Leverage Real USB Devices for USB Host DUT verification

Suchir Gupta Amit Sharma
Synopsys Inc Synopsys Inc
suchiir@synopsys.com amits@synopsys.com
Sunnyvale, CA, USA Sunnyvale, CA, USA

Abstract – In Transaction based verification, virtual transactors are leveraged to model real-world peripherals. Though the Transactor provides the Bus Functional Model interfaces, typically user-level testbenches are required to be created to model the actual traffic over that protocol interface. This is not a trivial task, and the complexity increases substantially for advanced protocols such as USB and PCIe. These complex software testbenches control and program the Transactor that interacts with a Design Under Test (DUT) in simulation or emulation domain. This paper explores a novel method how a real USB device (enabling real world stimulus) can be interfaced with the USB Device Transactor verifying a USB Host DUT so that a user-level testbench does not have to be written, and at the same time we can leverage the full benefits of Transactor based verification.

I. INTRODUCTION

A Transactor is a transaction-based verification solution that is developed for verifying Design Under Test (DUT). A verification engineer creates a user-level testbench that would instantiate a specific protocol transactor and would leverage the transactor APIs to drive stimulus to the DUT and respond to requests from DUT.

Universal Serial Bus (USB) provides an interface to connect multiple peripheral devices (USB Device) with personal computer (USB Host). A USB Device is characterized by its class code, sub class code, and protocol code. During enumeration, USB Host reads USB Device characteristics. If the USB Host supports USB device functionality it loads host-side device drivers needed for communicating with the device. As of today, USB supports 21 device classes, each device class supports multiple device sub classes, and each sub class supports multiple protocols.

A verification engineer verifying a host-side device driver needs to develop a device testbench that fully implements peripheral device's functionality, which is a complex task. Here we look at how we can have a virtual solution that enables a USB Device Transactor to interface with a real USB Device to avoid the complex task of creating multiple device testbenches while ensuring all the different use cases can be validated. The solution leverages Linux USB host subsystem to create a virtual adapter that can connect USB Device Transactor software with a real USB device. Currently the solution is developed for USB mass storage device that supports Bulk-Only-Transport (BBB) [3] protocol. The solution can work in both emulation and simulation domain. This approach can potentially be extended to leverage other protocol transactors to connect to respective real-world interfaces.

II. USB DEVICE TRANSACTOR

The USB Device Transactor would typically consist of a hardware and a software component. The hardware is written in Synthesizable SystemVerilog (SV) while software is written in C/C++ language. The hardware part of the transactor connects to its corresponding software part through SV direct programming interface (SV DPI) [7]. This communication across the HW and SW boundaries is at a transaction level leveraging SV DPI. This also ensures that the USB Device Transactor can run in both simulation and emulation.

The USB Device Transactor software in our framework consists of two parts – C library and a device testbench. USB Device Transactor C library has DPI functions and user-level APIs. The device testbench calls user-level APIs to configure USB Device Transactor and register user-level C callback functions for different USB requests. Since most of the USB requests require user inputs, hardware part of USB Device Transactor passes USB requests to transactor C library via SV DPI functions. The C library invokes a registered user callback C functions in the device testbench. Device testbench understands USB requests, prepares response, and queues them back to USB Host DUT via USB Device Transactor. A USB request can be a standard descriptor request or any device class specific request.

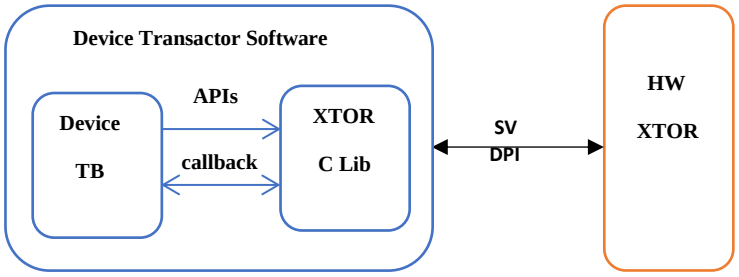

Figure 1. Device Transactor (XTOR) testbench

III. USB SUPPORT IN LINUX

It is important to understand the USB support in Linux OS, and the available and relevant Open-Source libraries that helps interface with device drivers to define how a "Virtual Device Transactor" can be created.

