Integrating DesignWare USB3.0 Device Controller In a UVM-based Testbench

Ning Guo

Paradigm Works

Andover MA, U.S.A

www.paradigm-works.com

ABSTRACT

DesignWare core USB3.0 Controller (DWC_usb3) can be configured as a USB3.0 Device Controller. When verifying a system that comprises a DWC_usb3 Device Controller, the verification environment is responsible for bringing up the DWC_usb3 Device Controller to its proper operation mode to communicate with the USB3.0 Host.

This paper describes the process for configuring and driving the DWC_usb3 Device Controller in a UVM-based verification environment.

Although the UVM-based environment presented in this paper uses specific CPU and external memory interfaces of the DWC_usb3 Device Controller, the flow of the configuration and operation is applicable to the cases when different CPU and external memory interfaces are used.

This paper also shows that a UVM-based sub-environment can be setup to make the DWC_usb3 Device Controller integration process more reusable.

Table of Contents

Table of Figures

1. Introduction

The Synopsys DesignWare Cores SuperSpeed USB3.0 Controller has four configurations: USB3.0 Device Controller, USB3.0 Host Controller, USB3.0 Static Dual-Role Device Controller and USB3.0 Hub Controller [1]. The focus of this paper is the USB3.0 Device Controller configuration, which will be referred to as *DWC_usb3 Device Controller* in the content of this paper.

To verify a system that consists of a DWC_usb3 Device Controller, the verification environment must be able to configure and drive the DWC_usb3 Device Controller properly so that the system behavior can be validated. It is the system verification engineers' responsibility to integrate the DWC_usb3 Device Controller into the verification environment and bring it up to various operational modes based on the system verification requirements. This integration process could be very different from one system to another system. However, as the verification industry is converging on the standard verification methodology, i.e., UVM – Universal Verification Methodology, it becomes possible for verification environments of different systems to have consistent look and feel.

This paper discusses the process of integrating, configuring and driving the DWC_usb3 Device Controller in an example UVM-based environment. Section 2 talks about the structure of the example environment and how to physically connect the DWC_usb3 Device Controller into the environment. Section 3 describes how to configure and drive the DWC_usb3 Device Controller from the environment. Section 4 evaluates the reusability of the DWC_usb3 Device Controller integration process as well as the example environment so that users can identify what information presented in the example environment may be applicable to their verification needs.

2. Integrating DWC_usb3 Device Controller in a UVM-based Environment

2.1 System Level Block Diagram

Figure 2-1 is the system level block diagram of the example UVM-based environment.

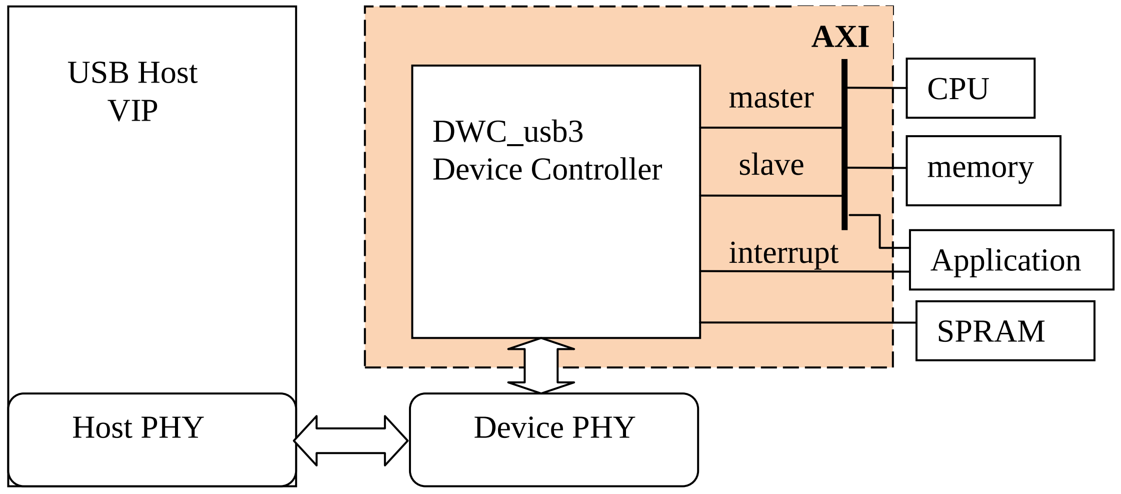

Figure 2-1 System Level Block Diagram of the Example UVM-based Environment

In Figure 2-1, the DWC_usb3 Device Controller is connected to the environment through five interfaces: AXI master interface, AXI slave interface, interrupt interface, RAM interface and USB3 PHY interface.

The DWC_usb3 Device Controller supports both USB2.0 and SuperSpeed functions. The focus of the example environment is on the SuperSpeed only. Hence the PHY interface used is PIPE3.

The Host side of the example environment is implemented using DesignWare USB VIP. Other systems may use a RTL design or a different vendor's verification IP as the Host side.

On the Device side of the example environment, the following setups are made:

- A DesignWare AXI master VIP is used as the CPU to access the DWC_usb3 Device Controller's registers and RAMs
- A DesignWare AXI slave VIP with built-in memory model is used to store Device Controller's event information, transfer descriptors and data.
- A single Verilog SPRAM model is used to store endpoint configurations
- The application layer includes a UVM Device Driver, a transfer response sequencer and a configuration descriptor. The details of the Device Driver will be discussed in section 3.

The DesignWare AXI VIPs and the DesignWare USB VIP used in the example UVM environment have native UVM support. Their UVM supports are in the early adopter phase.

2.2 Integrating the DWC_usb3 Device Controller

Integrating the DWC_usb3 Device Controller includes the following steps:

1. Instantiating and connecting the DWC_usb3 Device Controller module to the top level testbench module
2. Connecting the DWC_usb3 Device Controller to the application layer of the system environment
3. Configuring the DWC_usb3 Device Controller from the environment using the Device Driver
4. Driving the DWC_usb3 Device Controller to respond to Host transfer requests using the Device Driver

2.2.1 Integration at the Module Level

Assuming a DWC_usb3 Device Controller IP is installed in the user's $DESIGNWARE_HOME and is configured to the default setup through the coreConsultant tool. (Refer to DWC_usb3 Device Controller Databook[1] for details about how to install and configure the DWC_usb3 Device Controller IP)

In order to integrate the DWC_usb3 Device Controller to the UVM-based system environment, a SystemVerilog wrapper module – *DWC_usb3_sv_wrapper* is created to connect the I/Os of the DWC_usb3 Device Controller to the SystemVerilog interfaces.

Besides the interfaces such as PIPE3, AXI, etc, the DWC_usb3 Device Controller also has reset and clock signals that need to be driven by the top level testbench. Different approaches can be used to bring reset and clock signals up to the top level testbench. Two of them are,

1) Define internal wires and registers in the *DWC_usb3_sv_wrapper*. Connect these internal wires and registers with the corresponding reset and clock signals of the DWC_usb3 Device Controller. The top level testbench accesses the *DWC_usb3_sv_wrapper*'s internal reset and clock through hierarchical paths;

2) Bring the reset and clock signals to the port list of the *DWC_usb3_sv_wrapper*. The top level testbench defines wires and registers to connect to the *DWC_usb3_sv_wrapper*'s reset and clock ports and access them directly.

The example UVM environment uses the first approach for reset access and the second approach for clock access. Figure 2-2 shows the code snippet of the *DWC_usb3_sv_wrapper*.

```
module DWC_usb3_sv_wrapper  (input             clk,
                             svt_usb_if        usb_ss_mac_if,
                             svt_usb_if        usb_ss_phy_if,
                             rtl_dut_intr_if   dut_intr_if,
                             svt_axi_master_if axi_master_if,
                             svt_axi_slave_if  axi_slave_if);

 // Define internal wires/regs for resets
 wire tb_pipe3_reset_n;    // DWC_usb3’s output
 reg  tb_vcc_reset_n;      // input
 reg  tb_pipe3_reset_n;    // input

 // Instantiate DWC_usb3
 DWC_usb3  dut (…,
                .pipe3_reset_n(dut_pipe3_reset_n),
                .vcc_resetn_n(dut_vcc_reset_n),
                …);

 // Connect internal reset wires/regs to the DWC_usb3
 assign dut_vcc_reset_n  = tb_vcc_reset_n;
 assign tb_pipe3_reset_n = dut_pipe3_reset_n;

 // Connect clock to the DWC_usb3
 assign dut_bus_clk_early  = clk;

 // Connect interfaces to the DWC_usb3
 assign usb_ss_phy_if.pipe3_dut_mac_if.TxData = dut_pipe3_TxData;
 …
 assign dut_pipe3_RxValid = usb_ss_phy_if.pipe3_dut_mac_if.RxValid;
…
endmodule
```

