Taming a Complex UVM Environment

Manjunath Shetty, *Sr. Staff Engineer*
Ramamurthy Gorti, *Sr. Manager*
Broadcom Corporation

Abstract- **Setting up an effective UVM-based verification environment can be challenging due to various reasons. This paper discusses an approach that can be used to improve controllability and scalability of tests while maximizing the effect of constrained randomness and reusability in any complex UVM-based verification environment. This paper also discusses the advantages and challenges of using this approach in the verification of an xHCI-based USB 3.0 host controller.**

I. Introduction

Most UVM-based verification projects start with tests targeting specific DUT features of interest, with some tests exercising basic design features and others more complex features. Soon, the project will have thousands of such directed tests [1]. Though these directed scenarios help in cleaning bring-up related bugs in the DUT, maintaining all tests and running them periodically becomes challenging as the project matures. As new features are added to the DUT, testing new features interleaved with all existing verified features becomes a major challenge.

A. VERIF_MODE

The proposed approach is to have multiple runtime switches, termed as "verif_modes", corresponding to every design feature. Each verif_mode enables a sequence or a set of sequences to create the scenarios required to verify the design feature and also optionally enables checks related to the feature. The verif_modes can be enabled to test directed scenarios or concurrently with other verif_modes to create interleaved constrained random scenarios.

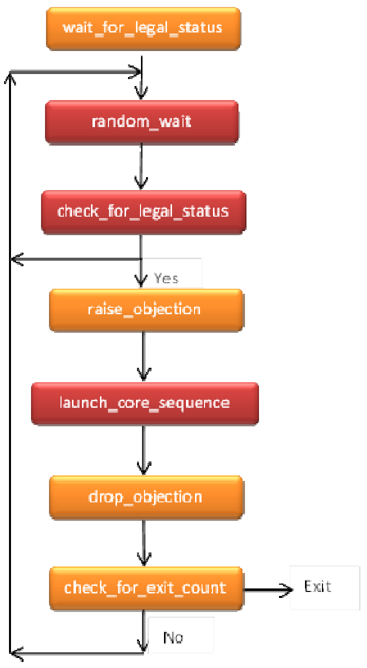

Figure 1: General VERIF_MODE Flow Diagram

Figure 1 shows a general flow diagram used for implementing verif_modes. Appendix A shows pseudo-code for an example verif_mode. The first step is to identify different scenarios/features to be tested from the testplan. The section below discusses a few verif_modes used for an xHCI-based USB 3.0 host controller. The core_sequence, which exercises the minimal iteration of the identified scenario, needs to be implemented. Appendix B shows pseudo-code for an example core_sequence of a verif_mode. The verif_mode sequence following the generic flow diagram in Figure 1 needs to be implemented. Using bench_status from the monitor, every verif_mode sequence waits for some legal state during which it can exercise the scenario. The sequence waits for a random time that helps in creating corner cases. The sequence then again checks for a legal condition for exercising the verif_mode. The verif_mode sequence then launches the core_sequence. The verif_mode sequence could, optionally, raise/drop an objection before/after running the core_sequence. There is some background activities related to verif_modes for which an objection should be omitted. After running the core_sequence, the verif_mode sequence could either repeat the iteration depending on the intended number of times the verif_mode needs to be exercised in the test.