The Linux operating system (OS) provides support for both USB host and USB device. The Linux USB host subsystem consists of three layers of drivers – host controller driver (HCD), `usbcore`, and device drivers. The HCD and `usbcore` operate in kernel space while a device driver can operate either in kernel-space or in user-space. A device driver, which is the top- layer in USB stack, controls the attached USB device via host OS.

Linux also provides a virtual file system switch (VFS) software that provides the filesystem interface to user space programs, allowing them to access kernel internals. `usbcore` creates USB VFS `usbfs` through which it exports kernel data structure and attached USB device information to user applications. A device driver operating in user space generally use files in USB VFS `usbfs` to interact with attached USB devices. Fig. 2 shows Linux USB file system connecting user space application in user space and USB framework in kernel space.

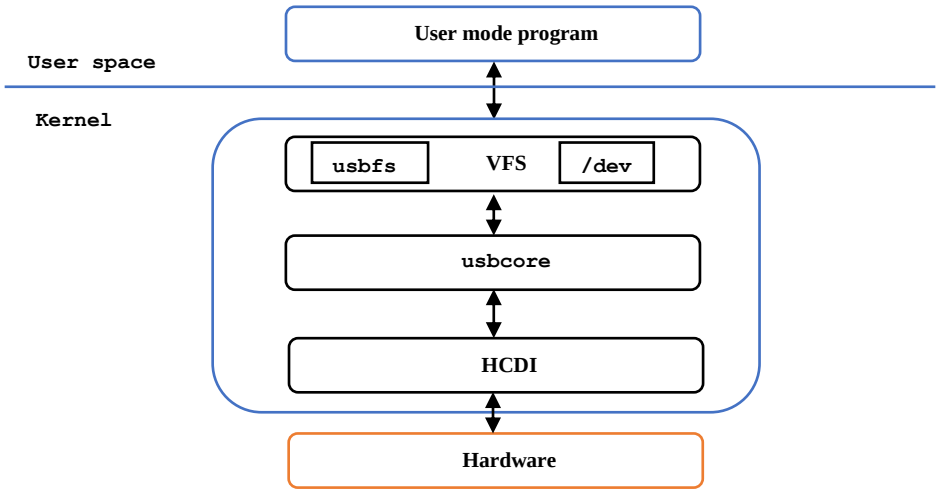

Figure 2. Linux file system connecting user application with Linux USB framework.

`libusb` is an open-source host-side C library [5] that provides application programming interfaces (APIs) for device driver development in user space. The APIs are cross-platform and user-mode. The `libusb` library is a wrapper that encapsulates OS level details and USB protocol complexities from device driver developers. In Linux platform, the `libusb` library access kernel data structure and attached USB device information through Linux USB filesystem file `usbfs`. Fig. 3 shows `libusb` interaction with user application and Linux USB file system.

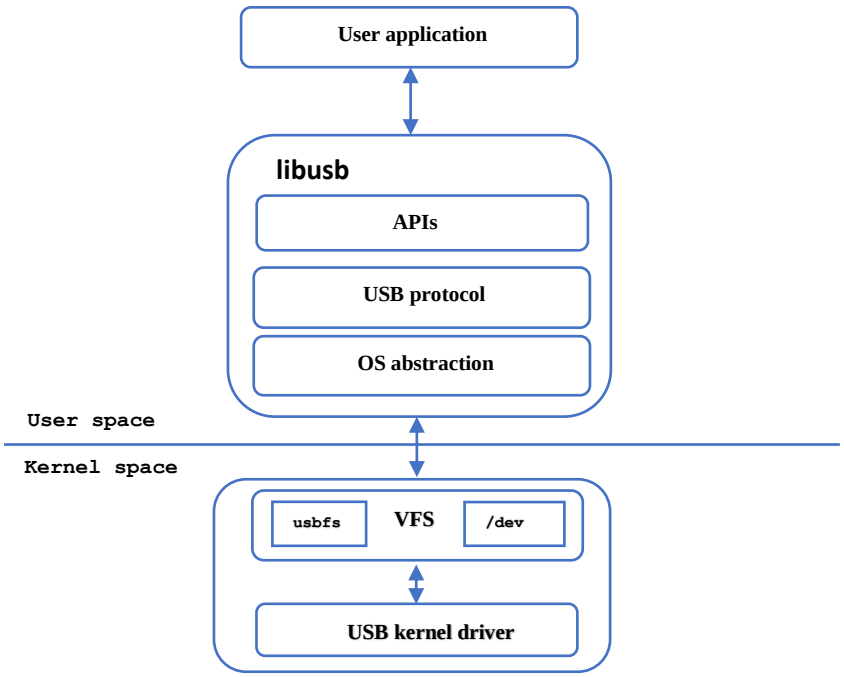

Figure 3. Host-side C library interconnecting user application with USB filesystem.

