Verification Continuum™

ZeBu® USB Host

User Guide

Version Q-2020.06, August 2020

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com

Contents

. Overview 17
. Related Documentation 17
1.1. Overview 20
1.2. USB Interfaces Compliance 21
1.3. API Layers 22
1.4. Features 22
1.4.1. List of Features 22
1.4.2. List of Integration Components 22
1.4.3. APIs 23
1.5. Requirements 24
1.5.1. FLEXlm License Features 24
1.5.2. ZeBu Software Compatibility 24
1.5.3. Knowledge 24
1.5.4. Software 25
1.6. Performance 26
1.6.1. Serial Interface 26
1.6.2. UTMI Interface 26
1.7. Limitations 27
1.7.1. USB Features 27
1.7.2. USB Transactor API 27
1.7.2.1. USB Channel API 27
1.7.2.2. URB API 27
1.7.3. USB Log 27
2.1. Installing the USB Host Transactor Package 30
2.2. Package Description 31
2.3. File Tree 32
3.1. Architecture with the UTMI DUT Interface 36
3.2. Architecture with the ULPI DUT Interface 37
3.3. Architecture with the USB Cable DUT Interface 38
4.1. Interface Overview 40
4.1.1. UTMI Interface 40
4.1.2. DP/DM Serial Interface 43
4.1.3. ULPI Interface 43

4.2. Connecting the Transactor's Clocks 44
4.2.1. Overview 44
4.2.2. Signal List 44
4.2.3. Verification Environment Example 45
4.2.3.1. hw_top File Example 45
4.2.3.2. Defining the Clocks in the designFeatures File 47
4.3. USB Cable Models 48
4.3.1. PHY UTMI Cable Model 48
4.3.2. PHY ULPI Cable Model 51
4.3.3. PHY Serial Cable Model 55
4.3.4. Connection/Disconnection to the USB Device of the DUT 57
4.3.5. PHY UTMI Interface Checker 58
4.3.5.1. Description 59
4.3.5.2. Interface 59
4.3.5.3. Advanced Debugging 62
4.4. USB Bus Logging 63
4.4.1. Verification Environment for the USB Bus Logging Feature 63
4.4.1.1. Adding a Logging Block in the DUT 64
4.4.1.2. The usb_monitor is a SystemVerilog module, which is available in the misc directory. Compiling with zCui64
4.4.2. Bus Logging Control 64
5.1. USB Channel API 66
5.2. USB Request Block (URB) API 67
6.1. Using the USB Host Channel API 69
6.1.1. Libraries 69
6.1.2. Data Alignment 69
6.2. USB Host Channel API Class Description 70
6.3. UsbHost Class 71
6.3.1. Description 71
6.3.2. Transactor Initialization and Control Methods 73
6.3.2.1. Constructor and Destructor Methods 73
6.3.2.2. init() Method 74
6.3.2.3. reInit() Method 74
6.3.2.4. config() Method 74
6.3.2.5. loop() Method 75
6.3.2.6. delay() Method 75
6.3.2.7. registerCallBack() Method 76
6.3.2.8. log() Method 76
6.3.2.9. setLogPrefix() Method 77
6.3.2.10. getVersion() Method 77

6.3.3. Channel Management Methods 78
6.3.3.1. channel() Method 78
6.3.3.2. **releaseChannel() Method 78**
6.3.4. Channel Operation Methods 78
6.3.4.1. hcInit() Method 78
6.3.4.2. hcHalt() Method 79
6.3.4.3. sendSetupPkt() Method 79
6.3.4.4. requestData() Method 80
6.3.4.5. sendStatusPkt() Method 80
6.3.4.6. sendData() Method 81
6.3.4.7. rcvData() Method 81
6.3.4.8. enableReception() Method 82
6.3.4.9. actualXferLength() Method 82
6.3.4.10. waitStatusPkt() Method 82
6.3.5. USB Operations 82
6.3.5.1. USBPlug() Method 83
6.3.5.2. USBUnplug() Method 83
6.3.5.3. usbReset() Method 83
6.3.6. USB Cable Model Status Methods 84
6.3.6.1. portEnabled() Method 84
6.3.6.2. isDeviceAttached() Method 84
6.3.7. USB Protocol Information 84
6.3.7.1. portSpeed() Method 84
6.3.7.2. maxPktSize() Method 84
6.3.7.3. isPingSupported() Method 84
6.3.8. General Purpose Vectors Operations 85
6.3.8.1. writeUserIO() Method 85
6.3.8.2. readUserIO() Method 85
6.4. HostChannel Class 86
6.4.1. Description 86
6.4.2. Detailed Methods 87
6.4.2.1. getChNumber() Method 87
6.4.2.2. dataPresent() Method 87
6.4.2.3. xferComplete() Method 87
6.4.2.4. errorHappened() Method 87
6.4.2.5. Halted() Method 88
6.4.2.6. Stall() Method 88
6.4.2.7. NAK() Method 88
6.4.2.8. ACK() Method 88
6.4.2.9. NYET() Method 88
6.4.2.10. xactError() Method 89

6.4.2.11. babbleError() Method 89
6.4.2.12. frameOverrun() Method 89
6.4.2.13. display() Method 89
6.5. Logging USB Packets Processed by the Transactor 90
6.5.1. USB Packet Processing Logs 90
6.5.2. Log Types 90
6.5.2.1. Host Controller Log 90
6.5.2.2. USB Transactor Log 90
6.5.2.3. Reference Clock Time Log 90
6.5.2.4. Full Logs 91
6.5.3. USB Log Example 91
6.6. Example 93
6.6.1. Host Process of the Testbench 93
6.6.2. USB Host Testbench Example 94
7.1. Using the URB API 100
7.1.1. Libraries 100
7.1.2. Data Alignment 100
7.2. URB API Class and Structure Description 102
7.3. UsbHost Class 103
7.3.1. Description 103
7.3.2. Transactor Initialization and Control Methods 106
7.3.2.1. Constructor/Destructor Methods 106
7.3.2.2. `Init()` Method 106
7.3.2.3. `InitBFM()` Method 107
7.3.2.4. `Loop()` Method 108
7.3.2.5. `DiscoverDevice()` Method 108
7.3.2.6. `Delay()` Method 108
7.3.2.7. `UDelay()` Method 109
7.3.2.8. `RegisterCallBack()` Method 109
7.3.2.9. `SetDebugLevel()` Method 110
7.3.2.10. `SetLogPrefix()` Method 110
7.3.2.11. `SetLog()` Method 111
7.3.2.12. `SetTimeOut()` Method 111
7.3.2.13. `RegisterTimeOutCB()` Method 111
7.3.2.14. `getVersion()` Method 112
7.3.3. USB Device Information Methods 112
7.3.3.1. `DevicePresent()` Method 112
7.3.3.2. `GetDeviceSpeed()` Method 113
7.3.3.3. `GetMaxPacketSize()` Method 113
7.3.4. USB Operations 113

7.3.4.1. USBPlug() Method 113
7.3.4.2. USBUnplug() Method 113
7.3.5. USB Device Configuration Management Methods 114
7.3.5.1. SetDeviceAddress() Method 114
7.3.5.2. GetDeviceAddress() Method 114
7.3.5.3. SetDeviceConfig() Method 114
7.3.5.4. GetDeviceConfig() Method 115
7.3.5.5. GetRawConfiguration() Method 115
7.3.5.6. SetDeviceInterface() Method 115
7.3.5.7. GetDeviceAltInterface() Method 116
7.3.6. General Purpose Vectors Operations 116
7.3.6.1. writeUserIO() Method 116
7.3.6.2. readUserIO() Method 116
7.3.7. Advanced User Methods 116
7.3.7.1. usbReset() Method 116
7.3.7.2. portEnabled() Method 117
7.3.7.3. isDeviceConnected() Method 117
7.3.7.4. BypassHubReset() Method 117
7.3.7.5. SetFastTimer() Method 117
7.3.7.6. Example 118
7.4. ZebuUrb Structure 119
7.5. URB API Enum Definitions 122
7.5.1. zusb_status Enum 122
7.5.2. zusb_transfer_status Enum 122
7.6. Managing USB Host URBs 124
7.6.1. Creating and Destroying ZeBu URBs 124
7.6.1.1. AllocUrb() Method 124
7.6.1.2. FreeUrb() Method 124
7.6.1.3. AllocBuffer() Method 125
7.6.1.4. FreeBuffer() Method 125
7.6.1.5. FillBulkUrb() Method 126
7.6.1.6. FillIntUrb() Method 126
7.6.1.7. FillControlUrb() Method 127
7.6.1.8. FillIsoUrb() Method 128
7.6.1.9. FillIsoPacket() Method 129
7.6.2. Submitting ZeBu URBs 130
7.6.3. Cancelling ZeBu URBs 130
7.6.4. USB Transfers without ZeBu URBs 130
7.6.4.1. SendControlMessage() Method 131
7.6.4.2. SendBulkMessage() Method 132

7.6.5. Examples 132
7.6.5.1. Bulk URB Transfer with ZeBu URBs 133
7.6.5.2. Bulk URB Transfer without ZeBu URBs 134
7.6.5.3. Isochronous URB Transfer 135
8.1. Description 139
8.2. URB API Methods List 140
8.2.1. EnableWatchdog() Method 140
8.2.2. SetTimeOut() Method 140
8.2.3. RegisterTimeOutCB() Method 140
8.3. USB Host Transactor Default Behavior 142
8.4. Logging USB Transfers Processed by the Transactor 143
8.4.1. USB Requests Processing Logs 143
8.4.2. Log Types 143
8.4.2.1. Main Info Log 143
8.4.2.2. USB Request Log 143
8.4.2.3. Controller Core Log 143
8.4.2.4. Data Buffer Log 144
8.4.2.5. Reference Clock Time Log 144
8.4.3. USB Requests Log Example 144
8.5. Using the USB Host Transactor 146
8.5.1. Initializing the USB Host Transactor 146
8.5.1.1. Initializing the ZeBu Board and USB Host Transactor 146
8.5.1.2. Initializing the Transactor Host Controller 146
8.5.1.3. Connecting the USB Host Transactor to the Cable 147
8.5.1.4. USB Device Discovery 147
8.5.2. Managing USB Device Configuration and Interfaces 148
8.5.2.1. Managing the Device Address 148
8.5.2.2. Managing the Device Configuration 149
8.5.2.3. Managing the Device Interface 149
8.5.2.4. Example of Configuration Parsing 149
8.5.2.5. Disconnecting the USB Host Transactor from the Cable 151
9.1. Prerequisites 153
9.1.1. USB Bus Log Hardware Instantiation 153
9.1.2. USB Host Transactor Initialization Constraint 153
9.2. Using the USB Logging Feature with USB Channel API 154
9.2.1. Starting and Stopping Log 154
9.2.2. Defining the Log File Name 154
9.2.3. Description of the USB Channel API Interface for USB Bus Logging 154
9.2.3.1. setBusMonFileName() Method 155
9.2.3.2. startBusMonitor() Method 155

9.2.3.3. stopBusMonitor() Method 156
9.3. Using the USB Logging Feature with URB API 157
9.3.1. Starting and Stopping Logging 157
9.3.2. Defining the Log File Name 157
9.3.3. Description of the URB API Interface for USB Bus Logging 157
9.3.3.1. SetBusMonFileName() Method 158
9.3.3.2. StartBusMonitor() Method 158
9.3.3.3. StopBusMonitor() Method 159
10.1. Description 161
10.2. Prerequisite 162
10.3. Using the URB Logging Feature 163
10.3.1. Starting and Stopping Logging 163
10.3.2. Defining the Log File Name 163
10.3.3. Description of the URB API Interface for URB Logging Feature 163
10.3.3.1. SetUrbMonFileName() Method 163
10.3.3.2. StartUrbMonitor() Method 164
10.3.3.3. StopUrbMonitor() Method 164
11.1. phy_utmi 168
11.1.1. File Tree 168
11.1.2. Overview 169
11.1.3. ZeBu Design 169
11.1.4. Software Testbench 170
11.2. phy_cable 171
11.2.1. File Tree 171
11.2.2. Overview 171
11.2.3. ZeBu Design 172
11.2.4. Software Testbench 172
11.3. Running the Examples 174
11.3.1. Setting the ZeBu Environment 174
11.3.2. Compiling the Designs 174
11.3.3. Running the phy_utmi Example 174
11.3.4. Running phy_cable Example 174

Related Documentation 17
ZeBu Compatibility 24
ZeBu Compatibility 26
UTMI Interface 26
UTMI Hardware Interface 40
UTMI Size Interface 42
DP/DM Serial Interface 43
USB Host Transactor Clock Signals List 45
PHY UTMI2UTMI Cable Interface (HostToDevice) 49
PHY UTMI2ULPI cable interface 52
PHY Serial Cable Interface Signals 56
PHY UTMI2UTMI Cable Checker Interface 60
USB Host Channel API Class Description 70
Types for UsbHost Class 71
Methods for UsbHost Class 71
75
Log levels for log method 77
Types for HostChannel Class 86
Methods for HostChannel Class 86
URB API Class description 102
URB API Structures description 102
URB Type 103
USB Structures 103
Methods for UsbHost Class 104
Log levels for SetDebugLevel Method 110
zusb_status Enum Description 122
zusb_transfer_status Enum Description 122
USB Host URB Creation and Destruction Methods 124
USB Host URB Filling Methods 125
Methods for Watchdogs and Timeout Detection 140
Methods for Bus Logging using Channel API 154
Bus Logging methods using URB API 157
ZeBu URB Logging API Methods 161

ZeBu Host Transactor Application 20
USB Host Transactor With the UTMI DUT Interface 36
USB Host Transactor with the ULPI DUT Interface 37
USB Host Transactor with the USB Cable DUT Interface 38
Connecting the USB Clocks 44
USB PHY UTMI2UTMI Cable 48
PHY UTMI2UTLPI Cable 51
PHY UTMI2ULPI Connection to Standard Device DUT 52
PHY Serial Cable 55
Device Connection/Disconnection Waveforms 58
Device Connection/Disconnection in the USB Bus Log 58
UTMI USB Cable Checker 59
Verification Environment for the USB Bus Logging Feature 64
USB Host Transactor Channel API 66
USB Host transactor URB API 67
Libraries Hierarchy Overview 100
phy_utmi example overview 169
phy_cable example overview 172

About This Manual

Overview

This manual describes how to use the ZeBu USB Host Transactor with your design being emulated in ZeBu.

Related Documentation

The following table lists the reference document names and their availability.

TABLE 1 Related Documentation

Document Name	Description
ZeBu Release Notes	Provides information about the ZeBu supported features and limitations. Available in the ZeBu documentation package corresponding to your software version.
Using Transactors	Provides relevant information about the usage of the present transactor. Available in the training material.
ZeBu USB Device Transactor User Manual	Provides information about USB Device transactor functioning.

1 Introduction

This section describes the ZeBu USB Host transactor and its features. In addition, this section provides an overview of the components required for using the USB Host transactor.

This section consists of the following sub-sections:

- *Overview*
- *USB Interfaces Compliance*
- *API Layers*
- *Features*
- *Performance*
- *Limitations*

1.1 Overview

The USB bus (Universal Serial Bus) is used for data transfers from a host to a device with bandwidths ranging from 1.5 to 480 Mb/s (Low-, Full-, and Hi-Speed devices). USB protocol is defined within 3 standards: USB 1.1, USB 2.0, and USB OTG (On-The-Go), which is an extension of USB 2.0 to interconnect peripheral devices.

The following figure illustrates the ZeBu Host Transactor application:

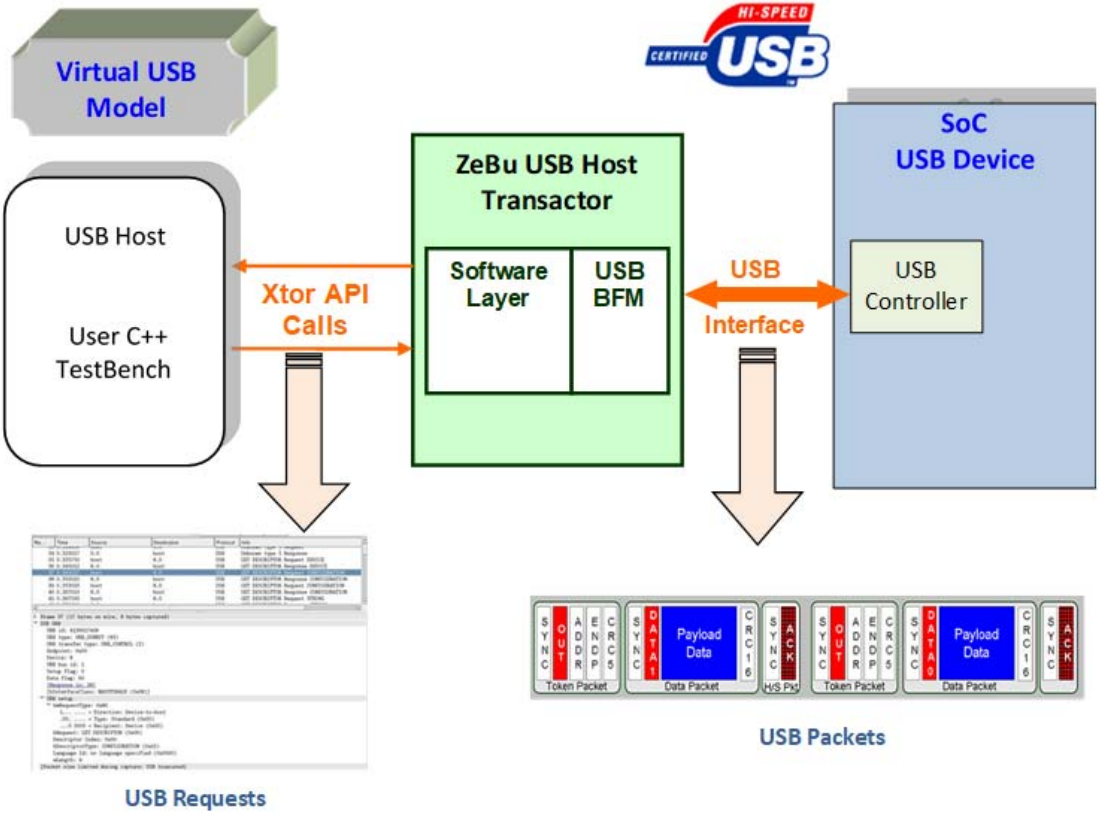

FIGURE 1. ZeBu Host Transactor Application

1.2 USB Interfaces Compliance

The ZeBu USB Host synthesizable transactor is compliant with USB 2.0 standard protocol. It implements the following USB interfaces:

- UTMI Level 3 interface (USB Transceiver Macrocell Interface).

  This is a UTMI parallel interface, coming in 8- and 16-bit data interfaces.
- ULPI interface (UTMI Low Pin Interface).
- USB differential links, also known as DP/DM Serial Interface.

For the transactor to work with the UTMI, ULPI, DP/DM interfaces, cable models should be implemented for a correct connection of the transactor with the DUT. The following cable models are provided in the transactor package:

- PHY UTMI2UTMI
- PHY UTMI2ULPI
- PHY CABLE

For more information on the DUT-Transactor integration, see *DUT-Transactor Integration with Cable Models*.

For a description of each cable model, see *USB Cable Models*.

1.3 API Layers

The USB Host transactor offers two levels of C++ Application Program Interface (API):

- **USB Channel level**: for transfer of USB packets for low-level USB software stack integration.
- **USB Request Block (URB) Level**: for USB driver integration at kernel structure level.

1.4 Features

1.4.1 List of Features

The ZeBu USB Host synthesizable transactor has the following features:

- USB 2.0 compliant.
- UTMI USB PHY DUT-side interface is compliant with UTMI+ Level 3.
- ULPI USB PHY DUT-side interface is compliant with ULPI revision 1.1.
- USB DP/DM cable DUT-side interface is compliant with USB 2.0 signaling.
- Controllable USB disconnection.
- Supports both USB Full-Speed (FS) and USB Hi-Speed (HS) modes.
- Configurable support of PING protocol.
- Supports BULK, Interrupt and Isochronous USB transfers.
- Supports up-to-512-byte USB packets.
- 16 USB channels available.

1.4.2 List of Integration Components

The ZeBu USB Host synthesizable transactor also provides the following components for integration with the user design:

- Synthesizable model for USB cable with:

  - UTMI Level 3 interface for transactor connection and UTMI Level 3 interface for DUT connection.
  - UTMI Level 3 interface for transactor connection and ULPI interface for DUT connection.
- Synthesizable model for USB cable with DP/DM cable serial interface.
- USB cable checker for UTMI interface.
- USB Bus log for UTMI and ULPI interfaces.

1.4.3 APIs

The USB Host transactor provides the following two levels of APIs with the following functions for its different API layers:

- Channel API features:
  - Control and Bulk USB transfer types.
- USB Request Block (URB) API features:
  - Control, Bulk, Interrupt, Isochronous OUT USB transfer types.
  - Device configuration and interface management.
  - Linux driver behavior for kernel version 2.6.18.
  - URB log.

1.5 FLEXlm License Features

You need the following FLEXlm license features for the USB Host Transactor.

- For ZS4 platform, the license feature used zip_USB2XtorZS4.
- For ZS3 platform, the license feature used is zip_USBHostXtor.

Note *If zip_USB2XtorZS4 license feature is not available, the zip_ZS4XtorMemBaseLib license feature is checked out.*

1.6 Performance

Performance is measured with a 3.3 GHz Linux PC with 64 GByte RAM and a ZeBu Server-3 system using software release V8_0_0.

The following tables provide information about the ZeBu USB Host transactor performance in various conditions.

