UVM Co-Emulation Utility Library

The UVM Co-Emulation Utility Library is a collection of code utilities which perform common tasks needed when running a UVM testbench with a Veloce emulator. The utilities are freely available and modifiable and come as is with no warranty. The Utility Library this far includes:

- a Clock Utility
- two Reset Utilities
- a Memory Load Utility
- a UVM Register Backdoor Access File Utility

Utility Architecture

The utilities follow the BFM-Proxy Pair Pattern at their heart and the Utility Pattern directly. These two patterns are part of the Verification Academy Patterns Library which is freely available and contributable. This implies that the utilities having code which is instantiated and used in an HVL top (simulator code) and code which is instantiated and used in a separate HDL top (emulator code). The HVL code is SystemVerilog class-based which inherits from `uvm_object`. This is the Utility Proxy code that the rest of the testbench can use to interact with HDL code. The HDL code is a SystemVerilog interface that the HVL code will communicate with through a virtual interface handle. The HDL code may further contain a back-pointer to the Utility Proxy. This HDL code is called the Utility Interface and communication between the Utility Proxy and Utility Interface is done via function and task calls. Figure 1 shows a basic diagram of this architecture.

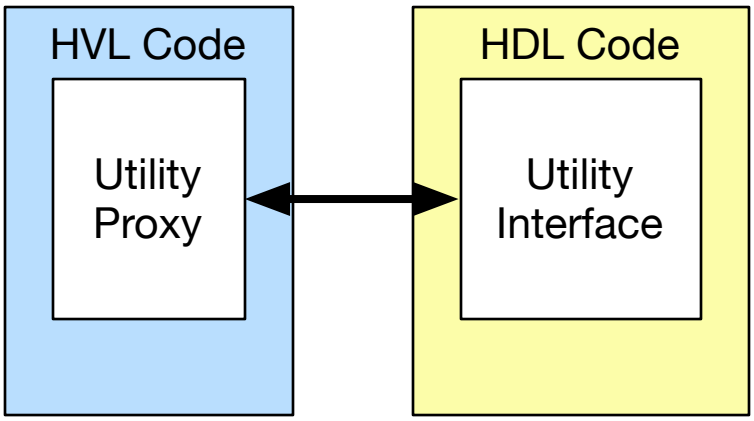

Figure 1 Utility Architecture

Digging deeper into the Utility Proxy side reveals more detailed information. As mentioned previously, the Utility Proxy inherits from `uvm_object` which allows for seamless integration into a UVM environment because the proxy objects can be registered with the UVM Factory and passed through the UVM Configuration Database, etc. A hierarchy of classes are defined with a base class defined first which inherits from `uvm_object`. The base class is a non-parameterized class which defines the API as a set of virtual functions and tasks. The concrete

proxy class inherits from the proxy base class and implements the API methods. The concrete proxy class also may have parameters to declare a correspondingly parameterized virtual interface handle of a parameterized Utility Interface. The Utility Interface is often parameterized due to emulation requirements. Additionally, there may be multiple concrete proxy classes defined which all implement the same API such as in the Reset Utility. Figure 2 shows this inheritance structure.