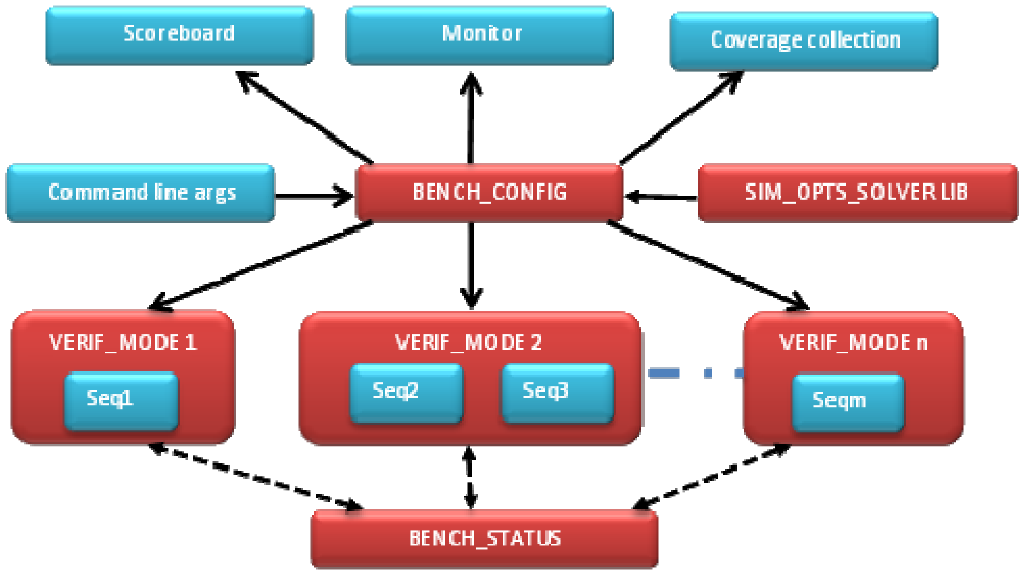

Figure 2: Verification Environment Block Diagram using VERIF_MODES

A singleton class called "bench_config" is used to control verif_modes. The bench_config class provides a centralized way to control all of the simulation options and their usage throughout the environment. Each verif_mode in bench_config class can be configured using command line options or using "sim_opts_solver" class. The uvm_cmdline_processor class can be used to set each verif_mode to desired values from simulation options provided by the user. For an automated way of controlling the verif_modes, the "sim_opts_solver" class is used. The sim_opts_solver class consists of constraints to configure all the verif_modes in different combinations to create interesting legal corner cases and to take care of any spatial and/or temporal dependencies. One can have a library of such sim_opts_solvers targeting single/multiple features in the design while enabling or disabling other background verif_modes. The sim_opts_solver also provides a way to deal with mutually exclusive and dependent verif_modes. Appendix C shows an example constraint from sim_opts_solver class used to handle few mutual dependent verif_modes. Figure 2 shows bench_config and sim_opts_solvers used to control verif_modes. Any specific verif_mode could also be set from simulation options using uvm_cmdline_processor. The verif_mode configured using command line option takes precedence over randomized value from sim_opts_solver class. This also provides the option of disabling scenarios when needed, either due to a known design issue or when a verif_mode sequence itself has issues. It results in a drastic reduction in time required for signoff regressions. This approach allows us to run as many random tests in regression as time permits, including all the scenarios of interest.

Figure 2 also shows the usage of the "bench_status" class. This is a data structure maintained in monitor and is used by all verif_mode sequences to check and update the status of the bench and DUT at any point in time during the run_phase of the test. Each verif_mode uses this during check_for_legal_status and also updates it according the verif_mode during the running of core_sequence. Appendix D shows bench_status class used for USB3.0 host controller.

Monitor can optionally maintain a data structure for each verif_mode indicating the number of times the expected scenario was hit. This can be used during the check_phase() of the test to make sure all the intended scenarios were hit during the test, at least once. It can also be used during the report_phase() to print statistics of various verif_modes enabled in the test and the number of times each scenario was hit during the test. Appendix F shows excerpt from the log file showing typical information printed during a report_phase() of a test.

B. xHCI-based USB 3.0 host controller

The host controller is the specific hardware implementation of the host controller architecture. There is one host controller specification for the USB 3.0 host controller, which enables support for Low-, Full-, High- and SuperSpeed devices. The interface presented by the xHC to the system is referred to as the eXtensible Host Controller Interface or the xHCI. The USB3.0 Host controller implements the serial interface for USB3.0 and the xHCI interface for communicating with host drivers. The host controller has many features like plug-n-play of devices, backward compatibility with USB2.0, Power Management, Virtual Memory, fine-grain scatter/gather for data transfers etc. Rest of the paper discusses use of verif_modes to effectively verify the host controller.

C. Template use cases of VERIF_MODES:

Depending on the DUT, different types of verif_modes need to be identified. Detailed review of testplan is required to identify various verif_modes and to categorize them as following. Here we discuss various verif_modes used for an xHCI-based USB 3.0 host controller. Appendix E shows excerpt from log file printing all the verif_mode details from the bench_config class used for the test which can be used for post analysis of the test. Following categorization can be used as a template to create verif_modes for any UVM-based verification environment.