Figure 2-2 Code Snippet of the DWC_usb3_sv_wrapper module

Figure 2-1 shows that the DWC_usb3 Device Controller also connects with a SPRAM to store endpoint specific configurations. The SPRAM model used by the example environment is a SystemVerilog model. It is directly instantiated in the *DWC_usb3_sv_wrapper* and connected with the DWC_usb3 Device Controller as shown in Figure 2-3.

```
module DWC_usb3_sv_wrapper  (input             clk,
                             svt_usb_if        usb_ss_mac_if,
                             svt_usb_if        usb_ss_phy_if,
                             rtl_dut_intr_if   dut_intr_if,
                             svt_axi_master_if axi_master_if,
                             svt_axi_slave_if  axi_slave_if);
    ...
    spram_model#(`DWC_USB3_RAM0_DEPTH,32) ram0(.clk(clk),
                                              .rst_n(tb_vcc_reset_n),
                                              .cs_n(dut_ram0_p1_ce_n),
                                              .wr_n(dut_ram0_p1_wr_n),
                                              .rw_addr(dut_ram0_p1_addr),
                                              .data_in(dut_ram0_p1_wdata),
                                              .data_out(dut_ram0_p1_rdata)
                                              );
     …
endmodule
```

Figure 2-3 SPRAM Connection in the DWC_usb3_sv_wrapper

The top level testbench instantiates the *DWC_usb3_sv_wrapper* module and the interfaces needed by the DWC_usb3 Device Controller. It then connects the interfaces with the *DWC_usb3_sv_wrapper*. The top level testbench also contains initial blocks to generate the clocks needed by the system. Figure 2-4 illustrates the structure of the top level testbench module.

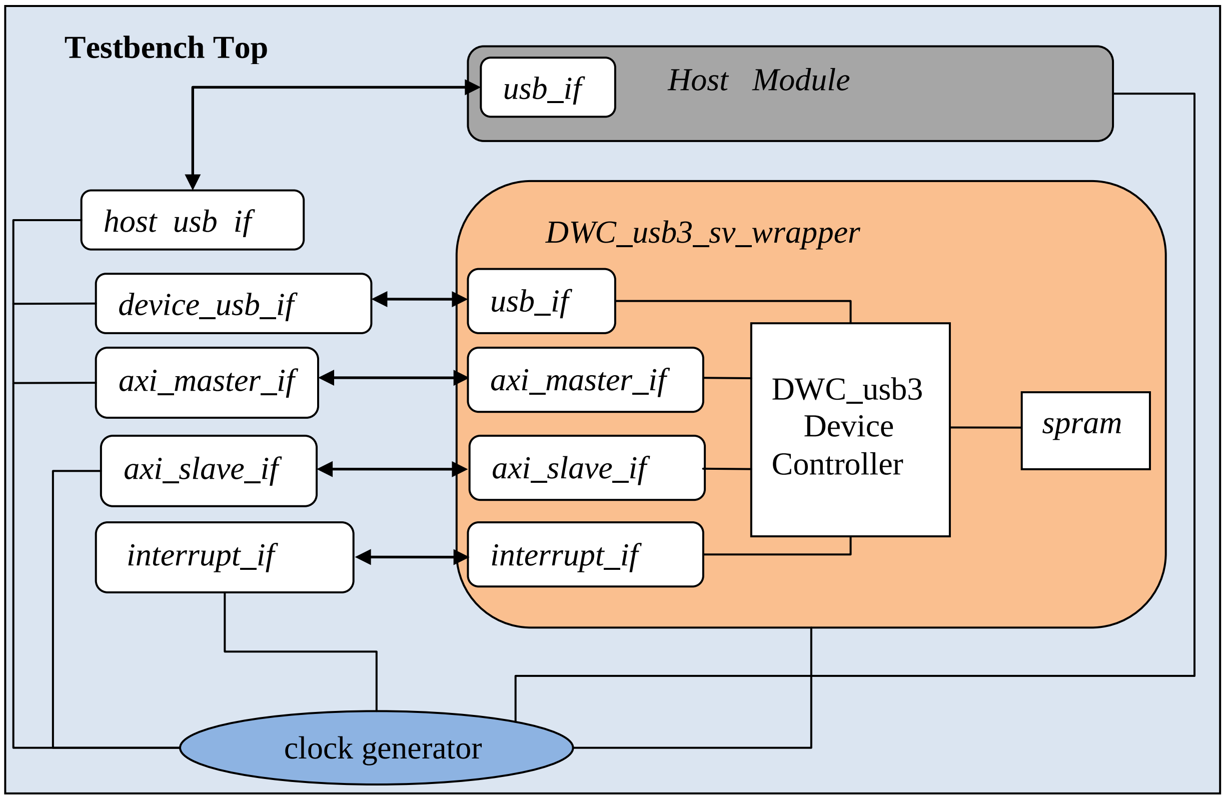

Figure 2-4 The Top Level Testbench Module of the example UVM-based environment

One thing to mention is that the connections inside the *DWC_usb3_sv_wrapper* are not visible from the top level testbench module. They are displayed in Figure 2-4 only to demonstrate how the DWC_usb3 Device Controller is integrated into the testbench.

2.2.2 Integration at the Environment level

After integrating DWC_usb3 Device Controller into the top level testbench module, the system environment can be constructed to access the DWC_usb3 Device Controller through the interfaces. The system environment, i.e., the *usb_system_env*, extends from the **uvm_env** base class. Figure 2-5 shows the structure of the *usb_system_env* class.

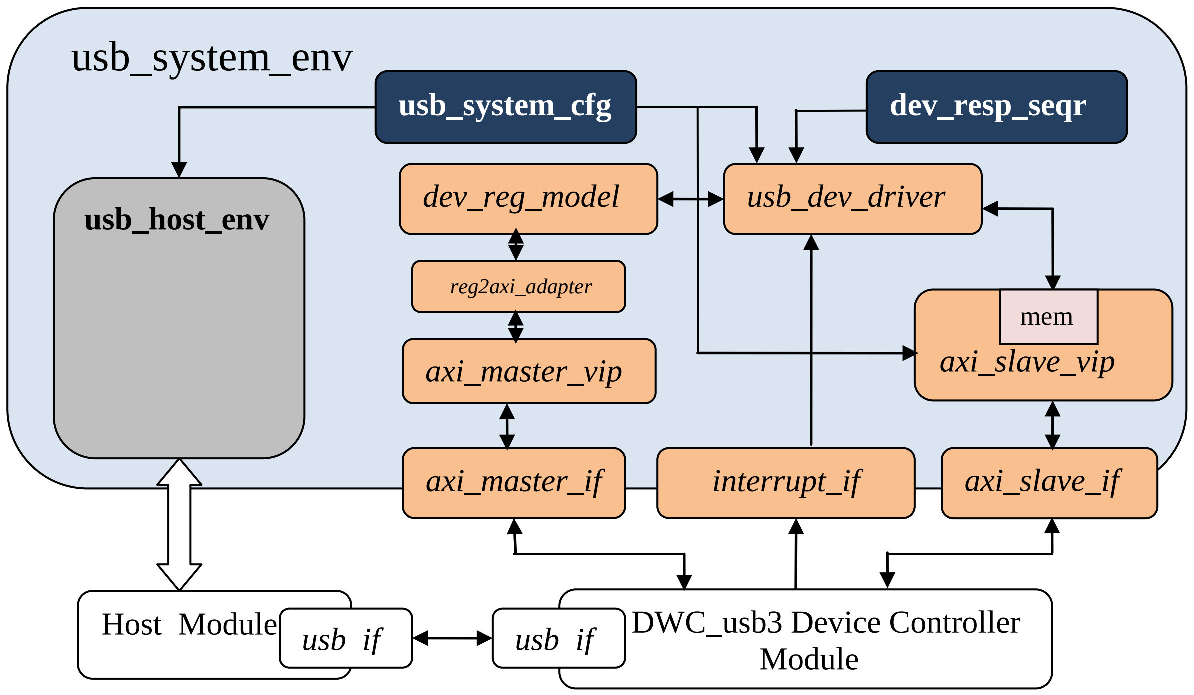

Figure 2-5 UVM-based System Environment Block Diagram

In Figure 2-5, the blocks outside of the *usb_system_env* block are part of the top level testbench module. They are shown here to illustrate how the *usb_system_env* class is connected to the testbench module through interfaces.