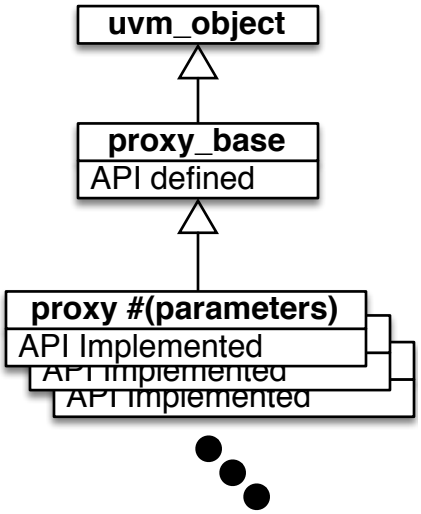

Figure 2 Utility Proxy Inheritance

This inheritance structure exists to ease usage of the utilities in a UVM testbench. Because concrete proxy classes are parameterized to enable connection to the parameterized Utility Interfaces, every time a concrete proxy class is used the parameters must also be present. To alleviate this parameter proliferation complication as much as possible, a `proxy_base` class handle should be used throughout the testbench other than in the top level test where the concrete proxy class can be created with its parameters and then placed into the configuration database using a `proxy_base` class handle. Because the API is defined in the `proxy_base` class using virtual methods, polymorphism will handle calling the correct concrete proxy methods even when called through a `proxy_base` class handle. This results in code as follows:

```
00 class base_test extends uvm_test;
01   `uvm_component_utils(base_test)
02
03   proxy #(shared_params_pkg::PARAM1) proxyA;
04
05   function void build_phase(uvm_phase phase);
06     virtual proxy_bfm #(shared_params_pkg::PARAM1) tmp_proxy_bfm;
07
08     proxyA = proxy#(shared_params_pkg::PARAM1)::type_id::create("proxyA");
09
10     //Set the bfm handle in the clk_ctrl
11     if (!uvm_config_db#(virtual proxy_bfm #(shared_params_pkg::PARAM1))
12                         ::get(this, "", "proxy_if_h", tmp_proxy_bfm)) begin
```

```
13        `uvm_fatal(report_id, "Could not get the Proxy BFM Interface")
14      end
15      proxyA.set_bfm(tmp_proxy_bfm);
16
17      uvm_config_db #(proxy_base)::set(this, "*", "proxyA", proxyA);
18    endfunction : build_phase
19
20 endclass : base_test
```

- Line 3 defines the concrete proxy handle including any parameters.
- Line 6 uses the same parameters to define a temporary virtual interface handle.
- Line 8 constructs the concrete proxy class using the UVM Factory. Notice again that the parameters are used.
- Lines 11 & 12 access the parameterized virtual interface handle which points to the Utility Interface BFM.
- Line 15 calls the `set_bfm()` function to set the virtual interface handle in the proxy class. This also sets any back-pointer values in the Utility Interface BFM if required.
- Line 17 uses the `proxy_base` class as the type to put the concrete proxy into the UVM Configuration Database to allow other parts of the testbench to access the proxy and utilize the utility's functionality.

Clock Utility

The clock utility follows the architecture described above. The Utility Interface BFM code of the clock utility contains two parameters which define the starting phase offset for the clock and the initial half period of the clock. The initial clock half period is set to 2000 ps. The phase offset is set by default to be equal to the clock half period. The phase offset cannot be 0 per Veloce requirements. The BFM has a single output: the generated clock.

```
interface clock_bfm #(int unsigned INIT_CLOCK_HALF_PERIOD = 2000,
                      int unsigned PHASE_OFFSET_IN_PS = INIT_CLOCK_HALF_PERIOD) (
   output logic clock
  );
```

The proxy base class is called `clock_ctrl_base` and defines the following API:

```
function update_half_period_in_ps ( int unsigned half_period );
function update_clock_enable      ( bit enable );
task     wait_clocks              ( int unsigned numClocks = 1 );
```

- The `update_half_period_in_ps()` function allows the user of the proxy to change the half period of the clock that is generated. This function can be called at any point in the simulation although it should be called only when no outstanding `wait_clocks()` are pending. If called while one or more `wait_clocks()` calls are pending, the exact return time of each of those calls is not predictable. When `update_half_period_in_ps()` is called, the period will be updated on the next rising edge of the generated clock. If the clock is static, then the UVM build_phase() is a

good place to call this function. If the clock changes period as time advances, then call this function when needed.

- The `update_clock_enable()` function allows for gating of the clock that is generated.
- The `wait_clocks()` task allows for time to be advanced by the proxy by waiting for the specified number of clock edges in the BFM. A queuing structure is implemented as part of the proxy to enable multiple calls to the `wait_clocks()` task to be made from anywhere within the testbench and from any number of threads in the testbench. Even with this capability, it is recommended to use this capability sparingly for performance considerations.

The proxy class is called `clock_ctrl` and takes the same parameter values for the phase offset and initial clock half period. In addition to the API functions defined above, it also defines a `set_bfm()` function used to pass the `clock_bfm` virtual interface handle in to the `clock_ctrl` object after it has been created.
Finally, all of the proxy side code is compiled into the `clock_pkg`. Users should wildcard import `clock_pkg` where needed.

Instantiation and Usage

The clock utility requires the `clock_bfm` to be instantiated and if control is needed, the `clock_ctrl` to be instantiated. If only a `clock_bfm` is used, then it will output a fixed frequency clock for the duration of the simulation and doesn't require registration into the `uvm_config_db`. The `clock_bfm` should be instantiated in the HDL top (the top level that will reside on the Veloce emulator).

