sqr`

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

  env e;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = env::type_id::create("e", this);
  endfunction

  // Initialize the vseq_base handle
  function void init_seq(seq_base seq);
    seq.SQR = e.agnt.sqr;
  endfunction
endclass
```

Example 18 - seq.start(null) – test_base example

6.1.4 test

The `test1` class shown in Example 19 extends the `test_base` class from Example 18, and calls the `init_seq()` method on the `write_read seq` handle, which will set the `SEQ` handle in all derivative sequences extended from the `seq_base` class to the sequencer located at `e.agnt.sqr`.

```
class test1 extends test_base;
  `uvm_component_utils(test1)

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

  task run_phase(uvm_phase phase);
    write_read seq = write_read::type_id::create("seq");
    phase.raise_objection(this);
      if (!seq.randomize())
          `uvm_error("RAND", "Failed randomization");
      init_seq(seq);
      seq.start(null);
    phase.drop_objection(this);
  endtask
endclass
```

Example 19 - seq.start(null) – test1 extended from test_base example

> Now all subsequences started on the `SQR` handle will run on the `e.agnt.sqr tb_sequencer`.
>
> After calling `init_seq(seq)`, `test1` calls `seq.start(null)`, which sets the `m_sequencer` handle in the `write_read` sequence to `null`, but the `write_read` sequence executes its subsequences on the `SQR`-handle and not on the `m_sequencer` (`null`) handle.

7. Typical Virtual Sequence / Virtual Sequencer Technique

One very common technique for creating virtual sequences that are used to coordinate sequence activity across multiple DUT interfaces is to use a virtual sequencer, which declares and holds handles to the subsequencers that are required by the virtual sequences. Our DVCon 2016 paper described this technique in detail [1].

The term subsequencer simply refers to other agent sequencers that are controlled by virtual sequences, using the agent sequencer handles stored in an upper-level sequencer (the virtual sequencer).

The virtual sequencer example shown in Figure 5 is based off the virtual sequence example shown on Verification Academy [3], except this version of the example uses a virtual sequencer as opposed to the `init_vseq()` method described on Verification Academy.

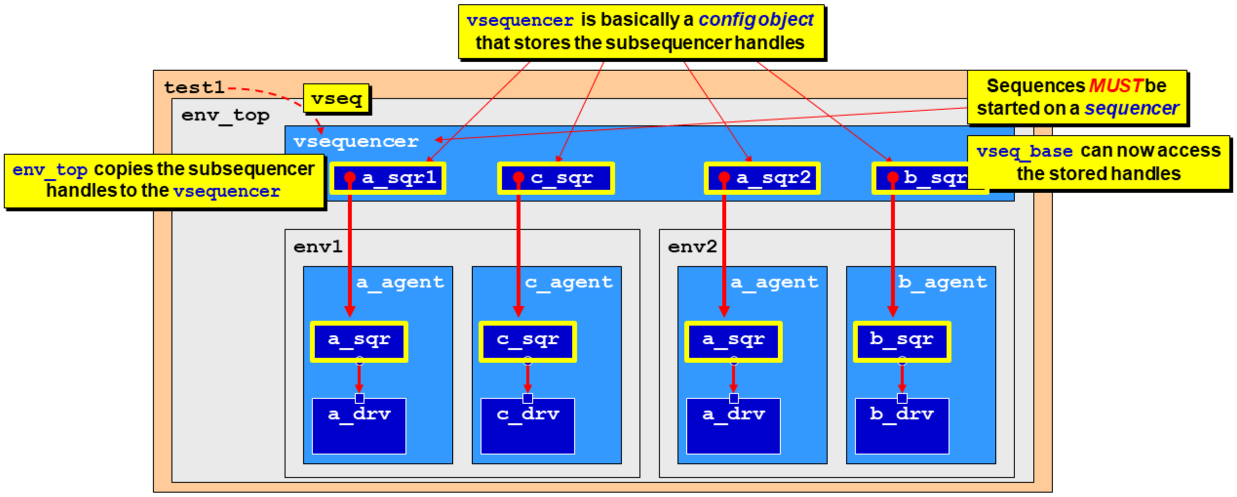


Figure 5 - Virtual sequencer example – block diagram

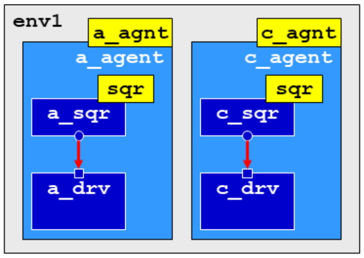
The virtual sequencer technique needs to access subsequencer handles from a central location. UVM stored variables and handles are typically kept in a config object, but sequences cannot be started on a config object, so the subsequencer handles are stored in a sequencer-type container; the virtual sequencer, as shown in Example 20.