IV. The USB Device Virtual Solution

To enable the functionality to leverage the transactor to interface with real world devices, we highlight the approach we have taken and the overall architecture. We have leveraged Linux USB framework and `libusb` to create a USB Device Virtual Solution. The solution acts as a bridge between real USB mass storage device and USB Device Transactor. The USB Device Transactor C library exports USB requests from USB Device Transactor hardware to the Virtual Adapter. The Virtual Adapter is a user space device driver that processes the information, communicates with real USB device through `libusb` and provides USB response back to USB Device Transactor hardware.

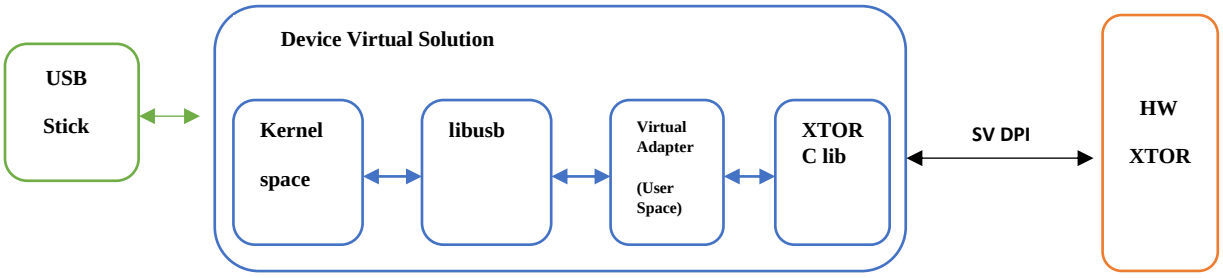

Figure 4. Device virtual solution connecting USB mass storage device and Device Transactor

For OUT transfers, the data bytes are stored in attached real USB device, while for IN transfers, the data bytes are fetched from attached real USB device. Fig. 4 showcases device virtual adapter that links USB device transactor C library and real mass storage device connected to Host PC.

A. Virtual Adapter

This section describes how Virtual Adapter (VA) interfaces with USB Device Transactor C library on one side and libusb on the other side. Users can adopt the similar procedure to develop a VA that interface with user's BFM on one side and libusb on the other side. User's VA interface to `libusb` should be analogues to the VA interface described here, but its interface to user's BFM can be different. Since such a USB Device transactor needs to support both emulation and simulation, we have employed SV DPI interface to communicate across transactor's SW and HW components. For pure simulation, other mechanism such as SV interface can also be used to connect SW and HW parts. The VA consists of four parts – initialization, enumeration, and data transfers, and destruction.

1. Initialization –

a. Create and configure USB Device Transactor object or user device's SW.
b. Initialize `libusb` library using `libusb_init()`. This API creates data structures and must be called before calling any other function.

```
#include "libusb.h"
libusb_context* m_libusbContext;
ret = libusb_init(m_libusbContext);
```

Figure 5. `libusb` initialization.

c. Connect `libusb` library with real USB device using `libusb_open_device_with_vid_pid`. User can get vendor ID and product ID using `lsusb` command on Linux terminal. `lsusb` lists all the USB devices attached to the USB Host. On successful connection with real USB device, we get real USB device context corresponding to USB device. The context is used for all future interactions with the real USB device. There are other APIs as well which user can use to connect `libusb` with attached real USB device.

```
libusb_device_handle*  m_libusbHandle = 0;
m_libusbHandle = libusb_open_device_with_vid_pid(m_libusbContext, VENDORID,
PRODUCTID);
```

