Verification Continuum™

ZeBu® USB 2.0 OTG

User Guide

Version Q-2020.06, August 2020

Contents

List of Tables

List of Figures

About This Manual

Overview

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

Related Documentation

For details about the ZeBu supported features and limitations, you should refer to the ***ZeBu Release Notes*** in the ZeBu documentation package which corresponds to the software version you are using.

You can find relevant information for usage of the present transactor in the training material about ***Using Transactors***.

Please consult the ***ZeBu USB Device Transactor User Manual*** for further details on the USB Device Transactor functioning.

1 Introduction

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

This section consists of the following sub-sections:

- *Overview*
- *USB Interfaces Compliance*
- *API Layers*
- *Features*
- *Performance*
- *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 OTG Transactor application:

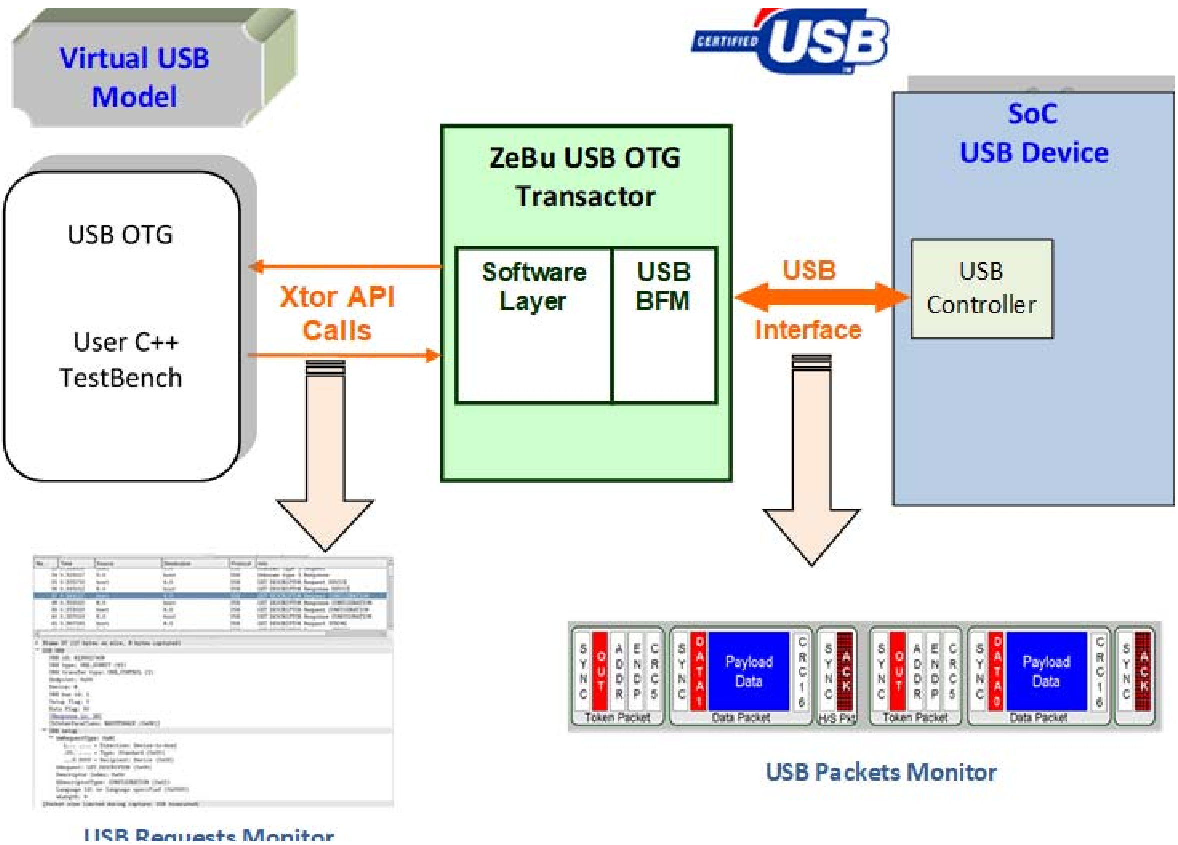

FIGURE 1. ZeBu OTG Transactor Application

1.2 USB Interfaces Compliance

The ZeBu USB OTG 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).

For the transactor to work with the UTMI, ULPI, 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

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 OTG transactor offers the following two levels of C++ Application Program Interface (API):

- USB Channel level for USB packet transfers 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 OTG 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.
- 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.
- HNP protocol
- SRP protocol

1.4.2 List of Integration Components

The ZeBu USB OTG 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.

- USB cable checker for UTMI interface.
- USB Bus log for UTMI and ULPI interfaces.

1.4.3 APIs

The USB OTG 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 USB transfer types.
  - Device configuration and interface management.
  - Linux driver behavior for kernel version 2.6.18.
  - URB monitor.

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_USBDeviceXtor.

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 OTG transactor performance in various conditions.

1.6.1 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 1 UTMI Interface

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

1.7 Limitations

This section explains the currently unsupported *USB Features* and *USB Transactor API*.

1.7.1 USB Features

The following USB features are 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 DP/DM is not supported with UBS OTG Transactor.
- Permits transfer of only one packet per microframe. Does not support high-bandwidth transactions (two or three packet per microframe).

1.7.2 USB Transactor API

The following features are not supported:

- **USB Channel API**: Does not support the Isochronous and interrupt transfers.
- **USB Log**: Permits only one URB OTG transactor per Linux process.

2 Installation

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

- *Installing the USB OTG Transactor Package*
- *Package Description*

2.1 Installing the USB OTG 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 OTG 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 OTG transactor is successfully installed, the package contains the following elements:

- `libUsb.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 OTG transactor:
  - Verilog file for the UTMI cable.
  - Verilog file for the ULPI cable.
- Example applications with the USB OTG transactor and the UTMI USB cable model mapped in ZeBu:
  - Channel API.
  - URB API.

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*

3.1 Architecture with the UTMI DUT Interface

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

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

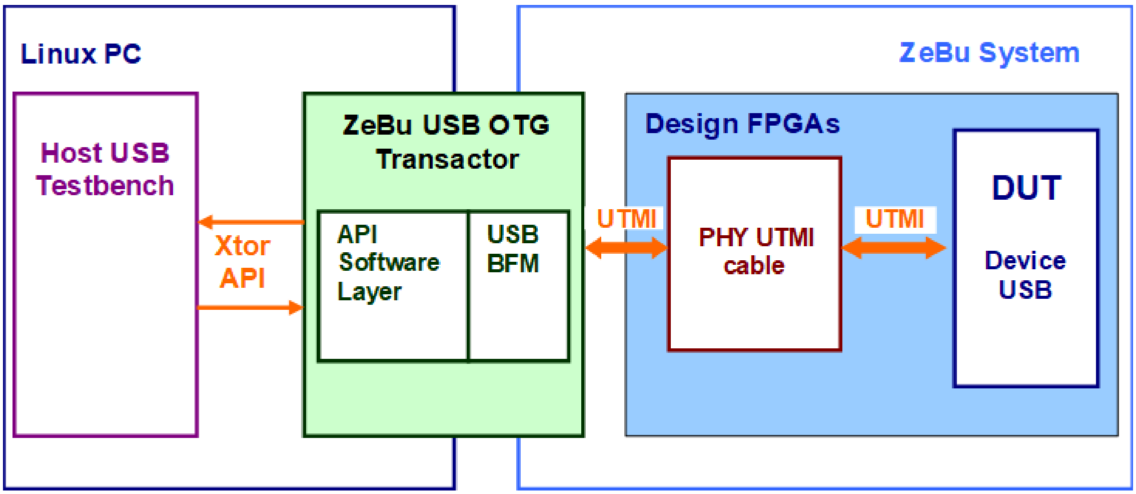

FIGURE 2. USB OTG Transactor With the UTMI DUT Interface

Note *The misc/phy_utmi2utmi directory contains an example of this cable model architecture: phy_utmi2utmi_otg.sem.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.