```
class vsequencer extends uvm_sequencer;
  `uvm_component_utils(vsequencer)

  a_sequencer a1_sqr;
  a_sequencer a2_sqr;
  b_sequencer  b_sqr;
  c_sequencer  c_sqr;

  function new(string name, uvm_component parent); ...
endclass
```

Example 20 - vsequencer (virtual sequencer) class example

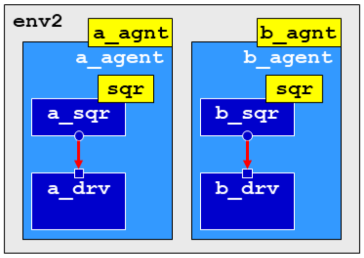
Using this technique, virtual sequences are started on the virtual sequencer, which gives them access to the stored subsequencer handles. In essence, the virtual sequencer takes the place of a config object to hold accessible subsequencer handles that can be accessed from a single location.

In this example, there are two sub-environments coded as shown in Example 21 and Example 22.



```
class env1 extends uvm_env;
  `uvm_component_utils(env1)
  a_agent a_agnt;
  c_agent c_agnt;

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

  function void build_phase(uvm_phase phase);
    a_agnt = a_agent::type_id::create("a_agnt", this);
    c_agnt = c_agent::type_id::create("c_agnt", this);
  endfunction
endclass
```

Example 21 - env1 block diagram and env1 class example



```
class env2 extends uvm_env;
  `uvm_component_utils(env2)
  b_agent b_agnt;
  a_agent a_agnt;

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

  function void build_phase(uvm_phase phase);
    b_agnt = b_agent::type_id::create("b_agnt", this);
    a_agnt = a_agent::type_id::create("a_agnt", this);
  endfunction
endclass
```

Example 22 - env2 block diagram and env2 class example

The sub-environments include two agents each and both include their own copy of an `a_agent` component.

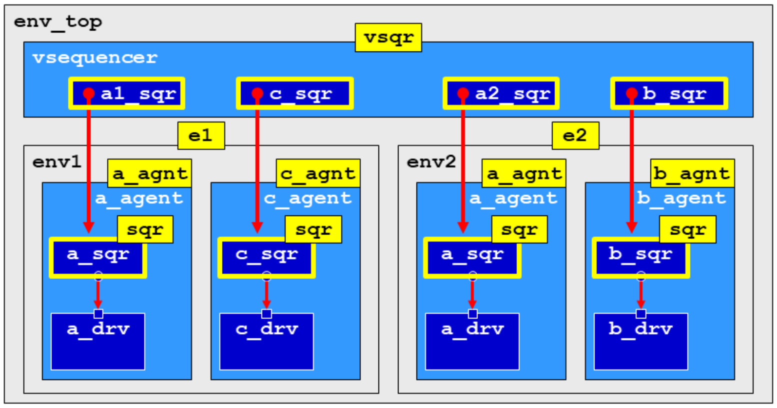


Figure 6 - env_top block diagram – connect_phase() used to copy subsequencer handles to vsequncer handles

The `env_top` class is responsible for building the `env1`, `env2` and `vsequencer` components. After the components are all created during the `build_phase()`, the `env_top` uses the `connect_phase()` to copy the component subsequencer handles to the subsequencer handles declared in the `vsequencer`, as shown in Example 23.

```
class env_top extends uvm_env;
  `uvm_component_utils(env_top)

  env1       e1;
  env2       e2;
  vsequencer vsqr;

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

  function void build_phase(uvm_phase phase);
    e1   =       env1::type_id::create("e1",   this);
    e2   =       env2::type_id::create("e2",   this);
    vsqr = vsequencer::type_id::create("vsqr", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    vsqr.a1_sqr = e1.a_agnt.sqr;
    vsqr.c_sqr  = e1.c_agnt.sqr;
    vsqr.a2_sqr = e2.a_agnt.sqr;
    vsqr.b_sqr  = e2.b_agnt.sqr;
  endfunction
endclass
```

Example 23 - env_top class that sets all the subsequencer handles in the vsequencer

The environments, all testbench subcomponents, and the virtual sequencer make up the structural portions of the virtual sequencer environment. Now the question becomes, how to pass the subsequencer handles to the virtual sequences themselves?

This is where the `` `uvm_declare_p_sequencer `` macro and `p_sequencer` handles are used.