1) **General verif_modes**: These are the verif_modes that are needed for all tests. Most of these are used to pass a particular runtime argument to the bench.
    a. **Controlling the speed**: BENCH_PORT_SPEED is used to specify the speed of the device model connected to each of the downstream ports. RANDOM_SPEED_SELECT_INCLUDE can be used

if the test writer wants the bench to select the speed of the device model from a provided list of speeds. As per USB3.0 specification, 4 is used for Super Speed, 3 for High Speed, 1 for Full Speed, and 2 for Low Speed. The number of downstream ports is controlled using defines from the DUT's configuration file. Example usage: In a 2port test, the test writer could use "+BENCH_PORT_SPEED=34" to have SuperSpeed device model connected on first port and HighSpeed device model connected on the second. The test write could also use "+RANDOM_SPEED_SELECT_INCLUDE=4321" to have bench select any random speed for both the ports

b. **Controlling the device model configuration**: DEVICE_ATTRIBUTE_LIBRARY_SELECT can be used to have the bench select the device configuration from a library of different device configurations maintained in the bench. DEN_PS_USB_EP_DESCR_FACTORY is used to specify the constraints used to calculate the endpoint parameters. The constraints are to set MaxBurstSize, Mult, MaxPktSize, and MaxESIT parameters of all the endpoints without violating the bandwidth requirement.

c. **Controlling the quantity of traffic**: DMVC_STOP_AFTER_N_TDS_PER_EP specifies the number of transfer_descriptors needed to be enqueued by the bench for each of the configured endpoints. The bench maintains a pending objection until all the expected traffic on all the configured endpoints is complete

d. **Controlling the type of traffic**: TD_FACTORY specifies the exact transfer_descriptor type to be used for all the configured endpoints in the test. TD_FACTORY_LIBRARY_SELECT can have the bench select random transfer_descriptor types from a library of transfer_descriptors maintained in the bench. DMVC_ENQUEUE_TD_DELAY_FACTORY can be used to specify a range of delays to be used between consecutive enqueue of transfer_descriptors onto the transfer_ring. TR_RING_FACTORY_RANDOM_SELECT can be used to have the bench select a random transfer_ring from a library of different transfer_ring types maintained in the bench

e. **Controlling external latencies**: INT_LATENCY_RANDOM_SELECT can be used to specify the range of latency to be used by the bench before processing an interrupt from the controller, emulating software behavior. AXI_SLAVE_CFG_RANDOM_SELECT can be used to specify the type of latencies to be used for AXI traffic. The bench maintains various types of constraints for latencies like low latency, high latency, and realistic latency.

2) **Special features verif_modes:** These are the verif_modes identified from the specification targeting a specific scenario and needs a sequence implemented as per Figure 1.

   a. **Device disconnect**: The USB specification allows the device to be unplugged and replugged at any time. NUM_DISCONNECT_CONNECT is used to specify the number of times the device model is disconnected and reconnected during the test. A non-zero NUM_DISCONNECT_CONNECT triggers sequences both on the device model side and the controller driver side to initiate device disconnect and handling. On the device model side, the sequence initiates the device disconnect and changes the severity of all the possible false errors that can occur during disconnect to warnings. During reconnection, the sequence initiates reconnection from the device model and also reverts back the severity of the error messages. On the controller driver side, any device disconnect indicated by the controller initiates a disconnect routine sequence as mentioned in the xHCI specification. STOP_ALL_EP_BEFORE_DISABLE_SLOT can be used to have the bench stop all the endpoints on a device before disabling the slot during a disconnect handling routine. The sequence also cleans up all the data structures maintained for the port and drops all the objections pending for the port after successful disconnection. DISCONNECT_ON_LNK_NOT_U0 can be used to make the bench initiate disconnect when the LTSSM link state of the downstream port is any state other than U0. This helps in hitting corner cases during LTSSM state machine testing. Monitor maintains a variable indicating the number of successful disconnects during the test on a particular port.

b. **Port Reset**: The USB specification provides two types of resets for the downstream ports:Hot Reset and Warm Reset. NUM_PORT_RESET is used to specify the number of times the controller is made to issue resets to the device models. PORT_RESET_TYPE can be used to specify either Hot or Warm Reset. A non-zero NUM_PORT_RESET launches sequences on the controller driver side and device model side. On the controller driver side, the sequence initiates the port reset and reduces the severity of all the false error messages, clears all the internal queues, and drops all pending objections. After the port reset, the severity of all the error messages are reverted back. On the device model side, the sequence expects a Hot Reset or Warm Reset, depending on PORT_RESET_TYPE and the current status of the port. It changes the severity of all the false error messages related to the device model to warnings, while the reset is in process. Monitor maintains a variable indicating the number of successful resets on a particular port. RESET_ON_LNK_NOT_U0 can be used to make the bench initiate reset when the LTSSM link state of the downstream is any state other than U0.