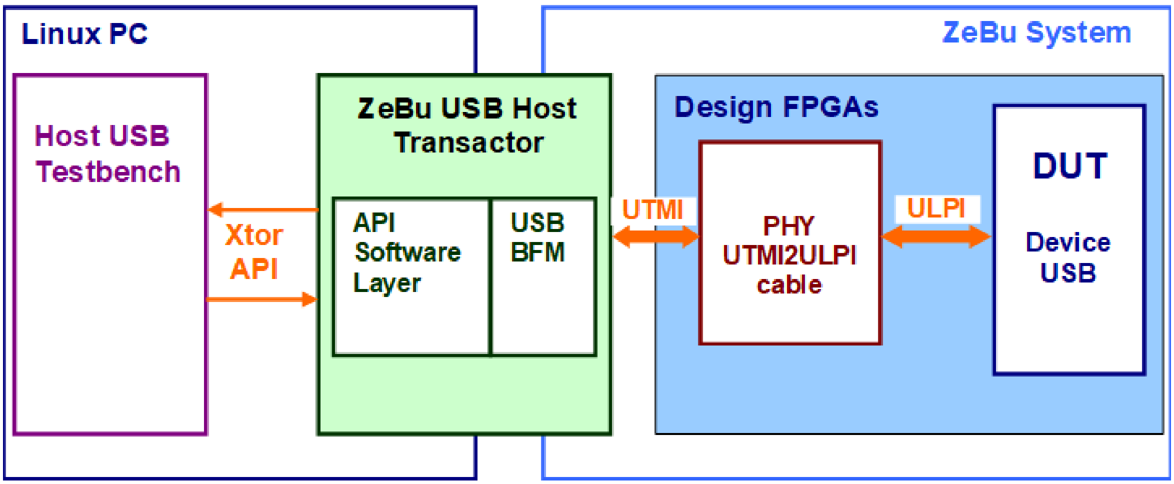

FIGURE 3. USB OTG Transactor with the ULPI DUT Interface

Note *The misc/phy_utmi2ulpi directory contains an example of this cable model architecture: phy_utmi2ulpi_otg.sem.v.*

4 Hardware Interface

This section explains the following topics:

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

4.1 Interface Overview

The USB OTG transactor has the following types of interfaces:

- *UTMI Interface*
- *ULPI Interface*

4.1.1 UTMI Interface

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

TABLE 3 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
utmi_linestate	2	I	O	PHY line state (dp is bit 0, dm is bit 1)

TABLE 3 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
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
utmiotg_iddig	1	O	I	Indicates whether the connected plug is a A-Device or B-Device

TABLE 3 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
utmiotg_avalid	1	O	I	The AValid signal is used to indicate if the session for an A-peripheral is valid
utmiotg_bvalid	1	O	I	The BValid signal is used to indicate if the session for a B-peripheral is valid
utmiotg_vbusvalid	1	O	I	The VbusValid signal is used to determine whether or not the voltage on Vbus is at a valid level for operation
utmisrp_sessend	1	O	I	The SessEnd signal is used to determine if the voltage on Vbus is below its B-Device Session End threshold.
utmiotg_idpullup	1	O	I	Signal that enables the sampling of the analog Id line.
utmiotg_dppulldown	1	O	I	This signal enables the 15k Ohm pull-down resistor on the DP line.
utmiotg_dmpulldown	1	O	I	This signal enables the 15k Ohm pull-down resistor on the DM line.
utmiotg_drvvbus	1	O	I	The DrvVbus is an enable signal to drive 5V on Vbus
utmisrp_chrgvbus	1	O	I	The signal enables charging Vbus.
utmisrp_dischrgvbus	1	O	I	The signal enables discharging Vbus.
device_con	1	O	I	Simulate device connection/ disconnection

TABLE 3 UTMI Hardware Interface

Signal	Size	Type (XTOR)	Type (Cable DUT)	Description
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
xtor_type	1	O	I	Transactor Type

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

TABLE 4 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

4.1.2 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 OTG transactor uses 3 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.

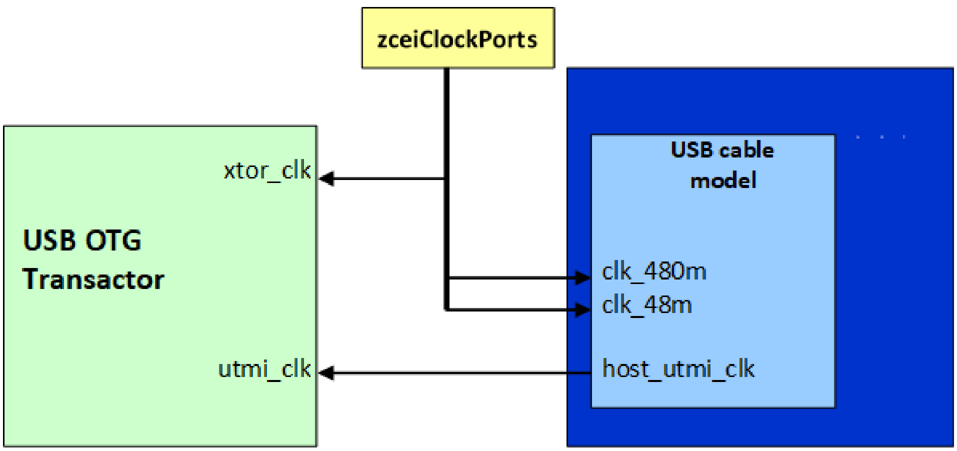

FIGURE 4. Connecting the USB Clocks

4.2.2 Signal List

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

TABLE 5 USB OTG 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 Top Verilog File Example

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

```
zceiClockPort usb_host_zceiClockPort
    (.cclock  (host_xtor_clk ),
     .cresetn (xtor_resetn   ));
zceiClockPort clk_48m ( // connected to the PHY cable model
   .cclock (clk_48m) );
zceiClockPort clk_480m (
   .cclock (clk_480m) );




usb_driver_Utmi_otg u_usb_driver_Utmi_Host
    (.xtor_clk            (host_xtor_clk            ),
     .xtor_resetn         (xtor_resetn              ),
     .xtor_type           (xtor_type                ),
     .utmi_clk            (host_utmi_clk            ),
```

```
      .prst_n             (cable_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 ),

      .utmiotg_idpullup   (host_utmiotg_idpullup   ), // OTG ID Pullup
      .utmiotg_dppulldown (host_utmiotg_dppulldown ), // OTG Data Plus
Pulldown Enable
      .utmiotg_dmpulldown (host_utmiotg_dmpulldown ), // OTG Data Minus
Pulldown Enable
      .utmiotg_iddig      (host_utmiotg_iddig      ), // OTG ID Digital
      .utmiotg_drvvbus    (host_utmiotg_drvvbus    ), // OTG Drive VBUS
      .utmisrp_chrgvbus   (host_utmisrp_chrgvbus   ), // OTG Charge VBUS
      .utmisrp_dischrgvbus (host_utmisrp_dischrgvbus ), // OTG Discharge
VBUS
      .utmiotg_bvalid     (host_utmiotg_bvalid     ), // OTG B-Session
Valid
      .utmisrp_sessend    (host_utmisrp_sessend    ), // OTG Session End
```

```
      .utmiotg_avalid      (host_utmiotg_avalid      ), // OTG A-Session
Valid
      .utmiotg_vbusvalid   (host_utmiotg_vbusvalid   ), // OTG VBUS Valid


//--- ULPI Connection
      .ulpi_clk            (0         ), // ULPI Clock
      .ulpi_datain         (0         ), // ULPI Data In
      .ulpi_dir            (0         ), // ULPI Data Direction Control
      .ulpi_nxt            (0         ), // ULPI Next Da
      //.ulpi_stp            (device_ulpi_stp         ), // ULPI Stop
      //.ulpi_dataout        (device_ulpi_datain[7:0] ), // ULPI Data Out




      .device_con          (host_device_con          ),
      .mon_config          (host_mon_config          ),
      .bus_I               ({3'b0, device_device_con}),

 `ifndef ZEBU_NO_RTB
      .cable_version       (cable_version[7:0]       ),
      .xtor_cclock0        (host_xtor_clk            ));

`else
   .cable_version       (cable_version[7:0]       ));
   defparam u_usb_driver_Utmi_Host.clkCtrl="usb_host_zceiClockPort";
`endif
```

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 OTG 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 OTG transactor at UTMI level.