Note: Both parameters are being set in this example. Only the half period is required.

```
module hdl_top();
  // pragma attribute hdl_top partition_module_xrtl

  // Wires
  logic       clk;

  // Declare the pin interfaces
  clock_bfm #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
              shared_params_pkg::CLK_PHASE_OFFSET_IN_PS) clk_if_h(clk);


  // DUT instantiate, etc.
  ...


  initial begin //tbx vif_binding_block
    import uvm_pkg::uvm_config_db;
      // Set the interface into the DB for retrieval by the TB
      //Clock & Reset
      uvm_config_db #(
           virtual clock_bfm #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
                               shared_params_pkg::CLK_PHASE_OFFSET_IN_PS)
                     )::set(null, "", "clk_if_h", clk_if_h);
  end

endmodule : hdl_top
```

The `clock_ctrl` object should be created in the `base_test` in the UVM test hierarchy where the BFM handle will be set and then placed into the `uvm_config_db` using the `clock_ctrl_base` as the type to allow for unparameterized access to clocks where needed in the rest of the testbench.

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

  //variable: clk_ctrl
  //Clock Proxy Object used to control the Clock
  // Must be extended clock_ctrl object and not clock_ctrl_base because
  // bfm is set here.  Usage elsewhere in testbench can just use a
  // clock_ctrl_base handle.
  clock_ctrl #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
               shared_params_pkg::CLK_PHASE_OFFSET_IN_PS) clk_ctrl;

  ...

  function void build_phase(uvm_phase phase);
    //Temp handle to clock_bfm
    virtual clock_bfm #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
                        shared_params_pkg::CLK_PHASE_OFFSET_IN_PS) clk_bfm;

    // Create the clock ctrl
    clk_ctrl =
      clock_ctrl #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
                   shared_params_pkg::CLK_PHASE_OFFSET_IN_PS)::
                                              type_id::create("clk_ctrl");
```

```
    //Set the bfm handle in the clk_ctrl
    if (!uvm_config_db #(
          virtual clock_bfm #(shared_params_pkg::CLK_INIT_HALF_PERIOD_IN_PS,
                              shared_params_pkg::CLK_PHASE_OFFSET_IN_PS) )::
                 get(this, "", "clk_if_h", clk_bfm)) begin
      `uvm_fatal(report_id, "Could not get the Clock BFM Interface")
    end
    clk_ctrl.set_bfm(clk_bfm);

    //Place the clk_ctrl in the uvm_config_db
    // The clock_ctrl_base handle is used here because it contains the API
    // while not requiring parameters to be passed to the rest of the TB
    uvm_config_db #(clock_ctrl_base)::set(this, "*", "clk_ctrl", clk_ctrl);

  endfunction : build_phase

  ...

endclass : base_test
```

Now with this one time setup done, any object in the UVM testbench can get access to the clock_ctrl object and call wait_clocks(), enable or disable the clock, etc. by simply doing a uvm_config_db::get() call.
First a handle will need to be created of type clock_ctrl_base.

```
clock_ctrl_base clk_ctrl;
```

For uvm_components such as drivers, scoreboards, etc. the call to the uvm_config_db would be placed in the build_phase or connect_phase and would look like this:

```
uvm_config_db #(clock_ctrl_base)::get(this, "", "clk_ctrl", clk_ctrl);
```

Sequences would place the call to uvm_config_db::get() in the beginning of the body() task or in the pre_body() task and look like this:

```
uvm_config_db #(clock_ctrl_base)::get(m_sequencer, "", "clk_ctrl", clk_ctrl);
```

Finally, remember that the rules of the uvm_config_db still apply. If a uvm_config_db::set() is performed with `this` as the first argument, then only object below `this` in the UVM hierarchy will be able to access the clock control handle placed in the uvm_config_db.

Reset Utilities

The reset utilities are a family of utilities which all utilize the same API. There are two types of reset utility: sync reset and async reset. Let's review their common API first, and then inspect the differences between the two.
The proxy base class is called `reset_ctrl_base` and defines the following API:

```
task     wait_reset_assertion   ();
task     wait_reset_asserted    ();
task     wait_reset_deassertion ();
task     wait_reset_deasserted  ();
function toggle_reset           ( int idle_cycles = -1,
                                  int num_clks_active = -1 );
function configure              ( int idle_cycles = -1,
                                  int num_clks_active = -1 );
function set_manual_control     ( bit mc );0
function assert_reset           ();
function deassert_reset         ();
```

- The `wait_reset_assertion()` task waits until the next active reset edge. This means that if the reset is currently active, this task does not return until reset is asserted again.
- The `wait_reset_asserted()` task returns immediately if reset is currently active or else it waits until the next active reset edge.
- The `wait_reset_deassertion()` task waits until the next inactive reset edge. This means that if the reset is currently inactive, this task does not return until reset is de-asserted again.
- The `wait_reset_deasserted()` task returns immediately if reset is currently inactive or else it waits until the next inactive reset edge.
- The function `toggle_reset()` causes a reset to be generated. If no arguments are supplied, the function uses the currently configured values for `idle_cycles` before reset assertion and for `num_clks_active` to control how many clock cycles the reset is active. If arguments are supplied, then the configured values are overwritten and used.
- The `configure()` function changes the values for `idle_cycles` and `num_clks_active` if supplied.
- The function `set_manual_control()` controls whether the reset generator will automatically generate a reset at time 0 or not. When configured for manual control, the reset is controlled via the `assert_reset()` and `deassert_reset()` functions.
- The `assert_reset()` function manually asserts reset when in manual control mode.
- The `deassert_reset()` function manually de-asserts reset when in manual control mode.

All of the proxy side code is compiled into the `reset_pkg`. Users should wildcard import `reset_pkg` where needed.

Synchronous Reset Utility

The sync reset utility utilizes the architecture and API described above. The Utility Interface BFM code of the sync reset utility requires three parameters. The first parameter called `RESET_POLARITY` controls whether an active reset is high or low. The second parameter called `INITIAL_IDLE_CYCLES` sets the initial configuration for the number of idle cycles before a reset is asserted from time 0 and then from when the `toggle_reset()` function is

called. The third parameter called RESET_ACTIVE_CYCLES sets the initial configuration for the number of active cycles before a reset is de-asserted from an assertion. This value is also stored as the initial configuration which is used when toggle_reset() is called unless overridden.
The BFM has a single input and output, namely the clock which is used to synchronize reset and the generated reset value, respectively.

```
interface sync_reset_bfm #(bit RESET_POLARITY = 1,
                           int INITIAL_IDLE_CYCLES = 0,
                           int RESET_ACTIVE_CYCLES = 10) (
   input bit    clock,
   output logic reset
  );
```

The proxy class is called sync_reset_ctrl and takes the same parameter values as the sync_reset_bfm. In addition to the API functions defined above, it also defines a set_bfm() function used to pass the sync_reset_bfm virtual interface handle in to the sync_reset_ctrl object after it has been created.

Asynchronous Reset Utility

The async reset utility utilizes the architecture and API described above. The Utility Interface BFM code of the async reset utility requires three parameters. The first parameter called RESET_POLARITY controls whether an active reset is high or low. The second parameter called INITIAL_IDLE_TIME_IN_PS sets the amount of time in picoseconds before reset is asserted asynchronously from time 0. The third parameter called RESET_ACTIVE_TIME_IN_PS and controls the amount of time in picoseconds that the reset is active after it has been asserted. The function toggle_reset() does not use these values because it only generate synchronous resets. Even though this is an async reset utility, the async reset can only be generated once from time 0. Any additional resets initiated after time 0 are synchronous to the input clock. This is a limitation imposed by the current capabilities of Veloce.
The BFM has a single input and output, namely the clock which is used to synchronize any reset generated after the initial async reset and the generated reset value, respectively.

```
interface async_reset_bfm #(bit RESET_POLARITY = 1,
                            int INITIAL_IDLE_TIME_IN_PS = 1000,
                            int RESET_ACTIVE_TIME_IN_PS = 20000) (
   input bit    clock,
   output logic reset
  );
```

The proxy class is called async_reset_ctrl and takes the same parameter values as the async_reset_bfm. In addition to the API functions defined above, it also defines a set_bfm() function used to pass the async_reset_bfm virtual interface handle in to the async_reset_ctrl object after it has been created.

Instantiation and Usage

The reset utilities follow the same pattern of instantiation and usage as the clock utility. Of course either a `sync_reset_bfm` or an `async_reset_bfm` would be instantiated instead of a `clock_bfm`. Also, a `sync_reset_ctrl` or an `async_reset_ctrl` would be created in the `base_test`. Finally, regardless of whether the reset is synchronous or asynchronous, a `reset_ctrl_base` type would be used when placing the proxy object into the `uvm_config_db` for other testbench object usage.

Memory Load Utility

The memory load utility provides a single function used to load a memory from a file when called from testbench code. The single function is a DPI function which calls C code. The C code is customized whether the user is running on Questa or running on Veloce. A Makefile included in the *memload* directory shows the proper compilation procedure required.
The API function provided is as follows:

```
import "DPI-C" function void memLoad ( input bit[128*8-1:0]  fname_in,
                                       input bit [128*8-1:0] path_in,
                                       input int             startAddr,
                                       input int             endAddr,
                                       input bit [3*8-1:0]   format_in);
```

The first argument called `fname_in` is the file name of the file which contains the hex or binary formatted memory data to be loaded. The second argument called `path_in` is the path to the file. The third argument called `startAddr` is the starting address where memory should be loaded from the file. The fourth argument called `endAddr` is the ending address where memory should be loaded from the file. The final argument called `format_in` is either "hex" or "bin" to specify if the file is formatted with hexadecimal data or binary data.
This function is contained in a package called `memload_pkg` which should be imported where needed.

Access Statistics Collection

The clock and reset utilities have an additional capability to capture statistics, disabled by default. When enabled, the number of calls to the API methods which instigate action are recorded. For the clock utility, calls to `wait_clocks()` are recorded. For the reset utilities, calls to `toggle_reset()`, `assert_reset()` and `deassert_reset()` are recorded.
The following functions are used with this capability:

```
static function set_stat_collection ( input bit val );

static function print_stats         ();
```

- The `set_stat_collection()` function enables or disables statistics collection. This function should be called before any time consuming phases in UVM is started.

- The `print_stats()` function displays the statistics results recorded so far. This function is usually called in the `report_phase()` of the top level UVM test after all time consuming phases have run.

Notice that both functions are static functions. This allows for collection of statistics across all instances of a specific type of utility. This also means that invocation of the functions is typically done in this form:

```
clock_ctrl_base::set_stat_collection(1);
reset_ctrl_base::print_stat();
```

UVM Register Backdoor Access File Utility

The UVM register backdoor access file utility is a little different from the other utilities. It is meant to be run before a simulation run to create a file for Questa and/or a file for Veloce to ensure that registers with a backdoor access path specified are accessible. It can also create a TCL file which can be used to initialize the Visualizer register view.

For Questa, the utility creates a `reg_acc.f` file which contains `+acc=rn+<path/to/reg>` lines. These lines instruct Questa's optimization engine to preserve the registers and the paths to the registers when performing optimization.

For Veloce, the utility creates a `forcesetget_nets.sigs` file which contains `<path.to.reg>` lines. The file is then used by adding

```
rtlc -forceset_nets_file forcesetget_nets.sigs   #Allows for forcing of regs
rtlc -get_nets_file forcesetget_nets.sigs        #Allows for getting of regs
```

to the `veloce.config` file. By adding these two lines, Veloce builds the infrastructure that is needed to allow for UVM backdoor register accesses to succeed.

For Visualizer, the utility creates a `vis_regs_tcl` file which can be used following these three steps:

1) Open Visualizer
2) Execute "do vis_regs.tcl" at the Visualizer command prompt
3) Go to View -> Register Viewer -> UVM Registers to open the register window

To use the utility, a simple module should be created which has a function with instantiates the user's register model, builds it and then passes it to one or more calls of the `create_acc_file()` function which is defined in the `print_uvm_reg_acc_pkg`. The code below illustrates that process. Notice the code in blue. In many cases, this is the only code that will need to be changed to make this work.

```
module create_access_file ();
  function automatic bit auto_gen();
    dut_registers_pkg::reg_block user_block = new();
    user_block.build();
    print_uvm_reg_acc_pkg::create_acc_file(user_block,
                                           print_uvm_reg_acc_pkg::QUESTA);
    print_uvm_reg_acc_pkg::create_acc_file(user_block,
                                           print_uvm_reg_acc_pkg::VELOCE);
    print_uvm_reg_acc_pkg::create_acc_file(user_block,
                                          print_uvm_reg_acc_pkg::VISUALIZER);

    return 1;
  endfunction : auto_gen

  bit auto_generate = auto_gen();

endmodule :  create_access_file
```

The `create_acc_file()` function takes two arguments:

```
function automatic void create_acc_file(uvm_reg_block reg_model,
                                        file_mode_e mode = VELOCE);
```

The first argument called `reg_model` is used to pass in a handle to the constructed UVM register model that contains backdoor access paths. The second argument is called `mode` and is used to control whether the output file will be for Questa or Veloce. This is an enumerated type with either `QUESTA`, `VELOCE`, or `VISUALIZER` as acceptable values.

Further Work

We would like to continue to expand the UVM Co-Emulation Utility Library. If you have any suggestions for additions or improvements, please let your Mentor AE know.