c. **Stopping the endpoints**: ISSUE_RAND_STOP_EP_CMD can be used to control the issuance of stop endpoint commands for the configured endpoints randomly when the endpoint has traffic running.

d. **Flow control on the endpoints**: The USB specification allows flow control on the endpoints using NRDY-ERDY in Super Speed and uses NAKs in USB2 speeds. FLOW_CONTROL_BUFFER can be used to configure the device model to randomly do flow control on all the configured endpoints. On the receive side (RX endpoints), this mode can be used to make the device model issue random data size packets exercising various data size handling in the controller

e. **Low power modes**: The USB specification allows three low-power modes for Super Speed (U1, U2, and U3) and two low-power modes for High Speed, Full Speed, and Low Speed (L1 and L2). ENABLE_FAST_U1_ENTRY and ENABLE_FAST_U2_ENTRY can be used to enable U1 entries in the test. The sequence sets U1/U2/L1/L2 timeout to a small value in both DUT and device model. Proper DMVC_ENQUEUE_TD_DELAY_FACTORY has to be used and USE_LARGE_DELAY_IN_BKGRND_MODES needs to be set to make sure test has enough idle time for the host to initiate specified low-power modes. RANDOM_U3_DURING_TRAFFIC can be used to randomly initiate U3 during the traffic. Once the link is in U3, ISSUE_REMOTE_WAKE can be used specify the type of resume, either device model initiated remote wakeup of controller initiated resume.

3) **Background activity verif_modes:** These verif_modes are identified from the specification to create random activities in the test to help create legal corner-case scenarios. These are mostly light on sequence implementation and exercise simple design features, but when enabled together, are powerful in creating corner cases.
   a. **Device issuing ERDYs**: The USB specification allows devices connected on downstream ports to issue back-to-back ERDYs on endpoints, irrespective of the endpoint being in flow control. ENABLE_RAND_ERDY can be used to enable a sequence that configures the device model to issue ERDYs on the endpoint while the endpoint is not in flow control. ISSUE_MULTI_DUMMY_ERDY can be used to specify the number of back-to-back dummy ERDYs on the endpoint
   b. **Device issuing LGO_Ux**: The USB specification allows devices to issue LGO_U1s and LGO_U2s to the host, irrespective of U1/U2 enabled on the port. The host is supposed to reject/accept such LGO_Ux. RAND_DEVICE_TX_LGOU1 and RAND_DEVICE_TX_LGOU2 can be used to enable a sequence that configures the device model to random LGO_U1 and LGO_U2.
   c. **Delayed device connection**: The USB protocol allows devices to be plugged in any time during the test as against in simulation where the device models are connected during bring up. RAND_DELAYED_DEVICE_CONNECT can be used to make a bench connect device models after some random wait time other than 0-time.
   d. **Ports with no device connected**: In the case of downstream multiport tests, only a few of the available downstream ports could be connected. DEVICE_ENABLE_MASK can be used to emulate such behavior. The constraints or the test writer must ensure that at least one device model is connected.
   e. **Vendor Endpoint0 transfers**: The USB protocol defines vendor class transfers for the default control endpoint. ENABLE_BACKGROUND_EP0_TRAFFIC can be used to set up vendor class traffic on the default control endpoint after initial enumeration of the device. RAND_VENDOR_DSTAGE_LEN can be used to control the size of transfers done by the sequence.
   f. **Control transfers with address=0**: The USB protocol allows the get_descriptor command for a default control endpoint before the device is assigned any address.

GET_DESC_BEFORE_ADDR_DEV can be used to change the initial enumeration process to do an address_device command with BSR=1 followed by the get_descriptor command.

g. **Error injection**: The bench consists of many sub-sequences that use BFM-vendor provided error injection routines to inject individual errors and handling. All these sub-sequences are added to uvm_sequence_library. Using ENABLE_RANDOM_ERROR_INJECTION, the uvm_sequence_libarry is run in UVM_SEQ_LIB_RAND mode [3]. This injects various errors, depending on the error injection sub-sequence selected and traffic running in the test at the time.

h. **LMP packets**: The xHCI specification defines the PORTPMSC register, and any update to this register results in the controller issuing an LMP packet to the device. RAND_PORTPM_WRITE_BACK could be used to randomly do a register read followed by a write back of the same data for the PORTPMSC register. This creates interesting link-related scenarios.

i. **Device notification packets**: The USB specification provides device notification packets, which the device can issue to the host at any time. ISSUE_RAND_DEV_NOTIFY can be used to make the device model randomly issue any of the three possible device notification packets: BusIntervalAdjustment, LatencyToleranceMsg, or FunctionWake.