The following figure illustrates the USB PHY UTMI2UTMI cable interface:

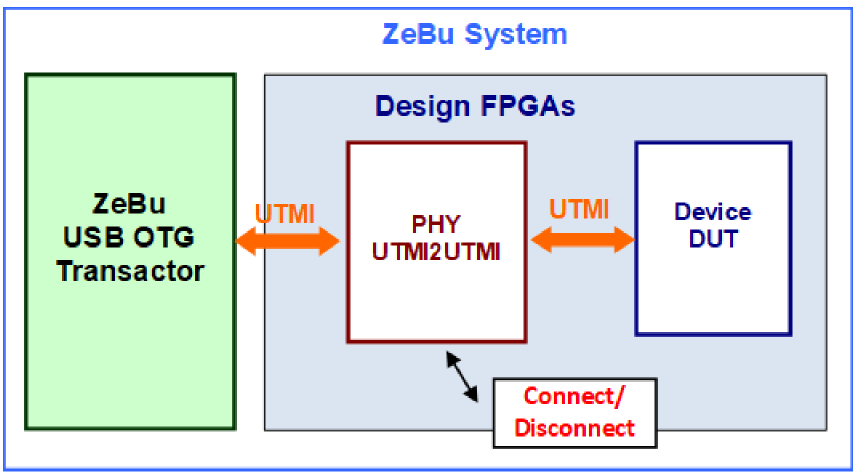

FIGURE 5. USB PHY UTMI2UTMI Cable

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 top Verilog File.
- When Side = D, the signal is driven by the DUT.

TABLE 6 PHY UTMI2UTMI Cable Interface

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 6 PHY UTMI2UTMI Cable Interface

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
utmiotg_iddig	1	O	X/D	Indicates whether the connected plug is a A-Device or B-Device
utmiotg_avalid	1	O	X/D	The AValid signal is used to indicate if the session for an A-peripheral is valid

TABLE 6 PHY UTMI2UTMI Cable Interface

Signal	Size	Type	Side	Description
utmiotg_bvalid	1	O	X/D	The BValid signal is used to indicate if the session for a B-peripheral is valid
utmiotg_vbusvalid	1	O	X/D	The VbusValid signal is used to determine whether or not the voltage on Vbus is at a valid level for operation
utmisrp_sessend	1	O	X/D	The SessEnd signal is used to determine if the voltage on Vbus is below its B-Device Session End threshold.
utmiotg_idpullup	1	O	X/D	Signal that enables the sampling of the analog Id line.
utmiotg_dppulldown	1	O	X/D	This signal enables the 15k Ohm pull-down resistor on the DP line.
utmiotg_dmpulldown	1	O	X/D	This signal enables the 15k Ohm pull-down resistor on the DM line.
utmiotg_drvvbus	1	O	X/D	The DrvVbus is an enable signal to drive 5V on Vbus
utmisrp_chrgvbus	1	O	X/D	The signal enables charging Vbus.
utmisrp_dischrgvbus	1	O	X/D	The signal enables discharging Vbus.
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
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

TABLE 6 PHY UTMI2UTMI Cable Interface

Signal	Size	Type	Side	Description
tx_mon_data	128	O	D	Log internal bus
xtor_type	1	I	X	Transactor Type

4.3.2 PHY ULPI Cable Model

The USB OTG 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:

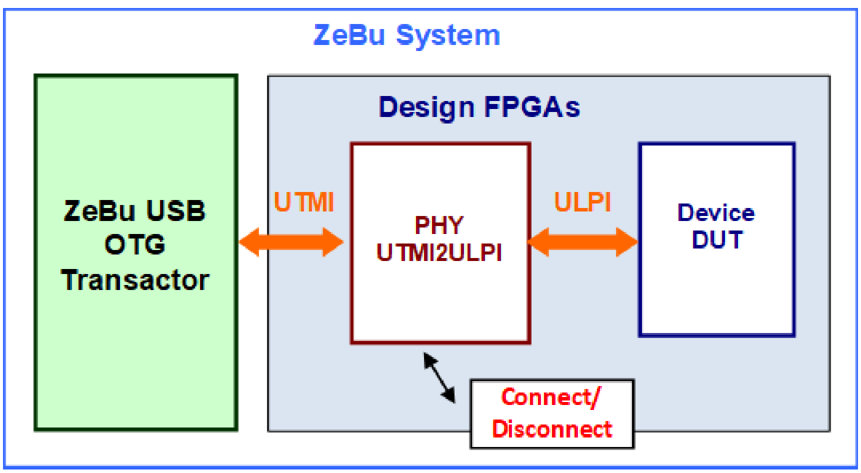

FIGURE 6. PHY UTMI2UTLPI Cable

The following figure illustrates the ULPI interface connection with the:

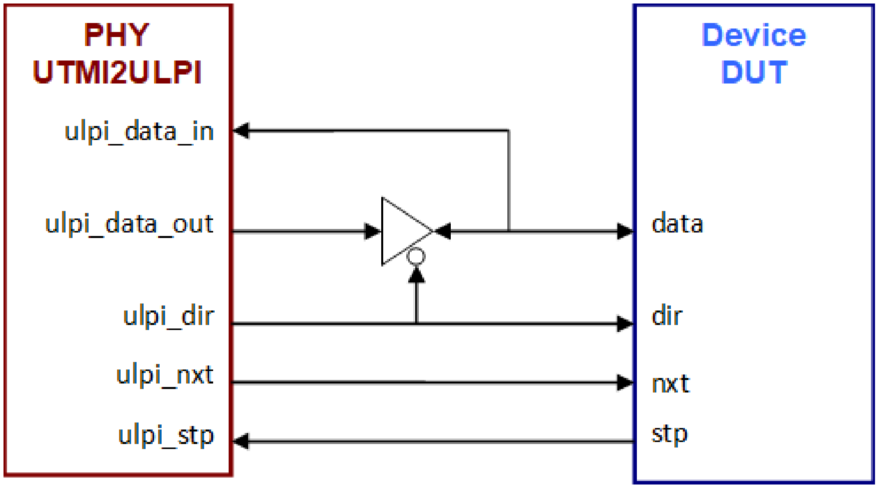

FIGURE 7. PHY UTMI2ULPI Connection to Standard Device DUT

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 7 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 7 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 7 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)
utmiotg_iddig	1	O	X	Indicates whether the connected plug is a A-Device or B-Device
utmiotg_avalid	1	O	X	The AValid signal is used to indicate if the session for an A-peripheral is valid
utmiotg_bvalid	1	O	X	The BValid signal is used to indicate if the session for a B-peripheral is valid
utmiotg_vbusvalid	1	O	X	The VbusValid signal is used to determine whether or not the voltage on Vbus is at a valid level for operation
utmisrp_sessend	1	O	X	The SessEnd signal is used to determine if the voltage on Vbus is below its B-Device Session End threshold.