1.6.1 Serial Interface

Tests in USB Full-Speed mode for Bulk OUT and Bulk IN transfers. The environment uses the transactor with a serial interface and a USB serial clock running at 15 MHz.

TABLE 2 ZeBu Compatibility

USB Full-Speed Mode Transfer Type	USB Host Transactor
BULK OUT	252 kbit/s
BULK IN	252 kbit/s

1.6.2 UTMI Interface

Tests in USB Hi-Speed mode for Bulk OUT and Bulk IN transfers. The environment uses the UTMI synthesizable model for the USB cable mapped in ZeBu, with a USB PHY clock running at 1.0625 MHz.

TABLE 3 UTMI Interface

USB High-Speed Mode Transfer Type	USB Host Transactor
BULK OUT	2.62 Mbit/s
BULK IN	2.00 Mbit/s

1.7 Limitations

1.7.1 USB Features

The following USB features are not supported:

- OTG mode is not supported.
- USB Low-Speed transfers are not supported.
- The transactor using the USB serial interface supports only the USB Full-Speed mode. For Full-Speed and Hi-Speed support, the use of the DP/DM cable model is mandatory.
- The UTMI PHY cable checker is only available for the 8-bit UTMI interface.
- The USB serial cable does not contain the connect/disconnect control signal available into the serial DP/DM resolution module.
- Only one packet is transmitted per microframe. High-bandwidth transactions (two or three packet per microframe) are not supported.

1.7.2 USB Transactor API

1.7.2.1 USB Channel API

Isochronous and interrupts transfers are not supported by the USB Channel API.

1.7.2.2 URB API

Only one URB Host transactor per Linux process is allowed.

1.7.3 USB Log

This feature requires the **zFAST** synthesis tool and is only available for ZeBu Server-1.

2 Installation

This section explains the process of installing the USB Host transactor. It covers the following topics:

- *Installing the USB Host Transactor Package*
- *Package Description*
- *File Tree*

2.1 Installing the USB Host Transactor Package

Prerequisites

Ensure that you have WRITE permissions to the IP directory and the current directory.

To install the USB Host transactor, perform the following steps:

1. Download the transactor compressed shell archive (.sh).
2. Install the USB Host transactor as follows:

```
$ sh USB.<version>.sh install [ZEBU_IP_ROOT]
```

where:

- [ZEBU_IP_ROOT] is the path to your ZeBu IP directory:
  - If no path is specified, the ZEBU_IP_ROOT environment variable is used automatically.
  - If the path is specified and a ZEBU_IP_ROOT environment variable is also set, the transactor is installed at the defined path and the environment variable is ignored.

The following message is displayed when the installation process is successfully completed:

```
USB v.<version> has been successfully installed.
```

An error message is displayed, if there is an error during the installation. For example:

```
ERROR: /auto/path/directory is not a valid directory.
```

2.2 Package Description

After the USB Host transactor is successfully installed, the package contains the following elements:

- .so shared library of the transactor API.
- Header files of API for the transactor.
- EDIF and NGO description for the transactor.
- Gate-level description for USB cable models used with the USB Host transactor:
  - EDIF file for the USB serial cable.
  - EDIF file for the UTMI cable.
  - EDIF file for the ULPI cable.
  - EDIF file for the UTMI PHY cable checker.
- Examples:
  - Channel API application with the USB Host transactor and the UTMI USB cable model mapped in ZeBu.
  - URB API application with the USB Host transactor and the UTMI USB cable model mapped in ZeBu.
  - Channel API application with USB Host transactor with UTMI interface connected through PHY cable.

2.3 File Tree

Here is the USB Host transactor file tree after package installation.

Note *The provided package contains the files for both the ZeBu USB Host and ZeBu USB Device transactors. Only the ZeBu USB Host transactor files are given in the file tree below.*

```
$ZEBU_IP_ROOT
 `-- XTOR
     `--USB.<version>
        |-- components
        |   |-- usb_driver_FS_Serial_Host.v
        |   `-- usb_driver_Utmi_Host.v
        |-- uc_xtor
        |   |-- usb_driver_FS_Serial_Host.v
        |   `-- usb_driver_Utmi_Host.v
        |-- doc
        |   |-- pdf
        |   |   |-- VSXTOR013_UM_USB_Host_Transactor_<revision>.pdf
        |   |   |-- VSXTOR013_USB_Host_URB_API_Reference_Manual.pdf
        |   |   `-- VSXTOR013_USB_Host_USB_API_Reference_Manual.pdf
        |   `-- html
        |   |   |-- VSXTOR013_USB_Host_URB_API_Reference_Manual.html
        |   |   |-- VSXTOR013_USB_Host_URB_API_Reference_Manual
        |   |   |-- VSXTOR013_USB_Host_USB_API_Reference_Manual.html
        |   |   `-- VSXTOR013_USB_Host_USB_API_Reference_Manual
        |-- drivers
        |   |-- dve_templates
        |   |   |-- usb_driver_FS_Serial_Host_dve.help
        |   |   `-- usb_driver_Utmi_Host_dve.help
        |   |-- usb_driver_FS_Serial_Host.<version>.install
        |   |-- usb_driver_FS_Serial_Host.install
```

```
|   |-- usb_driver_Utmi_Host.<version>.install
|   `-- usb_driver_Utmi_Host.install
|-- drivers.zse
|   |-- usb_driver_FS_Serial_Host.<version>.install
|   |-- usb_driver_FS_Serial_Host.install
|   |-- usb_driver_Utmi_Host.<version>.install
|   `-- usb_driver_Utmi_Host.install
|-- example
|   |-- phy_cable
|   `-- phy_utmi
|-- gate
|   |-- USB_OTG_DUAL.ngo
|   |-- usb_driver_FS_Serial_Host.<version>.edf
|   `-- usb_driver_Utmi_Host.<version>.edf
|-- include
|   |-- HostChannel.<version>.hh
|   |-- Usb.<version>.hh
|   |-- UsbHost.<version>.hh
|   |-- UsbStruct.<version>.hh
|   |-- UsbUrbCommon.<version>.hh
|   `-- UsbUrbHost.<version>.hh
|-- lib32
|   |-- libUsb.<version>_6.so
|   |-- libUsbUL.<version>_6.so
|   `-- libUsbUrb.<version>_6.so
|-- lib64
|   |-- libUsb.<version>.so
|   |-- libUsbUL.<version>.so
|   `-- libUsbUrb.<version>.so
`-- misc
    |-- log
    |   `-- usb_monitor.sv
    |-- phy_cable
```

```
|   |-- phy_cable_HostToDevice.edf
|   |-- phy_cable_HostToDevice_blackbox.v
|   |-- phy_cable_HostToDevice_blackbox.vhd
|   `-- src
|-- phy_utmi2ulpi
|   |-- phy_utmi2ulpi_HostToDevice.edf
|   `-- phy_utmi2ulpi_HostToDevice_blackbox.v
`-- phy_utmi2utmi
    |-- phy_checker.edf
    |-- phy_checker_blackbox.v
    |-- phy_utmi2utmi_HostToDevice.edf
    `-- phy_utmi2utmi_HostToDevice_blackbox.v
```

During installation, symbolic links are created in the `$ZEBU_IP_ROOT/drivers`, `$ZEBU_IP_ROOT/gate`, `$ZEBU_IP_ROOT/include` and `$ZEBU_IP_ROOT/lib` directories for an easy access from all ZeBu tools.

For example, zCui automatically looks for drivers of all transactors in `$ZEBU_IP_ROOT/drivers` if `$ZEBU_IP_ROOT` was properly set.

3 DUT-Transactor Integration with Cable Models

To properly integrate the transactor with the DUT, you must implement the appropriate cable model, which are discussed below.

This section explains the following topics:

- *Architecture with the UTMI DUT Interface*
- *Architecture with the ULPI DUT Interface*
- *Architecture with the USB Cable DUT Interface*

3.1 Architecture with the UTMI DUT Interface

The USB Host transactor implements a USB Host/Device bridge between the USB application software (host testbench) and the DUT, through the USB transceiver interface (UTMI).

The following figure illustrates the USB Host transactor with the UTMI DUT interface:

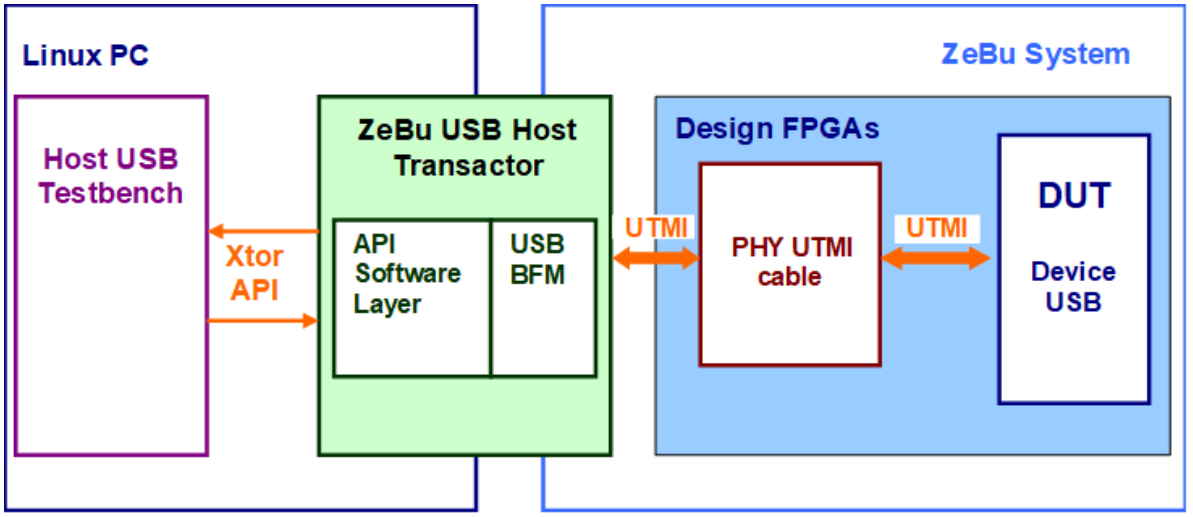

FIGURE 2. USB Host Transactor With the UTMI DUT Interface

Note *The misc/phy_utmi2utmi directory contains an example of the cable model architecture, phy_utmi2utmi_HostToDevice_blackbox.v.*

3.2 Architecture with the ULPI DUT Interface

The USB Host transactor implements a USB Host/Device bridge between the USB application software (Host testbench) and the DUT, through the USB Low-Pin transceiver Interface (ULPI).

The following figure illustrates the USB Host transactor with the ULPI DUT interface.

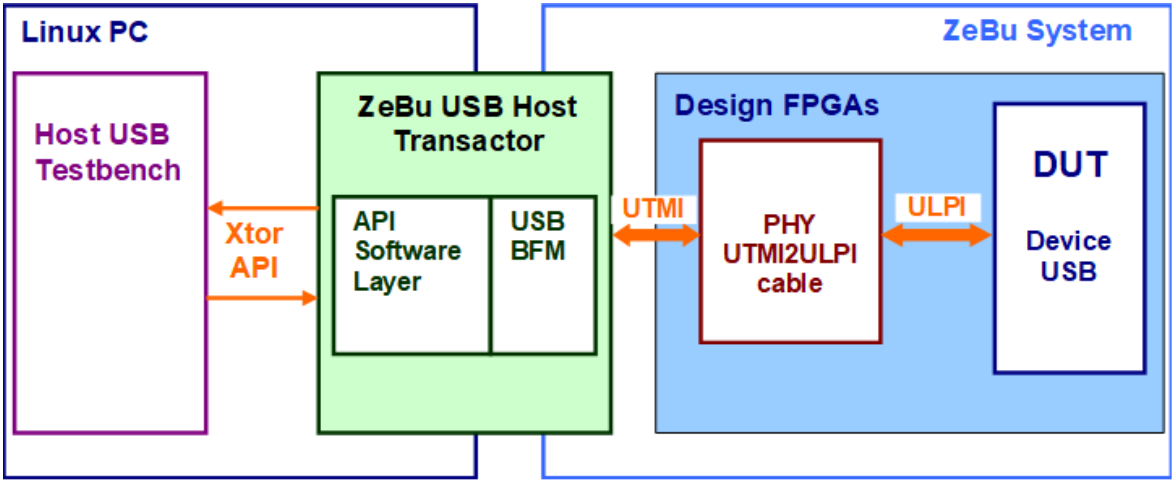

FIGURE 3. USB Host Transactor with the ULPI DUT Interface

Note *The misc/phy_utmi2ulpi directory contains an example of the cable model architecture, phy_utmi2ulpi_HostToDevice_blackbox.v.*

3.3 Architecture with the USB Cable DUT Interface

The USB Host transactor implements a USB Host/Device bridge between the USB application software (Host testbench) and the DUT, through USB cable signals (DP/DM).

The following figure illustrates the USB Host transactor with the USB cable DUT interface.

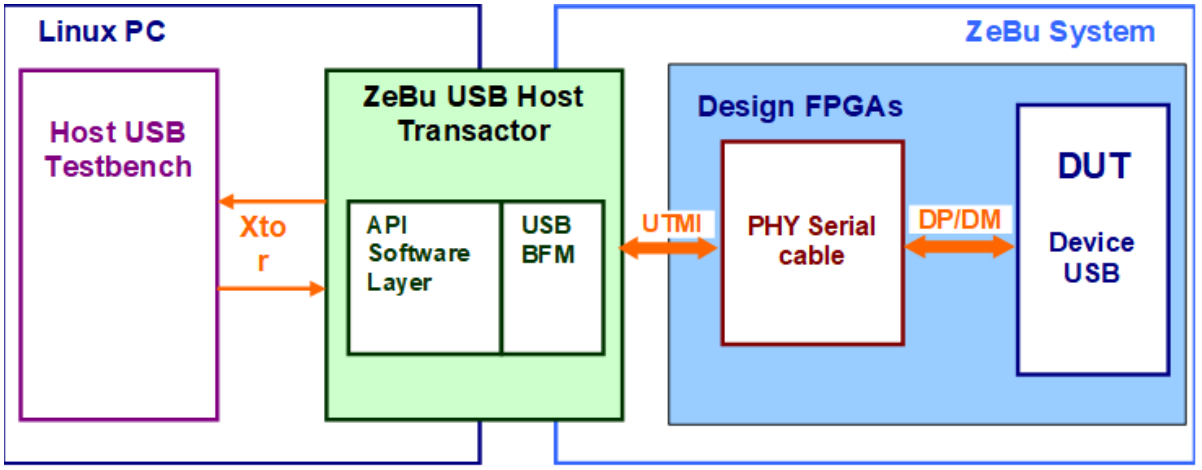

FIGURE 4. USB Host Transactor with the USB Cable DUT Interface

The `misc/phy_cable` directory contains an example of this cable model architecture, `phy_cable_HostToDevice_blackbox.v`.

4 Hardware Interface

This section explains the following topics:

- *Interface Overview*
- *Connecting the Transactor's Clocks*
- *USB Cable Models*
- *USB Bus Logging*

4.1 Interface Overview

The USB Host transactor has the following three types of interfaces:

- *UTMI Interface*
- *DP/DM Serial Interface*
- *ULPI Interface*

4.1.1 UTMI Interface

The following table lists the signals in the UTMI hardware interface:

TABLE 5 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
xtor_clk	1	I	NA	Transactor clock
xtor_resetn	1	I	NA	Transactor reset (active low)
utmi_clk	1	I	I	USB PHY clock
prst_n	1	I	I	PHY reset (active low)
utmi_txready	1	I	O	PHY ready for next packet to transmit
utmi_datain	16	I	O	8/16 bits read from the PHY Low/high bits are asserted valid by utmi_rxvalid/ utmi_rxvalidh respectively
utmi_rxvalid	1	I	O	utmi_datain[7:0] contains valid data
utmi_rxvalidh	1	I	O	utmi_datain[15:8] contains valid data
utmi_rxactive	1	I	O	PHY is active
utmi_rxerror	1	I	O	PHY has detected a receive error

TABLE 5 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
utmi_linestate	2	I	O	PHY line state (dp is bit 0, dm is bit 1)
utmi_dataout	16	O	I	Data transmitted to the PHY
utmi_txvalid	1	O	I	utmi_dataout[7:0] contains valid data
utmi_txvalidh	1	O	I	utmi_dataout[15:8] contains valid data
utmi_opmode	2	O	I	PHY operating mode • 2'b00: normal operation • 2'b01: non-driving • 2'b10: disable bit stuffing and NRZI encoding
utmi_suspend_n	1	O	I	PHY is in suspend mode : disables the clock
utmi_termselect	1	O	I	Selects HS/FS termination: • 1'b0: HS termination enabled • 1'b1: FS termination enabled
utmi_xcverselect	2	O	I	Selects HS/FS transceiver • 2'b00: HS transceiver enabled • 2'b01: FS transceiver enabled
utmi_hostdisconnect	1	I	O	Peripheral disconnect indicator to host
utmi_fsls_low_power	1	O	I	PHY Low Power Clock Select - selects PHY clock mode for power saving
utmi_fslsserialmode	1	O	I	PHY Interface Mode Select - tied low for parallel interface

TABLE 5 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
device_con	1	O	I	Simulate device connection/ disconnection
mon_config	2	O	I	Bus log configuration
bus_O	4	O	I	4-bit bus dedicated to drive signals to the DUT. Output controlled from the API which uses the writeUserIO() method.
bus_I	4	I	O	4-bit bus dedicated to read signals from the DUT. Input controlled from the API that uses the readUserIO() method.
cable_version	8	I	O	Cable version verification.
utmi_word_if	1	O	I	Data bus width: • 1'b0: 8-bit interface • 1'b1: 16-bit interface

The following table describes the UTMI Size interface associated with the UTMI cable:

TABLE 6 UTMI Size Interface

Signal	Type (XTOR)	Type (Cable DUT)	Description
UTMI2UTMI	UTMI-8	UTMI-8	Connects only 8-bit LSB of cable model
UTMI2UTMI	UTMI-16	UTMI-16	Connects 16-bit of cable model
UTMI2ULPI	UTMI-8	ULPI	Connects only 8-bit LSB of cable model to ULPI DUT interface
Serial	UTMI-8	DP/DM	Connects only 8-bit LSB of cable model to DUT serial interface

4.1.2 DP/DM Serial Interface

The following table describes the signal list of serial interface:

TABLE 7 DP/DM Serial Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
xtor_clk	1	I	NA	Transactor clock
xtor_resetn	1	I	NA	Transactor reset (active low)
phy_clk	1	I	I	USB PHY clock
prst_n	1	I	I	PHY reset (active low)
DP_in	1	I	O	DP line
DM_in	1	I	O	DM line
DP	1	O	I	Full speed DP driver
DP_en	1	O	I	Full speed DP enable
DM	1	O	I	Full speed DM driver
DM_en	1	O	I	Full speed DM enable

4.1.3 ULPI Interface

The ULPI interface is available for the transactor through a combined UTMI-ULPI USB cable model, which connects a DUT with ULPI interface to a USB transactor with UTMI Level 3 interface.

Usage of the UTMI-ULPI cable model is described in PHY ULPI Cable Model section.

4.2 Connecting the Transactor's Clocks

4.2.1 Overview

For the synthesizable cable model, the USB Host transactor uses 4 primary clocks, which belong to the same domain but have different frequency ratios.

These clocks are declared as `zceiClockPort` instances in the **hw_top** file and their characteristics are defined for run-time in the `designFeatures` file, as shown in the following figure. See *Verification Environment Example* for `hw_top` and `designFeatures` file examples.

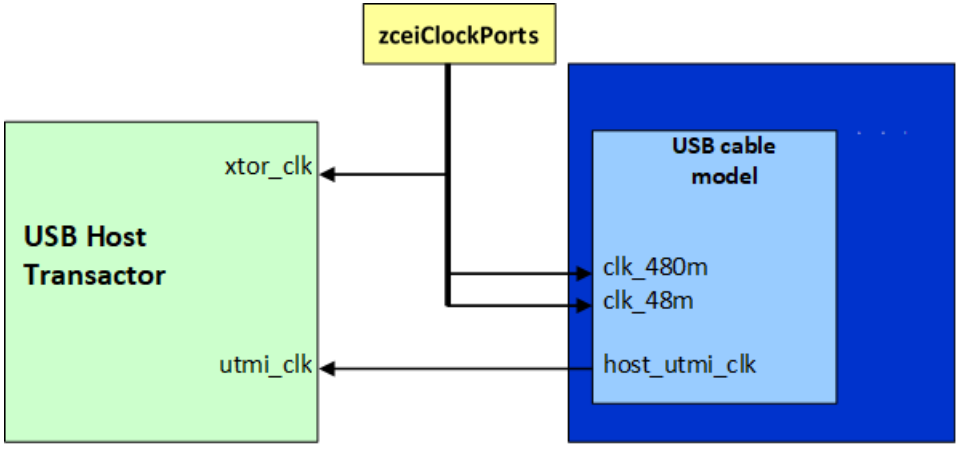

FIGURE 5. Connecting the USB Clocks

4.2.2 Signal List

The following table lists the clock signals for the USB Host transactor:

TABLE 8 USB Host Transactor Clock Signals List

Primary Clock	Description
clk_480m	Represents the USB wire clock for USB synthesizable cable model. Its virtual frequency 480 MHz.
clk_48m	Represents the full-speed USB clock for the USB synthesizable cable model. The virtual frequency is 48 MHz. Thus its virtual frequency is the clk_480m clock frequency divided by 10.
xtor_clock	Represents the application layer clock for the USB transactor. Manages all USB requests to/from the software API with a virtual frequency of 240 MHz.