```
class vseq_base extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(vseq_base)
  `uvm_declare_p_sequencer(vsequencer)

  a_sequencer A1;
  a_sequencer A2;
  b_sequencer B;
  c_sequencer C;

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

  task body;
    A1 = p_sequencer.a1_sqr;
    A2 = p_sequencer.a2_sqr;
    B  = p_sequencer.b_sqr;
    C  = p_sequencer.c_sqr;
  endtask
endclass
```

Example 24 - vseq_base (virtual sequence base) class using p_sequencer handles

In Example 24, the subsequencer handles are declared. Declaring subsequencer handles is easy! Now how are the handles properly set to point to the actual subsequencers? Since virtual sequences will be started on this virtual sequencer, the virtual sequence base class will have an `m_sequencer` handle that points to the `e_top.vsqr`. Using the `` `uvm_declare_p_sequencer `` macro, the `vseq_base` class has declared `vsequencer p_sequencer` and has cast `m_sequencer` to `p_sequencer` so that the `p_sequencer` handle now points to the same `vsequencer` as the `m_sequencer` handle.

The `body()` task of the `vseq_base` class, now uses the `p_sequencer` handle to copy the stored subsequencer handles to the locally declare subsequencer handles. That answers the question, "how are the handles properly set so that they point to the actual subsequencers?"

```
class vseq_A1_B_A2_A1 extends vseq_base;
  `uvm_object_utils(vseq_A1_B_A2_A1)

  function new(string name="vseq_A1_B_A2_A1");
    super.new(name);
  endfunction

  task body();
    a_seq  a = a_seq::type_id::create("a");
    b_seq  b = b_seq::type_id::create("b");
    a_seq a2 = a_seq::type_id::create("a2");
    super.body();
    a.start(A1);
    fork
       b.start(B);
      a2.start(A2);
    join
    a.start(A1);
  endtask
endclass
```

Example 25 - Virtual sequence example #1 extended from vseq_base class (vseq_A1_B_A2_A1)

```
class vseq_A1_B_C extends vseq_base;
  `uvm_object_utils(vseq_A1_B_C)