TABLE 7 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
utmiotg_idpullup	1	I	X	Signal that enables the sampling of the analog Id line.
utmiotg_dppulldown	1	I	X	This signal enables the 15k Ohm pull-down resistor on the DP line.
utmiotg_dmpulldown	1	I	X	This signal enables the 15k Ohm pull-down resistor on the DM line.
utmiotg_drvvbus	1	I	X	The DrvVbus is an enable signal to drive 5V on Vbus
utmisrp_chrgvbus	1	I	X	The signal enables charging Vbus.
utmisrp_dischrgvbus	1	I	X	The signal enables discharging Vbus.
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
rx_mon_data	128	O	D	Log internal bus

TABLE 7 PHY UTMI2ULPI cable interface

Signal	Size	Type	Side	Description
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
xtor_type	1	I	X	Transactor Type

4.3.3 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:

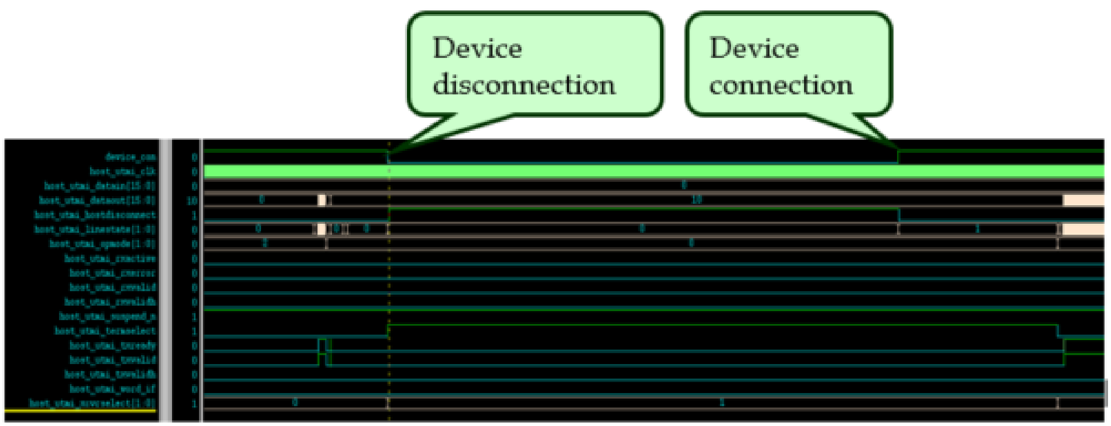

FIGURE 8. Device Connection/Disconnection Waveforms

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

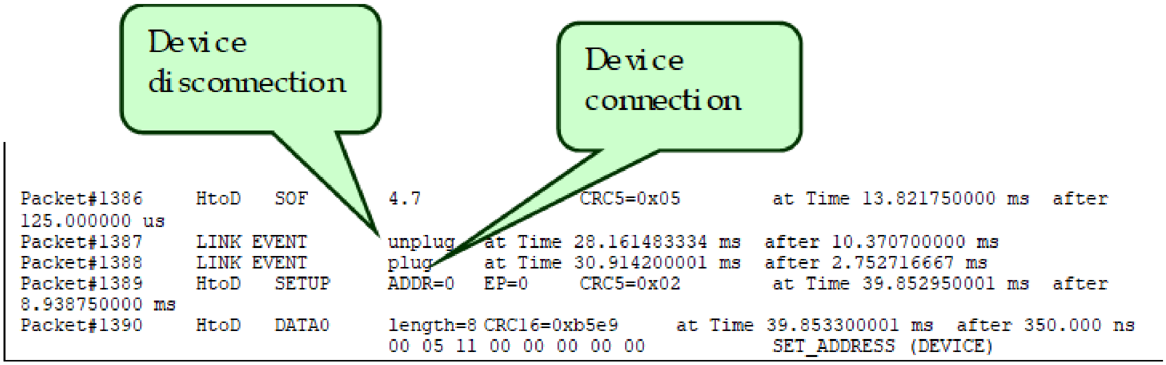

FIGURE 9. Device Connection/Disconnection in the USB Bus Log

4.3.4 PHY UTMI Interface Checker

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

4.3.4.1 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 OTG 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 OTG 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 OTG or USB Device side.

The following figure illustrates the UTMI USB Cable Checker:

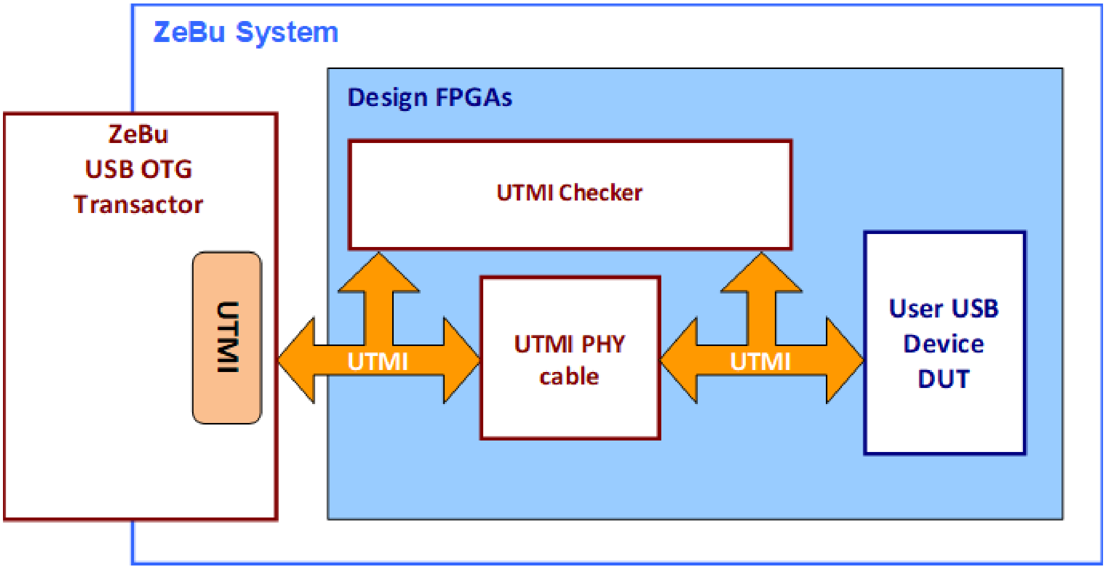

FIGURE 10. UTMI USB Cable Checker

4.3.4.2 Interface

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

TABLE 8 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

TABLE 8 PHY UTMI2UTMI Cable Checker Interface

Signal	Size	Type	Description
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 OTG side. Synchronized on hst_utmi_clk.
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

TABLE 8 PHY UTMI2UTMI Cable Checker Interface

Signal	Size	Type	Description
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.

4.3.4.3 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 top Verilog module:

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 Monitoring

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 Monitoring Feature

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

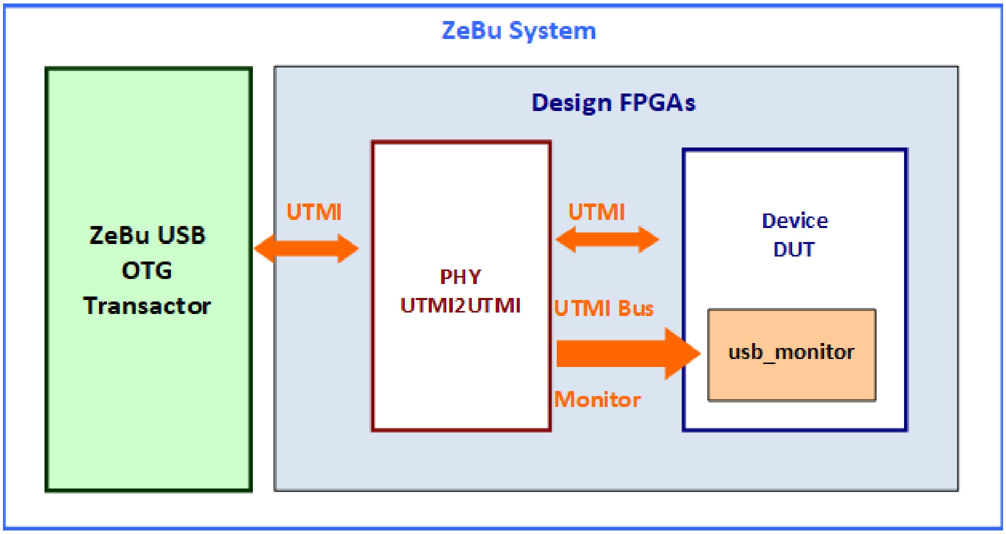

FIGURE 11. Verification Environment for the USB Bus Logging Feature

4.4.1.1 Adding a Monitor Block in the DUT

The usb_monitor is a SystemVerilog module which can be found in the misc directory. It must be instantiated in the DUT, implemented in ZeBu, and connected to the USB PHY cable model using USB monitor tx_mon_* output signals, as shown in the example below:

```
module usb_cable_top (mon_config, …)
input [1:0] mon_config;
input …
output …

  phy_utmi2utmi_HostToDevice U_phy (
    .cable_resetn(cable_resetn),
    .dut_utmi_clk(dut_utmi_clk),
    …
    .mon_config  (mon_config[1:0]),
    .mon_clk     (mon_clk)
```

```
    .rx_mon_trig (rx_mon_trig),
    .rx_mon_sel  (rx_mon_sel[3:0]),
    .rx_mon_data (rx_mon_data[127:0]),
    .tx_mon_trig (tx_mon_trig),
    .tx_mon_sel  (tx_mon_sel[3:0]),
    .tx_mon_data (tx_mon_data[127:0]);

  usb_monitor monitor_host(
    .resetn      (cable_resetn),
    .phy_clk     (mon_clk),
    .rx_mon_trig (rx_mon_trig),
    .rx_mon_sel  (rx_mon_sel[3:0]),
    .rx_mon_data (rx_mon_data[127:0]),
    .tx_mon_trig (tx_mon_trig),
    .tx_mon_sel  (tx_mon_sel[3:0]),
    .tx_mon_data (tx_mon_data[127:0]);
```

4.4.2 Bus Monitoring Control

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

5 Software Interface

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

- *USB Channel API - USB OTG A-Device*
- *USB Endpoint API - USB OTG B-Device*

5.1 USB Channel API - USB OTG A-Device

For USB OTG transactor, an A-Device refers to a USB Host.

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 OTG Software Channel/Endpoint API* for a detailed API description.

The following figure illustrates the USB OTG Transactor Channel API:

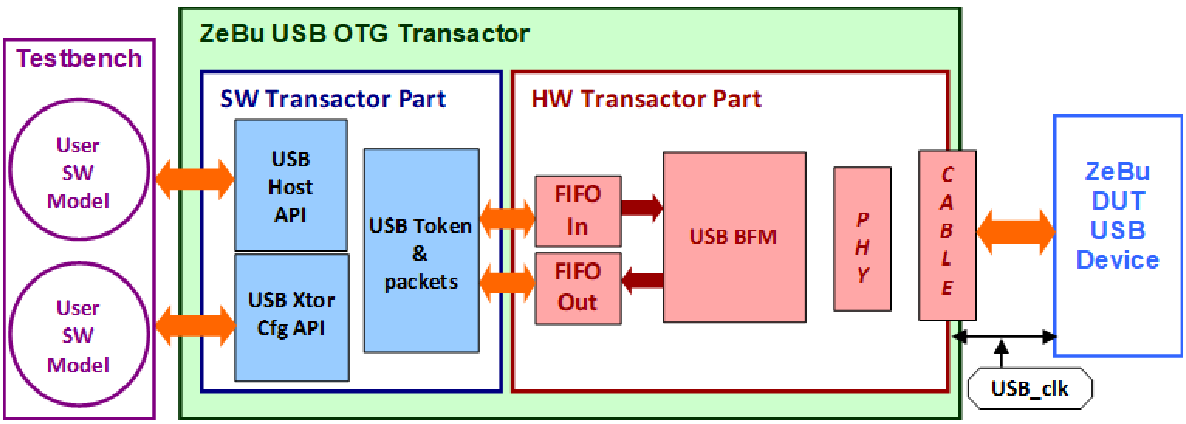

FIGURE 12. USB OTG Transactor Channel API - USB OTG A-Device

5.2 USB Endpoint API - USB OTG B-Device

For USB OTG transactor, a B-Device refers to a USB Device

The testbench directly controls USB endpoint transfers to the USB Host implemented in the DUT. The testbench is responsible for managing the low-level activity transfers , such as, splitting data into single operations, checking endpoint status, and sending data again in case of NAK response.

All those methods are provided in the USB Device Endpoint API of the transactor, described in *USB OTG Software Channel/Endpoint API*.

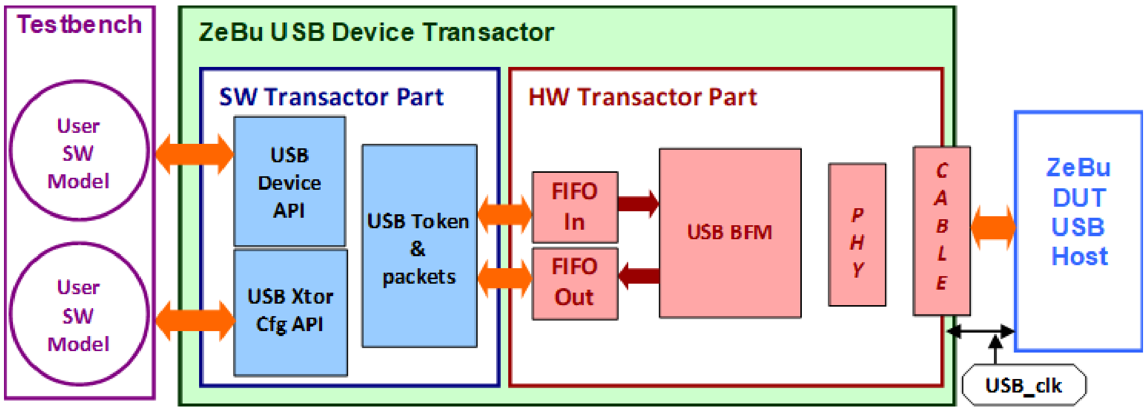

FIGURE 13. USB Device transactor Endpoint API

5.3 USB Request Block (URB) API

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

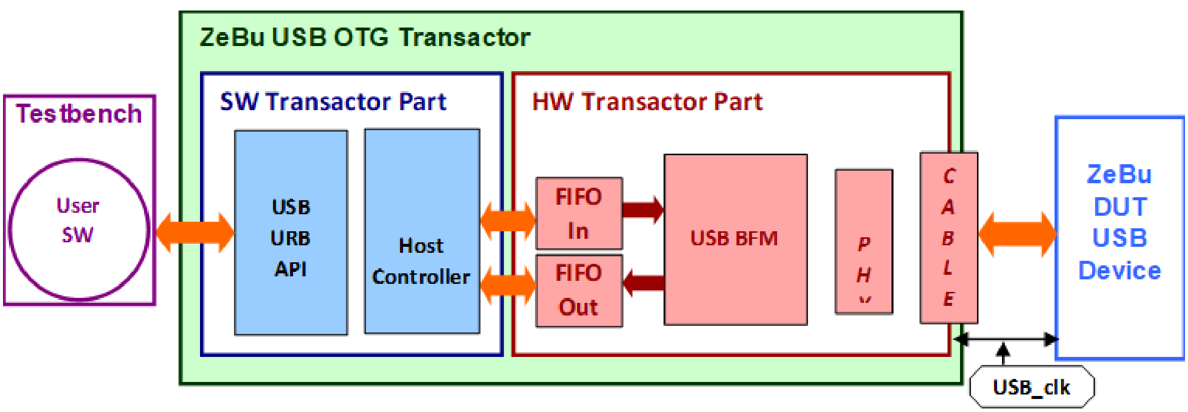

FIGURE 14. USB Request Block (URB) API

6 USB OTG Software Channel/ Endpoint API

The USB OTG 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.