4.2.3 Verification Environment Example

4.2.3.1 hw_top File Example

The following hw_top file example connects the USB Host transactor using the UTMI 16-bit interface:

```
zceiClockPort clk_48m ( // connected to the PHY cable model
    .cclock (clk_48m) );
zceiClockPort clk_480m (
    .cclock (clk_480m) );

//Host driver part
usb_driver_Host usb_driver_host_inst
     (
                .xtor_clk           (host_xtor_clk),
                .xtor_resetn        (xtor_resetn),
                .utmi_clk           (host_utmi_clk),
                .prst_n             (xtor_resetn),
                .utmi_txready       (host_utmi_txready),
                .utmi_datain        (host_utmi_datain[15:0]),
```

```
                .utmi_rxvalidh       (host_utmi_rxvalidh),
                .utmi_rxvalid        (host_utmi_rxvalid),
                .utmi_rxactive       (host_utmi_rxactive),
                .utmi_rxerror        (host_utmi_rxerror),
                .utmi_linestate      (host_utmi_linestate[1:0]),
                .utmi_dataout        (host_utmi_dataout[15:0]),
                .utmi_txvalidh       (host_utmi_txvalidh),
                .utmi_txvalid        (host_utmi_txvalid),
                .utmi_opmode         (host_utmi_opmode[1:0]),
                .utmi_suspend_n      (host_utmi_suspend_n),
                .utmi_termselect     (host_utmi_termselect),
                .utmi_xcvrselect     (host_utmi_xcvrselect[1:0]),
                .utmi_word_if        (host_utmi_word_if),
                .utmi_hostdisconnect (host_utmi_hostdisconnect),
                .utmi_fsls_low_power (host_utmi_fsls_low_power),
                .utmi_fslsserialmode (host_utmi_fslsserialmode),
                .device_con          (device_con),
                .mon_config          (mon_config[1:0],
                .bus_I               (bus_I[3:0]),
                .bus_O               (bus_O[3:0]),
                .cable_version       (cable_version[7:0]),
      );
defparam usb_driver_host_inst.clkCtrl = host_xtor_clk;
defparam usb_driver_host_inst.process = "usb_driver_inst_host";

zceiClockPort usb_host_zceiClockPort (
    .cclock  (host_xtor_clk),
    .cresetn (xtor_resetn)
  );

assign cable_resetn        = xtor_resetn;
assign cable_resetn_host   = xtor_resetn;
assign cable_resetn_device = xtor_resetn;
```

4.2.3.2 Defining the Clocks in the designFeatures File

```
$B0.host_xtor_clk.VirtualFrequency = 240;
$B0.host_xtor_clk.GroupName = "phy_clk";
$B0.host_xtor_clk.Waveform = "_-";
$B0.host_xtor_clk.Mode = "controlled";

$B0.clk_48m.VirtualFrequency = 48;
$B0.clk_48m.GroupName = "phy_clk";
$B0.clk_48m.Waveform = "_-";
$B0.clk_48m.Mode = "controlled";

$B0.clk_480m.VirtualFrequency = 480;
$B0.clk_480m.GroupName = "phy_clk";
$B0.clk_480m.Waveform = "_-";
$B0.clk_480m.Mode = "controlled";
```

4.3 USB Cable Models

4.3.1 PHY UTMI Cable Model

The USB Host transactor provides a synthesizable cable model for the UTMI interface.

The cable model is available in the misc/phy_utmi2utmi directory of the transactor package as an encrypted EDIF module for ZeBu compilation.

Compile the USB cable model on ZeBu within the DUT to connect to the USB Host transactor at UTMI level.

The following figure illustrates the USB PHY UTMI2UTMI cable interface:

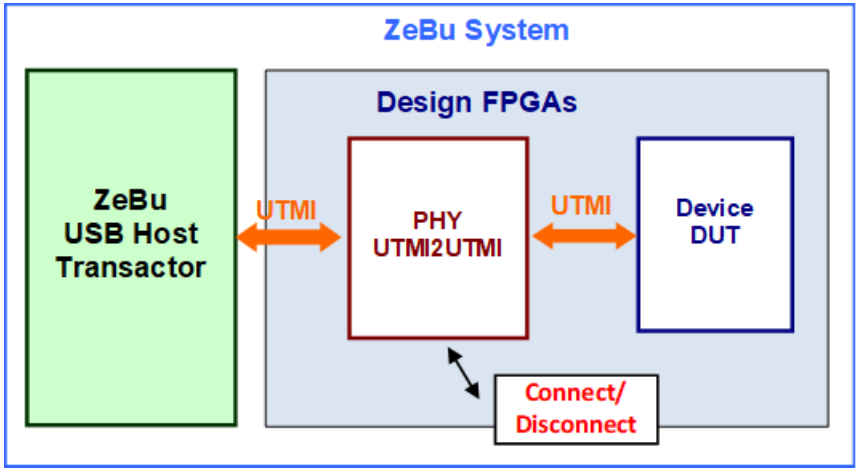

FIGURE 6. USB PHY UTMI2UTMI Cable

The following table describes the PHY UTMI cable model interface. In this table:

- When Side = X, the signal is driven by the transactor or the hw_top.
- When Side = D, the signal is driven by the DUT.

TABLE 9 PHY UTMI2UTMI Cable Interface (HostToDevice)

Signal	Size	Type	Side	Description
cable_resetn	1	I	X	Cable reset (active low)
cable_resetn_device	1	I	X	Cable reset - device side (active low)
cable_resetn_host	1	I	X	Cable reset - host side (active low)
device_con	1	I	X	Connection to the USB device. See *Connection/Disconnection to the USB Device of the DUT* for further details.
clk_48m	1	I	X	48 MHz clock
clk_480m	1	I	X	480 MHz clock
cable_version	8	O	X/D	Cable version check
probe	32	O		Debug bus
UTMI Interface				
utmi_clk	1	O	X/D	USB PHY clock
utmi_txready	1	O	X/D	PHY ready for next packet to transmit
utmi_datain	16	O	X/D	8/16 bits read from the PHY Low/high bits are asserted valid by `utmi_rxvalid`/ `utmi_rxvalidh` respectively
utmi_dataout	16	I	X/D	8/16 bits sent to the PHY Low/high bits are asserted valid by `utmi_rxvalid`/ `utmi_rxvalidh` respectively
utmi_rxvalid	1	O	X/D	`utmi_datain[7:0]` contains valid data
utmi_rxvalidh	1	O	X/D	`utmi_datain[15:8]` contains valid data

TABLE 9 PHY UTMI2UTMI Cable Interface (HostToDevice)

Signal	Size	Type	Side	Description
utmi_rxactive	1	O	X/D	PHY is active
utmi_rxerror	1	O	X/D	PHY has detected a receive error
utmi_linestate	2	O	X/D	PHY line state (dp is bit 0, dm is bit 1)
utmi_txvalid	1	I	X/D	utmi_dataout[7:0] contains valid data
utmi_txvalidh	1	I	X/D	utmi_dataout[15:8] contains valid data
utmi_opmode	2	I	X/D	PHY operating mode: 2'b00: normal operation 2'b01: non-driving 2'b10: disable bit stuffing and NRZI encoding
utmi_suspend_n	1	I	X/D	PHY is in suspend mode : disables the clock
utmi_termselect	1	I	X/D	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled
utmi_xcverselect	2	I	X/D	Selects HS/FS transceiver 2'b00: HS transceiver enabled 2'b01: FS transceiver enabled
utmi_word_if	1	I	X/D	Selects 8/16 bits interface 1'b0: 8-bit interface 1'b1: 16-bit interface
utmi_hostdisconnect	1	O	X/D	Peripheral disconnect indicator to host
Transactor Log				
mon_config	2	I	X	Log configuration.
rx_mon_trig	1	O	D	Log internal signal
rx_mon_sel	4	O	D	Log internal bus

TABLE 9 PHY UTMI2UTMI Cable Interface (HostToDevice)

Signal	Size	Type	Side	Description
rx_mon_data	128	O	D	Log internal bus
tx_mon_trig	1	O	D	Log internal signal
tx_mon_sel	4	O	D	Log internal bus
tx_mon_data	128	O	D	Log internal bus

4.3.2 PHY ULPI Cable Model

The USB Host transactor provides a synthesizable cable model with a UTMI interface on the transactor side and a ULPI interface on the DUT side.

The cable model is available in the misc/phy_utmi2ulpi directory as an encrypted EDIF module for ZeBu compilation.

The following figure illustrates the USB PHY UTMI2ULPI cable interface:

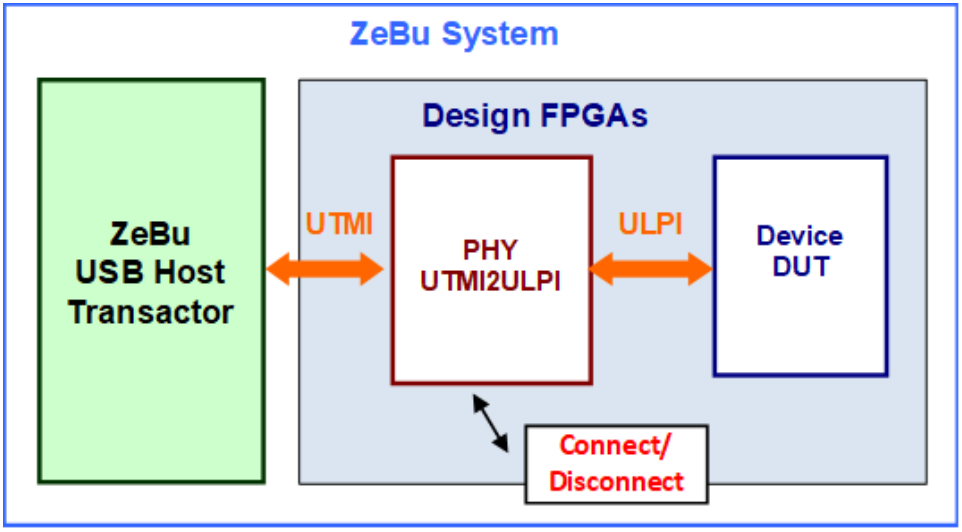

FIGURE 7. PHY UTMI2UTLPI Cable

The following figure illustrates the ULPI interface connection with the:

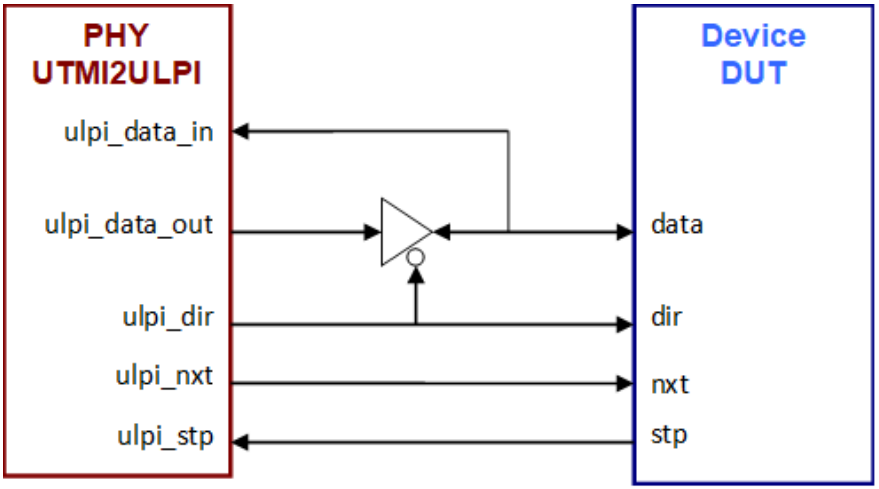

FIGURE 8. PHY UTMI2ULPI Connection to Standard Device DUT

The following table describes the PHY ULPI cable model interface. In this table:

- When Side = X, the signal is driven by the transactor and the hw_top.
- When Side = D, the signal is driven by the DUT.

TABLE 10 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
cable_resetn	1	I	X	Cable reset (active low)
device_con	1	I	X	Device connection. See *Connection/Disconnection to the USB Device of the DUT* for further details.
clk_48m	1	I	X	48 MHz clock
clk_480m	1	I	X	480 MHz clock
UTMI Interface for XTOR				
utmi_clk	1	O	X	USB PHY clock

TABLE 10 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
utmi_txready	1	O	X	PHY ready for next packet to transmit
utmi_datain	16	O	X	8/16 bits read from the PHY. Low/high bits are asserted valid by utmi_rxvalid/ utmi_rxvalidh respectively
utmi_dataout	16	I	X	8/16 bits sent to the PHY. Low/high bits are asserted valid by utmi_txvalid/ utmi_txvalidh respectively
utmi_rxvalid	1	O	X	utmi_datain[7:0] contains valid data
utmi_rxvalidh	1	O	X	utmi_datain[15:8] contains valid data
utmi_rxactive	1	O	X	PHY is active
utmi_rxerror	1	O	X	PHY has detected a receive error
utmi_linestate	2	O	X	PHY line state (dp is bit 0, dm is bit 1)
utmi_txvalid	1	I	X	utmi_dataout[7:0] contains valid data
utmi_txvalidh	1	I	X	utmi_dataout[15:8] contains valid data
utmi_opmode	2	I	X	PHY operating mode: 2'b00: normal operation 2'b01: non-driving 2'b10: disable bit stuffing and NRZI encoding
utmi_suspend_n	1	I	X	PHY is in suspend mode: disables the clock

TABLE 10 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
utmi_termselect	1	I	X	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled
utmi_xcverselect	2	I	X	Selects HS/FS transceiver 2'b00: HS transceiver enabled 2'b01: FS transceiver enabled
utmi_word_if	1	I	X	Selects 8/16 bits interface 1'b0: 8-bit interface 1'b1: 16-bit interface
utmi_hostdisconnect	1	O	X	Peripheral disconnect indicator to host
utmi_fsls_low_power	1	I	D	PHY Low Power Clock Select (selects PHY clock mode for power saving)
utmi_fslsserialmode	1	I	D	PHY Interface Mode Select (tied low for parallel interface)
ULPI Interface for DUT				
ulpi_stp	1	I	D	Stops output control
ulpi_data_in	8	I	D	ULPI data input
ulpi_data_out	8	O	D	ULPI data output
ulpi_clk	1	O	D	ULPI clock
ulpi_dir	1	O	D	Data bus control 1'b0: the Host/Device is the driver 1'b1: the PHY is the driver
ulpi_nxt	1	O	D	Next data control
Transactor Log				
mon_config	2	I	X	Log configuration.
rx_mon_trig	1	O	D	Log internal signal
rx_mon_sel	4	O	D	Log internal bus

TABLE 10 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
rx_mon_data	128	O	D	Log internal bus
tx_mon_trig	1	O	D	Log internal signal
tx_mon_sel	4	O	D	Log internal bus
tx_mon_data	128	O	D	Log internal bus

4.3.3 PHY Serial Cable Model

The USB transactor provides a synthesizable cable model with a UTMI interface on the transactor side and a serial interface on the DUT side, as shown in the following figure:

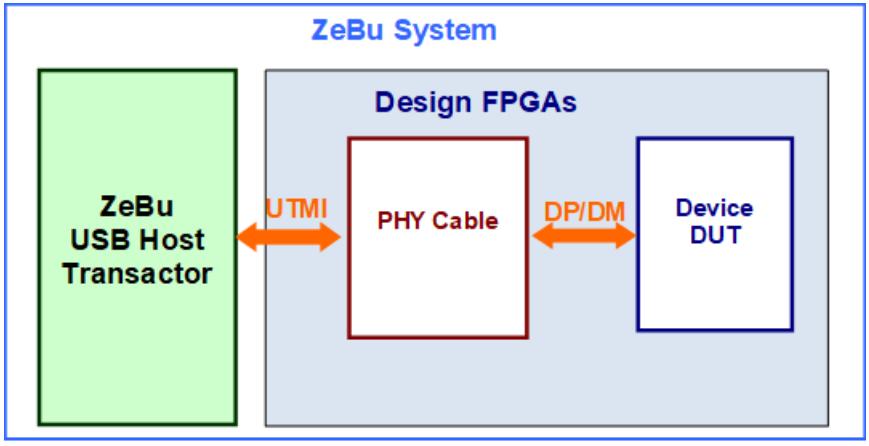

FIGURE 9. PHY Serial Cable

The cable model is available in the `misc/phy_cable` directory as a dedicated encrypted EDIF module for ZeBu compilation.

The following table lists the PHY serial cable interface signals:

TABLE 11 PHY Serial Cable Interface Signals

Signal	Size	Type	Description
cable_resetn	1	I	Cable reset (active low)
UTMI Interface for Xtor side			
utmi_clk	1	O	USB PHY clock
utmi_txready	1	O	PHY ready for the next packet to be transmitted
utmi_datain	16	I	8/16 bits sent to the PHY. Low/high bits are asserted valid by utmi_txvalid/utmi_txvalidh respectively
utmi_dataout	16	O	8/16 bits read from the PHY. Low/high bits are asserted valid by utmi_rxvalid/utmi_rxvalidh respectively
utmi_rxvalid	1	O	utmi_dataout[7:0] contains valid data
utmi_rxvalidh	1	O	utmi_dataout[15:8] contains valid data
utmi_rxactive	1	O	PHY is active
utmi_rxerror	1	O	PHY has detected a receive error
utmi_linestate	2	O	PHY line state (dp is bit 0, dm is bit 1)
utmi_txvalid	1	I	utmi_datain[7:0] contains valid data
utmi_txvalidh	1	I	utmi_datain[15:8] contains valid data
utmi_opmode	2	I	PHY operating mode 2'b00: normal operation 2'b01: non-driving 2'b10: disable bit stuffing and NRZI encoding
utmi_suspend_n	1	I	PHY is in suspend mode: disables the clock
utmi_termselect	1	I	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled

TABLE 11 PHY Serial Cable Interface Signals

Signal	Size	Type	Description
utmi_xcverselect	2	I	Selects HS/FS transceiver 2'b00: HS transceiver enabled 2'b01: FS transceiver enabled
utmi_word_if	1	I	Selects 8/16 bits interface 1'b0: 8-bit interface 1'b1: 16-bit interface
utmi_hostdisconnect	1	O	Peripheral disconnect indicator to host
Serial Interface for DUT side			
dm	1	I	DM line
dp	1	I	DP line
LSFS_DP_tx	1	O	Full-speed DP driver
LSFS_DM_tx	1	O	Full-speed DM driver
LSFS_DP_en	1	O	Full-speed DP enable
LSFS_DM_en	1	O	Full-speed DM enable
HS_DP_tx	1	O	High-speed DP driver
HS_DM_tx	1	O	High-speed DM driver
HS_DP_en	1	O	High-speed DP enable
HS_DM_en	1	O	High-speed DM enable
clk_48m	1	I	48 MHz clock
clk_480m	1	I	480 MHz clock

4.3.4 Connection/Disconnection to the USB Device of the DUT

From the Host transactor with PHY UTMI or PHY ULPI cable model only, connection/disconnection to the USB device is enabled using the `device_con` input signal of the cable model. This input is driven by the transactor via the `USBPlug` and `USBUnplug` API functions.

The following figure describes a sequence of device disconnection/connection:

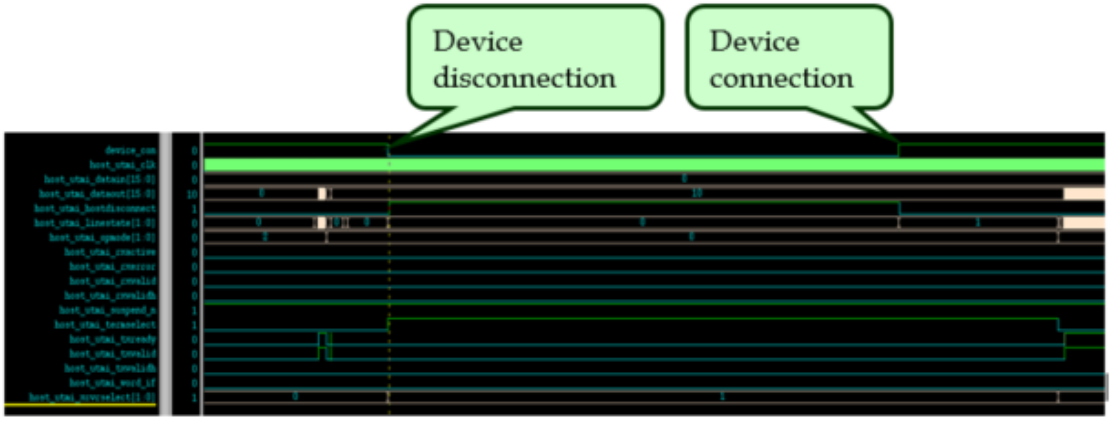

FIGURE 10. Device Connection/Disconnection Waveforms

The same sequence from the USB Bus point of view is shown below:

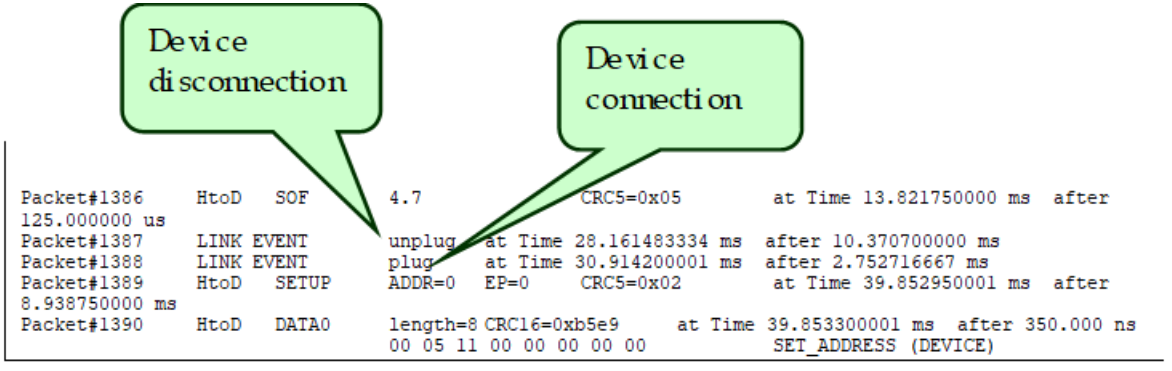

FIGURE 11. Device Connection/Disconnection in the USB Bus Log

4.3.5 PHY UTMI Interface Checker

Note *This interface checker only checks the phy_utmi2utmi interface.*

4.3.5.1 Description

A PHY cable checker module, `phy_checker`, is provided with the transactor package in the `misc/phy_utmi2utmi` directory. It checks that the data managed by the USB cable synthesizable model is transferred correctly from the USB Host to the USB Device and vice-versa. This verification is made at the UTMI interface level and is helpful to validate the integration of the USB Host transactor with the DUT.