j. **Evaluate context command:** The xHCI specification defines the evaluate context command for software to change the Max Exit Latency value in slot context, which affects the periodic endpoints. ISSUE_RAND_EVALUATE_CONTEXT can be used to randomly issue an evaluate context command to change MEL in a legal range of values.

k. **Lane Inversion:** For Super Speed, TX and TX_ connection could be inverted. INVERT_LANE_POLARITY could be used to emulate lane polarity inversion at the beginning of the test. This mode also inverts the lane polarity on disconnect and reconnection.

l. **Deferred packets:** The HUBs could revert back the packets to the host with a deferred bit set when any of the target devices on the path are in low-power mode. ENABLE_DEFERRED_MODE can be used to randomly emulate this behavior from the device model.

D. Conclusion:

The proposed approach has been very effective in verifying an xHCI-based Host Controller. It facilitated modular verification of incremental design/test-bench enhancements such as Streams protocol, Broadcom PHY, and USB2.0 support. This also resulted in faster coverage-driven signoff regressions compared to directed tests, while still covering more corner cases. With the flexibility of fine controlling the verif_modes, this approach facilitated our verification team to work on different design scenarios, independently. This approach has proven to be of very high quality and has managed to catch over 300 RTL and corner cases, many of which were obscure and would not easily have been hit with a simpler test bench. This approach also promotes easy integration into the top-level bench of our customers within Broadcom to quickly bring up their own verification environment to meet their simulation goals.

Appendix A:

```
//***********************************************************************//
// Random Disconnect-Reconnect verif_mode sequence                        //
//***********************************************************************//
class broadcomUsbVirSeq_random_disconnect_reconnect extends broadcomUsbVirSeqBase;

  `uvm_object_utils(broadcomUsbVirSeq_random_disconnect_reconnect)
  `uvm_declare_p_sequencer(broadcomUsbVirtualSequencer)

   virtual task body();
        ......
    fork
      begin
          //---------------------------------------------------------------------//
          // Wait_for_legal_status
          //---------------------------------------------------------------------//
          wait(`MONITOR_PATH.bench_status.port_enabled[p_sequencer.port_id-1]==1)
          //---------------------------------------------------------------------//
          // Repeat the dis-connect sequence for NUM_DISCONNECT_CONNECT times
          // check_for_exit_count
          //---------------------------------------------------------------------//
          while(`MONITOR_PATH.num_disconnect_reconnect<usb_bench_cfg.NUM_DISCONNECT_CONNECT) begin
            //---------------------------------------------------------------------//
            // Waiting for random delay before initiating dis-connect/re-connect
            // random_wait
            //---------------------------------------------------------------------//
            assert(std::randomize(rnd_wait) with {rnd_wait inside {[200:350]};});
            repeat(rnd_wait) #1us;
            //---------------------------------------------------------------------//
            // check_for_legal_status
            //---------------------------------------------------------------------//
            if(`MONITOR_PATH.bench_status.port_enabled[p_sequencer.port_id-1]==1 &&
               `MONITOR_PATH.bench_status.port_being_in_loopback[p_sequencer.port_id-1]==0 &&
               `MONITOR_PATH.bench_status.port_in_loopback[p_sequencer.port_id-1]==0 &&
               `MONITOR_PATH.bench_status.port_ss_being_disabled[p_sequencer.port_id-1]==0 &&
               `MONITOR_PATH.bench_status.port_ss_disabled[p_sequencer.port_id-1]==0) begin
              //---------------------------------------------------------------------//
              // Raise the objection for disconnect sequence
              //---------------------------------------------------------------------//
              phase.raise_objection(this);
              //---------------------------------------------------------------------//
              // Perform a disconnect & re-connect
              // launch_core_sequence
              //---------------------------------------------------------------------//
              `uvm_do(disconnect_reconnect)
                 ->`MONITOR_PATH.disconnect_reconnect_done;
                 `MONITOR_PATH.num_disconnect_reconnect++;
              //---------------------------------------------------------------------//
              // Wait for port enabled by DUT after re-connect and went to U0 state
              //---------------------------------------------------------------------//
              wait(`DMVC_MONITOR_PATH.port_enabled_by_dmvc[p_sequencer.port_id-1] ==1);
              //---------------------------------------------------------------------//
              // Drop the objection for disconnect sequence
              //---------------------------------------------------------------------//
              phase.drop_objection(this);
            end
          end
         end
      join_none
    `endif
  endtask
endclass
```