The Host side of the *usb_system_env* is shown as a single gray box. It could be a RTL design or a verification IP. The example system environment uses the Synopsys DesignWare USB VIP which has native UVM support to implement its Host side.

The Device side of the *usb_system_env* consists of a Device Driver, a Device Register Model, an AXI master VIP and an AXI slave VIP. The AXI master and slave VIPs are also Synopsys DesignWare VIPs with native UVM support. The AXI master VIP is responsible for writing and reading registers and memories of the DWC_usb3 Device Controller. The AXI slave VIP has an internal memory model. The DWC_usb3 Device Controller writes and reads to the memory through the AXI slave interface. The AXI slave VIP is responsible for handling memory write/read requests properly.

The Device Register Model, *reg_model*, is an extension from **uvm_reg_block** base class. It contains all the DWC_usb3 Device Controller internal registers and fields. Using the Register Model, the DWC_usb3 Device Controller's registers can be accessed by the following method calls:

```
<reg_name>.write(status,data);
<reg_name>.read(status,data);
```

<reg_name> is the register name including the hierarchical path from the top level of the Register Model.

For example, the top level of the *reg_model* consists of two register blocks: *usb3_map_DWC_usb3_block_gbl* and *usb3_map_DWC_usb3_block_dev*. *usb3_map_DWC_usb3_block_gbl* contains all the global registers of the DWC_usb3 Device Controller. *usb3_map_DWC_usb3_block_dev* contains all the common device specific registers of the DWC_usb3 Device Controller. A write to the global register GCTL can be done by issuing:

```
reg_model.usb3_map_DWC_usb3_block_gbl.GCTL.write(status,data);
```

The *write/read* methods are defined by the base **uvm_reg** class. All the register models of the DWC_usb3 Device Controller are derived from the **uvm_reg** base class. The *status* indicates the write/read access status. Its values include: *UVM_IS_OK*, *UVM_NOT_OK*, *UVM_HAS_X*. Details of the UVM Register Layer can be found in the UVM Reference Manual [2]. The *data* is the write data or read return data.

In order to convert the generic read/write operations of the *reg_model* to protocol-specific address/data/rw operations, a protocol specific extension from the **uvm_reg_adapter** base class needs to be used. In the example environment, the *reg2axi_adapter* is instantiated in the *usb_system_env*. The *reg2axi_adapter* defines the two main methods for converting a generic read/write operation to and from the AXI master operations: **reg2bus()** and **bus2reg()**. The *reg2axi_adapter* is constructed and connected to the *reg_model* in the **connect_phase()** of the *usb_system_env*.

Figure 2-6 shows the code snippet of constructing and connecting the *reg2axi_adapter* to the *reg_model*.

```
class uvm_system_env;
  ral_sys_DWC_usb3  reg_model;
  reg2axi_adapter   reg2axi;
  …
  task connect_phase(uvm_phase phase);
   if (reg_model != null) begin
     if (reg_model.get_parent() == null) begin
       //Register block is the Root block
       reg2axi = reg2axi_adapter::type_id::create(“reg2axi”);
       //
       reg_model.default_map.set_sequencer(axi_mstr.sequencer,reg2axi);
     end
   end
   …
  endtask
endclass
```

Figure 2-6 Code Snippet for connecting reg2axi_adapter with reg_model

Another important component on the device side is the Device Driver, i.e., *usb_dev_driver*. It functions as the application layer of the USB3.0 device and is responsible for bringing up the USB3 device to its proper operational mode. The Device Driver is specific to the corresponding USB device. Section 3 will be focusing on the Device Driver that is specific to the DWC_usb3 Device Controller.

3. Configuring and Operating DWC_usb3 Device Controller

Section 2 described how to integrate the DWC_usb3 Device Controller into the top level testbench module and connect it with the system environment through interfaces. This section will describe how to configure the DWC_usb3 Device Controller and manage its operations through the *usb_dev_driver*.

3.1 Device Driver Overview

The *usb_dev_driver* is the key component for configuring and operating the DWC_usb3 Device Controller.

Figure 3-1 shows the class members and the processes defined in the *usb_dev_driver*.

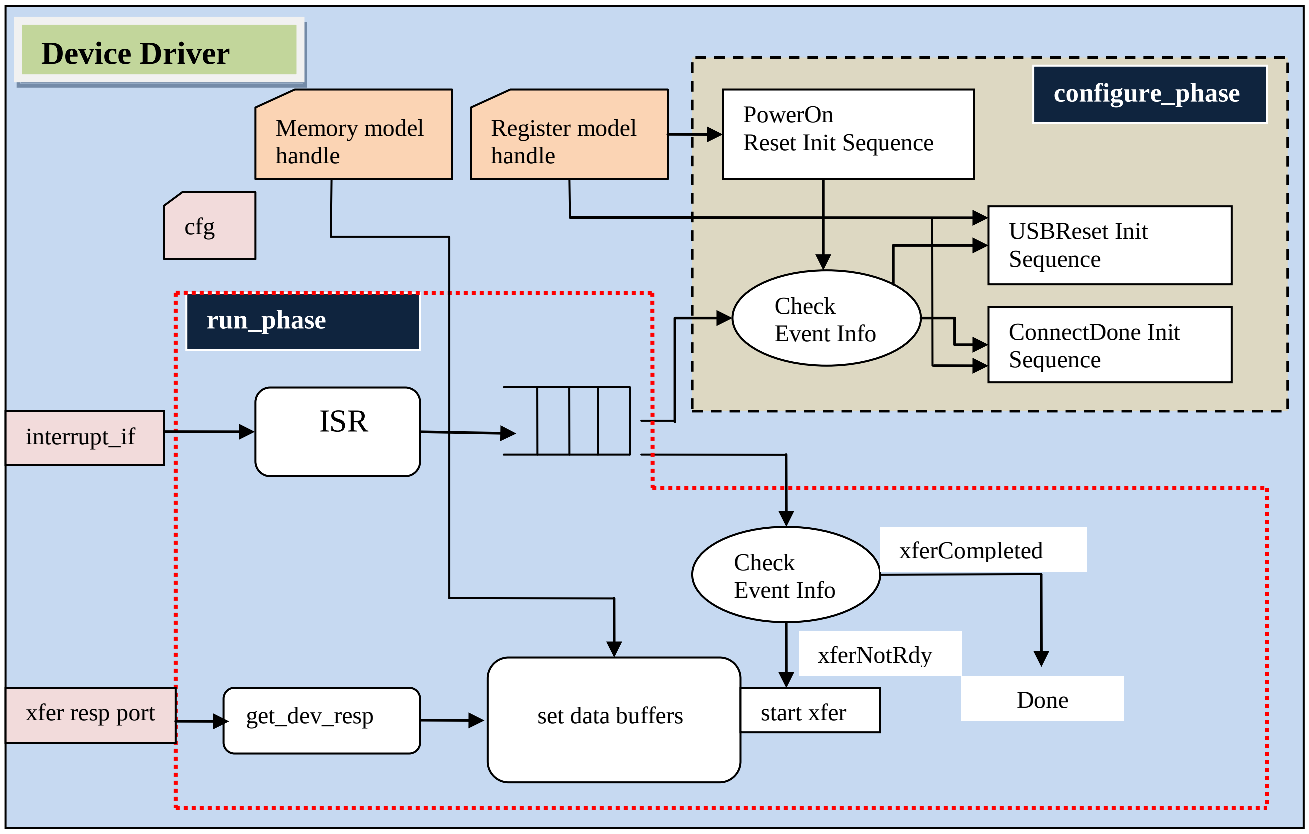

Figure 3-1 Class Members and Processes of the Device Driver

Among the class members shown in Figure 3-1, the *register model handle* is used by the three initialization sequences in the **configure_phase**. The *memory model handle* is used for backdoor memory accesses to set up data buffers in the **run_phase**.

The processes defined in the *usb_dev_driver* include:

1) The three initialization sequences in the **configure_phase**
2) The interrupt service routine (ISR) in the **run_phase**
3) The device response sequence process in the **run_phase**
4) The event checking process that is used by both **configure_phase** and **run_phase**.

3.2 Configuring the DWC_usb3 Device Controller

To configure the DWC_usb3 Device Controller, the environment needs to issue one or multiple register access sequences. Such sequences are defined and issued by the *usb_dev_driver* during the **configure_phase**.

As shown in Figure 3-1, there are three main configuration sequences: PowerOn Reset, USB Reset and Connect Done. Each sequence needs to be issued at different state of the DWC_usb3 Device Controller.