You can plug this additional checking module to the USB cable interface, to report any link or data error on the USB Host or USB Device side.

The following figure illustrates the UTMI USB Cable Checker:

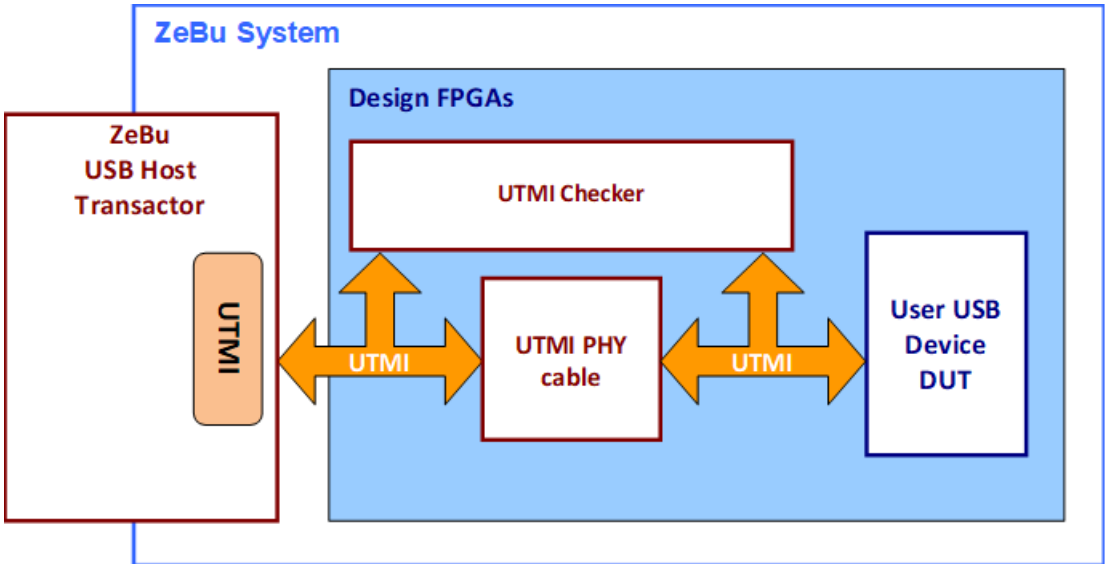

FIGURE 12. UTMI USB Cable Checker

4.3.5.2 Interface

The following table lists the signals in the PHY UTMI2UTMI cable checker interface:

TABLE 12 PHY UTMI2UTMI Cable Checker Interface

Signal	Size	Type	Description
rstn	1	I	Transactor reset (active low)
hst_utmi_clk	1	I	USB PHY clock
hst_utmi_txready	1	I	PHY ready for next packet to transmit
hst_utmi_data_in	16	I	8/16 bits read from the PHY. Low/High bits are asserted valid by utmi_rxvalid/utmi_rxvalidh.
hst_utmi_rxvalid	1	I	utmi_datain[7:0] contains valid data
hst_utmi_rxvalidh	1	I	utmi_datain[15:8] contains valid data
hst_utmi_rxactive	1	I	PHY is active
hst_utmi_data_out	16	I	Data transmitted to the PHY
hst_utmi_txvalid	1	I	utmi_dataout[7:0] contains valid data
hst_utmi_txvalidh	1	I	utmi_dataout[15:8] contains valid data
hst_utmi_opmode	2	I	PHY operating mode 2'b00: normal operation 2'b01: non-driving 2'b10: disable bit stuffing and NRZI encoding
hst_utmi_termselect	1	I	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled
hst_utmi_word_if	1	I	Data bus width 1'b0: 8-bit interface 1'b1: 16-bit interface
hst_link_error	1	O	1'b1: link error for host 1'b0: no error
hst_data_error	32	O	Number of errors detected on USB Host side. Synchronized on hst_utmi_clk.

TABLE 12 PHY UTMI2UTMI Cable Checker Interface

Signal	Size	Type	Description
dev_utmi_clk	1	I	USB PHY clock.
dev_utmi_txready	1	I	PHY ready for next packet to transmit.
dev_utmi_data_in	16	I	8/16 bits read from the PHY. Low/High bits asserted valid by utmi_rxvalid/utmi_rxvalidh.
dev_utmi_rxvalid	1	I	utmi_datain[7:0] contains valid data
dev_utmi_rxvalidh	1	I	utmi_datain[15:8] contains valid data
dev_utmi_rxactive	1	I	PHY is active
dev_utmi_data_out	16	I	Data transmitted to the PHY
dev_utmi_txvalid	1	I	utmi_dataout[7:0] contains valid data
dev_utmi_txvalidh	1	I	utmi_dataout[15:8] contains valid data
dev_utmi_opmode	2	I	PHY operating mode 2'b00: normal operation 2'b01: non-driving 2'b10: disable bit stuffing and NRZI encoding
dev_utmi_termselect	1	I	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled
dev_utmi_word_if	1	I	Data bus width 1'b0: 8-bit interface 1'b1: 16-bit interface
dev_link_error	1	O	1'b1: link error for device 1'b0: no error
dev_data_error	32	O	Number of errors detected on the device side. Synchronized on dev_utmi_clk.

4.3.5.3 Advanced Debugging

You can connect the error detection of the PHY UTMI Interface Checker to the module for a more efficient debugging.

Additionally, you can connect the outputs of the checker module to an SRAM-trace driver, as shown in the following hw_top file example:

Note *SRAM-trace uses static-probes.*

```
SRAM_TRACE checker(
  .output_bin({
    hst_data_error[31:0],
    hst_link_error,
    dev_data_error[31:0],
    dev_link_error
}));
```

4.4 USB Bus Logging

Note *This feature is only available for the 64-bit version of ZeBu Server-1. It requires the use of ZeBu zFAST synthesis tool.*

The USB signals and packets logging feature allows the viewing/analyzing of USB low-level protocol transactions exchanged over the USB cable model between the host and the device.

It enables the logging of the USB Bus at UTMI level. It enables you to report the different USB tokens and packets sent to and received from the design. This is available for debugging at packet level or at transaction level.

This feature is available for both the PHY UTMI2UTMI and PHY UTMI2ULPI USB cable models. It can be controlled by the software API and dynamically enabled/disabled during the run.

4.4.1 Verification Environment for the USB Bus Logging Feature

The following figure illustrates the verification environment for the USB Bus logging:

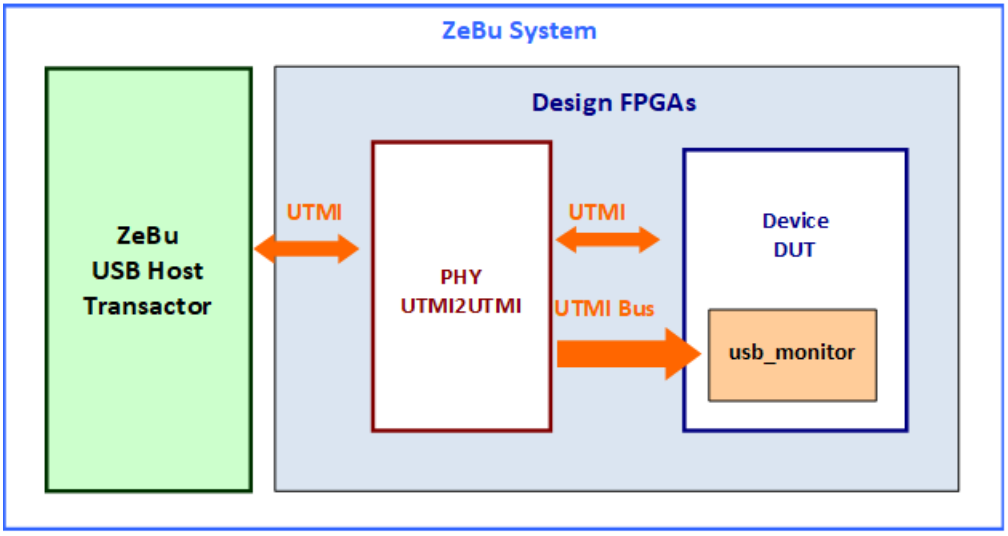

FIGURE 13. Verification Environment for the USB Bus Logging Feature

4.4.1.1 Adding a Logging Block in the DUT

4.4.1.2 The usb_monitor is a SystemVerilog module, which is available in the misc directory. Compiling with zCui

In order to enable the USB bus logging feature, perform the following steps in zCui:

1. **RTL Group Properties -> Main** tab: select the zFAST synthesizer.
2. **RTL Group Properties -> zDPI** tab: select Activate zDPI then Synthesize All.
3. Set the monitor.v source file as SystemVerilog (right-click on the file in the file tree and select **Set File Type** as **SystemVerilog**).

4.4.2 Bus Logging Control

The log is controlled through the USB Host transactor software API. See *USB Bus Monitoring Feature* for details about the methods available in the API.

5 Software Interface

The USB Host transactor provides the following two APIs allowing two different abstraction layers for USB operations and transfer control:

- *USB Channel API*
- *USB Request Block (URB) API*

5.1 USB Channel API

The testbench directly controls the USB channel transfers to the USB device implemented in the DUT. There is no software host controller embedded in the transactor. The testbench splits the data into single operations, checking channel status, and sending data again in case of NAK response.

See *USB Channel Software API* for a detailed API description.

The following figure illustrates the USB Host Transactor Channel API:

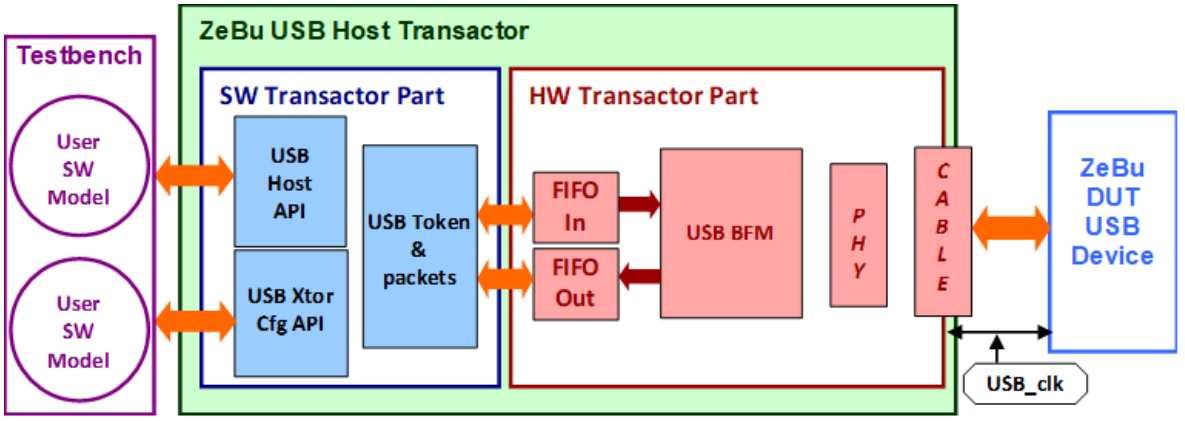

FIGURE 14. USB Host Transactor Channel API

5.2 USB Request Block (URB) API

The testbench is written like a Linux USB driver and the transactor behaves as a USB Host controller. It uses high-level USB Request Blocks to transfer data to/from the USB device implemented in the DUT.

See *URB Software API* for a detailed API description.

The following figure illustrates the URB API for the USB Host Transactor.

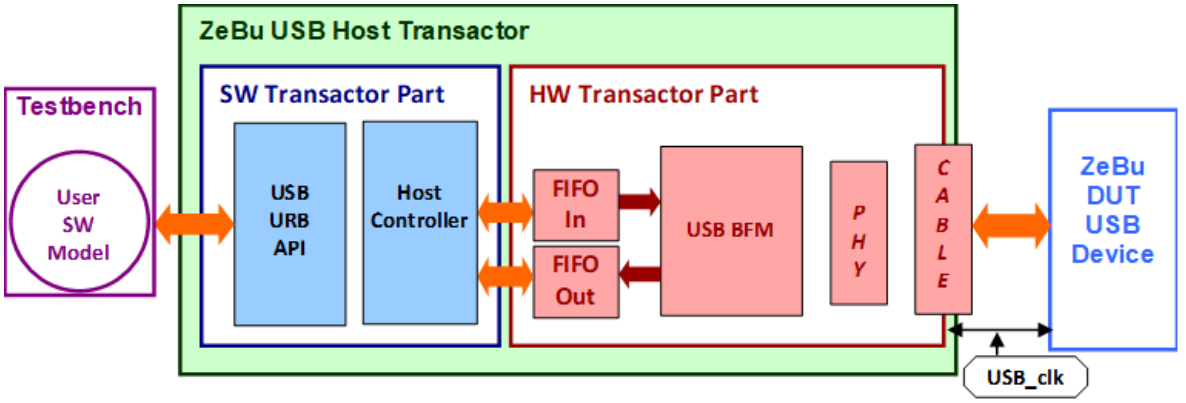

FIGURE 15. USB Host transactor URB API

6 USB Channel Software API

The USB Host Channel API provides a way to communicate with the DUT inside ZeBu using Channel and Endpoint objects.

The testbench is written like a Host controller driver at single USB transfer level.

6.1 Using the USB Host Channel API

6.1.1 Libraries

The USB Host Channel API is defined in `UsbHost.hh` and `UsbStruct.hh` header files.

Use the following libraries:

- `libUsb.so`: 64-bit gcc 3.4 library
- `libUsb_6.so`: 32-bit gcc 3.4 library

6.1.2 Data Alignment

The data received and transmitted over the USB BFM are packed into `unsigned char` arrays or `unsigned int` arrays. The LSB byte or the first word of the USB packet data is stored at location 0 of the array.

6.2 USB Host Channel API Class Description

The USB Host transactor can be instantiated and accessed using the following C++ classes:

TABLE 13 USB Host Channel API Class Description

Class	Description
UsbHost	Represents the USB Host transactor. It manages directly USB status, USB packets and then requests to or from the DUT processed by the USB Host transactor. It also controls and configures the USB Host transactor.
HostChannel	Represents the data exchange over the USB Host transactor. It provides information on the status of each USB Host channel.

6.3 UsbHost Class

6.3.1 Description

UsbHost class contains methods to control the USB Host transactor and to send/receive data packets over the USB link. It also manages the USB flow control mechanism.

The following tables give an overview of available types (defined in UsbStruct.hh) and methods for the USB API UsbHost class.

TABLE 14 Types for UsbHost Class

Type	Description
usbHaltStatus_t	Reason for channel halting
HostStatus_t	USB Host status
XferTyp_t	USB transfer type
HCSpeed_t	USB channel speed
bmReqDir_t	USB control transfer direction
bmReqType_t	USB setup request type
bmReqRecipient_t	USB setup request recipient
bReqcode_t	USB setup request
logMask_t	USB Log Mask
utmi_type_t	UTMI interface width in bit (8 or 16)
speed_conf_type_t	Transactor speed support configuration

TABLE 15 Methods for UsbHost Class

Method	Description
Transactor Initialization and Control	
UsbHost	Constructor.
~UsbHost	Destructor.

TABLE 15 Methods for UsbHost Class

Method	Description
init	ZeBu Board attachment.
reInit	Re-initializes the transactor. It must be called before any new call to USBPlug() after an USBUnplug() call.
config	USB transactor configuration and initialization.
loop	Blocking method: it processes all the pending Tx or Rx USB packets. and waits for BFM events or interrupts.
delay	Inserts a delay with respect to the USB clock (in milliseconds).
registerCallback	Registers a service loop callback.
log	Selects the logging mode.
setLogPrefix	Sets a log prefix to be used in log information.
getVersion	Returns either a string or a float corresponding to the transactor version.
Channel Management	
channel	Gets channel handler (0 ? n ? 15).
releaseChannel	Releases the channel used by the Host.
Channel Operations	
hcInit	Prepares and configures a channel for transmission.
hcHalt	Sends a HALT over the specified channel.
sendSetupPkt	Sends a setup, or USB request transaction given as parameter over the specified channel.
requestData	Sends a data request transaction of N bytes over the specified channel.
sendStatusPkt	Sends a status packet over the specified channel.
sendData	Sends a data bytes' buffer over the specified channel.
rcvData	Gets the received data buffer from the specified channel.
enableReception	Enables data reception over the specified channel.

TABLE 15 Methods for UsbHost Class

Method	Description
actualXferLength	Returns the number of bytes sent/received over a specified channel.
waitStatusPkt	Prepares the transactor for USB status packet reception phase.
USB Operations	
USBPlug	Connects the USB Host to the cable.
USBUnplug	Disconnects the USB Host from the cable.
usbReset	Sends the USB Reset sequence to the USB Device.
USB Cable Model Status	
portEnabled	Returns true if the USB port is enabled.
isDeviceAttached	Returns true if a device is connected on the USB bus.
USB Protocol Information	
portSpeed	Returns the port speed.
maxPktSize	Returns the maximum packet size.
isPingSupported	Returns true if the PING protocol is supported.
General Purpose Vectors Operations	
writeUserIO	Drives the general purpose bus_O vector.
readUserIO	Reads the general purpose bus_I vector.

6.3.2 Transactor Initialization and Control Methods

6.3.2.1 Constructor and Destructor Methods

Allocates/frees the structures for the USB Host transactor.

```
UsbHost (void)
~UsbHost (void)
```

6.3.2.2 `init()` Method

Connects to the ZeBu board.

```
void init (Board *zebu, const char *driverName,
           const char *clkName=NULL);
```

Parameter Name	Parameter Type	Description
`zebu`	`Board *`	Pointer to the ZeBu board structure.
`driverName`	`const char *`	Name of the transactor instance in hw_top file.
`clkName`	`const char *`	Name of the primary clock used as a time reference in debug messages (optional).

Note *When the Bus logging feature is used (see USB Bus Monitoring Feature), clkName must be specified and must have the same name as the transactor clock. In the examples shown in the Verification Environment Example section , the name is host_xtor_clk.*

6.3.2.3 reInit() Method

Re-initializes the transactor after an unplug sequence. This function must be called before any new call to `USBPlug()` after an `USBUnplug()` call.

```
void reInit (void);
```

6.3.2.4 config() Method

Configures the USB Host transactor and allocates software structures accordingly.

```
uint config (speed_conf_type_t spdSupport, utmi_type_t phyIf);
```

TABLE 16

Parameter Name	Parameter Type	Description
spdSupport	speed_conf_type_t	Transactor speed selection (optional): • spd_full: the transactor supports only Full Speed transfers, whatever the speed mode of the USB device. • spd_high (default): the transactor supports Full and High Speed transfers. • spd_high_ping: the transactor supports Full Speed transfers and High Speed transfers with PING protocol.
phyIf	utmi_type_t	Type of interface (optional): • utmi_8 (default): for UTMI 8-bit interface and Serial Cable interface • utmi_16: for UTMI 16-bit interface

This method must be called after `init()` and before performing any operation on the USB Host transactor. If Hi-Speed support is enabled, the transactor supports both Full- and Hi-Speed connections. Else it supports Full-Speed connections only.

This method also configures the UTMI interface in 8 or 16-bit mode.

It returns `0` if an error occurs.

6.3.2.5 loop() Method

Checks the USB Host transactor status and performs internal operations. If an event occurs during the loop call, software structures and returned value are set accordingly. This method must be called regularly to allow the controlled clock to advance

.

```
HostStatus_t loop (void);
```

6.3.2.6 delay() Method

Allows the USB clock to advance for the specified number of milliseconds before

performing the next USB operation.

```
void delay (uint32_t msec, bool blocking);
```

Parameter Name	Parameter Type	Description
msec	uint32_t	Delay of the UTMI clock in milliseconds
blocking	bool	Operation type: • false (default): non-blocking operation; the method returns immediately • true: blocking operation; the method returns after the operation is completed

6.3.2.7 registerCallBack() Method

Registers a callback to be called during service loop.

```
void registerCallBack (void (*userCB)(void* CBParams),
                       void* CBParams);
```

Parameter Name	Parameter Type	Description
userCB	void (*fct)(void*)	Pointer to the function
CBParams	void *	Context to be used in the callback

6.3.2.8 log() Method

Configures the log mechanism.

```
void log (logMask_t mask);
```

where mask is the log level as defined hereafter:

TABLE 17 Log levels for log method

mask values	Description
logOff	No information.
logCore	Information about the Host controller.
logInfo	Information about Transactor steps.
logTime	Information about reference clock value at different stages of the run.
logAny	Selects all logs above

See *Logging USB Packets Processed by the Transactor* for further details.