Appendix B:

```
//************************************************************************//
// Disconnect-Reconnect verif_mode core_sequence                           //
//************************************************************************//
class broadcomUsbVirSeq_disconnect_reconnect extends broadcomUsbVirSeqBase;

virtual task body();
      .....
      //------------------------------------------------------------------------//
      // Raise the objection for disconnect sequence
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Starting disconnect-reconnect routine on port_id=%0d", 
p_sequencer.port_id));
      uvm_report_info(get_name(), $psprintf("deviceState= %s, device_linkState=%s, 
hostMonitor_linkState=%s",deviceState.name(), device_linkState.name(),hostMonitor_linkState.name()));
      phase.raise_objection(this);

      //------------------------------------------------------------------------//
      //Disable known false error which can occur during disconnect
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Disabling known false error messages from denali model during 
disconnect and re-connect of device in port_id=%0d", p_sequencer.port_id));
      .....

      //------------------------------------------------------------------------//
      // Turning off the VBUS for device model
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Turning device VBUS OFF for device in port_id=%0d", 
p_sequencer.port_id));
      `DMVC_MONITOR_PATH.port_being_disconnected[p_sequencer.port_id-1]=1;
      ....

      //------------------------------------------------------------------------//
      // Wait for the Host S/W to detect the device dis-connect
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Waiting for host to detect device dis-connect,on 
port_id=%0d", p_sequencer.port_id));
      ....
      uvm_report_info(get_name(), $psprintf("Host detected device dis-connect,on port_id=%0d", 
p_sequencer.port_id));
      //------------------------------------------------------------------------//
      // Wait for Random Delay before connecting the device back
      //------------------------------------------------------------------------//
      wait_in_us = $urandom_range(100,300);
      repeat(wait_in_us) #1us;
      uvm_report_info(get_name(), $psprintf("Finished waiting for %0dus in device disconnected state,on 
port_id=%0d", wait_in_us,p_sequencer.port_id));

      //------------------------------------------------------------------------//
      // Change the device configuration before reconnect
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("changing the configuration of the device,on 
port_id=%0d",p_sequencer.port_id));
      ....

      //------------------------------------------------------------------------//
      // Turn ON the VBUS for device model
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Trigerring device re-connect,on 
port_id=%0d",p_sequencer.port_id));
      uvm_report_info(get_name(), $psprintf("Turning device VBUS ON for device in port_id=%0d", 
p_sequencer.port_id));
      .....
      `DMVC_MONITOR_PATH.port_being_disconnected[p_sequencer.port_id-1]=0;
      //------------------------------------------------------------------------//
      // Wait for Port to go to normal state after re-connect
      //------------------------------------------------------------------------//
      ......

      //------------------------------------------------------------------------//
      // Revert back to enabling of all known false error after dis-connect
      //------------------------------------------------------------------------//
      uvm_report_info(get_name(), $psprintf("Enabling known false error messages from denali model after
disconnect and re-connect of device in port_id=%0d", p_sequencer.port_id));

      //------------------------------------------------------------------------//
      // Drop the objection for dis-connect sequence
      //------------------------------------------------------------------------//
      phase.drop_objection(this);
  endtask
endclass
```

Appendix C:

```
//------------------------------------------------------------------------//
// when low power modes are enabled, sim_opts_solver can be used to make sure test has enough
// idle time to facilitate controller to enter low power mode
//------------------------------------------------------------------------//
   constraint fast_u1_or_u2_entry_requires {
      if (enable_fast_u2_entry_arg_value == 1) {
        if((u0Tou2_inactivity_timer_value >= 2) && (u0Tou2_inactivity_timer_value <= 254)) {
          enqueue_td_delay_factory_arg_value_int dist { 8 :/ 1, 9 :/ 1 };
        } else {
          enqueue_td_delay_factory_arg_value_int dist { 2 :/ 1, 7 :/ 1 };
        }
      use_large_delay_in_bkgrnd_modes_arg_value == 1;
      issue_multi_dummy_erdy_arg_value == 0;
      }
      else if (enable_fast_u1_entry_arg_value == 1) {
      enqueue_td_delay_factory_arg_value_int dist { 1 :/ 1, 6 :/ 1 };
      use_large_delay_in_bkgrnd_modes_arg_value == 1;
      issue_multi_dummy_erdy_arg_value == 0;
      }
      else {
      use_large_delay_in_bkgrnd_modes_arg_value==0;
      }
    // Solve fast U1/U2 before TD delay
    solve enable_fast_u1_entry_arg_value  before enqueue_td_delay_factory_arg_value_int;
    solve enable_fast_u2_entry_arg_value  before enqueue_td_delay_factory_arg_value_int;
    solve u0Tou2_inactivity_timer_value   before enqueue_td_delay_factory_arg_value_int;
    solve enable_fast_u1_entry_arg_value  before use_large_delay_in_bkgrnd_modes_arg_value;
    solve enable_fast_u2_entry_arg_value  before use_large_delay_in_bkgrnd_modes_arg_value;
    solve enable_fast_u1_entry_arg_value  before issue_multi_dummy_erdy_arg_value;
    solve enable_fast_u2_entry_arg_value  before issue_multi_dummy_erdy_arg_value;
   }
```