Figure 3-2 shows the PowerOn Reset flow which is issued right after the DWC_usb3 Device Controller comes out of the PowerOn Reset or Soft Reset. The highlighted strings are the registers of the DWC_usb3 Device Controller. These registers are described in the DWC_usb3 Databook [1].

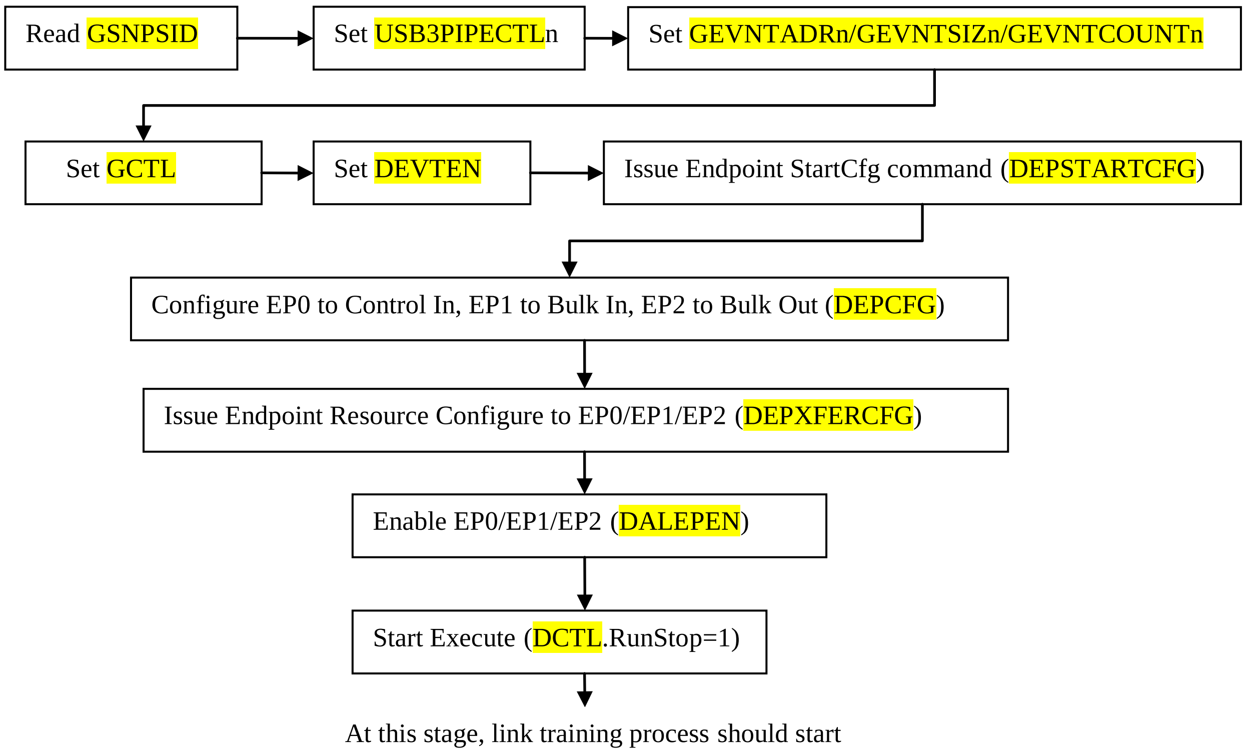
Read GSNPSID
Set USB3PIPECTLn
Set GEVNTADRn/GEVNTSIZn/GEVNTCOUNTn
Set GCTL
Set DEVTEN
Issue Endpoint StartCfg command (DEPSTARTCFG)
Configure EP0 to Control In, EP1 to Bulk In, EP2 to Bulk Out (DEPCFG)
Issue Endpoint Resource Configure to EP0/EP1/EP2 (DEPXFERCFG)
Enable EP0/EP1/EP2 (DALEPEN)
Start Execute (DCTL.RunStop=1)
At this stage, link training process should start

Figure 3-2 Device Driver PowerOn Reset Initialization Sequence

Two things to mention regarding the PowerOn Reset sequence.

1) As shown in Figure 2-5, the system environment consists of a configuration descriptor, *usb_system_cfg*. This configuration descriptor contains the configuration information for all components of the system. The configuration information of each component in *usb_system_cfg* is passed to that component using **uvm_config_db::set()** and **uvm_config_db::get()** methods. The PowerOn Reset sequence uses the configuration from the *usb_system_cfg* to determine what value the control registers need to be set to and how to configure the endpoints.

2) The register accesses physically happen at the AXI master interface of the DWC_usb3 Device Controller because the *reg2axi_adapter* is set to be the sequencer of the *reg_model* as shown in Figure 2-6.

Assuming both Host and Device are active, during the PowerOn Reset, the DWC_usb3 Device Controller generates two events: USB Reset and Connect Done. DWC_usb3 databook [1] describes what the application layer may need to do when these two events are observed.

Figure 3-3 shows the initialization sequences associated with the USB Reset and the Connect Done events.

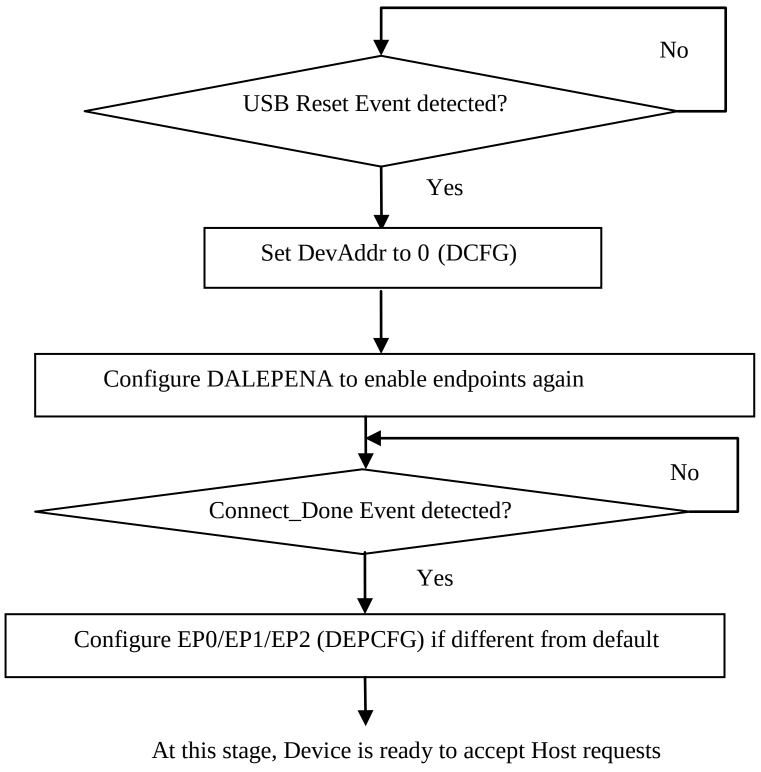
No
USB Reset Event detected?
Yes
Set DevAddr to 0 (DCFG)
Configure DALEPENA to enable endpoints again
No
Connect_Done Event detected?
Yes
Configure EP0/EP1/EP2 (DEPCFG) if different from default
At this stage, Device is ready to accept Host requests

Figure 3-3 USB Reset and Connect Done Process

As shown in Figure 3-3, at the end of the Connect Done, the DWC_usb3 Device Controller is in the normal operation mode and is ready to accept transfer requests from the Host.

3.3 Monitoring the DWC_usb3 Device Controller Events

The DWC_usb3 Device Controller generates various events to indicate its current operation state. If enabled, these events will trigger the interrupt signal(s) of the DWC_usb3 Device Controller so that the application layer will be notified about the Device Controller's state changes.

In order to capture the DWC_usb3 Device Controller's interrupts dynamically, the *usb_dev_driver* defines a task *isr()* to monitor the activities of the DWC_usb3 Device Controller's interrupt signal(s). Once an interrupt is observed, the *isr()* task processes the interrupt to find out the interrupt source, record the source information and then clear the interrupt. In Figure

3-1, the block ISR briefly illustrates the function of the *isr()* task. Figure 3-4 shows the detailed flow of the *isr()* task.