Figure 6. `libusb opening a USB device.`

2. USB Device Enumeration –
   a. Once link is established between USB Host DUT and USB Device Transactor, USB Host DUT device driver will enumerate the USB device through USB control transfers.
   b. VA will get these USB control transfers from Host DUT via Device Transactor software and invoke corresponding `libusb_*_descriptor` () APIs to retrieve all descriptors from real USB device. VA provides vast set of APIs [5] to get various device descriptors from attached USB device. Fig. 15 shows few examples how VA gets descriptors from real device using libusb APIs.

```
struct libusb_device_descriptor desc;
ret = libusb_get_device_descriptor(libusb_get_device(m_libusbHandle)
      ,&desc);

struct libusb_bos_descriptor * bosDesc;
ret = libusb_get_bos_descriptor(m_libusbHandle, &bosDesc);
```

Figure 7. `libusb` APIs getting USB device and BOS descriptor from attached USB device.

   c. Once USB Host DUT device driver has received all the descriptors, it will issue SET CONFIGURATION command, marking end of successful enumeration. VA issues `libusb_set_configuration`() and `libusb_claim_interface()` to set configuration and interface within real USB device. Fig. 8 shows VA setting configuration and interface in real device using libusb APIs.

```
ret = libusb_set_configuration(virtual_usb_device_imp_svs::
                               m_libusbHandle, config);
ret = libusb_claim_interface(m_libusbHandle, intf);
```

Figure 8. `libusb` API setting configuration in real USB device.

3. USB transfers –
   a. `libusb` provides two interfaces - synchronous and asynchronous - for USB data transfers. Synchronous interface is blocking, while asynchronous interface is non-blocking.
   b. When Host DUT sends a USB transfer, Device Transactor hardware relays the USB transfer information to VA using Device Transactor software. VA uses `libusb` asynchronous interface to

perform USB transfer operation with real attached USB device. The USB transfer through asynchronous transfer consists of five stages –

i. Create a callback function that gets called when a USB transfer is complete.
ii. Create a `libusb` transfer struct
iii. Fill `libusb` transfer struct. This step also registers the callback that needs to be called when USB transfer is complete.
iv. Submit the filled `libusb` transfer struct to libusb.
v. Once the USB transfer is complete, `libusb` invokes the callback indicating the status of USB transfer.

c. Fig. 9 shows how VA issues Bulk-Out transfer is performed. VA creates complete callback function `CmpltOutCB`. Once VA gets data bytes from USB Host DUT, it invokes `libusb_fill_bulk_transfer()` and `libusb_submit_transfer ()` to fill data information and submit the transfer. Once USB transfer was complete, `libusb` would call completion callback `CmpltOutCB` indicating USB transfer status. If `libusb` was able to successfully store data bytes in attached USB real device, status would be zero. If the USB transfer did not complete within timeout, VA cancels the transfer, and issues USB Stall command back to USB Host DUT. Similarly, USB Bulk-In transfers are also performed.

```
//USB transfer complete callback function.
int completed = 0;
static void libCmpltOutCB(struct libusb_transfer *transfer){
  else if (transfer->status != LIBUSB_TRANSFER_COMPLETED) {
    printf("libusb %s OUT transfer status %d\n"), transfer->status);
  }
  completed = true;
}
//Allocate USB transfer.
struct libusb_transfer* transfer = libusb_alloc_transfer(0);
//Fill USB transfer.
libusb_fill_bulk_transfer(transfer,
                          m_libusbHandle,
                          LIBUSB_ENDPOINT_OUT, //Target Out EP.
                          buf,                 //Data buffer
                          length,              //Length of data buffer.
                          libCmpltOutCB,       //Completion callback.
                          ep_numvusb,          //EP number
                          m_libusb_timeout);

//Submit USB transfer
int status = libusb_submit_transfer(transfer);
//Wait for completion of USB transfer.
while (!completed) {
  struct timeval tv;
  tv.tv_sec = 0;
  tv.tv_usec = 1;
  status = libusb_handle_events_timeout_completedm_libusbContext, &tv,
  completed);
  if (status < 0) {
    libusb_cancel_transfer(transfer);
    ibusb_free_transfer(transfer);
  }
}
```

Figure 9. `libusb` API setting configuration in real USB device.