Appendix D:

```
class bench_status
   int         port_to_slot_map [`NUM_DOWNSTREAM_PORTS];
   int         slot_to_port_map [`NUM_DOWNSTREAM_PORTS];
   int         port_to_den_inst_map [`NUM_DOWNSTREAM_PORTS];
   bit         port_enabled_by_dmvc [`NUM_DOWNSTREAM_PORTS];
   bit         port_defaulted_by_dmvc [`NUM_DOWNSTREAM_PORTS];
   bit         port_addressed_by_dmvc [`NUM_DOWNSTREAM_PORTS];
   bit         port_configured_by_dmvc [`NUM_DOWNSTREAM_PORTS];
   int         num_u1_entry [`NUM_DOWNSTREAM_PORTS];
   int         num_u2_entry [`NUM_DOWNSTREAM_PORTS];
   int         num_u3_entry [`NUM_DOWNSTREAM_PORTS];
   int         num_l2_entry [`NUM_DOWNSTREAM_PORTS];
   int         num_rand_stop_ep [`NUM_DOWNSTREAM_PORTS];
   int         num_background_ep0_transfer [`NUM_DOWNSTREAM_PORTS];
   int         num_error_injections [`NUM_DOWNSTREAM_PORTS];
   int         num_latency_tolerance_msg [`NUM_DOWNSTREAM_PORTS];
   int         num_bus_interval_adjustment_msg [`NUM_DOWNSTREAM_PORTS];
   int         num_deferred_pkts [`NUM_DOWNSTREAM_PORTS];
   bit         port_being_disconnected [`NUM_DOWNSTREAM_PORTS];
   bit         port_disconnected [`NUM_DOWNSTREAM_PORTS];
   bit         enable_do_remote_wake [`NUM_DOWNSTREAM_PORTS];
   int         num_function_wakeup_notification [`NUM_DOWNSTREAM_PORTS];
   bit         port_reset [`NUM_DOWNSTREAM_PORTS];
   bit         port_being_suspend [`NUM_DOWNSTREAM_PORTS];
   bit         port_suspend [`NUM_DOWNSTREAM_PORTS];
   bit         port_warm_reset [`NUM_DOWNSTREAM_PORTS];
   bit         port_being_reset [`NUM_DOWNSTREAM_PORTS];
   int         port_low_power_mode [`NUM_DOWNSTREAM_PORTS];
   bit         port_ss_being_disabled [`NUM_DOWNSTREAM_PORTS];
   bit         port_ss_disabled [`NUM_DOWNSTREAM_PORTS];
   int         port_speed_connected [`NUM_DOWNSTREAM_PORTS];
   bit         port_being_in_loopback [`NUM_DOWNSTREAM_PORTS];
   bit         port_in_loopback [`NUM_DOWNSTREAM_PORTS];
   bit         port_in_compliance [`NUM_DOWNSTREAM_PORTS];
   bit         compliance_test_done [`NUM_DOWNSTREAM_PORTS];
   bit         port_timeout_occurred [`NUM_DOWNSTREAM_PORTS];
 endclass
```