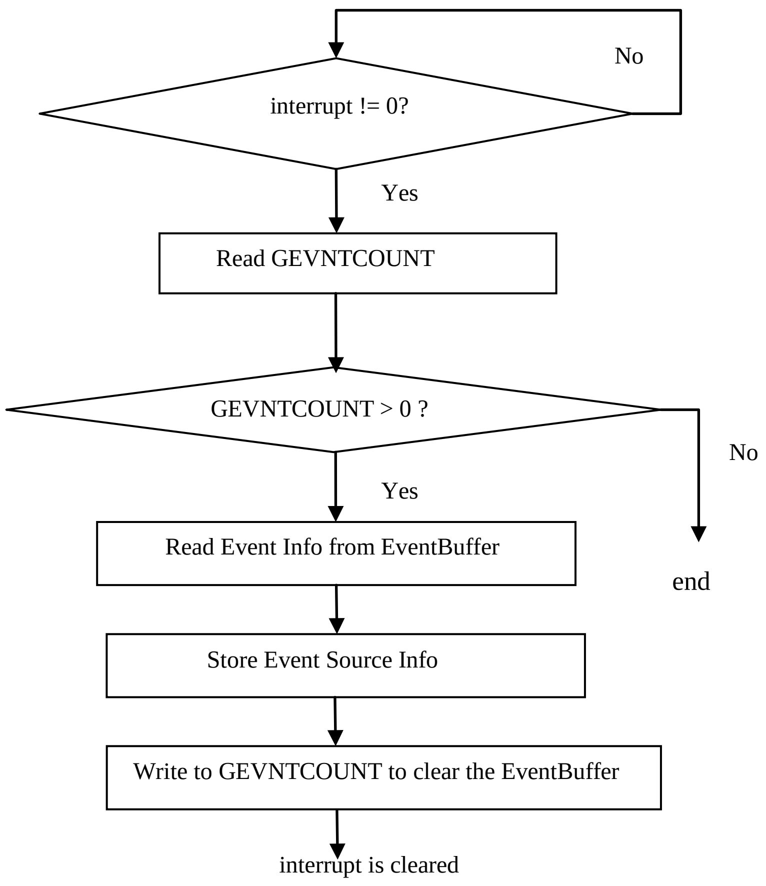

Figure 3-4 Interrupt Service Routine - *isr()*

The *isr()* task is invoked in a forever loop during the **run_phase** of the *usb_dev_driver* so that the interrupts can be monitored constantly.

Every time an interrupt is detected, the interrupt event information captured by the *isr()* task is pushed into a smart queue. This smart queue is then checked by the **configure_phase** to determine when to issue the USB Reset and the Connect Done sequences. The smart queue of the interrupt event information is also checked by the **run_phase** to control how the DWC_usb3 Device Controller responds to the transfer requests.

3.4 Performing Bulk Transfers

This section only covers Bulk transfers with streams not being enabled. Other kinds of transfers will follow similar processes but with some differences. Please refer to the DWC_usb3 databook [1] for detailed instructions on other kinds of transfers.

3.4.1 Bulk In Transfer Process

The DWC_usb3 databook[1] describes two scenarios for handling a Bulk In transfer request from a Host:

1) The application layer sets up data buffers before the host request.
2) The application layer sets up data buffers after the host request

The *usb_dev_driver* in the example environment is implemented based on the first scenario. However, the only difference between the implementations of the two scenarios is when to call the data buffer setup methods. Therefore, it should be fairly straightforward to modify the example *usb_dev_driver* to handle the second scenario.

Figure 3-5 shows the flow of a Bulk In transfer. In the example environment, the Endpoint1 (*EP1)* of the DWC_usb3 Device Controller is configured to be the BulkIn endpoint.

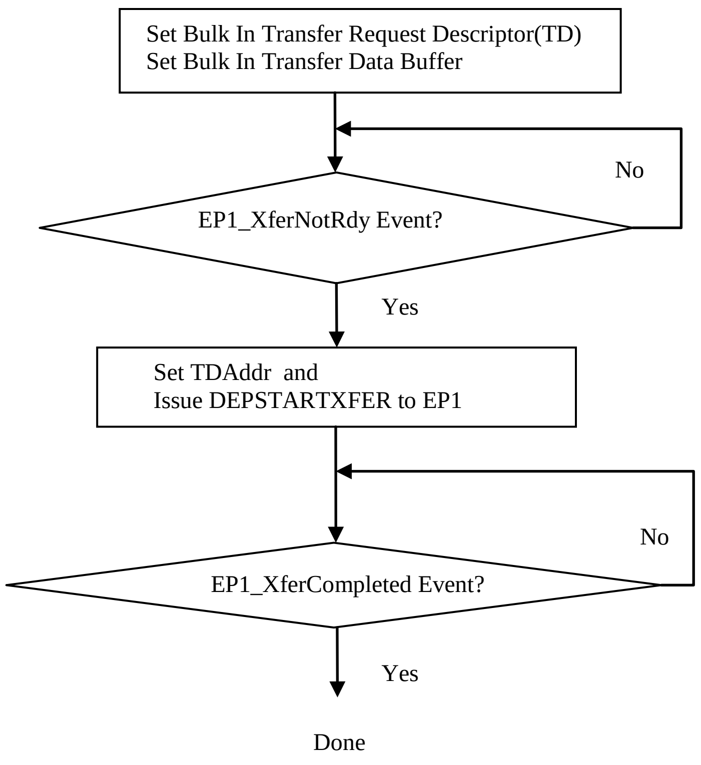
Set Bulk In Transfer Request Descriptor(TD)
Set Bulk In Transfer Data Buffer
No
EP1_XferNotRdy Event?
Yes
Set TDAddr and
Issue DEPSTARTXFER to EP1
No
EP1_XferCompleted Event?
Yes
Done

Figure 3-5 Bulk In Transfer Process in the *usb_dev_driver*

The above process is activated whenever the *dev_resp_seqr* (see Figure 2-5) in the environment has a request response item ready to pass to the *usb_dev_driver*.

3.4.2 Bulk Out Transfer Process

Similar to the Bulk In transfer, there are two scenarios for handling a Bulk Out transfer.

1) The application layer sets up data buffers before a host request.
2) The application layer sets up data buffers after a host request

Figure 3-6 shows the Bulk Out transfer process implemented in the *usb_dev_driver*. The implementation is based on the first scenario. In the example environment, Endpoint2 (*EP2*) is configured to be a BulkOut endpoint.

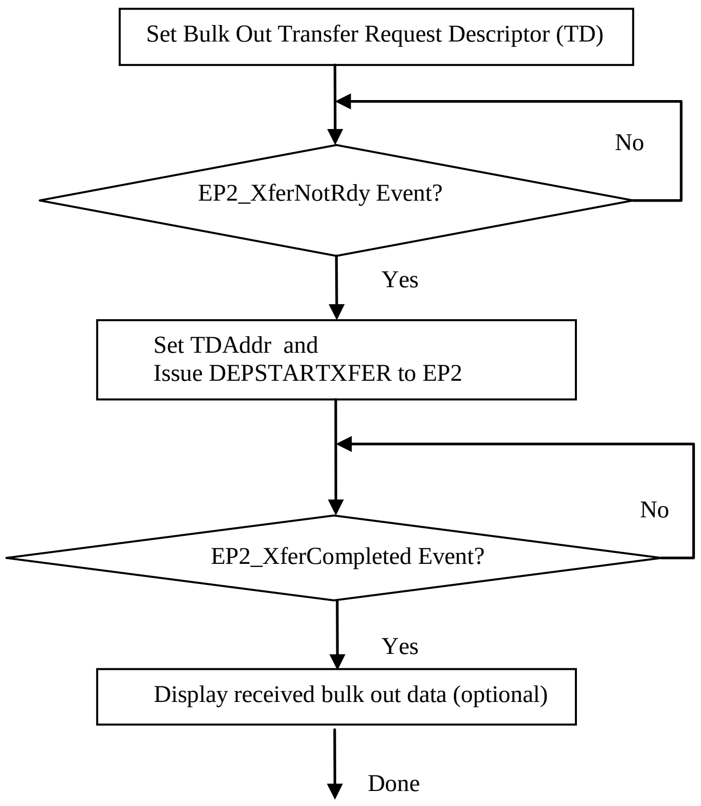
Set Bulk Out Transfer Request Descriptor (TD)
No
EP2_XferNotRdy Event?
Yes
Set TDAddr and
Issue DEPSTARTXFER to EP2
No
EP2_XferCompleted Event?
Yes
Display received bulk out data (optional)
Done

Figure 3-6 Bulk Out Transfer Process in the Device Driver

The above process is activated whenever the *dev_resp_seqr* (see Figure 2-5) has a request response item to pass to the *usb_dev_driver*.