6.3.2.9 setLogPrefix() Method

Sets a new log prefix for log.

```
void setLogPrefix (const char *prefix);
```

where prefix is the prefix to use in logs.

See *Logging USB Packets Processed by the Transactor* for further details on logs.

6.3.2.10 getVersion() Method

Two different prototypes exist for this method:

- Returns a string corresponding to the transactor version.

```
static const char* getVersion (void);
```

- Returns a float corresponding to the transactor version.

```
static void getVersion (float &version_num);
```

where version_num is the transactor version number.

6.3.3 Channel Management Methods

6.3.3.1 channel() Method

Returns a handler to the specified channel number. The channel number must range between 0 and 15 included. Channel number 0 is reserved for control transfer.

```
HostChannel *channel (uint8_t n);
```

where n is the channel number.

6.3.3.2 `releaseChannel()` Method

Releases the specified channel.

```
void releaseChannel (HostChannel *hc);
```

where hc is the channel handler.

6.3.4 Channel Operation Methods

6.3.4.1 hcInit() Method

Prepares a host channel for transferring packet to or from the specified endpoint of the specified device.

```
void hcInit (HostChannel *hc, XferTyp_t typ, HCSpeed_t spd,
             uint8_t dev_addr, uint8_t ep_num, bool ep_is_in,
             uint16_t max_pkt_size);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel handler
type	XferTyp_t	Transmission type

Parameter Name	Parameter Type	Description
spd	HCSpeed_t	Transmission speed
dev_addr	uint8_t	Destination device address
ep_num	uint8_t	Destination endpoint address
ep_is_in	bool	Transmission direction
max_pkt_size	uint16_t	Maximum packet size

6.3.4.2 hcHalt() Method

Attempts to halt a host channel. Halt completes when `HostChannel::halted()` returns true.

```
void hcHalt (HostChannel *hc, usbHaltStatus_t haltSts);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor
haltSts	usbHaltStatus_t	Reason of the halt

6.3.4.3 sendSetupPkt() Method

Sends a setup request over the specified channel. For details, see the ***USB Device Requests*** section in the USB 2.0 standard specification.

```
void sendSetupPkt (HostChannel *hc, bmReqDir dir, bmReqType typ,
                   bmReqRecipient rcpt, bReqCode code,
                   uint16_t val, uint16_t index, uint16_t length);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor.
dir	bmReqDir	Transfer direction.
typ	bmReqType	Request type.
rcpt	bmReqRecipient	Request recipient.
code	bReqCode	Request content.
val	uint16_t	Value.
index	uint16_t	Index.
length	uint16_t	Number of bytes to transfer if there is a data phase; 0 otherwise

6.3.4.4 requestData() Method

Initiates IN data phase over the specified channel.

```
void requestData (HostChannel *hc,uint32_tlen);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor.
len	uint	Number of expected bytes.

6.3.4.5 sendStatusPkt() Method

Sends a status packet over the specified channel. If no data is specified, a zero-length status packet is sent.

```
void sendStatusPkt (HostChannel *hc, uint32_t *data=NULL,
                    uint32_tlen=0);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor.
data	uint32_t *	Pointer to data buffer (optional) This parameter is ignored if data length (len) is zero. Default value is a NULL pointer.
len	uint	Data buffer length (optional). Default value is 0.

6.3.4.6 sendData() Method

Starts data transfer over the specified channel.

```
void sendData (HostChannel *hc, uint32_t *data,uint32_tlen);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor.
data	uint32_t *	Pointer to data buffer. This parameter is ignored if data length (len) is zero.
len	uint	Data buffer length in bytes.

6.3.4.7 `rcvData()` Method

Gets the received data. Call this method on data transfer completion, when both `HostChannel::XferComplete()` and `HostChannel::dataPresent()` returns true. Data pointed by `data` and `len` should be used/copied before next call of `UsbHost::loop()` since they belong to `UsbHost` class and may be overridden.

```
void rcvData (HostChannel *hc, uint32_t **data,uint32_t*len);
```

Parameter Name	Parameter Type	Description
hc	HostChannel *	Host channel descriptor.
data	uint32_t **	Pointer to the reception data buffer.
len	uint *	Length in bytes of the actual data received in the buffer.

6.3.4.8 enableReception() Method

Starts an IN data phase on the specified channel.

```
void enableReception (HostChannel *hc);
```

where hc is the Host channel descriptor.

6.3.4.9 actualXferLength() Method

Gets the number of bytes sent/received during the current transfer. This method is convenient when a NAK occurred to restart the transfer where it was halted.

```
uint32_t actualXferLength (HostChannel *hc);
```

where hc is the Host channel descriptor.

6.3.4.10 waitStatusPkt() Method

Sets up the host channel for status packet reception.

```
void waitStatusPkt (HostChannel * hc);
```

where hc is the Host channel descriptor.

6.3.5 USB Operations

6.3.5.1 USBPlug() Method

Connects the host to the USB cable with one of the following methods:

```
void USBPlug (uint32_t duration);
void USBPlug (uint32_t latency, uint32_t duration);
```

Parameter Name	Parameter Type	Description
duration	uint32_t	Time to wait after the connection of the host (in ms)
latency	uint32_t	Time to wait before connecting the host (in ms)

6.3.5.2 USBUnplug() Method

Disconnects the host from the USB cable with one of the following methods:

```
void USBUnplug (uint32_t duration);
void USBUnplug (uint32_t latency, uint32_t duration);
```

Parameter Name	Parameter Type	Description
duration	uint32_t	Time to wait after the host disconnection (in ms)
latency	uint32_t	Time to wait before disconnecting the host (in ms)

6.3.5.3 usbReset() Method

Initiates a USB reset on the bus.

```
void usbReset (uint8_t length=10);
```

where length is the reset duration in milliseconds. Default value is 10 ms. This

parameter is optional.

6.3.6 USB Cable Model Status Methods

6.3.6.1 portEnabled() Method

Returns `true` if the USB port on hardware transactor's side is enabled, `false` otherwise.

```
bool portEnabled (void);
```

6.3.6.2 `isDeviceAttached()` Method

Returns `true` when a device is connected on the USB bus, `false` otherwise.

```
bool isDeviceAttached (void);
```

6.3.7 USB Protocol Information

6.3.7.1 `portSpeed()` Method

Returns the device speed.

```
HCSpeed_t portSpeed (void);
```

6.3.7.2 `maxPktSize()` Method

Returns the maximum packet size. The value depends on the device speed.

```
uint32_t maxPktSize (void);
```

6.3.7.3 `isPingSupported()` Method

Returns `true` if the device supports the PING protocol, `false` otherwise.

```
bool isPingSupported (void);
```

6.3.8 General Purpose Vectors Operations

6.3.8.1 `writeUserIO()` Method

Drives the 4-bit general purpose bus_O vector with its specified value.

```
void writeUserIO (const uint32_t value);
```

where value is the vector's value. It ranges from 0 to 15.

6.3.8.2 `readUserIO()` Method

Reads the 4-bit general purpose bus_I vector.

```
uint32_t readUserIO (void);
```

6.4 HostChannel Class

6.4.1 Description

The HostChannel class contains all methods to access information about the activity on the USB channels associated with the USB Host transactor.

The following tables give an overview of available types and methods for the HostChannel class.

TABLE 18 Types for HostChannel Class

Type	Description
HCType	USB packet type
HCSpeed	USB channel speed
HCReqType	USB Request type
HCReqcode	USB Request code

TABLE 19 Methods for HostChannel Class

Type	Description
getChNumber	Returns the channel number
dataPresent	Returns true when incoming data is present
xferComplete	Returns true when current transfer is completed
errorHappened	Returns true when an error happens
Halted	Returns true when channel receives a HALT
Stall	Returns true when a STALL is received
NAK	Returns true when a NAK is received
ACK	Returns true when an ACK is received
NYET	Returns true when a NYET is received
xactError	Returns true when a transaction error occurs

TABLE 19 Methods for HostChannel Class

Type	Description
babbleError	Returns true when a babble error occurs
frameOverrun	Returns true when a frame overrun occurs
display	Displays the channel status (for debug purpose)

6.4.2 Detailed Methods

6.4.2.1 getChNumber() Method

Returns the channel number.

```
uint8_t getChNumber (void);
```

6.4.2.2 dataPresent() Method

Returns true if data has been received during the current transfer, false otherwise.

```
bool dataPresent (void);
```

6.4.2.3 xferComplete() Method

Returns true if the current transfer is complete, false otherwise.

```
bool xferComplete (void);
```

6.4.2.4 errorHappened() Method

Returns true if an error happened, false otherwise.

```
bool errorHappened (void);
```

The following are the possible errors:

- transaction error

- babble error
- frame overrun
- data toggle error

For more information, see *Detailed Methods* section.

6.4.2.5 Halted() Method

Returns `true` if the channel is halted, `false` otherwise.

```
bool Halted (void);
```

6.4.2.6 Stall() Method

Returns `true` if a Stall token was received, `false` otherwise.

```
bool Stall (void);
```

6.4.2.7 NAK() Method

Returns `true` if a NAK token was received, `false` otherwise.

```
bool NAK (void);
```

6.4.2.8 ACK() Method

Returns `true` if an ACK token was received, `false` otherwise.

```
bool ACK (void);
```

6.4.2.9 NYET() Method

Returns `true` if a NYET token was received, `false` otherwise.

```
bool NYET (void);
```

6.4.2.10 xactError() Method

Returns `true` if a transactor error occurred, `false` otherwise.

```
bool xactError (void);
```

6.4.2.11 `babbleError()` Method

Returns `true` if a babble error occurred, `false` otherwise.

```
bool babbleError (void);
```

6.4.2.12 `frameOverrun()` Method

Returns `true` if a frame overrun occurred, `false` otherwise.

```
bool frameOverrun (void);
```

6.4.2.13 `display()` Method

Shows information (transfer length, speed, endpoint num, etc.) about the Host channel status for debug purposes.

```
void display (void);
```

6.5 Logging USB Packets Processed by the Transactor

6.5.1 USB Packet Processing Logs

The USB Host transactor allows the logging of USB packets activity in a file or on standard output. Log options are set with the `log()` function.

- `log()` enables printing of logs into a file.
- `setLogPrefix()` sets a prefix which is added onto each line of the log (see *setLogPrefix() Method*).

Each log type is chosen independently.

6.5.2 Log Types

You can define log type using the enum `logMask_t` defined in the `UsbStruct.hh` file.

6.5.2.1 Host Controller Log

This log reports the main stages of the Host controller.

Use the `logCore` option to activate it.

6.5.2.2 USB Transactor Log

This log reports the main transactor steps.

Use the `logInfo` option to activate it.

6.5.2.3 Reference Clock Time Log

This log reports the reference clock counter (set by the user).

Use the `logTime` option to activate it.

6.5.2.4 Full Logs

This option sets all log levels detailed above.

Use the logAny option to activate it.

6.5.3 USB Log Example

The following example shows the sequence of a Host sending a control command to set the address of a device. logAny option is used.

```
Host   DEBUG          : clk_48m cycle - 427964
Host   DEBUG          : ZUSB OTG HCD QH Initialized
Host   DEBUG          : clk_48m cycle - 428009
Host   DEBUG          : hc start transfer
Host   DEBUG          : no split
Host   DEBUG          : Number of packet to be Tmited :
Host   DEBUG          : xfer_len   = 8
Host   DEBUG          : max_packet = 64
Host   DEBUG          : 1 packets
Host   DEBUG          : 8 bytes
Host   DEBUG          : Done
Host   DEBUG          : Core_if   : 0xaf07c0
Host   DEBUG          : HC        : 0
Host   DEBUG          : data_fifo : 00001000
Host   DEBUG          : data_fifo : 00001000
Host   DEBUG          : data_fifo : 0x1000
Host   DEBUG          : data_fifo : 0x1000
Host   DEBUG          : data_buff : 0xaf1860
Host                  :   Channel 0
Host                  :   Setup Data             = MSB  00 00 00 00 00 11
05 00
Host                  :   bmRequestType Tranfer   = Host-to-Device
Host                  :   bmRequestType Type      = Standard
```

```
Host                    :   bmRequestType Recipient = Device
Host                    :   bRequest                = SetAddress (0x5)
Host                    :   wValue                  = 0x11
Host                    :   wIndex                  = 0x0
Host                    :   wLength                 = 0x0
```

6.6 Example

This example uses the USB Host transactor with PHY UTMI cable model.

The software testbench is made of two processes: one for the Host part and one for the Device part. The Host sends data to the Device and the Device sends back the data to the Host. The testbench returns OK if the data sent by the Host and the data received from the Device are identical.

6.6.1 Host Process of the Testbench

The Host process is as follows:

1. Initializes the core.
2. Plugs device
3. Waits for device connection.
4. Sends a USB reset.
5. Waits for Port Enabled.
6. Device Enumeration (`Set_Address` and `Get_Descriptor`).
7. Starts `Bulk OUT` transfers by sending to the device bytes read from the file test.in. The main loop in the Host testbench responds to the Host channel events.
   16 channels are available for transfer as well as 16 endpoints on the device side. If the testbench receives an event on the channel configured as `Bulk OUT`, it handles it depending on the status of the channel (`Transfer Complete`, `ACK received`, `Channel HALTED`).
   It is the same for the channel configured for `Bulk IN`: if the testbench receives a `Transfer Complete` event, it writes the received data to the `test.out` file. If no data is present, transfer is completed and the testbench is over. See *Connection/ Disconnection to the USB Device of the DUT*.
8. Unplugs device.

Note

For Bulk OUT transfers, when the Host receives a NAK from the device, it is responsible for re-sending the data. This means that the host must halt the corresponding channel and restart the transfer.

For Bulk IN transfers, a NAK received from the device is automatically handled by the core. You do not have to restart the transfer.

6.6.2 USB Host Testbench Example

Here is an example of testbench using an USB Host transactor. This example is a snippet of the example/phy_utmi/src/bench/tb_host.cc file:

```
UsbHost *usbHst = (UsbHost*)usb;
uint8_t devAddr = 0x11;
HCSpeed_t usbSpeed;

printf(" -- HST -- Starting USB host TB\n");

// Initializes the core in High speed supporting PING protocol
// and uses utmi_16 interface
usbHst->config(spd_high, utmi_16);

printf(" -- HST -- Config done !\n");

// Wait 3ms and plug the device
// Then wait 10 ms before continuing
usbHst->USBPlug(3, 10);

// Wait for device connection
while (!usbHst->isDeviceAttached()) {
  usbHst->loop();
}
printf(" -- HST -- Device connection detected, Sending USB Reset\n");

// Sends USB reset to the device
usbHst->usbReset(10);

printf(" -- HST -- Waiting for port enable\n");

// Wait for the port availability
```

Example

```
  while (!usbHst->portEnabled()) {
    usbHst->loop();
  }
  printf(" -- HST -- Port enabled\n");
  usbSpeed = usbHst->portSpeed();
  printf("Device speed detected : %s\n",
         (usbSpeed == HCSpd_High) ? "High Speed" :
         (usbSpeed == HCSpd_Full) ? "Full Speed" :
         (usbSpeed == HCSpd_Low)  ? "Low Speed"  : "Unknown"
         );

  host_set_address(usbHst, devAddr);  // SET_ADDRESS Transfer

  printf(" -- HST -- Starting GET_DESC\n");
  host_get_descriptor(usbHst);        // GET_DESCRIPTOR Transfer

  printf(" -- HST -- GET_DESC Done\n");

  while (!tbDone) {          // Main loop for events processing

  HostStatus_t  stat = usbHst->loop();
    if (stat.d32 != 0 || !bulk_out_started) {
      if (!bulk_out_started || (stat.d32 & (0x1<<bulkout_hcnum))) {
        host_handle_bulk_out(usbHst, bulkout_epnum);
// Handles BULK OUT events on channel 1
      }
      if (!bulk_in_started || (stat.d32 & (0x1<<bulkin_hcnum))) {
        host_handle_bulk_in(usbHst, bulkin_epnum);
// Handles BULK IN events on channel 2
      }
    }
    fflush(stdout);
  }
```

```
  // Unplug the device
  USBUnplug(0);


////////////////////////////////////
// BULK IN processing function
////////////////////////////////////

void host_handle_bulk_in( UsbHost *usbHst, uint8_t bulkin_epnum)
{
  int               rsl = 0;
  static uint32_t   xfersize;
  static uint32_t*  buff = NULL;
  static hcState_t xferstate = HC_IDLE;

  bool newXfer  = false;
  bool getData = false;

  HostChannel* hc = usbHst->channel(bulkin_hcnum);

  switch (xferstate) {  // State machine for a transfer
  case HC_IDLE:
    printf(" -- HST -- BULK IN State : IDLE \n");
    newXfer    = true;
    xferstate  = HC_XFER;
    break;
  case HC_XFER:
    printf(" -- HST -- BULK IN State : XFER \n");
    if (hc->xferComplete()) {
      getData  = true;
      newXfer    = true;   // Transfer Completed – Ready for new one
    } else if (hc->errorHappened()) {
```

```
      printf(" -- Error happened during BULK IN transfer\n");
      hc->display();
      xferstate = HC_IDLE;
      rsl = -1;
    }
    break;
  case HC_HALT:
    printf(" -- HST -- BULK IN State : NAK \n");
    if (hc->Halted()) {
      printf(" -- HST -- Halt Complete\n");
      newXfer  = true;
      xferstate = HC_XFER;
    }
    break;
  default:
    printf(" -- HST -- BULK IN State : Unknown \n");
    xferstate = HC_IDLE;
  }

  if (getData) {
    if (hc->dataPresent()) {  // Data present in the packet
      usbHst->rcvData(hc, &buff, &xfersize);
      bufdisp ((uint8_t*)buff, xfersize);
      printf(" -- HST -- Received Data : %d bytes\n", xfersize);
      ssize_t w = fwrite((void*)buff, 1, xfersize, outFile);
      fflush(outFile);
      if (w<0) {
        rsl = -1;
      } else if (w==0) {
        printf(" -- HST -- No data dumped in out file(received %d
bytes)\n",xfersize);
        tbDone = 1;
        rsl = -1;
```

```
      } else if (w<xfersize) {
        printf(" -- HST -- Only %d bytes out of %d were dump in out
file\n",w,xfersize);
      }
    } else {
      printf(" -- HST -- BULK IN completed with no data\n");
      tbDone = true;
    }
  }
  if (newXfer) {
    printf(" -- HST -- Preparing host channel for BULK IN reception\n",
xfersize);
    usbHst->hcInit ( hc, XferTyp_Bulk , UsbSpeed, device_addr,
bulkin_epnum, 1, usbHst->maxPktSize());
// Initializes Host channel for BULK IN Transfer
    usbHst->requestData(hc, xfersizemax);  // Requests for data
    bulk_in_started = 1;
  }
}
```

You can find another testbench example in the `example/phy_cable/src/bench` directory, also named tb_host.cc.

7 URB Software API

A USB Request Block (URB) is the Linux representation of a USB data transfer.

The URB API provides a high-level representation of USB data transfers that the USB Host transactor can use to manipulate USB data transfers.

The URB API software interface allows the writing of testbenches like any other Linux USB driver. The transactor controller behavior is based on Linux kernel revision 2.6.18.

7.1 Using the URB API

7.1.1 Libraries

The URB API is defined in the `UsbUrbHost.hh` and `UsbUrbCommon.hh` header files.

The available library files are:

- `libUsbUrb.so`64-bit gcc 3.4 library
- `libUsbUrb_6.so`32-bit gcc 3.4 library

The following libraries contain the Linux 2.6.18 adaptor part and are dynamically loaded by the `libUsbUrb` libraries:

- `libUsbUL.so`64-bit gcc 3.4 library
- `libUsbUL_6.so`32-bit gcc 3.4 library

The figure below is an overview of the library dependencies when building a software testbench using the USB transactor.

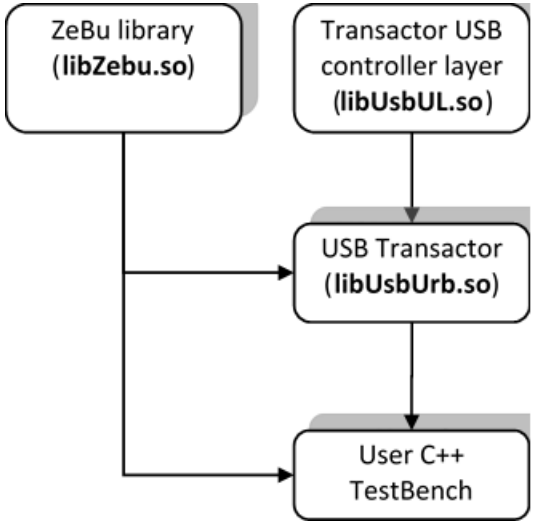

FIGURE 16. Libraries Hierarchy Overview

7.1.2 Data Alignment

The data received and transmitted over the USB BFM are packed into `unsigned char` arrays or `unsigned int` arrays. The LSB byte or the first word of the USB packet data is stored at location 0 of the array.

7.2 URB API Class and Structure Description

The USB Host transactor can be instantiated and accessed using the following C++ classes and structures:

TABLE 20 URB API Class description

Class	Description
UsbHost	Represents the USB Host transactor

TABLE 21 URB API Structures description

Class	Description
`ZebuUrb`	Represents a USB Request Block. This structure is equivalent to the Linux URB structure. It can be used to send/receive data to/from an endpoint of a USB device. It contains information about the addressed endpoint, the type of transfer to submit, and some rules related to the type of transfer.