The testbench is responsible for managing the low-level activity transfers, such as, splitting data into single operations, checking endpoint status, and sending data again in case of NAK response.

6.1 Using the USB OTG Channel API

6.1.1 Libraries

The USB OTG Channel API is defined in `UsbOTG.hh` and `UsbStruct.hh` header files.

Use the following library:

- `libUsb.so`: 64-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 OTG API Class Description

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

TABLE 9 USB OTG Channel API Class Description

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

6.3 UsbOTG Class

6.3.1 Description

`UsbOTG` class contains methods to control the USB OTG 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 `UsbOTG` class.

TABLE 10 Types for UsbHost Class

Type	Description
usbHaltStatus_t	Reason for channel halting
OTGStatus_t	USB OTG 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 11 Methods for UsbOTG Class

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

TABLE 11 Methods for UsbOTG 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.
setAddress	Sets the device address - only for OTG B-Device
Channel Management - A-Device (Host)	
channel	Gets channel handler (0 ? n ? 15).
releaseChannel	Releases the channel used by the Host.
Channel Operations A-Device (Host)	
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.

TABLE 11 Methods for UsbOTG Class

Method	Description
enableReception	Enables data reception over the specified channel.
actualXferLength	Returns the number of bytes sent/received over a specified channel.
waitStatusPkt	Prepares the transactor for USB status packet reception phase.
Endpoint Management B-Device (Device)	
Endpoint	Get specific endpoint handler ($0 \leq n \leq 15$)
Endpoint Operation B-Device (Device)	
epEnable	Prepares and configures an endpoint for transmission.
enableReception	Sends a setup or USB request transaction given as parameter over the specified channel.
sendData	Sends a data request transaction of N bytes over the specified channel.
rcvData	Sends a status packet over the specified channel.
sendStatusPkt	Sends a data bytes buffer over the specified channel.
rcvSetup	Gets the received data buffer from the specified channel.
USB Operations A-Device (Host)	
USBPlug	Connects the USB OTG to the cable.
USBUnplug	Disconnects the USB OTG from the cable.
usbReset	Sends the USB Reset sequence to the USB Device.
USB Cable Model Status A-Device (Host)	
portEnabled	Returns true if the USB port is enabled.
isDeviceAttached	Returns true if a device is connected on the USB bus.
USB Cable Model Status B-Device (Device)	
isHostAttached	Returns true if a host is connected on the USB bus
USB Protocol Information	

TABLE 11 Methods for UsbOTG Class

Method	Description
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.
OTG specific methods	
endOfSession	End of session method request.
srpDetected	Detects an SPR.
enableHNP	Enables the HNP.
srp_req	Session Request Protocol requested.
hnp_req	Host Negotiation Protocol requested.

6.3.2 Transactor Initialization and Control Methods

6.3.2.1 Constructor and Destructor Methods

Allocates/frees the structures for the USB OTG transactor.

```
UsbOTG (usbType type)
~UsbOTG (void)
```

Where, usbType is the type of Device at the begin of the run (A-Device or B-Device)

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 OTG transactor and allocates software structures accordingly.

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

TABLE 12 config() Parmeters

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 OTG 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 OTG 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

.

```
OTGStatus_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 13 Log levels for log method

`mask` values	Description
`logOff`	No information.
`logInfo`	Information about USB OTG detection and configuration.
`logUrb`	Information on URBs.
`logBuf`	Information on data transferred through the bus.
`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.

6.3.2.11 setAddress() Method

Sets the device address. The SET_ADDRESS requests are not automatically handled by the transactor, so the application has to use this method upon a SET_ADDRESS request reception to set the device address. It also allows the application to force the device address.

```
void setAddress (uint8_t addr);
```

6.3.3 Channel Management Methods - A-Device (Host)

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 A-Device (Host)

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
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 `UsbOTG::loop()` since they belong to `UsbOTG` 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 Endpoint Management Methods B-Device (Device)

The endpoint method returns a handler to the specified endpoint number. Specify an endpoint number between 0 and 15. Endpoint number 0 is reserved for control transfer.

```
DevEndpoint *endpoint (uint8_t n);
```

where n is the endpoint number.

6.3.6 Endpoint Operation Methods B-Device (Device)

6.3.6.1 epEnable() Method

Sets the endpoint characteristics. The control endpoint (endpoint number 0) is set automatically and cannot be set by the application.

```
void epEnable (DevEndpoint *ep, XferTyp_t typ, bool ep_is_in,
               uint16_t max_pk_size);
```

ep	DevEndpoint *	Device endpoint handler
typ	XferTyp_t	Transmission type

ep_is_in	Bool	Transmission direction
max_pkt_size	uint16_t	Maximum packet size

6.3.6.2 enableReception() Method

Prepares endpoint for data reception (OUT transfer). The endpoint reception needs to be enabled before each OUT data transfer.

```
void enableReception (DevEndpoint* ep, uint32_t len);
```

ep	DevEndpoint *	Device endpoint descriptor
len	uint32_t	Expected transfer size in bytes

The Control endpoint (endpoint number 0) is always enabled so there is no need to use this method for endpoint 0.

6.3.6.3 sendData() Method

Starts sending data over a specified endpoint.

```
void sendData (DevEndpoint *ep, uint32_t *data, uint32_t len);
```

ep	DevEndpoint *	Device endpoint descriptor
data	uint32_t *	Pointer to data buffer. Ignored if data length is zero
len	uint32_t	Expected transfer size in bytes

6.3.6.4 rcvData() Method

Gets the received data. This method should be called upon data transfer completion (when DevEndpoint::XferComplete() and DevEndpoint::dataPresent() both return true).

The data pointed by data and len should be used or copied before the next call of UsbDevice::loop() method since they belong to the UsbDevice class and therefore might be overridden.

```
rcvData (DevEndpoint *ep, uint32_t **data, uint32_t *len);
```

Parameter Name	Parameter Type	Description
ep	DevEndpoint *	Device endpoint descriptor
data	uint32_t **	Pointer to reception data buffer pointer
len	uint32_t *	Pointer to received data length in bytes

6.3.6.5 `sendStatusPkt()` Method

Sends a zero-length status packet over the specified endpoint.

```
sendStatusPkt (DevEndpoint *ep);
```

Parameter Name	Parameter Type	Description
ep	DevEndpoint *	Device endpoint descriptor

6.3.6.6 `rcvSetup()` Method

Gets the received setup request packets. This method should be called upon setup transfer completion (when both `DevEndpoint::setupReceived()` and `DevEndpoint::setupDone()` return `true`).

```
void rcvSetup (DevEndpoint *ep, CtrlReq_t *req);
```

ep	DevEndpoint *	Device endpoint descriptor
req	CtrlReq *	Request structure pointer

6.3.7 USB Operations

6.3.7.1 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)

6.3.7.2 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)

6.3.7.3 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.

6.3.8 USB Cable Model Status Methods A-Device (Host)

6.3.8.1 portEnabled() Method

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

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

6.3.8.2 isDeviceAttached() Method

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

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

6.3.9 USB Cable Model Status Method B-Device (Device)

The `isHostAttached` method returns `true` when a host is connected on the USB bus, `false` otherwise.