3.5 Device Driver Implementation

The following code snippet shows the class declaration and the main class members of the *usb_dev_driver* for the DWC_usb3 Device Controller. These members are also illustrated in Figure 3-1.

```
 class usb_dev_driver extends uvm_component;
  ral_sys_DWC_usb3  reg_model;      // register model
  svt_mem           mem_model;      // memory model
  svt_usb_agent_configuration  cfg; // device configuration descriptor

  // TLM port to pull transfer responses from the sequencer
  uvm_seq_item_pull_port #(svt_usb_transfer) xfer_resp_port;

  // Interrupt interface to the DWC_usb3 Device Controller
  interrupt_itf     isr_if;

  // Smart queue to store event source
  local string      evt_src[$];
  …
endclass
```

Figure 3-7 and Figure 3-8 are the **configure_phase** and **run_phase** tasks of the *usb_dev_driver*.

```
task configure_phase(uvm_phase phase);

  // Call implementation in parent class
  super.configure_phase(phase);

  // Raise objection to prevent simulation from termination
  phase.raise_objection(this,”usb_dev_driver::configure_phase”);

  // PowerOn Reset or soft reset initialization
  poweron_sreset_cfg(cfg);

  while (!connect_done) begin
    wait(evt_src.size()>0);
    evt_str = evt_src.pop_front();
    // Call USB Reset initialization when USB_Reset event happens
    if (evt_str == “USB_Reset”)  usb_reset_cfg(cfg);

    // Call Connect Done initialization when Connect Done event happens
    if (evt_str == “Connect_Done”) begin
      connect_done_cfg(cfg);
      connect_done = 1;
    end
  end

  // Drop objection so simulation can terminate
  phase.drop_objection(this,”usb_dev_driver::configure_phase”);
endtask
```

Figure 3-7 configure_phase of the Device Driver

```
task run_phase(uvm_phase phase);

  // Call implementation in parent class
  super.run_phase(phase);

  fork
    forever isr();   // ISR routine
    forever drive_xfer_response();  // Drive transfer response
  join_none

endtask
```

Figure 3-8 run_phase of the Device Driver

The *isr()* and *drive_xfer_response()* processes invoked by the **run_phase()** are implemented as follows.

```
task isr();
  wait(isr_itf.intr_in !== 0); // wait for interrupt indication
  while (isr_itf.intr_in !== 0) begin
    // read GEVNTCOUNT
    reg_model.DWC_usb3_map_DWC_usb3_block_gbl.GEVNTCOUNT.read(status,da
ta);

    if (data>0) begin  // Detected an event
      // Figure out the event source
      evt_str = “event_src”;

      // Push event source info to evt_src queue
      evt_src.push_back(evt_str);
    end

    // write to GEVNTCOUNT to remove 4bytes from event buffer
    data = 32’h4;
    reg_model.DWC_usb3_map_DWC_usb3_block_gbl.GEVNTCOUNT.write(status,d
ata);
  end
endtask
```

Figure 3-9 ISR routine of the Device Driver

```
task drive_xfer_response();
  svt_usb_transfer xfer;

  xfer_resp_port.get_next_item(xfer);

  case (xfer.xfer_type)
    svt_usb_transfer::BULK_IN_TRANSFER: begin
     // write transfer descriptor to data buffer
     // write transfer data to data buffer
    end
    svt_usb_transfer::BULK_OUT_TRANSFER: begin
     // write transfer descriptor to data buffer
    end
  endcase

  // check event info
  …
  case (evt_str)
     “EP1_XferNotRdy”: begin
        // Issue DEPXFERSTART to EP1
        // Wait for EP1_XferCompleted event, then stop
     end
     “EP2_XferNotRdy”: begin
        // Issue DEPXFERSTART to EP2
        // Wait for EP2_XferCompleted event, then stop
     end
  endcase
  xfer_resp_port.item_done(xfer);
endtask
```

Figure 3-10 drive_xfer_response() task of the Device Driver

4. Reusability of the example UVM-based system environment

The following two scenarios are used for evaluating the reusability of the example UVM-based system environment presented in sections 2 and 3.

1) DWC_usb3 Device Controller is the USB3 Device in the system. But different CPU and/or memory interfaces are used in the system.

2) Different USB3 Device is used in the system.

4.1 DWC_usb3 Device Controller in different systems

Assuming the DWC_usb3 Device Controller in a different system also operates in the Super-Speed mode, its interface to the USB PHY component remains to be PIPE3. The main differences between the two systems are the interfaces to the CPU and/or the memories.

In terms of the CPU interfaces, besides the AXI Master and Slave interfaces discussed in the example environment, the DWC_usb3 Device Controller supports the AHB Master and Slave interfaces, the Native Master and Slave interfaces, as well as the Non-processer interface (NPI).

Besides the single one port RAM structure in the example environment, the DWC_usb3 Device Controller can connect to multiple RAMs or a single RAM with multiple ports.

Figure 4-1 shows the generic environment structure of a system which uses the DWC_usb3 Device Controller as its USB3 Device. It is modified from Figure 2-5, i.e., the environment structure of the example system.

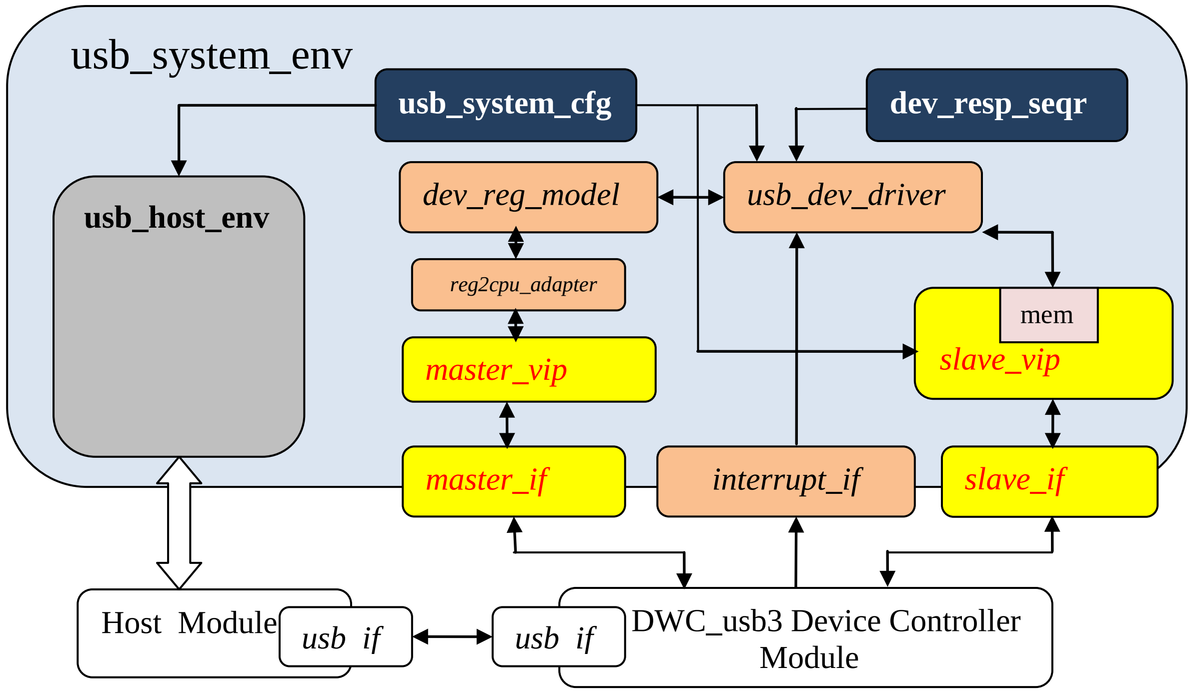

Figure 4-1 DWC_usb3 Device Controller with different CPU and memory interfaces

The main differences between Figure 4-1 and Figure 2-5 are the four yellow blocks.

Additional changes needed to support the CPU master and slave VIP variations include:

a. Replace the *reg2axi_adapter* of the example environment with the new CPU interface specific *reg2cpu_adapter*. The *reg2cpu_adapter* implements interface specific **reg2bus()** and **bus2reg()** methods.

b. Modify the *usb_system_cfg* descriptor to contain configurations that are specific to the new CPU master and/or slave VIPs.

Figure 4-2 is modified from Figure 2-4. It shows the testbench module of a system with different RAM connections to the DWC_usb3 Device Controller.