4. Destructor –

a. Once the testbench has called VA's destructor, call `libusb_close()`. This will destroy all the handles corresponding to `libusb`.

```
libusb_close(m_libusbHandle);
```

Figure 10. `libusb` API closing USB real device.

VI. USB Host DUT Verification with USB Device Virtual Solution

Now that we have the USB Virtual Device Transactor Solutions, we will describe how this can be leveraged to verify a USB Host DUT and how such a verification environment needs to be created

The Host DUT verification environment would consist of a Host DUT, Device Transactor, Real USB device, and a Device Virtual Solution as primary components amongst others. For demonstration, Host DUT software and hardware have been replaced with Host Transactor (XTOR).

In emulation domain, Host DUT and Device transactor hardware parts are integrated, compiled, and synthesized on emulator, while their corresponding software parts execute on Host PC. The real USB device is attached to Host PC. Fig. 11 shows interconnection among different components in Host DUT verification environment in emulation domain.

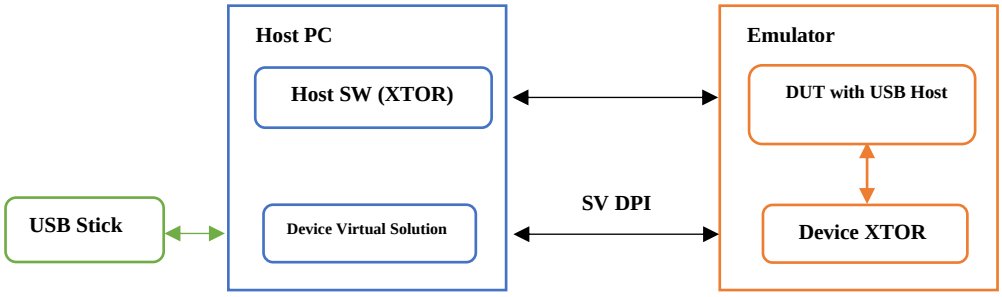

Figure 11. Host DUT (XTOR) verification environment with Device Virtual Solution in emulation domain.

In the simulation domain, hardware and software parts of both Host DUT and Device Transactor execute on Host PC. The real USB device is attached to Host PC. Fig. 12 shows interconnection among different components in Host DUT verification environment in simulation domain.

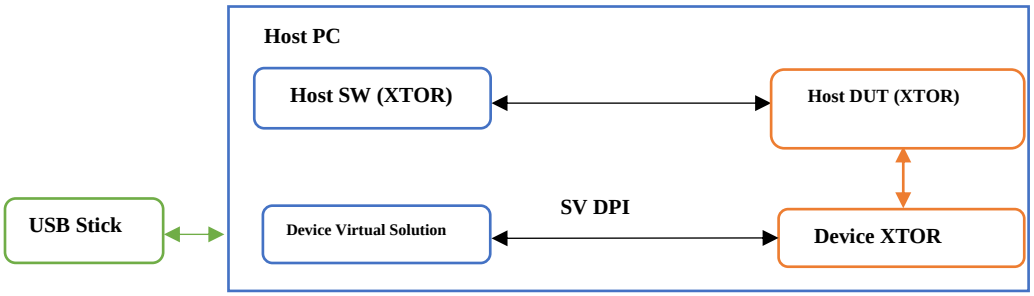

Figure 12. Host DUT (XTOR) verification environment with Device Virtual Solution in simulation domain.

A. Verification Environment setup

Following is the code flow for USB Host DUT (XTOR) verification environment.

1. Testbench Environment Settings –
    a. Connect a real USB mass storage device in Host PC.
    b. Obtain USB real device vendor ID and product ID. Linux command `lsusb` can be used to get attached devices' vendor ID and product ID.
2. Host DUT – Device Transactor Testbench

a. Integrate USB Host DUT and Device Transactor hardware in the user environment.
b. Compile the Host DUT and Device Transactor.
c. Create Host DUT software that will enable Host DUT.
d. Wait for link up between Host DUT and Device transactor.
e. Initiate USB data transfers from Host DUT.