```
bool isHostAttached (void);
```

6.3.10 USB Protocol Information

6.3.10.1 portSpeed() Method

Returns the device speed.

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

6.3.10.2 maxPktSize() Method

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

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

6.3.10.3 `isPingSupported()` Method

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

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

6.3.11 General Purpose Vectors Operations

6.3.11.1 `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`.

6.3.11.2 `readUserIO()` Method

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

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

6.3.12 OTG Methods

This section explains the OTG-specific methods:

6.3.12.1 End of Session Method

Enables you to switch off the A-Device (Host):

```
void endOfSession   () ;
```

6.3.12.2 SRP Detection Method

Enables the A-Device (Host) to detect a SRP:

```
bool srpDetected    ();
```

6.3.12.3 Enable HNP Method

This method enables the HNP protocol.

```
void enableHNP   ();
```

6.3.12.4 Session Request Protocol (SRP) Request Method

Enables the B-Device (Device) to request a SRP:

```
void srp_req ()  ;
```

6.3.12.5 Host Negotiation Protocol (HNP) Request Method

Use this method to request a HNP.

```
bool hnp_req ()  ;
```

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 USB OTG A-Device (Host) transactor.

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

TABLE 14 Types for HostChannel Class

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

TABLE 15 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 15 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 DevEndpoint Class

6.5.1 Description

The `DevEndpoint` class contains all methods to access information about the activity on the USB endpoints associated with the USB OTG B-Device (Device) transactor.

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

TABLE 16 Types for DevEndPoint Class

Type Name	Description
XferTyp_t	USB transfer type
HCSpeed_t	USB channel speed
HCReqType	USB Request type
HCReqcode	USB Request code

TABLE 17 Methods for DevEndPoint Class

Method Name	Description
getEpNumber	Returns endpoint number
dataPresent	Returns true when incoming data is present
xferComplete	Returns true when current transfer is completed
setupReceived	Returns true when setup packet has been received
setupDone	Returns true when setup stage is complete
isActive	Returns true when endpoint is active
`isDisabled`	Returns true when endpoint is disabled
`display`	Displays endpoint status for debug purpose

6.5.2 Detailed Methods

6.5.2.1 getEpNumber() Method

Returns the endpoint number.

```
uint8_t getEpNumber (void);
```

6.5.2.2 dataPresent() Method

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

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

6.5.2.3 xferComplete() Method

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

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

6.5.2.4 setupReceived() Method

Returns `true` if the setup packet has been received, `false` otherwise.

```
bool setupReceived (void);
```

6.5.2.5 setupDone() Method

Returns `true` if the setup phase is complete, `false` otherwise.

```
bool setupDone (void);
```

6.5.2.6 isActive() Method

Returns `true` if the endpoint is active, `false` otherwise.

```
bool isActive (void);
```

6.5.2.7 isDisabled() Method

Returns `true` if the endpoint is disabled, `false` otherwise.

```
bool isDisabled (void);
```

6.5.2.8 display() Method

Displays the endpoint status for debug purposes.

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

6.6 Logging USB Packets Processed by the Transactor

6.6.1 USB Packet Processing Logs

The USB OTG 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.

6.6.2 Log Types

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

6.6.2.1 Host Controller Log

This log reports the main stages of the Host controller.

Use the `logCore` option to activate it.

6.6.2.2 USB Transactor Log

This log reports the main transactor steps.

Use the `logInfo` option to activate it.

6.6.2.3 Reference Clock Time Log

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

Use the `logTime` option to activate it.

6.6.2.4 Full Logs

This option sets all log levels detailed above.

Use the `logAny` option to activate it.

6.6.3 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
```

Example : Using the USB OTG Transactor with the PHY UTMI cable model

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

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.

The software testbench consists of the following two processes:

- *Device Process of the Testbench*
- *Host Process of the Testbench*

In addition to the process, this section explains the *USB Device Testbench Example* and *USB OTG Testbench Example*

6.6.4 Device Process of the Testbench

Perform the following steps for the device process of the testbench:

1. Initialize the core.
2. Plug the device.
3. Waits for the events.

   When an event is detected:

   - On endpoint 0: Perform a specific action depending on the type of request (SET_ADDRESS, GET_DESCRIPTOR). Enable used endpoints, either directly or in function of the SET_CONFIGURATION or SET_INTERFACE requests for example.
   - At the endpoint configured as Bulk OUT (in our case it is EP #1): the device stores the data in a list, if data is present.

- At the endpoint configured as `Bulk IN` (EP #2): the device sends the data to that endpoint, if the list is not empty.

6.6.5 USB Device Testbench Example

The following is the example of testbench (a sample from the `example/otg/src/bench/tb_dev.cc` file) using a USB Device transactor:

```
  UsbOTG *usbDev = (UsbOTG*)usb;

  DevEndpoint* ctrlep;
  DevEndpoint* bulkinep;
  DevEndpoint* bulkoutep;
  OTGStatus_t stat;

        if(verbose) printf(" -- DEV -- Start device configuration\n");
#ifdef HS
      usbDev.config(spd_high, utmi_8);  //Device supports High Speed mode
#else
      usbDev.config(spd_full, utmi_8);  //Device supports only Full Speed
mode
#endif

        if(verbose) printf(" -- DEV -- Config done !\n");

      usbDev.USBPlug(0, 0); // Plug device to the cable

      ctrlep    = usbDev.endpoint(0);

      while (1) {
        stat = usbDev.loop();
       if (stat.EpEvent.d32 != 0 || (!bulkin_started && dataQueueLength >
0))          {
```

```
          if (stat.EpEvent.b.DevEp0event) {
            if(verbose) printf(" -- DEV -- Handling CTRL Endpoint event 
\n");
            handle_ctrlep(&usbDev, ctrlep);
            if (configDone == 1) {
              configDone = 0;
              // Setup BULKIN Endpoint
              bulkinep  = usbDev.endpoint(bulkin_epnum);
              usbDev.epEnable(bulkinep ,  XferTyp_Bulk, 1, maxpktsize);

              // Setup BULKOUT Endpoint
              bulkoutep = usbDev.endpoint(bulkout_epnum);
              usbDev.epEnable(bulkoutep , XferTyp_Bulk, 0, maxpktsize);

              usbDev.enableReception(bulkoutep,xfersizemax);
            }

          }
          if ( stat.EpEvent.d32 & (0x1<<bulkin_epnum)) {
            if(verbose)
              printf(" -- DEV -- Handling BULK IN Endpoint event \n");
            handle_bulkinep(&usbDev, bulkinep);
          }
          if (stat.EpEvent.d32 & (0x1<<bulkout_epnum)) {
            if(verbose)
              printf(" -- DEV -- Handling BULK OUT Endpoint event \n");
            handle_bulkoutep(&usbDev, bulkoutep);
          }
        }
        fflush(stdout);
      }
```

Note *Functions in this example that are described neither in the ZeBu USB OTG API Reference Manuals nor in this manual are specific testbench functions described in the tb_dev.cc file.*

You can find another testbench example in the example/otg/src/bench directory, also named tb_dev.cc.

6.6.6 Host Process of the Testbench

Perform the following steps for the host process of the testbench:

1. Initialize the core.
2. Plug the device
3. Wait for the device connection.
4. Send a USB reset.
5. Wait for the Port Enabled.
6. Perform 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. Unplug the device.

6.6.7 USB OTG Testbench Example

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.

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

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

printf(" -- HST -- Starting UsbOTG 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
  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

  OTGStatus_t  stat = usbHst->loop();
    if (stat.HcEvent.d32 != 0 || !bulk_out_started) {
      if (!bulk_out_started || (stat.HcEvent.d32 & (0x1<<bulkout_hcnum)))
{
        host_handle_bulk_out(usbHst, bulkout_epnum);
// Handles BULK OUT events on channel 1
      }
```

```
      if (!bulk_in_started || (stat.HcEvent.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( UsbOTG *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.