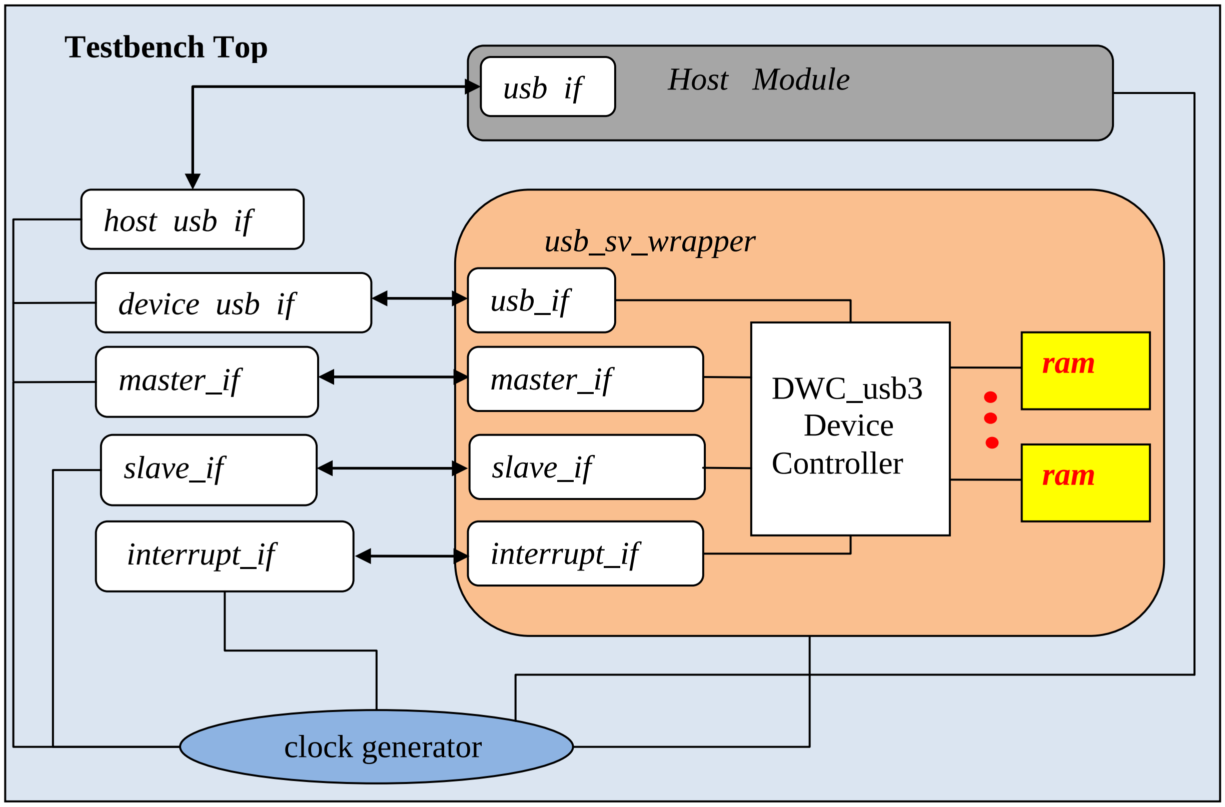

Figure 4-2 Testbench Module of the DWC_usb3 Device Controller with different RAM connections

Since the RAM connections are encapsulated in the *usb_sv_wrapper* module, it is transparent to the upper layer of the system environment.

In summary, when integrating the DWC_usb3 Device Controller DUT in a UVM-based system environment, the top level testbench module can be constructed as shown in Figure 4-2. The implementation details can be found in section 2.2.1. The RAM interface variations can be handled by implementing system specific *usb_sv_wrapper* module.

The system environment can be constructed as shown in Figure 4-1. The implementation details can be found in section 2.2.2. The *dev_resp_seqr*, *dev_reg_model* and the *usb_dev_driver* can be reused without modifications. The Master/Slave VIPs will be system specific. The *reg2cpu_adapter* and the *usb_system_cfg* need to be modified based on the Master/Slave VIPs.

4.2 Different USB3 Device DUT

Figure 4-3 shows the top level testbench module for the system that includes a different USB3 Device DUT.

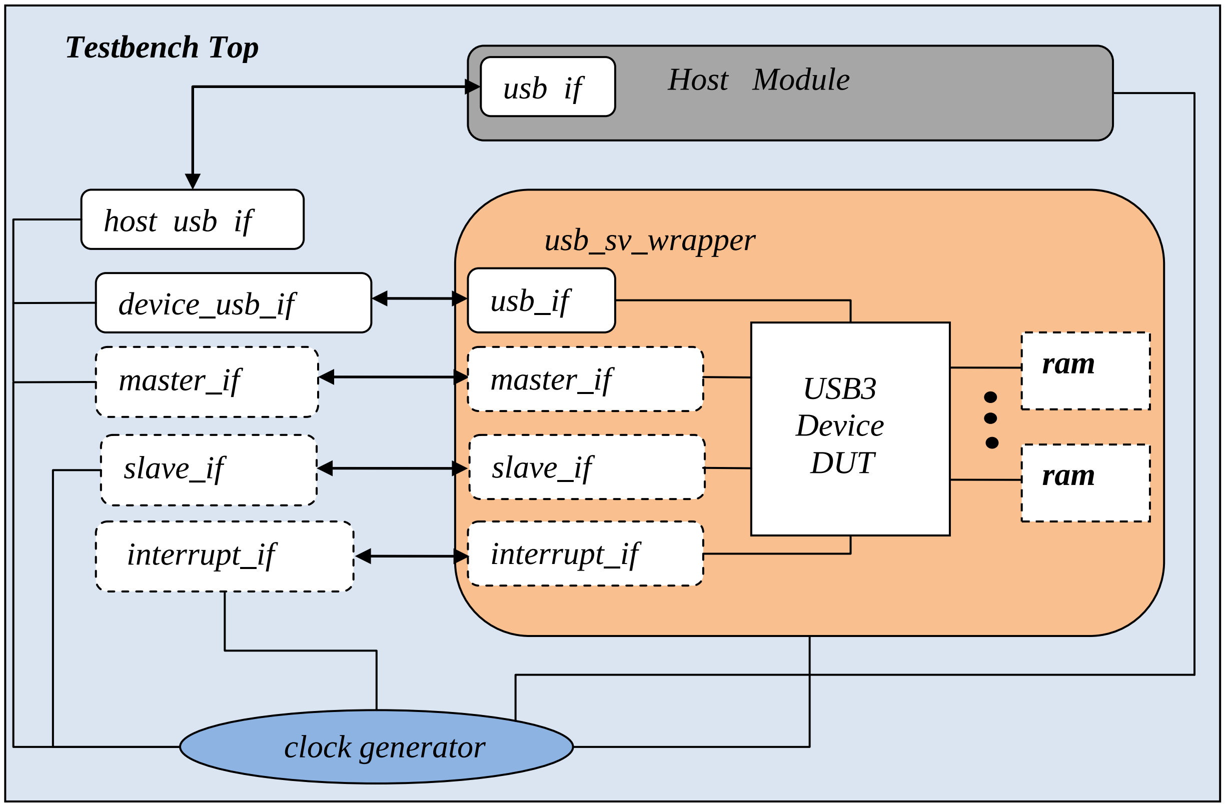

Figure 4-3 Testbench module for a USB3 Device DUT verification environment

The dashed blocks in Figure 4-3 may or may not be all applicable to a specific DUT. But the **clock generator, Host module** and the **usb_sv_wrapper** blocks as well as the **host_usb_if** and **device_usb_if** will be needed. Using *usb_sv_wrapper* to encapsulate the USB3 Device DUT and its interfaces to the external environment makes it easy to reuse the Figure 4-3 testbench module in various USB3 Device DUT verification environments.

In terms of the upper level verification environment, Figure 4-1 is redrawn below with DUT name changed to "USB3 Device DUT Module".