7.3 UsbHost Class

7.3.1 Description

The methods associated with the `UsbHost` object allows you to manage the transactor and control its status using a standard USB Request Block layer as found in the most common USB device driver software. The `UsbHost` class is defined in the `UsbUrbHost.hh` file.

The following table describes the URB type available for the URB API `UsbHost` class defined in `UsbUrbCommon.hh`:

TABLE 22 URB Type

Type	Description
`zusb_Complete`	Pointer to completion function of a ZeBu URB.

The following table lists the standard USB structures defined in `UsbUrbCommon.hh` available for the URB Host API:

TABLE 23 USB Structures

Structure	Description
`zusb_Direction`	Transfer direction, IN or OUT
`zusb_DeviceSpeed`	USB device speed
`zusb_TransferType`	Isochronous, interrupt, control, or Bulk
`zusb_RequestType`	USB Hub request type (standard, class or vendor)
`zusb_Request`	USB standard control requests
`zusb_DescriptorType`	Type of the descriptor accessed
`zusb_ClassCode`	Class of the device (For example, Audio, Printer, and so on)
`zusb_TransferFlags`	Information about the URB transfer
`zusb_RequestRecipient`	Recipient of the request

TABLE 23 USB Structures

Structure	Description
zusb_IsoSync	Synchronization type for isochronous transfers
zusb_isoPacketDescriptor	Descriptor for isochronous packets
zusb_DescriptorHeader	Common header of descriptors
zusb_DeviceDescriptor	Device descriptor
zusb_EndpointDescriptor	Endpoint descriptor
zusb_InterfaceDescriptor	Interface descriptor
zusb_ConfigDescriptor	Configuration descriptor
zusb_ControlSetup	Description of control transfers setup commands

The following table gives an overview of the available methods for the URB API UsbHost class.

TABLE 24 Methods for UsbHost Class

Method	Description
Transactor Initialization and Control	
UsbHost	Constructor.
~UsbHost	Destructor.
Init	ZeBu Board attachment.
InitBFM	USB transactor configuration & initialization.
Loop	Processes all pending Tx/Rx USB packets, waits for BFM events/interrupts.
DiscoverDevice	Checks device connection. Enables port and resets the USB device if present. Returns ZUSB_STATUS_SUCCESS if the device is ready for transfers.
Delay	Inserts a delay in milliseconds of USB clock.
UDelay	Inserts a delay for of USB clock in microseconds.

TABLE 24 Methods for UsbHost Class

Method	Description
RegisterCallBack	Registers a service loop callback.
SetDebugLevel	Selects the logging mode.
SetLogPrefix	Sets a log prefix to be used in log information.
SetLog	Sets a file where logs will be saved.
SetTimeOut	Sets the time of the watchdog.
RegisterTimeOutCB	Registers a callback to be called if watchdog is activated.
getVersion	Returns either a string or a float corresponding to the transactor version.
USB Device Information	
DevicePresent	Returns true if the device is connected and initialized.
GetDeviceSpeed	Returns device speed (High or Full).
GetMaxPacketSize	Returns the maximum packet size of a transfer.
USB Operations	
USBPlug	Connects the USB Host to the cable.
USBUnplug	Disconnects the USB Host from the cable.
USB Device Configuration Management	
SetDeviceAddress	Sets the device address.
GetDeviceAddress	Gets the device address.
SetDeviceConfig	Sets the configuration number.
GetDeviceConfig	Gets the selected configuration number.
GetRawConfiguration	Returns the full device configuration in raw format.
SetDeviceInterface	Selects a new interface and alternate setting.
GetDeviceAltInterface	Gets the current alternate setting of an interface.
General Purpose Vectors Operations	
writeUserIO	Drives the general purpose bus_0 vector with the specified value

TABLE 24 Methods for UsbHost Class

Method	Description
`readUserIO`	Reads the general purpose `bus_I` vector
Advanced User Methods	
`usbReset`	Sends the USB Reset sequence to the USB Device.
`portEnabled`	Indicates if the USB port is enabled.
`isDeviceConnected`	Indicates if a device is connected to the USB bus.
`BypassHubReset`	Bypasses the reset activated on the `discoverDevice` method
`SetFastTimer`	Reduces internal delays

7.3.2 Transactor Initialization and Control Methods

7.3.2.1 Constructor/Destructor Methods

Allocates/frees the structures for the USB Host transactor.

```
UsbHost (void)
~UsbHost (void)
```

7.3.2.2 `Init()` Method

Connects to the ZeBu board.

```
void Init (Board *zebu, const char *driverName,
           const char *clkName=NULL);
```

Parameter Name	Parameter Type	Description
zebu	Board *	Pointer to the ZeBu board structure.
driverName	const char *	Name of the transactor instance in the hw_top file.
clkName	const char *	Name of the primary clock used as a time reference in debug messages (optional).

Note *When the Bus logging feature is used (see USB Bus Monitoring Feature), the last argument (clkName) must be specified and must have the same name as the transactor clock. In the examples shown in Verification Environment Example, the name is host_xtor_clk.*

7.3.2.3 `InitBFM()` Method

Configures the USB Host transactor and allocates software structures accordingly. This method needs to be called after `Init()` and before any operation is performed on the USB Host transactor.

```
zusb_status InitBFM (bool highSpeedSupport = true, bool IsUtmi16);
```

Parameter Name	Parameter Type	Description
highSpeedSupport	bool	Configures the controller to support: • `true`: High-Speed and Full-Speed devices • `false`: Full-Speed devices only
IsUtmi16	bool	Configures the physical UTMI width: • `false` (default): 8 bits • `true`: 16 bits.

It returns ZUSB_STATUS_SUCCESS if successful, error status otherwise (See *zusb_status Enum*).

7.3.2.4 Loop() Method

Checks the USB Host transactor status and performs any needed operations. If an event occurs during the loop call, software structures and returned value are set accordingly. This method must be called regularly to allow the controlled clock to advance and to check events.

```
zusb_status Loop (void);
```

If successful, the method returns ZUSB_STATUS_SUCCESS.

Otherwise, it returns any other status of the zusb_status enum (see *zusb_status Enum*).

7.3.2.5 `DiscoverDevice()` Method

Checks for device connections on controller ports.

```
zusb_status DiscoverDevice (void);
```

If a device is connected, the port is enabled and a USB reset is sent to the device.

If the method returns ZUSB_STATUS_SUCCESS, the device is ready to be accessed.

7.3.2.6 `Delay()` Method

Allows the USB clock to advance for the specified number of milliseconds before performing the next USB operation.

```
void Delay (uint32_t msec, bool blocking = false);
```

Parameter Name	Parameter Type	Description
msec	uint32_t	Delay of the UTMI clock in milliseconds.
blocking	bool	Operation type: • false (default): non-blocking operation; the method returns immediately • true: blocking operation; the method returns after the operation is completed

To set a delay in microseconds, use the UDelay() method (See *UDelay() Method*).

7.3.2.7 UDelay() Method

Allows the USB clock to advance for the specified number of microseconds before performing the next USB operation.

```
void UDelay (uint32_t usec);
```

Parameter Name	Parameter Type	Description
usec	uint32_t	Delay of the UTMI clock in milliseconds.

To set a delay in milliseconds, use the Delay() method (See *Delay() Method*).

7.3.2.8 RegisterCallBack() Method

Registers a callback to be called during service loop.

```
void RegisterCallBack (void (*userCB)(void* CBParams),
                       void *CBParams);
```

Parameter Name	Parameter Type	Description
userCB	void (*fct)(void*)	Pointer to function
CBParams	void*	Context to be used in callback

7.3.2.9 SetDebugLevel() Method

Enables and configures the transactor's log mechanism.

```
void SetDebugLevel (uint32_t val);
```

where val is the log level as defined hereafter:

TABLE 25 Log levels for SetDebugLevel Method

val value	Description
logOff	No information.
logInfo	Information about USB device detection and configuration.
logUrb	Information on URBs.
logCore	Information on controller events (channels, endpoints, etc.)
logBuf	Information on data transferred through the bus.
logTime	Information on reference clock value at different stages of the run.

See *Logging USB Transfers Processed by the Transactor* for further details on the available levels.

7.3.2.10 SetLogPrefix() Method

Sets a new log prefix for log.

```
void SetLogPrefix (const char *prefix);
```

where prefix is the prefix to use in logs.

7.3.2.11 SetLog() Method

Sets the file where to save logs.

```
void SetLog(FILE *stream, bool stdoutDup);
```

Parameter Name	Parameter Type	Description
stream	FILE *	File name.
stdoutDup	bool	Activates/deactivates the standard output: • true: log information is output to the standard output as well as in the log file • false: log information is output only to the log file.

7.3.2.12 SetTimeOut() Method

Sets the value of the watchdog timeout in milliseconds.

```
void SetTimeOut(uint32_t msecs);
```

where msecs is the timeout value in milliseconds.

7.3.2.13 RegisterTimeOutCB() Method

Registers a callback to be called when the timeout has been reached.

```
void RegisterTimeOutCB(bool (*timeoutCB) (void*), void* context);
```

Parameter Name	Parameter Type	Description
timeoutCB	bool (*fctPT) (void*)	Pointer to callback function
context	void*	Context to be passed to callback

7.3.2.14 getVersion() Method

Two different prototypes exist for this method:

- Returns a string corresponding to the transactor version.

```
static const char* getVersion (void);
```

- Returns a float corresponding to the transactor version.

```
static void getVersion (float &version_num);
```

where version_num is the transactor version number.

7.3.3 USB Device Information Methods

These methods get information about the characteristics of the attached USB Device.

7.3.3.1 DevicePresent() Method

Indicates if a USB device is connected.

```
bool DevicePresent () const;
```

It returns true if the USB device is successfully attached, false otherwise.

7.3.3.2 GetDeviceSpeed() Method

Returns the speed mode (High or Full) of the attached USB Device.

```
zusb_DeviceSpeed GetDeviceSpeed (void);
```

Returned values are of zusb_DeviceSpeed type.

7.3.3.3 GetMaxPacketSize() Method

Returns the maximum packet size. The value depends on the device speed.

```
uint32_t GetMaxPacketSize (void);
```

7.3.4 USB Operations

7.3.4.1 USBPlug() Method

Connects the host to the USB cable with one of the following methods:

```
zusb_status USBPlug (uint32_t duration);
zusb_status USBPlug (uint32_t latency, uint32_t duration);
```

Parameter Name	Parameter Type	Description
duration	uint32_t	Time to wait after the connection of the host (in ms).
latency	uint32_t	Time to wait before connecting the host (in ms).

7.3.4.2 USBUnplug() Method

Disconnects the host from the USB cable with one of the following methods:

```
zusb_status USBUnplug (uint32_t duration);
zusb_status USBUnplug (uint32_t latency, uint32_t duration);
```

Parameter Name	Parameter Type	Description
duration	uint32_t	Time to wait after the host disconnection (in ms).
latency	uint32_t	Time to wait before disconnecting the host (in ms).

7.3.5 USB Device Configuration Management Methods

7.3.5.1 `SetDeviceAddress()` Method

Sets an expected address to the USB device. It must be called before the device discovery.

```
zusb_status SetDeviceAddress (uint8_t addr);
```

where addr is the address to set. It can range from 1 to 127.

7.3.5.2 `GetDeviceAddress()` Method

Gets the address of the connected USB device.

```
uint8_t GetDeviceAddress (void) const;
```

7.3.5.3 `SetDeviceConfig()` Method

Defines an expected configuration number for the USB device. It must be called before the device discovery.

```
zusb_status SetDeviceConfig (uint32_t config);
```

where config is the number of the configuration to set.

7.3.5.4 GetDeviceConfig() Method

Gets the configuration selected for the connected device.

```
uint32_t GetDeviceConfig () const;
```

7.3.5.5 GetRawConfiguration() Method

Gets the full configuration of the device as a uint8_t* buffer. The buffer contains the full device descriptor as defined in the USB 2.0 specification. It is composed of all standard descriptors required to communicate with the device, meaning all configuration descriptors, interface descriptors, endpoint descriptors, string descriptors, etc. It is the user's responsibility to parse the configuration; the size of the buffer depends on the device itself.

```
uint8_t* GetRawConfiguration ();
```

7.3.5.6 SetDeviceInterface() Method

Sets an expected interface and alternate setting number for the device. It must be called when the device is discovered and initialized.

```
zusb_status SetDeviceInterface (uint32_t intf, uint32_t altsetting);
```

Parameter Name	Parameter Type	Description
intf	uint32_t	Interface to set.
altsetting	uint32_t	Number of the alternate setting to set.

7.3.5.7 `GetDeviceAltInterface()` Method

Gets the chosen alternate interface used for a specific interface.

```
uint32_t GetDeviceAltInterface (uint32_t intf);
```

where `intf` is the user interface.

7.3.6 General Purpose Vectors Operations

7.3.6.1 `writeUserIO()` Method

Drives the 4-bit general purpose `bus_O` vector with the specified value.

```
void writeUserIO (const uint32_t value);
```

where `value` is the vector's value. It ranges from `0` to `15`.

7.3.6.2 `readUserIO()` Method

Reads the 4-bit general purpose `bus_I` vector.

```
uint32_t readUserIO (void);
```

7.3.7 Advanced User Methods

7.3.7.1 `usbReset()` Method

Initiates a USB Reset sequence on the bus to the USB device.

```
void usbReset (uint8_t length=10);
```

where `length` is the reset duration in milliseconds. Default value is `10`. This parameter is optional.

7.3.7.2 portEnabled() Method

Indicates if the USB port is enabled.

```
bool portEnabled (void);
```

The method returns:

- `true` when the USB port is enabled
- `false` when the USB port is disabled

7.3.7.3 isDeviceConnected() Method

Checks if the USB device is physically connected.

```
bool isDeviceConnected (void);
```

This method returns:

- `true` when a device is connected to the USB bus
- `false` when no device is connected to the USB bus.

7.3.7.4 BypassHubReset() Method

Bypasses the internal reset activated on the `discoverDevice` method for emulation speed up

```
void BypassHubReset(int value = 0);
```

where `value` must be set to:

- `1` for bypass
- `0` to keep reset (default)

7.3.7.5 SetFastTimer() Method

Reduces internal delays for emulation speed up

```
void SetFastTimer(int value = 0);
```

where value must be set to:

- 1 to reduce internal delays
- 0 to disable this feature (default)

7.3.7.6 Example

```
usb_reset(&usbHst, 10);

//--- wait_dev_attach
while (!hst->IsDeviceConnected()) {
     sched_yield();
   }

//wait_port_enable
  while (!hst->PortEnabled()) {
  hst->UDelay(10);
  }

  // wait hub attach
  int32_t st = ZUSB_STATUS_NO_DEVICE;
  hst->BypassHubReset(1);
  hst->SetFastTimer(1);

  do {
    st = hst->DiscoverDevice();
  } while(st == ZUSB_STATUS_NO_DEVICE);
```

7.4 ZebuUrb Structure

The ZeBuUrb structure contains USB request block information as shown below. This structure is described in the UsbUrbHost.hh file:

```
struct ZebuUrb
{
  uint32_t                 endpoint_num;
  zusb_TransferType        type;
  zusb_Direction           dir;
  int32_t                  status
  uint32_t                 transfer_flags
  uint8_t*                 transfer_buffer;
  int32_t                  transfer_buffer_length;
  int32_t                  actual_length;
  uint8_t*                 setup_packet;
  int32_t                  start_frame;
  int32_t                  number_of_packets;
  int32_t                  interval;
  int32_t                  error_count;
  void*                    context;
  zusb_Complete            complete;
  zusb_isoPacketDescriptor* iso_frame_desc;
 } ;
```

Parameter	Description
endpoint_num	Device Endpoint address (0 to 15)
type	Transfer type (Isochronous, Interrupt, Control, or Bulk)
dir	Transfer direction (IN or OUT)

Parameter	Description
status	Current status of the ZebuUrb block. The status returned is one of the zusb_transfer_status enum.
transfer_flags	Transfer option: • ZUSB_SHORT_NOT_OK: read operations smaller than the Endpoint max packet size are not supported • ZUSB_ISO_ASAP: isochronous transfer as soon as possible (ignoring interval parameter)
transfer_buffer	Pointer to the transfer buffer
transfer_buffer_length	Length of the data pointed by transfer_buffer (in bytes)
actual_length	Length of the data effectively transferred. For Non-isochronous transfers only.
setup_packet	Pointer to the setup transfer buffer
start_frame	Sets/returns initial frame for isochronous transfer (isochronous only)
number_of_packets	Number of isochronous packets.
interval	Specifies polling interval for interrupt/isochronous transfers. Unit is: • Microframe (1/8 millisecond) for High Speed devices • Frame (1 millisecond) for Full Speed devices
error_count	Returns the number of ISO transfer that reported an error
context	Pointer to a data context that can be used in completion function
complete	Pointer to a completion handler function that is called when the ZeBu URB transfer is completed or when an error occurs during transfer.
iso_frame_desc	Array of isochronous transfer buffer and status (isochronous only). See description below.

iso_frame_desc is structured as follows:

```
struct zusb_isoPacketDescriptor {
    uint32_t offset;
    uint32_t length;
    uint32_t actual_length;
    uint32_t status;
  };
```

Parameter	Description
offset	Offset inside a ZeBu URB transfer_buffer memory block.
length	Size of this isochronous packet.
actual_length	Length of the data effectively received in transfer buffer.
status	Status of the individual isochronous transfer.

7.5 URB API Enum Definitions

All the following enums are described in the `UsbUrbCommon.hh` file.

7.5.1 `zusb_status` Enum

`zusb_status` enum contains the statuses returned by the URB API methods. They are listed in the following table:

TABLE 26 zusb_status Enum Description

Parameter	Description
ZUSB_STATUS_SUCCESS	Function success
ZUSB_STATUS_INVALID_PARAM	Invalid parameter passed to the function
ZUSB_STATUS_NO_DEVICE	No device connected yet
ZUSB_STATUS_NOT_FOUND	Request not found
ZUSB_STATUS_OVERFLOW	Requested data out of range
ZUSB_STATUS_NOT_SUPPORTED	Command not supported
ZUSB_STATUS_OTHER	Generic error status

7.5.2 zusb_transfer_status Enum

`zusb_transfer_status` enum contains the statuses of transfers returned by the `ZebuUrb` structure. They are listed in the following table:

TABLE 27 zusb_transfer_status Enum Description

Parameter	Description
ZUSB_TRANSFER_COMPLETED	ZeBu USB transfer completed successfully
ZUSB_TRANSFER_PENDING	ZeBu USB transfer is pending
ZUSB_TRANSFER_ERROR	ZeBu USB transfer finished with error
ZUSB_TRANSFER_TIMED_OUT	ZeBu USB transfer timed out

TABLE 27 zusb_transfer_status Enum Description

Parameter	Description
ZUSB_TRANSFER_CANCELLED	ZeBu USB transfer is cancelled
ZUSB_TRANSFER_STALL	ZeBu USB transfer is stalled
ZUSB_TRANSFER_NO_DEVICE	ZeBu USB transfer cancelled because device was disconnected
ZUSB_TRANSFER_OVERFLOW	ZeBu USB transfer data overflow

7.6 Managing USB Host URBs

This section explains the methods to send or receive USB blocks to/from the device.

7.6.1 Creating and Destroying ZeBu URBs

You should allocate and destroy ZeBu URBs using the `AllocUrb()` and `FreeUrb()` functions. Allocating the structure statically is not allowed.

It is mandatory to use the functions `AllocBuffer()` and `FreeBuffer()` to allocate and delete data buffers of the `ZebuUrb` structures.

TABLE 28 USB Host URB Creation and Destruction Methods

Method	Description
`AllocUrb`	Allocates a ZeBu URB structure
`FreeUrb`	Frees a ZeBu URB structure
`AllocBuffer`	Allocates a buffer for a ZeBu URB
`FreeBuffer`	Frees a buffer of a ZeBu URB

7.6.1.1 `AllocUrb()` Method

Allocates a `ZebuUrb` structure.

```
static ZebuUrb* AllocUrb (int32_t iso_packets);
```

where `iso_packets` is the number of ISO packets this ZeBu URB should contain (Isochronous transfers only). It should be set to 0 for transfers other than isochronous.

This method returns a pointer to ZeBu URB in case of success, `NULL` otherwise.

7.6.1.2 `FreeUrb()` Method

Frees a `ZeBuUrb` structure.

```
static void FreeUrb (ZebuUrb* zurb) ;
```

where zurb is the pointer to the ZebuUrb to delete.

7.6.1.3 AllocBuffer() Method

Allocates a buffer for a ZeBu URB.

```
uint8_t* AllocBuffer (int32_t size) ;
```

where size is the buffer size to allocate in bytes.

7.6.1.4 FreeBuffer() Method

Frees a buffer of a ZeBu URB.

```
void FreeBuffer (uint8_t* buffer) ;
```

where buffer is the pointer to the buffer structure to delete.

TABLE 29 USB Host URB Filling Methods

Method	Description
FillBulkUrb	Properly fills a ZeBuUrb structure to be sent to a Bulk endpoint.
FillIntUrb	Properly fills a ZeBuUrb structure to be sent to an interrupt endpoint.
FillControlUrb	Properly fills a ZeBuUrb structure to be sent to a control endpoint.
FillIsoUrb	Properly fills a ZeBuUrb structure to be sent to an isochronous endpoint.
FillIsoPacket	Must be used to add an isochronous packet to a created isochronous URB.