3. Device Transactor Testbench
   a. The APIs mentioned in this section are specific to Device virtual solution. These APIs encapsulate `libusb` APIs. These APIs would be different for users developing their own virtual solution. They are showcased here to highlight that device testbench logic would be minimal when `libusb` and real USB device are leveraged to verify Host DUT.
   b. Create and configure USB Device Virtual Solution object. At this point, VA will initialize `libusb`.
   c. Connect USB Device Virtual Solution. At this point USB Host DUT (Transactor) will be able to detect USB Device and establish connection with it. During the execution of this API, VA will open real USB device and will be able to send response for all control transfers.
   d. Run USB Device Virtual Solution in a continuous loop. Whenever USB Device Virtual Solution hardware wants to interact with its software, it raises an interrupt. The interrupt is generally raised when hardware receives a USB request such as descriptor or USB transfer from USB Host DUT. With every loop call, USB Device Virtual Solution software checks whether hardware raised an interrupt. If the interrupt is raised, Device Virtual Solution software process the interrupt, calls `libusb` APIs to interact with attached real USB device. The loop should be called continuously in a loop for the full duration of test.

```
//Testbench code creating and configuring USB Device Virtual Solution.
usbDev = new virtual_usb_device_svs("xtor_usb_svs" , "hw_top.Device");

//Testbench code connecting Device Virtual Solution with Host DUT.
if(usbDev->USBPlug()) {
  printf("Virtual Device TB : Cannot connect the device.");
  return NULL;
}

//Testbench code running Device Virtual Solution in a loop.
do {
  usbDev->Loop();
}
while(tbDone != 1);
```

Figure 13. Testbench code running Device Virtual Solution.

   e. Device Virtual Solution will keep responding to Host DUT USB requests.

B. Verification Environment Results

A High speed (HS) USB mass storage device was inserted in Host PC. The real USB device supported full SCSI transparent command set using the BBB protocol. Device Virtual Solution established connection with USB mass storage device using device's vendor ID and product ID. After USB connection between USB Host Transactor and USB Device Transactor, USB Host Transactor testbench initialized and enumerated USB mass storage device. Post enumeration, USB Host Transactor ran few SCSI commands, and transferred and received 1800K bytes from USB mass storage device. Responses for all USB requests – USB Control and Bulk transfers -- came directly from USB mass storage device. Apart from initialization and running a service loop, verification engineer did not have to write any code logic in device testbench for generating responses. The test at HS speed completed in 5702 seconds in simulation, whereas it took 23 seconds in emulation, making it approximately 248 times faster in emulation. Similarly, USB Host Transactor was verified for super speed (SS) by replacing HS device with SS device. In SS speed, 3600K bytes were transferred. The test at SS speed completed in 4815 seconds in simulation, whereas it took 16 seconds in emulation, making it approximately 301 times faster in emulation.

VI Future Work

The current Device Virtual solution has been developed for mass storage device following the BBB protocol. Since libusb provides a generic framework for user mode USB driver development, the solution can be enhanced to verify Host DUT USB for other 21 device classes, each device class supporting multiple device sub classes, and each sub class supporting multiple protocols. A Similar approach can be taken up to enable interfacing with other real-world devices for the respective protocols' interfaces. Timers etc. can be configured if necessary to address slow simulation performance and timeouts when this solution is deployed in simulation for specific devices

VIII Conclusion

This paper shows how a USB host device driver and USB host DUT can be verified with real USB device using Device Virtual Solution in both simulation and emulation environments. Specific results using a real USB device were enumerated. Once basic Host DUT functionality is verified using traditional mechanisms, such a solution can easily cover multiple system level use cases easily, and effectively and helps the DUT respond to real world stimulus. In this case, verification teams using such a virtual solution does not need to write complex testbench and testcases. The paper presents a crisp set of steps on how users can develop their own virtual device solution using available open-Source libraries like `libusb`.

IX References

[1] USB Implementors forum, "Mass Storage Specification Overview", Version 1.4, 2010. Available: www.usb.org

[2] R. Regupathy, "Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB," 1st edition, Cengage Learning PTR; 2011.

[3] libusb, "libusb: A cross-platform library for USB device access," Available: https://libusb.info/

[4] Synopsys Inc, "ZeBu Virtual USB Bridge svs," Ver. Q-2020.06, Available: www.synopsys.com

[5] IEEE Standard Association, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," IEEE Standard 1800-2017, Dec. 2017 Available: https://standards.ieee.org