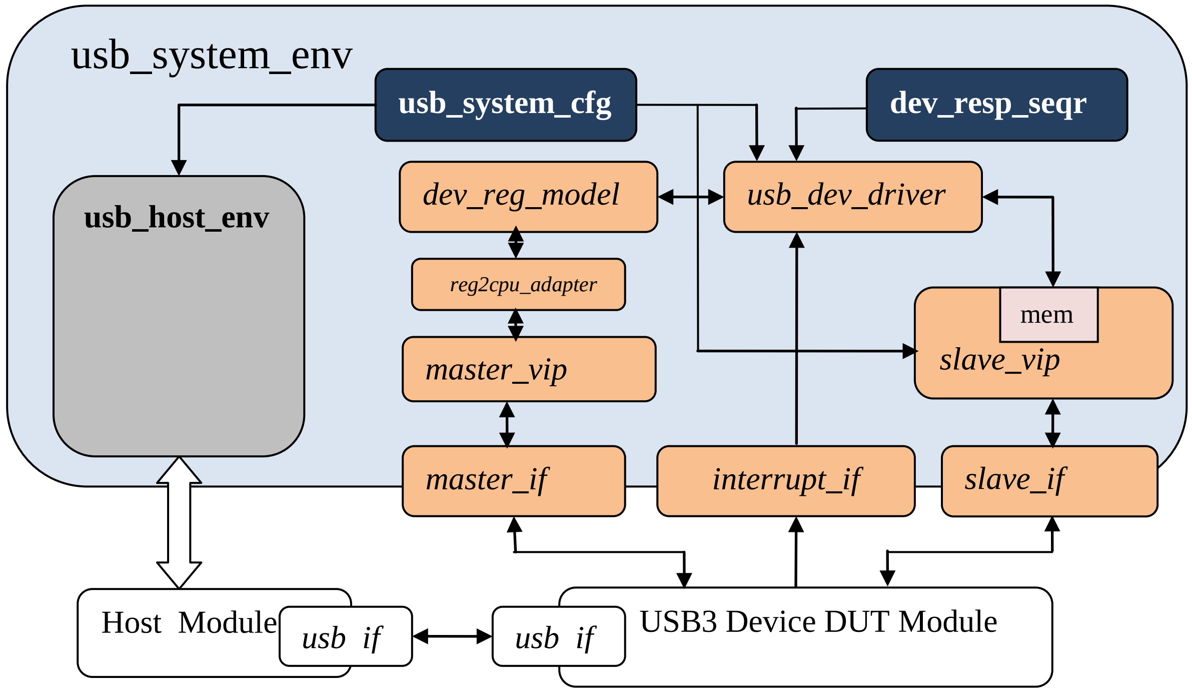

Figure 4-4 System Diagram of the USB3 Device DUT verification environment

The structure shown in Figure 4-4 can be reused by the system of a different USB3 Device DUT. However, the implementation of every block on the Device DUT side in Figure 4-4 needs to be evaluated to identify what needs to be changed. Below are the brief descriptions on what each block is depending.

- *usb_system_cfg* : Need DUT specific and VIP specific configuration information
- *dev_resp_seqr* : Need system specific request/response data structure information
- *dev_reg_model* : Need DUT specific register model
- *usb_dev_driver* : Depend on DUT's functionality and the *dev_reg_model*
- *reg2cpu_adapter* : Need DUT specific CPU/Memory interface information
- *master_vip, slave_vip, master_if, interrupt_if, slave_if* : Need DUT specific information

The example environment discussed in sections 2 and 3 uses DesignWare USB3 VIP as the Host side of the environment. For the systems that also employ the DesignWare USB3 VIP as the Host, the Host side implementation of the example environment may be reusable.

In summary, when building an environment for a system whose USB3 device DUT is not a DWC_usb3 Device Controller, the testbench module and the upper level environment structure described for the example environment can still be used. However, the implementation of each component on the device side will need to be revised based on the DUT's functionality and the interfaces used. The implementation provided by the example environment can be used as references.

4.3 Encapsulating Device Side Components in a Sub-Env

To make the upper level of the example environment more modular to further improve its reusability, a sub-environment class can be created to encapsulate the components on the device side of the system. Figure 4-5 shows the sub-environment around the DWC_usb3 Device Controller DUT, *usb_dev_env*.

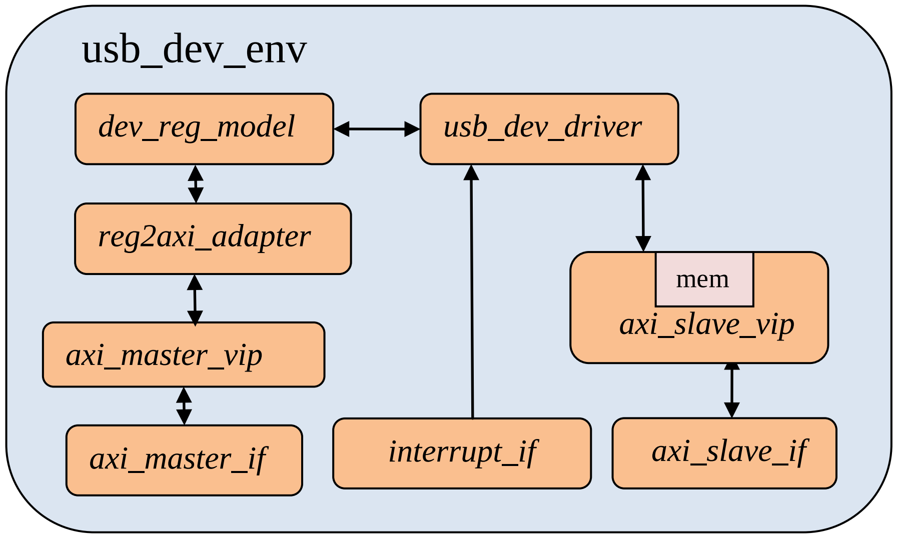

Figure 4-5 Sub-environment around the DWC_usb3 Device Controller DUT

By encapsulating the discrete components into the sub-environment, the system environment in Figure 4-1 can be redrawn as Figure 4-6.

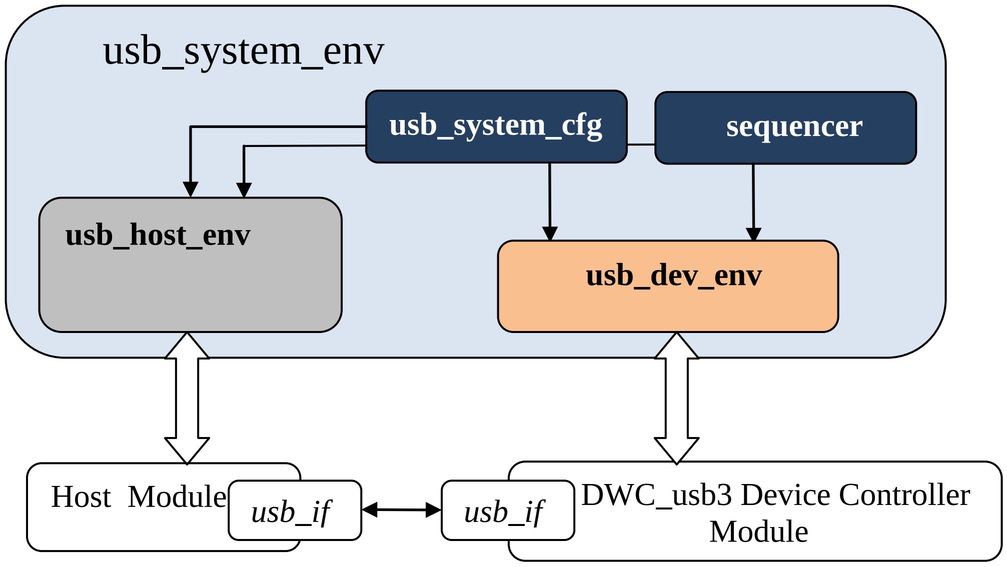

Figure 4-6 The USB3 System Environment consisting of the Device Sub-Env

The following steps are the top-down process for integrating a USB3 Device DUT into a UVM-based system environment using the device sub-environment.

Assuming the testbench module shown in Figure 4-4 has been constructed for the system.

Step 1: Create the *usb_system_env* by extending **uvm_env**.

- The *usb_system_env* consists of

- o The environment configuration descriptor - *usb_system_cfg*,
- o Host side sub-environment – *usb_host_env*
- o Device side sub-environment – *usb_dev_env*
- o The environment level sequencer which contains sub-sequencers for sub-environments.
- o Interfaces and scoreboards can also be instantiated

- *usb_system_env::***build_phase** constructs the scoreboards, sets the interfaces and configurations for the sub-environments using **uvm_config_db::set()**.
- *usb_system_env::***connect_phase** connects the sub-sequencers and the scoreboards to their corresponding sub-environments.

Step 2: Create the *usb_dev_env* sub-environment as shown in Figure 4-5.

Step 3: Assuming the VIPs and *reg2axi_adapter* in Figure 4-5 are provided by vendors, users need to create DUT specific register model and the corresponding *usb_dev_driver*. The implementation detail of the *usb_dev_driver* can refer to Figure 3-1 and section 3.5.

Once above steps are completed, the *usb_system_env* can be instantiated in a test class. The test class configures the *usb_system_env* and sets up the default sequence to run.

5. Conclusions

This paper goes through the details of integrating the DWC_usb3 Device Controller into a UVM-based verification environment. The reusability of the example environment is discussed so that testbench developers for systems that use DWC_usb3 Device Controller DUT or different USB3 Device DUT can evaluate their system requirements and determine how to make use of the information presented in this paper.

6. References

[1] DesignWare Cores SuperSpeed USB3.0 Controller Databook, *version 1.91a*

[2] UVM Reference Manual

[3]