Appendix E

```
UVM_INFO @ 0.000 ns: reporter [usb_bench_config]
 RANDOM_SIM_OPTS_ENABLE=1
 FAST_SIM_MODE_ENABLE=1
 ISSUE_RAND_STOP_EP_CMD=0
 FLOW_CONTROL_BUFFER=0
 ENABLE_SHORT_IN_PKT=0
 ENABLE_ZERO_LEN_IN_PKT=0
 ENABLE_RAND_ERDY=0
 ISSUE_MULTI_DUMMY_ERDY=0
 ENABLE_FAST_U1_ENTRY=1
 ENABLE_FAST_U2_ENTRY=1
 RAND_DEVICE_TX_LGOU1=0
 RAND_DEVICE_TX_LGOU2=0
 ENQUEUE_TD_DELAY_FACTORY=wait_400_to_500us_after_delay
 STOP_AFTER_N_TDS_PER_EP=3
 INT_LATENCY_RANDOM_SELECT=1
 AXI_SLAVE_CFG_RANDOM_SELECT=real_delay_axi_slave_cfg
 TR_RING_FACTORY_RANDOM_SELECT=1
 ERST_FACTORY_RANDOM_SELECT=1
 BIU_CONFIG_RANDOM_SELECT=1
 AIU_CONFIG_RANDOM_SELECT=1
 NUM_DISCONNECT_CONNECT=2
 CONTINUOUS_U3_DURING_TRAFFIC=0
 RANDOM_U3_DURING_TRAFFIC=0
 DEVICE_ATTRIBUTE_RANDOM_SELECT=1
 DEVICE_ENABLE_MASK=1111
 RAND_DELAYED_DEVICE_CONNECT=0
 BENCH_PORT_SPEED=1444
 RANDOM_SPEED_SELECT_INCLUDE=4321
 ENABLE_BACKGROUND_EP0_TRAFFIC=0
 ENABLE_RANDOM_ERROR_INJECTION=0
 NUM_PORT_RESET=2
 ENABLE_DEFERRED_MODE=0
 INVERT_LANE_POLARITY=0
 RAND_LANE_POLARITY_INVERSION=0
```

Appendix F:

```
uvm_test_top.env[final_phase] ********************************************************
uvm_test_top.env[final_phase] Test Summary
uvm_test_top.env[final_phase] Port[1]:
uvm_test_top.env[final_phase]   num_u1_entry=30
uvm_test_top.env[final_phase]   num_u2_entry=15
uvm_test_top.env[final_phase]   num_u3_entry=0
uvm_test_top.env[final_phase]   num_rand_stop_ep=15
uvm_test_top.env[final_phase]   num_background_ep0_transfer=20
uvm_test_top.env[final_phase]   num_error_injections=0
uvm_test_top.env[final_phase]   num_latency_tolerance_msg=3
uvm_test_top.env[final_phase]   num_bus_interval_adjustment_msg=3
uvm_test_top.env[final_phase]   num_function_wakeup_notification=8
uvm_test_top.env[final_phase]   num_deferred_pkts=0
uvm_test_top.env[final_phase] Port[2]:
uvm_test_top.env[final_phase]   num_u1_entry=26
uvm_test_top.env[final_phase]   num_u2_entry=12
uvm_test_top.env[final_phase]   num_u3_entry=0
uvm_test_top.env[final_phase]   num_rand_stop_ep=23
uvm_test_top.env[final_phase]   num_background_ep0_transfer=18
uvm_test_top.env[final_phase]   num_error_injections=0
uvm_test_top.env[final_phase]   num_latency_tolerance_msg=2
uvm_test_top.env[final_phase]   num_bus_interval_adjustment_msg=7
uvm_test_top.env[final_phase]   num_function_wakeup_notification=6
uvm_test_top.env[final_phase]   num_deferred_pkts=0
uvm_test_top.env[final_phase] num_disconnect_reconnect=2
uvm_test_top.env[final_phase] num_port_reset=2
uvm_test_top.env[final_phase] num_loopback_entry=0
uvm_test_top.env[final_phase] num_ssdisable_entry=0
uvm_test_top.env[final_phase] ********************************************************
uvm_test_top.env[final_phase] ********************************************************
uvm_test_top.env[final_phase]             Test Case Status  : PASSED
uvm_test_top.env[final_phase] ********************************************************
```

REFERENCES

[1] Abbas Khalili, Ryan Rhodes, Adrian Yu, Broadcom Corporation “Challenges in Verifying USB3.0 Host and Device Controller “ Paper Poster Presentation, 49th Design Automation Conference.

[2] Kathleen A Meade, Sharon Rosenberg, *A Practical Guide to Adopting the Universal Verification Methodology*

[3] John Aynsley, Doulos “The Finer Points of UVM: Tasting Tips for the Connoisseur” Paper presentation DVCON 2013

[4] eXtensible Host Controller Interface for Universal Serial Bus (xHCI), Intel Corporation

[5] Universal Serial Bus 3.0 Specification, usb.org

[6] Universal Serial Bus Specification, usb.org