7.6.1.5 `FillBulkUrb()` Method

Properly fills a `ZeBuUrb` structure to be sent to a `Bulk` endpoint.

```
zusb_status FillBulkUrb (ZebuUrb *zurb,
                      uint8_t endpoint_num,/* end point number */
                      zusb_Direction dir,/* direction */
                      uint8_t *transfer_buffer,
                      int32_t buffer_length,
                      zusb_Complete complete,
                      void *context);
```

Parameter Name	Parameter Type	Description
`zurb`	`ZebuUrb *`	Pointer to the URB to fill
`endpoint_num`	`uint8_t`	Endpoint address
`dir`	`zusb_Direction`	Direction of the transfer
`transfer_buffer`	`uint8_t*`	Pointer to the transfer buffer
`buffer_length`	`int32_t`	Length of the transfer buffer
`complete`	`zusb_Complete`	Pointer to the URB completion function
`context`	`void*`	Pointer to a data context that can be used in completion function

7.6.1.6 `FillIntUrb()` Method

Properly fills a `ZeBuUrb` structure to be sent to an interrupt endpoint.

```
zusb_status FillIntUrb (ZebuUrb *zurb,
                     uint8_t endpoint_num,/* end point number */
                     zusb_Direction dir,/* direction */
                     uint8_t *transfer_buffer,
                     int32_t buffer_length,
```

```
                    zusb_Complete complete,
                    void *context,
                    int32_t interval) ;
```

Parameter Name	Parameter Type	Description
zurb	ZebuUrb *	Pointer to the URB to fill
endpoint_num	uint8_t	Endpoint address
dir	zusb_Direction	Direction of the transfer
transfer_buffer	uint8_t*	Pointer to the transfer buffer
buffer_length	int32_t	Length of the transfer buffer
complete	zusb_Complete	Pointer to the URB completion function
context	void*	Pointer to a data context that can be used in completion function
interval	int32_t	Specifies the polling interval. Unit is: • Frame (1 millisecond) for Full Speed devices • Microframe (1/8 millisecond) for High Speed devices

7.6.1.7 `FillControlUrb()` Method

Properly fills a ZeBuUrb structure to be sent to a control endpoint.

```
zusb_status FillControlUrb (ZebuUrb *zurb,
                        uint8_t endpoint_num,/* end point number */
                        zusb_Direction dir,  /* direction */
                        uint8_t *setup_packet,
                        uint8_t *transfer_buffer,
                        int32_t buffer_length,
                        zusb_Complete complete,
                        void *context) ;
```

Parameter Name	Parameter Type	Description
zurb	ZebuUrb *	Pointer to the URB to fill.
dir	zusb_direction	Direction of the transfer.
setup_packet	uint8_t *	Pointer to the control setup buffer.
transfer_buffer	uint8_t *	Pointer to the transfer buffer.
buffer_length	int32_t	Length of the transfer buffer.
complete	zusb_Complete	Pointer to the URB completion function.
context	void *	Pointer to a data context that can be used in completion function.

7.6.1.8 FillIsoUrb() Method

Properly fills a ZeBuUrb structure to be sent to an isochronous endpoint.

```
zusb_status FillIsoUrb (ZebuUrb *zurb,
                    uint8_t endpoint_num,/* end point number */
                    zusb_Direction dir,/* direction */
                    uint8_t *transfer_buffer,
                    int32_t buffer_length,
                    int32_t number_of_packets,
                    zusb_Complete complete,
                    void *context,
                    int32_t interval);
```

Parameter Name	Parameter Type	Description
zurb	ZebuUrb *	Pointer to the URB to fill
endpoint_num	uint8_t	Endpoint address
dir	zusb_Direction	Direction of the transfer

Parameter Name	Parameter Type	Description
transfer_buffer	uint8_t *	Pointer to the transfer buffer
buffer_length	int32_t	Length of the transfer buffer
number_of_packets	int32_t	Number of ISO packets
complete	zusb_Complete	Pointer to the URB completion function
context	void *	Pointer to a data context that can be used in completion function
interval	int32_t	Specifies the polling interval. Unit is: • Frame (1 millisecond) for Full Speed devices • Microframe (1/8 millisecond) for High Speed devices

A helper function is available to fill IsoPackets.

7.6.1.9 FillIsoPacket() Method

Fills the packet of an isochronous transfer.

```
zusb_status FillIsoPacket (ZebuUrb *zurb,
             uint32_t pktnum,
             uint32_t offset,
             uint32_t length);
```

Parameter Name	Parameter Type	Description
zurb	ZebuUrb *	Pointer to the URB to fill
pktnum	uint32_t	Packet number (starting from 0 to number of ISO packets -1)
offset	uint32_t	Offset in transfer buffer
length	uint32_t	Length of the packet

7.6.2 Submitting ZeBu URBs

Once the ZebuUrb structure is properly filled using the above methods, it is ready to be transferred to the device. This is done using the SubmitUrb() function.

The SubmitUrb() function submits a ZeBuUrb to the device.

```
zusb_status SubmitUrb (ZebuUrb* zurb);
```

where zurb is the pointer to the URB to submit.

7.6.3 Cancelling ZeBu URBs

Cancels a transfer request for an endpoint using the submitted the URB. On successful cancellation, the URB terminates and the completion handler is called.

A failure indicates that the URB is either not submitted or the hardware has already processed it.Do not release the while the cancel is in progress, that is, the completion handler has not been called yet.

The cancelUrb function cancels an URB pointed by a ZeBuURB structure.

```
zusb_status cancelUrb (ZebuUrb* zurb);
```

where: zurb is the pointer to submitted URB.

7.6.4 USB Transfers without ZeBu URBs

Two functions are available to provide a simpler interface without manipulating ZeBu URBs. They are blocking functions and only return when transfer is finished.

Table 27: Methods for USB Transfer without ZeBu URBs

Method	Description
SendControlMessage	Sends a control message to the device
SendBulkMessage	Sends a BULK message to the device

7.6.4.1 SendControlMessage() Method

Sends a control message and waits for the message transfer to complete. Function parameters correspond to the zusb_ControlSetup structure fields (See the USB specification).

```
zusb_status SendControlMessage (zusb_Direction dir,
                                uint8_t request,
                                uint8_t requesttype,
                                uint16_t value,
                                uint16_t index,
                                uint8_t* data,
                                uint16_t size,
                                int32_t timeout);
```

Parameter Name	Parameter Type	Description
dir	zusb_direction	Direction of the transfer
request	uint8_t	Matches the USB bmRequest field
requesttype	uint8_t	Matches the USB bRequest
value	uint16_t	Matches the USB wValue field
index	uint16_t	Matches the USB wIndex field
data	uint8_t*	Pointer to the data to send
size	uint16_t	Length of the data to send in bytes
timeout	int32_t	Time (ms) to wait for message to complete before timeout. If value is 0, the default timeout value defined using SetTimeOut() method is used.

On success, this method returns the number of bytes that have been transferred.

On error, it returns one of the statuses of the zusb_status enum (see *zusb_status Enum*).

7.6.4.2 SendBulkMessage( ) Method

Sends a Bulk message and waits for the message to complete.

```
zusb_status SendBulkMessage (uint8_t endpoint_num,
                             zusb_Direction dir,
                             uint8_t *data,
                             int32_t len,
                             int32_t &actual_length,
                             int32_t timeout);
```

Parameter Name	Parameter Type	Description
endpoint_num	uint8_t	Endpoint number
dir	zusb_Direction	Direction of the transfer
data	uint8_t*	Data to transfer
len	uint32_t	Length of the data to transfer
actual_length	int32_t &	Actual data length transferred
timeout	int32_t	Time (ms) to wait for message to complete before timeout. If value is 0, the default timeout value defined using SetTimeOut( ) method is used.

On success, this method returns the number of bytes transferred.

On error, it returns the statuse of the zusb_status enum. For more information, see *zusb_status Enum*.

7.6.5 Examples

7.6.5.1 Bulk URB Transfer with ZeBu URBs

```
UsbHost usbXtor;
struct ctxStruct {
  uint32_t complete_val;
  UsbHost* xtor;
};
static void mycompletefct(ZebuUrb* zurb) {
  ctxStruct* theStruct = static_cast<ctxStruct*>(zurb->context);
  theStruct->complete_val = 1;
}
ctxStruct myStruct ;
myStruct.complete_val = 0;
myStruct.xtor = &usbXtor;

uint8_t*  buff = UsbHost::AllocBuffer(512);
int actual_length = 0;
ZebuUrb* myurb = UsbHost::AllocUrb(0);
usbHst.FillBulkUrb (myurb, 7 /*device addr*/, ZUSB_DIR_OUT, buff, 512,
mycompletefct, &myStruct);

myurb->transfer_flags = 0;
zusb_status ret = usbHst.SubmitUrb(myurb);
if(ret != ZUSB_STATUS_SUCCESS) {
  printf("Submit failed with code = %d\n", ret);
  return 1;
}
while(!myStruct.complete_val) {
  if((ret = usbHst.Loop()) != ZUSB_STATUS_SUCCESS) {
    if(ret == ZUSB_STATUS_NO_DEVICE) {
      printf("Device deconnection detected, exit!\n");
    }
    else {
```

```
      printf("Error detected, exit!\n");
    }
    return 1;
  }
}
if(myurb->status != 0) {
  printf("End bulk out transfer status = %d\n", myurb->status);
  return 1;
}
UsbHost::FreeUrb(myurb);
```

7.6.5.2 Bulk URB Transfer without ZeBu URBs

```
UsbHost usbXtor;
uint8_t*  buff = UsbHost::AllocBuffer(512);
int32_t actual_length = 0;
zusb_status ret = 0;

ret = usbHst.SendBulkMessage( 3 /*EP address*/,
                              ZUSB_DIR_OUT,
                              buff,
                              512,
                              &actual_length,
                              0);

if(ret != ZUSB_STATUS_SUCCESS) {
  printf("Bulk out transfer error status = %d\n", ret);
}
else {
  printf("Bulk out done\n");
}
```

7.6.5.3 Isochronous URB Transfer

In this example, the isochronous transfer is repeated indefinitely by calling a new submit inside the completion function.

```
#define NUM_ISOC 4
#define ISO_PACKET_SIZE 256

UsbHost usbXtor;

struct ctxStruct {
  uint32_t complete_val;
  UsbHost* xtor;
};
static void mycompleteIsocOutfct(ZebuUrb* zurb) {
  ctxStruct* theStruct = static_cast<ctxStruct*>(zurb->context);

  int j = 0;
  int k = 0;

  if(zurb->status != 0) {
    printf("ISOC OUT transfer error, ZURB status = %d\n", zurb->status);
    exit(1);
  }
  printf("ISOC OUT Transfer %u, packet size %u, num packets %u\n",
num_transfer, ISO_PACKET_SIZE, NUM_ISOC);
  for (j=k=0; j < NUM_ISOC; j++, k += ISO_PACKET_SIZE) {
    theStruct->xtor->FillIsoPacket(zurb, j, k, ISO_PACKET_SIZE);
  }

  // Submitting a new isochronous URB
  zurb->actual_length = 0;
  theStruct->xtor->SubmitUrb(zurb);
```

```
}

ctxStruct myStruct ;
myStruct.xtor = &usbXtor;

myurb = UsbHost::AllocUrb(NUM_ISOC);
uint8_t*  buff = UsbHost::AllocBuffer(ISO_PACKET_SIZE * NUM_ISOC);

usbHst.FillIsoUrb (myurb, (epIsocOut->bEndpointAddress) &
ZUSB_ENDPOINT_NUMBER_MASK, ZUSB_DIR_OUT, buff, ISO_PACKET_SIZE * NUM_ISOC,
NUM_ISOC, mycompleteIsocOutfct, &myStruct, 1/*interval*/);

myurb->transfer_flags = ZURB_ISO_ASAP;

for (j=k=0; j < NUM_ISOC; j++, k += ISO_PACKET_SIZE) {
  usbHst.FillIsoPacket(myurb, j, k, ISO_PACKET_SIZE);
}

myurb->actual_length = 0;
zusb_status ret = usbHst.SubmitUrb(myurb);
if(ret != ZUSB_STATUS_SUCCESS) {
  printf("Submit failed with code = %d\n", ret);
  return 1;
}

while(1) {
  if((ret = usbHst.Loop()) != ZUSB_STATUS_SUCCESS) {
    if(ret == ZUSB_STATUS_NO_DEVICE) {
      printf("Device deconnection detected, exit!\n");
    }
    else {
      printf("Error detected, exit!\n");
    }
```

```
    UsbHost::FreeUrb(myurb);
    return 1;
  }
}
```

8 Watchdogs and Timeout Detection

8.1 Description

To avoid deadlocks during USB transfers, the USB Host transactor includes internal watchdogs that can be configured from the application. By default, watchdogs are enabled with a timeout value of 180 seconds.

The URB API provides methods for managing watchdogs and timeout detection, which are explained in this section.

8.2 URB API Methods List

The following URB methods control watchdogs and timeout detection.

TABLE 30 Methods for Watchdogs and Timeout Detection

Method	Description
`EnableWatchdog`	Enables/disables watchdogs at any time.
`SetTimeOut`	Sets the watchdog timeout values in milliseconds.
`RegisterTimeOutCB`	Registers a callback which is called at each timeout occurrence.

8.2.1 EnableWatchdog() Method

Allows the application to enable/disable the watchdogs at any time.

```
void EnableWatchdog (bool enable);
```

If no argument is specified or if enable is `true`, watchdogs are enabled. Otherwise, they are disabled.

8.2.2 SetTimeOut() Method

Sets the watchdog timeout values in milliseconds (`msec`).

```
void SetTimeOut (uint32_t msec) ;
```

8.2.3 RegisterTimeOutCB() Method

Registers a callback, which is called at each timeout occurrence.

```
void RegisterTimeOutCB (bool (*timeoutCB) (void *context),
                        void *context);
```

Parameter Name	Parameter Type	Description
timeoutCB	bool	Pointer to callback function
context	void *	Pointer to the data passed to the callback

If the deadlock cannot be fixed from the callback, the method returns true and the transactor exits the application correctly.

If the callback returns false, the transactor executes the callback code and continues with a behavior equivalent to the default behavior described in USB Host Transactor Default Behavior.

Note *This feature can be disabled by registering a NULL pointer.*

8.3 USB Host Transactor Default Behavior

When a timeout occurs, the transactor displays a message indicating that a watchdog has been activated, re-arms the watchdog, and resumes.

However, if a timeout occurs during `SendControlMessage()` and `SendBulkMessage()` calls, the transactor displays a message indicating that the watchdog has been activated. In this case, it returns the control to the user by issuing a `ZUSB_TRANSFER_TIMEOUT` status.

8.4 Logging USB Transfers Processed by the Transactor

8.4.1 USB Requests Processing Logs

The USB Host transactor allows logging USB transfers activity in a file or on standard output.

- Log options are set with the `SetDebugLevel()` method.
- `SetLog()` enables printing of logs into a file.
- `SetLogPrefix()` sets a prefix which is added onto each line of the log.

Each log type is chosen independently.

8.4.2 Log Types

You can define log type using the `logMask_t` enum defined in the `UsbUrbCommon.hh` file.

8.4.2.1 Main Info Log

This log reports the main stages of device detection and initialization.

Use the `logInfo` option to activate it.

8.4.2.2 USB Request Log

This log reports the URB activity.

Use the `logUrb` option to activate it.

8.4.2.3 Controller Core Log

This log reports the host controller core activity (channel and endpoint management).

Use the `logCore` option to activate it.

8.4.2.4 Data Buffer Log

This log reports the data transferred between the USB host and the USB device.

Use the `logBuf` option to activate it.

8.4.2.5 Reference Clock Time Log

This log reports the reference clock counter, which is set by the user.

Use the `logTime` option to activate it.

8.4.3 USB Requests Log Example

The following example shows the sequence of a Host sending a control command to set the address of a device. All logs are set except `logBuf` (data buffer).

```
Host CTRL   Info    : Starting controller core initialization
Host CTRL   Info    : Core initialization done
Host CTRL   Info    : Starting HCD initialization
Host CORE   Info    : ZEBU USB Controller                                  logInfo
Host CORE   Info    : new USB bus registered, assigned bus number 1
Host CORE   Info    : configuration #1 chosen from 1 choice
Host CTRL   Info    : HCD initialization done
Host CORE   Info    : USB hub found
Host CORE   Info    : 1 port detected

...

Host CTRL   DEBUG   : Zebu URB unqueue
Host CTRL   DEBUG   :   Device address: 0
Host CTRL   DEBUG   :   Endpoint: 0, OUT
Host CTRL   DEBUG   :   Endpoint type: CONTROL                             logUrb
Host CTRL   DEBUG   :   Speed: HIGH
Host CTRL   DEBUG   :   Max packet size: 64
Host CTRL   DEBUG   :   Data buffer length: 0
Host CTRL   DEBUG   :   Interval: 0
Host CTRL   DEBUG   :   Setup pkt: ReqType: 0x00
Host CTRL   DEBUG   :                  Req: SET_ADDRESS (0x05)
Host CTRL   DEBUG   :                Value: 0x0009
Host CTRL   DEBUG   :                Index: 0x0000
Host CTRL   DEBUG   :               Length: 0x0000

Host CTRL   DEBUG   : Init Channel 3
Host CTRL   DEBUG   :   Device Addr: 0
Host CTRL   DEBUG   :   EP Num: 0
Host CTRL   DEBUG   :   DIR: OUT                                           logCore
Host CTRL   DEBUG   :   Max packet: 64
Host CTRL   DEBUG   : Channel 3 Start Xfer : NumBytes=8
                      DataPID=MDATA NumPackets:1
Host CTRL   DEBUG   : Transfering 8 bytes
Host CTRL   DEBUG   : Channel 3 Pending transfer
Host CTRL   DEBUG   : Control setup transaction done
Host CTRL   DEBUG   : Init Channel 4
Host CTRL   DEBUG   :   Device Addr: 0
Host CTRL   DEBUG   :   EP Num: 0
Host CTRL   DEBUG   :   DIR: IN
Host CTRL   DEBUG   :   Max packet: 64
Host CTRL   DEBUG   : Channel 4 Start Xfer : NumBytes=64 DataPID=D1 NumPackets:1

Host TIME   DEBUG   : clk_48m cycle - 5256487                              logTime

Host CTRL   DEBUG   : Channel 4 Rcv Data : NumBytes=0 DataPID=D2
Host CTRL   DEBUG   : Control transfer complete
Host CTRL   DEBUG   : Zebu URB complete urb, device 0, ep 0 OUT, status=0
```

8.5 Using the USB Host Transactor

8.5.1 Initializing the USB Host Transactor

The process of initializing the USB Host transactor is divided into the following steps:

1. *Initializing the ZeBu Board and USB Host Transactor*
2. *Initializing the Transactor Host Controller*
3. *Connecting the USB Host Transactor to the Cable*
4. *USB Device Discovery*

8.5.1.1 Initializing the ZeBu Board and USB Host Transactor

```
// Opening the ZeBu board
Board* board = Board::open(“zebu.work”, "designFeatures", "usb_driver");
  if (board == NULL) {
    cerr << "Could not open Zebu." << endl;
    return 1;
  }

// Initializing the USB transactor
  usbHst.Init(board, "usb_driver_host_inst", "clk_48m");

// Initializing the ZeBu board
  board->init(NULL);

// Registering a service loop callback
  usbHst.RegisterCallBack(&mycallback, (void *)("usb_callback"));
```

8.5.1.2 Initializing the Transactor Host Controller

`InitBFM()`starts the Host controller initialization sequence:

```
Zebu Host Controller initialization.
// Initialize the USB transactor BFM
// Setting High speed devices support mode
  if(usbHst.InitBFM(true) != ZUSB_STATUS_SUCCESS) { /* High speed support,
utmi 8*/
    cerr << "Initialization failed." << endl;
    return 1;
  }
```

8.5.1.3 Connecting the USB Host Transactor to the Cable

The `USBPlug()` method connects the USB Host transactor to the USB cable as shown below:

1. ZeBu Root Hub initialization. Assigned to USB bus number 1.
2. Hub USB Port power ON.

```
 // Connects the Host to the cable
  if(usbHst.USBPlug() != ZUSB_STATUS_SUCCESS) {
    cerr << "Plug failed." << endl;
    return 1; }
```

8.5.1.4 USB Device Discovery

The ZeBu USB Host transactor discovers the USB device using the same sequence as the Linux kernel probe function. The `DiscoverDevice()` method (see *`DiscoverDevice()` Method*) executes the following steps:

1. Calls the Hub Port Status.
2. Calls the Hub Clear Port feature with `C_PORT_CONNECTION`.
3. Executes a debounce checking by calling the Hub Port Status 4 times.
4. Executes the Hub Port feature `PORT_RESET`.
5. Detects Port Enable transition and then clears the port feature with `C_PORT_RESET`.
6. Sends a `GET_DESCRIPTOR` control request to the USB device.
7. Executes Hub Port `PORT_RESET`.

8. Detects Port Enable transition and then clears the port feature with C_PORT_RESET.
9. Sends a SET_ADDRESS control request to the device. Default address is 2 but it can be set to another value using SetDeviceAddress() (see *SetDeviceAddress() Method*).
10. Sends multiple GET_DESCRIPTOR control requests to discover the USB device (number of calls depends on the device).
11. Sends a SET_CONFIGURATION control request to select the configuration. Default is the first one available. Can be set to another value with SetDeviceConfig() (see *SetDeviceConfig() Method*).