  function new(string name = "vseq_A1_B_C");
    super.new(name);
  endfunction

  task body();
    a_seq a = a_seq::type_id::create("a");
    b_seq b = b_seq::type_id::create("b");
    c_seq c = c_seq::type_id::create("c");
    super.body();
    a.start(A1);
    fork
      b.start(B);
      c.start(C);
    join
  endtask
endclass
```

Example 26 - Virtual sequence example #2 extended from vseq_base class (vseq_A1_B_C)

The `vseq_base` class had done the hard work of retrieving the actual subsequencer handles. Now the virtual sequences[1] can extend the `vseq_base` class, inherit the declared subsequencer handles, and call the `super.body()` method to set the inherited handles in each virtual sequence class, as shown in Example 25 and Example 26.

This is a common technique that shows that a sequence base class can use the `` `uvm_declare_p_sequencer `` macro and `p_sequencer` handle to retrieve required testbench component hierarchy information to be used for coordinated virtual sequence testing.

All of these `p_sequencer` coding gymnastics are required because handles stored as resources in the `uvm_resource_pool` cannot be retrieved directly into the virtual sequence base class because the `uvm_config_db` API requires `get` commands to be called using a two-argument `cntxt` (component class handle), `"inst_name"` (string) that when combined must point to an actual component in the testbench hierarchy, and sequences to do not have testbench hierarchy-paths. Is there a better way?

8. Sequences can access resources directly

In our 2023 DVCon paper [2], we detailed how most UVM Verification engineers have been using the wrong UVM resources API for more than 10 years.

There is no `uvm_config_db`! `uvm_config_db` is the secondary API used to store and retrieve resources in the `uvm_resource_pool`, and this API has several restrictions that were included in the API to be backward compatible with obsolete OVM `set_config`/`get_config` commands.

`uvm_resource_db` is the primary API to store and retrieve resources in the `uvm_resource_pool`, and this API does not require the complex two-argument `cntxt`-component-handle, `"inst_name"` -string that must match an actual component path in the UVM testbench.

Engineers are encouraged to read our DVCon 2023 paper [2] to learn the proper way to use UVM resources. The `uvm_resource_db` is a much easier and more powerful way to interact with the `uvm_resource_pool`.

Using the `uvm_resource_db` API, environments can store subsequencer handles as resources that the virtual sequence base class can directly retrieve, also using the `uvm_resource_db` API.

---

[1] "**virtual sequence**: A conceptual term for a **sequence** that controls the execution of **sequence**s on other **sequencer**s" [4]. A virtual sequence coordinates the execution of other sequences on multiple subsequencers.

9. Summary & Conclusions

Every sequence has an `m_sequencer` handle. The `m_sequencer` handle is set to the path where the sequence runs. The `m_sequencer` handle is most often set using the command `sequence.start(`***full_path_to_sequencer***`)`.

The `p_sequencer` handle does not exist in sequences unless explicitly declared or unless the `` `uvm_declare_p_sequencer `` macro is used. Engineers use the `` `uvm_declare_p_sequencer `` macro and `p_sequencer` handle most often with virtual sequencers to ensure that a `vseq_base` class has been started on the correct virtual sequencer.

Engineers have used the `p_sequencer` handle to retrieve information that is stored in a sequencer to pass required testbench information to a sequence. This is unnecessary and has become a UVM poor-programming practice because engineers did not know there was an easier and better way to pass information to sequences.

The `uvm_resource_db` API can be used to store and retrieve resources in the `uvm_resource_pool`, which can be accessed from anywhere in a UVM testbench, including sequences; hence, there is no need to use the `` `uvm_declare_p_sequencer `` macro and `p_sequencer` handle.

Engineers are encouraged to read the Cummings, Glasser, Chambers, DVCon 2023 paper [2] to learn the proper way to use UVM resources. The `uvm_resource_db` is a much easier and more powerful way to interact with the `uvm_resource_pool`.

10. Acknowledgements

I would like to thank my friend and colleague Jeff Montesano for his review and valuable feedback on both this paper and the accompanying presentation slides.

11. References

[1] Clifford E. Cummings, Janick Bergeron, *"Using UVM Virtual Sequencers & Virtual Sequences,"* DVCon 2016 Proceedings, also available at: www.sunburst-design.com/papers/CummingsDVCon2016_Vsequencers.pdf

[2] Clifford E. Cummings, Mark Glasser, Heath Chambers, *"The Untapped Power of UVM Resources and Why Engineers Should Use the uvm_resource_db API,"* DVCon 2023 Proceedings, also available at: www.sunburst-design.com/papers/CummingsDVCon2023_uvm_resource_db_API.pdf

[3] Tom Fitzpatrick, *"Advanced UVM Layered Sequences,"* Verification Academy – https://verificationacademy.com

[4] Universal Verification Methodology (UVM) 1.2 Class Reference – June 2014

12. Author & Contact Information

Sunburst Design World Class Training

Sunburst Design merged with Paradigm Works in February of 2020 and still provides World Class SystemVerilog, Synthesis and UVM Verification training. For more information about training, contact Cliff Cummings (cliffc@sunburst-design.com) or Michael Hoyt (michael.hoyt@paradigm-works.com)

Paradigm Works Expert Design Services, Verification Services & IP

Paradigm Works provides expert services in Semiconductor Architecture, Design, Synthesis, Functional Verification, and DFT. For more information about Paradigm Works services, contact Michael Hoyt at michael.hoyt@paradigm-works.com

Cliff Cummings is Vice President of Training at Paradigm Works and Founder of Sunburst Design. Paradigm Works and Sunburst Design merged in February of 2020. Mr. Cummings has 42 years of ASIC, FPGA and system design experience and 32 years of combined Verilog, SystemVerilog, UVM verification, synthesis, and methodology training experience.

Mr. Cummings has presented more than 100 SystemVerilog seminars and training classes in the past 20 years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings participated on every IEEE & Accellera SystemVerilog, SystemVerilog Synthesis, SystemVerilog committee from 1994-2012, and has presented more than 50 papers on SystemVerilog & SystemVerilog related design, synthesis, and verification techniques.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

Email address: cliffc@sunburst-design.com

Last Updated: April 2024