The device is now ready to be accessed using ZeBu URBs:

```
printf("Checking device connection\n");
zusb_status st = ZUSB_STATUS_NO_DEVICE;
do {
  st = usbHst.DiscoverDevice();
}while(st == ZUSB_STATUS_NO_DEVICE);

// Device up and running
// Check its speed
if(usbHst.GetDeviceSpeed() == ZUSB_SPEED_HIGH) {
  printf("High speed device detected\n");
}
else {
  printf("Full speed device detected\n");
}
// user code
```

8.5.2 Managing USB Device Configuration and Interfaces

8.5.2.1 Managing the Device Address

The following methods enable you to manage the device address:

- `SetDeviceAddress(uint8_t address)` allows setting an address to the device. If not called, the first available address is used.
  This function must be called before the call to `DeviceDiscovery()`.
- `GetDeviceAddress()` returns the current device address.

8.5.2.2 Managing the Device Configuration

The following methods enable you to manage the device configuration:

- `SetDeviceConfig(uint32_t config)` allows to choose the device configuration number. If not called, the first available configuration is used. This function must be called before the call to `DeviceDiscovery()`.
- `GetDeviceConfig()` returns the current configuration number.
- `GetRawConfiguration()` returns the full device configuration. The representation follows the USB 2.0 specification. You are responsible for parsing this configuration to get device information (see *Managing the Device Configuration* for further details on device configuration information parsing).

8.5.2.3 Managing the Device Interface

The following methods enable you to manage the device interface:

- `SetDeviceInterface(uint32_t intf, uint32_t altsetting)` allows to choose an alternate setting for an interface. By default, alternate setting 0 is used. This function must be called after the call to `DeviceDiscovery()`. The endpoints associated with the previous alternate setting interface are disabled while the new endpoints are enabled.
- `GetDeviceInterface(uint32_t intf)` returns the current alternate setting used for an interface.

8.5.2.4 Example of Configuration Parsing

The following example illustrates a simple configuration parsing using the helper USB structures provided with the USB Host transactor. This parse assumes that there is only one configuration, one interface, and 2 `Bulk` endpoints available.

```
// Getting the used configuration
printf("Reading chosen configuration : %d\n", usbHst.GetDeviceConfig());
```

```
// Reading the USB device descriptors
printf("Parsing configuration : \n");
const uint8_t* confBuf =
    usbHst.GetRawConfiguration(usbHst.GetDeviceConfig());

uint8_t* ptBuf = confBuf;

zusb_DescriptorHeader headerDesc;
zusb_ConfigDescriptor configDesc;
zusb_InterfaceDescriptor itfDesc;
zusb_EndpointDescriptor* epBulkOut = 0;
zusb_EndpointDescriptor* epBulkIn = 0;
zusb_EndpointDescriptor epDesc;

  memcpy(&headerDesc, ptBuf, 2 /* header size */);
  do {
    if(headerDesc.bDescriptorType == ZUSB_DT_ENDPOINT) {
      memcpy(&epDesc, ptBuf, ZUSB_DT_ENDPOINT_SIZE);
      if((epDesc.bEndpointAddress & ZUSB_ENDPOINT_DIR_MASK) ==
ZUSB_DIR_IN) {
        if((epDesc.bmAttributes & ZUSB_ENDPOINT_XFERTYPE_MASK) ==
ZUSB_ENDPOINT_XFER_BULK) {
            epBulkIn = new zusb_EndpointDescriptor;
            memcpy(epBulkIn, &epDesc, ZUSB_DT_ENDPOINT_SIZE);
            printf("Found BULK IN Endpoint at address %d\n",
(epDesc.bEndpointAddress) & ZUSB_ENDPOINT_NUMBER_MASK);
        }
      }
      else {
        if((epDesc.bmAttributes & ZUSB_ENDPOINT_XFERTYPE_MASK) ==
ZUSB_ENDPOINT_XFER_BULK) {
            epBulkOut = new zusb_EndpointDescriptor;
            memcpy(epBulkOut, &epDesc, ZUSB_DT_ENDPOINT_SIZE);
```

```
            printf("Found BULK OUT Endpoint at address %d\n",
(epDesc.bEndpointAddress) & ZUSB_ENDPOINT_NUMBER_MASK);
        }
      }
    }
    else if(headerDesc.bDescriptorType == ZUSB_DT_CONFIG) {
      memcpy(&configDesc, ptBuf, ZUSB_DT_CONFIG_SIZE);
      if(configDesc.bConfigurationValue != usbHst.GetDeviceConfig()) {
        printf("Error, bad configuration number\n");
        exit(1);
      }
      printf("Number of interfaces = %d\n", configDesc.bNumInterfaces);
    }
    else if(headerDesc.bDescriptorType == ZUSB_DT_INTERFACE) {
      memcpy(&itfDesc, ptBuf, ZUSB_DT_INTERFACE_SIZE);
      printf("Reading interface number %d alt setting %d :\n",
itfDesc.bInterfaceNumber, itfDesc.bAlternateSetting);
      printf("Number of endpoints = %d\n", itfDesc.bNumEndpoints);
    }
    ptBuf = ptBuf + headerDesc.bLength;
    memcpy(&headerDesc, ptBuf, 2 /* header size */);
  }while(headerDesc.bLength != 0);
```

8.5.2.5 Disconnecting the USB Host Transactor from the Cable

The following is the sequence to disconnect the USB Host transactor.

```
  // Connects the Host to the cable
  if(usbHst.USBUnplug() != ZUSB_STATUS_SUCCESS) {
    cerr << "Unplug failed." << endl;
    return 1;
  }
```

The method returns ZUSB_STATUS_SUCCESS in case of success. The device is then

properly disconnected and internal transactor structures are updated.

8 USB Bus Monitoring Feature

Note *This feature is only available for the 64-bit version of ZeBu Server*

The USB Device transactor cable models include a USB Bus logging mechanism. This mechanism allows logging of USB traffic, i.e. USB Packets going in and out of the DUT.

Both USB Channel and URB APIs provide three methods dedicated to USB bus monitoring control as described in the following sections:

- *Prerequisites*
- *Using the USB Monitoring Feature with USB Channel API*
- *Using the USB Monitoring Feature with URB API*

8.1 Prerequisites

8.1.1 USB Bus Monitor Hardware Instantiation

To use the USB Bus logging feature, the USB Bus log must be correctly instantiated in the hardware side.

8.1.2 USB Device Transactor Initialization Constraint

When the USB Bus log is used, the `clkName` parameter of the transactor initialization `init()` method must specify the transactor clock name. This enables the transactor to correctly report times in the log files.

Otherwise, the time reported in the log file is not relevant.

8.2 Using the USB Monitoring Feature with USB Channel API

Use the following USB Channel API methods to manage USB monitoring:

TABLE 39 Methods for Bus Monitoring using Channel API

Method	Description
`setBusMonFileName`	Sets the log file name.
`startBusMonitor`	Starts the USB bus log.
`stopBusMonitor`	Stops the USB bus log.

8.2.1 Starting and Stopping Log

To start monitoring a USB link at a given point, use the `startBusMonitor()` method.

To stop monitoring, use the `stopBusMonitor()` method.

8.2.2 Defining the Monitor File Name

The USB Bus Logging API allows defining a filename for the log file generated by the transactor.

The `setBusMonFileName()` method sets a file name and controls filename indexing. If the `setBusMonFileName()` method is not used, the transactor uses the transactor instance name (defined in the Top Verilog file) as file name each time the `startBusMonitor` method is called. See *Description of the USB Channel API Interface for USB Bus Logging* for detailed information on the USB Channel API.

The log file generated is a text file (.bmn)and a FSDB File (.bnm.fsdb) using a proprietary file format, which can be read with Synopsys Protocol Analyzer tool.

8.2.3 Description of the USB Channel API Interface for USB Bus Logging

8.2.3.1 `setBusMonFileName( )` Method

Sets the log file name.

```
const char* name, bool _CreateFSDB = true , const char* ProgName= "NULL",
bool autoInc = false);
```

Parameter Name	Parameter Type	Description
`name`	`const char*`	File name without extension. The `.bmn` extension is appended automatically.
`CreateFSDB`	`bool`	Enable or disable the FSDB monitor for Protocol Analyzer
`ProName`	`const char`	Must be the name of testbench executable (mandatory if the internal detection fail - depend of the linux distribution)
`autoInc`	`bool`	Specifies if an index should follow the file name to avoid deletion of the previous file: • `true`: the index of the filename is incremented each time the `StartBusMonitor()` method is called. The transactor checks the current index for this file name and creates a new file with an incremented index (`myFileName.0.bmn`, `myFileName.1.bmn`, etc.). • `false` (default): the file is overwritten.

This method returns `0` on success, `-1` otherwise.

Note *Use the ProName argument if the PATH variable is not pointing to the run directory.*

8.2.3.2 startBusMonitor() Method

Opens a log file and starts activity logging.

```
int32_t startBusMonitor (busMonType type);
```

where type activates the monitor:

- MonitorData: logs data packet payload.
- MonitorNoData: logs only data packet header.

This method returns 0 on success, -1 otherwise.

Note *Call this method after the init() method and config() methods.*

8.2.3.3 stopBusMonitor() Method

Stops bus activity logging and closes the current binary file.

```
int32_t stopBusMonitor (void);
```

This method returns 0 on success, -1 otherwise.

8.3 Using the USB Monitoring Feature with URB API

The following URB API methods can be used to manage USB monitoring:

TABLE 40 Bus Monitoring methods using URB API

Method	Description
`SetBusMonFileName`	Sets the log file name.
`StartBusMonitor`	Starts the USB bus log.
`StopBusMonitor`	Stops the USB bus log.

8.3.1 Starting and Stopping Logging

To start logging at a given point, use the `StartBusMonitor()` method.

To stop logging, use the `StopBusMonitor()` method.

8.3.2 Defining the Log File Name

The URB API for bus logging allows defining a filename for the log file generated by the transactor.

The `SetBusMonFileName()` method sets a file name and controls the filename indexing. If the `SetBusMonFileName()` method is not used, the transactor uses the transactor instance name (defined in the hw_top file) as file name each time the `StartBusMonitor` method is called. See *Description of the URB API Interface for USB Bus Logging* for detailed information on the URB API for USB Bus logging.

The log file generated is a binary file (`.bmn`). You can read this file using the ZeBu USB Viewer. For more information, see ***ZeBu USB Viewer User Manual***.

8.3.3 Description of the URB API Interface for USB Bus Logging

8.3.3.1 SetBusMonFileName() Method

Sets the log file name.

```
int32_t SetBusMonFileName (const char* name, bool autoInc = false);
```

Parameter Name	Parameter Type	Description
name	const char*	File name without extension.
autoInc	bool	Specifies if an index should follow the file name to avoid deletion of the previous file: • true: the index of the filename is incremented each time the StartBusMonitor() method is called. The transactor checks the current index for this file name and creates a new file with an incremented index (myFileName.0.umn, myFileName.1.umn, etc.). • false (default): the file is overwritten.

This method returns 0 upon success, -1 otherwise.

8.3.3.2 StartBusMonitor() Method

Opens a log file and starts activity logging.

```
int32_t StartBusMonitor (zusb_mon_type type);
```

where type activates the log:

- MonitorData: logs data packet payload.
- MonitorNoData: logs only data packet header.

This method returns 0 on success, -1 otherwise.

8.3.3.3 `StopBusMonitor()` Method

Stops bus activity logging and closes the current binary file.

```
int32_t StopBusMonitor (void);
```

This method returns 0 on success, -1 otherwise.

10 URB Logging Feature

10.1 Description

The USB Host transactor includes a URB logging mechanism, which allows dumping of USB bus activity at a URB level of abstraction.

The URB API contains three API calls dedicated to URB logging:

TABLE 33 ZeBu URB Logging API Methods

Method	Description
`SetUrbMonFileName`	Sets the log file name.
`StartUrbMonitor`	Starts the URB log.
`StopUrbMonitor`	Stops the URB log

10.2 Prerequisite

When the USB Bus log is used, the `clkName` parameter of the transactor initialization `init()` method must include the transactor clock name for the transactor to correctly report times in the log files.

Otherwise, the time reported in the log files is not relevant.

10.3 Using the URB Logging Feature

10.3.1 Starting and Stopping Logging

To start logging at a given point, use the `StartUrbMonitor()` method.

To stop logging, use the `StopUrbMonitor()` method.

10.3.2 Defining the Log File Name

The URB Logging API allows defining a filename for the log file generated by the transactor.

The `SetUrbMonFileName()` method sets a file name and controls filename indexing. If the `SetUrbMonFileName()` method is not used, the transactor uses the transactor instance name (defined in the hw_top file) as file name each time the `StartUrbMonitor` method is called. See *Description of the URB API Interface for URB Logging Feature* for detailed information on the URB API for URB logging.

The log file generated is a binary file (`.bmn`). You can read this file using the ZeBu USB Viewer. For more information, see ***ZeBu USB Viewer User Manual***.

10.3.3 Description of the URB API Interface for URB Logging Feature

10.3.3.1 SetUrbMonFileName() Method

Sets the log file name.

```
int32_t SetUrbMonFileName (const char *name, bool autoInc = false);
```

Parameter Name	Parameter Type	Description
name	const char*	File name without extension. The .unm extension is appended automatically to the log filename.
autoInc	bool	Specifies if the file name should be followed by an index to avoid deletion of the previous file: • true: the index of the filename is incremented each time the StartBusMonitor() method is called. The transactor checks the current index for this file name and creates a new file with an incremented index (myFileName.0.umn, myFileName.1.umn, etc.) • false (default): the file is overwritten.

This method returns 0 on success, -1 otherwise.

10.3.3.2 StartUrbMonitor() Method

Opens a new log file and starts URB logging.

```
int32_t StartUrbMonitor ();
```

This method returns 0 upon success, -1 otherwise.

10.3.3.3 StopUrbMonitor() Method

Stops URB logging and closes the current log file.

```
int32_t StopUrbMonitor (void);
```

This method returns 0 upon success, -1 otherwise.

11 Tutorial

The example directory of the transactor package contains the following two examples:

- *phy_utmi*
- *phy_cable*

11.1 phy_utmi

This example shows how the transactor is used for an integration with the USB DUT device using the UTMI interface.

11.1.1 File Tree

The following files are provided for this example:

```
`-- example
    `-- phy_utmi
        |-- README
        |-- src
        |   |-- bench
        |   |   |-- tb_dev.cc
        |   |   |-- tb_host.cc
        |   |   |-- tb_urb_dev.cc
        |   |   |-- tb_urb_host.cc
        |   |   |-- usb_dev_access.cc
        |   |   |-- usb_dev_access.hh
        |   |   |-- usb_dev_descriptors.hh
        |   |   |-- usb_host_desc.cc
        |   |   |-- usb_host_desc.hh
        |   |   |-- usbstring.cc
        |   |   `-- usbstring.hh
        |   |-- dut
        |   |   `-- usb_cable_top.v
        |   |-- env
        |   |   |-- designFeatures_utmi
        |   |   |-- usb_driver.dve
        |   |   `-- usb_driver.zpf
        |   |-- gate
        |   |   `-- dut.edf
        |   `-- input
```

```
|       `-- test.i
`-- zebu
    `-- Makefile
```

11.1.2 Overview

The phy_utmi example consists of two testbench processes representing USB Host and a USB Device connected by a cable, as shown in the following example:

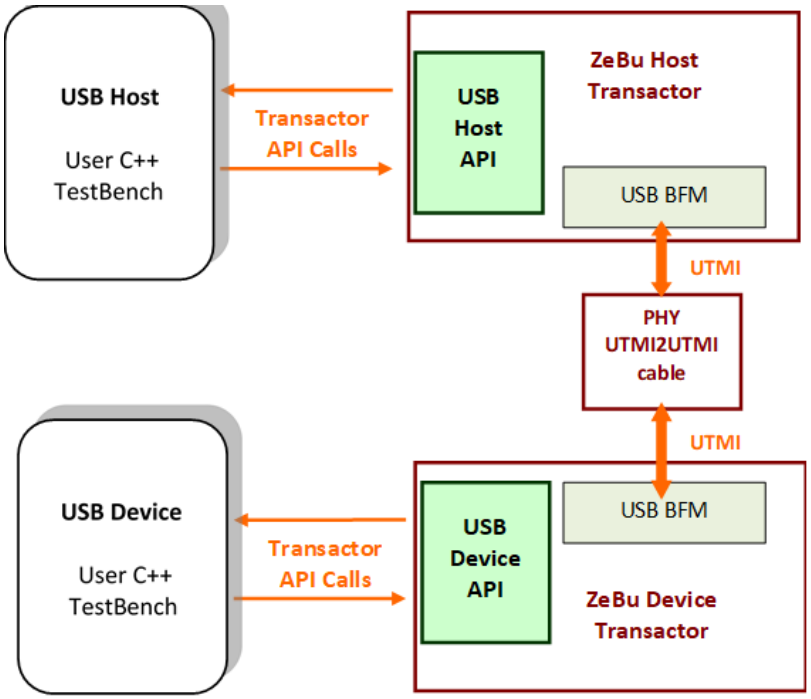

FIGURE 17. phy_utmi example overview

11.1.3 ZeBu Design

The ZeBu design consists of a:

- USB Host transactor connected to a cable side via a UTMI interface.
- USB Device transactor connected to the other side via a UTMI interface.

11.1.4 Software Testbench

You can run this example using either the Channel API or the URB API. The initialization sequence (Setting device address, and so on) is quite different between Channel and URB API testbenches but the demo sequence is the same:

1. The USB Host starts reading an input file (test.in) and sends data to the device using Bulk OUT transfers.
2. When all the data has been transferred, the Host starts reading back the data from the Device using Bulk IN transfers, and puts the data in an output file (test.out).
3. The files are compared to check that all worked fine.

11.2 phy_cable

This example explains how the transactor is used for an integration with the USB DUT device using the serial interface.

11.2.1 File Tree

The following files are provided for this example:

```
`-- example
    `-- phy_cable
        |-- README
        |-- src
        |   |-- bench
        |   |   `-- tb.cc
        |   |-- dut
        |   |   |-- tristate_resolv.v
        |   |   `-- usb_cable_top.v
        |   |-- env
        |   |   |-- designFeatures
        |   |   |-- usb_driver.dve
        |   |   `-- usb_driver.zpf
        |   |-- gate
        |   |   `-- dut.edf
        |   `-- input
        |       `-- test.in
        `-- zebu
            `-- Makefile
```

11.2.2 Overview

This example consists of a USB Host and a USB Device connected by a cable and running as two threads inside a single testbench process, as shown in the following example.

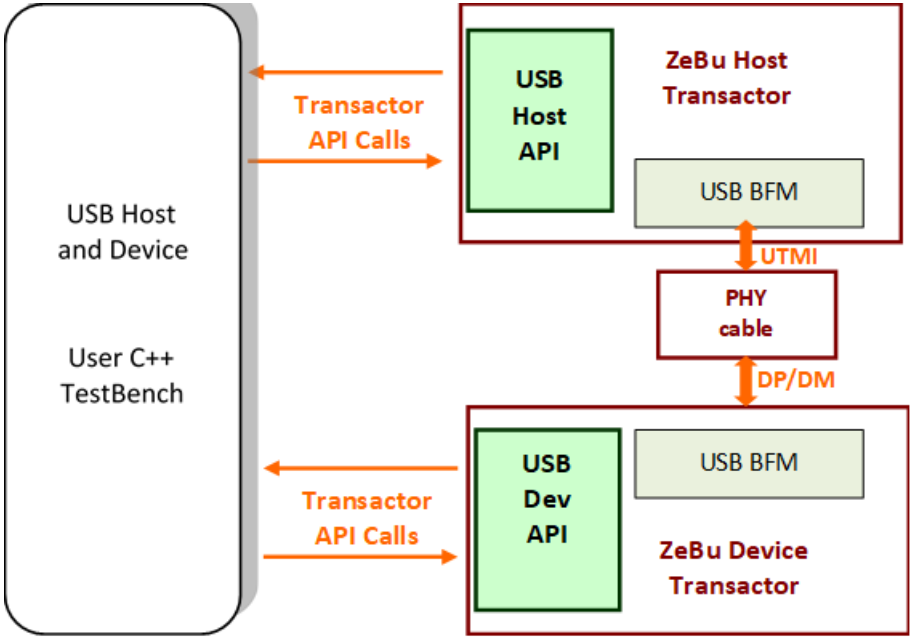

FIGURE 18. phy_cable example overview

11.2.3 ZeBu Design

The ZeBu design consists of a:

- USB Host transactor connected to a cable side through a UTMI interface.
- USB Device transactor connected to the other side via a serial interface.

11.2.4 Software Testbench

This example uses the Channel API. The initialization sequence is as follows:

1. The USB Host sets the device address, reads the device descriptor and then starts reading an input file (test.in). It then sends the data to the device using Bulk OUT transfers.

2. When all the data has been transferred, the Host starts reading back the data from the Device using Bulk IN transfers and puts the data in an output file (test.out).
3. The files are compared to check that all worked fine.

11.3 Running the Examples

11.3.1 Setting the ZeBu Environment

1. Set $ZEBU_ROOT to the ZeBu release home directory.
2. Set the following environment variables:

```
export FILE_CONF=<zebu_configuration_used>
export REMOTECMD=<remote_command_to_be_used>
export ZEBU_IP_ROOT=<transactor_install_dir>
export ARCH="<32/64>"
```

where ARCH is the 32-bit (32) or 64-bit (64; default value) Linux OS.

11.3.2 Compiling the Designs

1. Go to the example/phy_<utmi/cable>/zebu directory.
2. Type the following:

```
@> make compil
```

11.3.3 Running the phy_utmi Example

1. Go to the example/phy_utmi/zebu directory.
2. Type the following:

```
@> make run:    Run the Channel API example
@> make run_urb:    Run the URB API example
```

11.3.4 Running phy_cable Example

1. Go to the example/phy_cable/zebu directory.
2. Type the following:

```
@> make run
```