This section explains the following topics:

- *Using the URB API*
- *URB API Class, Structure and Type Description*
- *UsbOTG Class*
- *DeviceEP Class*
- *ZeBuRequest Structure*
- *ZebuUrb Structure*
- *URB API Enum Definitions*
- *Managing USB Host URBs*
- *Watchdogs and Timeout Detection*
- *Logging USB Transfers Processed by the Transactor*
- *Using the USB OTG Transactor when B-Device*
- *Using the USB OTG Transactor when A-Device*

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 following figure provides an overview of the library dependencies when building a software testbench using the USB transactor.

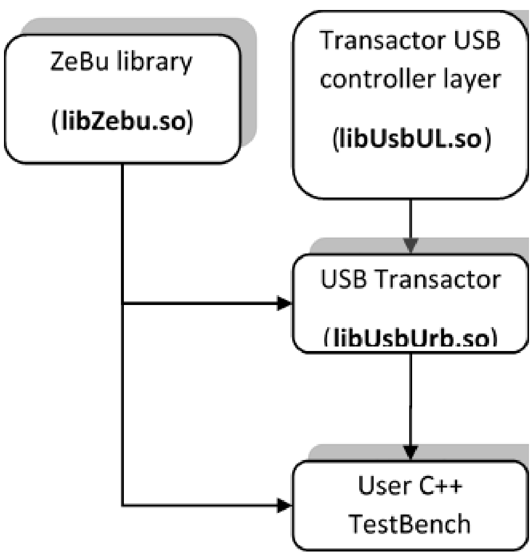

FIGURE 15. Libraries Hierarchy Overview

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, Structure and Type Description

You can instantiate and access the USB Device transactor using the following C++ classes and structures.

TABLE 18 URB API Class description

Class	Description
`UsbOTG`	Represents the USB OTG transactor
`DeviceEP`	Represents an Endpoint when B-Device

TABLE 19 URB API Structure description

Structure	Description
`ZebuRequest`	Represents the request of a B-Device endpoint
`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 a specific 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. Must be used when OTG is Host

The following table describes the types available for the URB API `UsbOTG` and `DeviceEP` classes, defined in `UsbUrbCommon.hh`:

TABLE 20 API Types

Type Name	Description
`zusb_ReqComplete`	Pointer to completion function of a ZeBu Request
`zusb_SetupCB`	Pointer to callback function called when setup access is required by Host
`zusb_Complete`	Pointer to completion function of a ZeBu URB.

The USB OTG transactor contains the definition of some standard USB structures.

TABLE 21 USB OTG transactor standard definitions

Type Name	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 (i.e. Audio, Printer, etc.)
zusb_TransferFlags	Information about the URB transfer
zusb_RequestRecipient	Recipient of the request
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

7.3 UsbOTG Class

This section explains the following topics:

- *Description*
- *Transactor Initialization and Control Methods*
- *USB Device Information Methods*
- *USB Operations*
- *USB Device Configuration Management Methods*
- *General Purpose Vectors Operations*
- *Advanced User Methods*
- *USB Device Setup Methods*

7.3.1 Description

The methods associated with the `UsbOTG` object 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 `UsbOTG` class is defined in the `UsbUrbOTG.hh` file.

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

TABLE 22 Methods for UsbDev Class

Method Name	Description
Transactor Initialization and Control	
`UsbOTG`	Constructor
`~UsbOTG`	Destructor
`Init`	ZeBu Board attachment
`InitBFM`	USB transactor configuration & initialization
`USBPlug`	Connects and attaches USB Device connector to the USB cable

TABLE 22 Methods for UsbDev Class

Method Name	Description
`USBUnplug`	Disconnects and detaches USB Device connector from the USB cable
`Loop`	Processes all the pending Tx or Rx USB packets, waits for BFM events or interrupts , and blocking methods
`DiscoverDevice`	Checks device connection. Enables port and resets the USB device if present. Returns ZUSB_STATUS_SUCCESS if the device is ready for transfers. Only when UsbOTG is A-Device
`Delay`	Inserts a delay in milliseconds of USB clock
`RegisterCallback`	Registers a service loop callback
`SetDebugLevel`	Select the logging mode
`SetLogPrefix`	Set a log prefix to be used in log information
`SetLog`	Sets a file where log messages will be saved
`SetTimeOut`	Sets the time of the watchdog.
`RegisterTimeOutCB`	Registers a callback to be called if watchdog is triggered.
`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 Device Setup	
`RegisterControlSetupCB`	Register callback called when a control setup access comes from Host
`GetEndpoint`	Returns an endpoint of the connected device
General Purpose Vector Operations	

TABLE 22 Methods for UsbDev Class

Method Name	Description
writeUserIO	Drives the general purpose bus_O vector with the specified value
readUserIO	Reads the general purpose bus_I vector
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.
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
USB OTG special feature	
endOfSession	Usb End Of Session
srpDetected	SRP detection – when A-Device
srp_req	SRP request – when B-Device
setXtorType	Set the type of Xtor (A-Device or B-Device)
disableHnpReq	Disable the HNP Request

TABLE 22 Methods for UsbDev Class

Method Name	Description
usbSuspend	Usb suspend
usbResume	Usb resume

7.3.2 Transactor Initialization and Control Methods

7.3.2.1 Constructor/Destructor Methods

Allocates/frees the structures for the USB Device transactor.

```
UsbOTG (usbType type)
~UsbOTG (void)
```

Where usbType can be usbTypHost or usbTypDev

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 ZeBu board structure
driverName	const char *	Name of the transactor instance in DVE file
clkName	const char *	Name of the primary clock used as a time reference in debug messages (optional).

Note *When Bus Monitoring 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 Section, the name is device_xtor_clk.*

7.3.2.3 `InitBFM()` Method

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

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

Parameter Name	Parameter Type	Description
`highSpeed`	`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.

7.3.2.4 `USBPlug()` Method

Connects the device to the USB cable using 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 device (in ms).
latency	uint32_t	Time to wait before connecting the device (in ms).

7.3.2.5 USBUnplug( ) Method

Disconnects the device from the USB cable using 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 disconnection of the device (in ms).
latency	uint32_t	Time to wait before disconnecting the device (in ms).

7.3.2.6 Loop() Method

Checks the USB Device 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 in order to allow the controlled clock to advance and to check events.

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

On success, the method returns ZUSB_STATUS_SUCCESS.

Otherwise, it returns any other zusb_status enum type.

7.3.2.7 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.

7.3.2.8 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.

7.3.2.9 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.

7.3.2.10 `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

7.3.2.11 `SetDebugLevel()` Method

Enables and configures the transactor with the level and specified message categories.

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

where `val` is the log level as defined hereafter:

TABLE 23 Log levels for SetDebugLevel Method

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

7.3.2.12 `SetLogPrefix()` Method

Sets a new log prefix for log.

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

where `prefix` is the prefix to use in logs.

7.3.2.13 `SetLog()` Method

Activates the log generation and defines where to print log messages.

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

Parameter Name	Parameter Type	Description
`stream`	`FILE *`	Path and name of the file where to print messages.
`stdoutDup`	`bool`	Activates/deactivates print to the standard output: • `true`: log information is output to the standard output as well as in the log file • `false` (default): log information is output only to the log file.

7.3.2.14 `SetTimeOut()` Method

Sets the value of the watchdog timeout in milliseconds.

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

where `msecs` is the timeout value in milliseconds.

7.3.2.15 `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

7.3.2.16 `getVersion()` Method

Two different prototypes exist for this function:

- 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 used 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);
```

7.3.8 USB Device Setup Methods

7.3.8.1 `RegisterControlSetupCB()` Method

Registers a callback called when a control setup command comes from the host.

```
void RegisterControlSetupCB (ZebuRequest *ctrl_req,
zusb_SetupCB setupCB, void *context);
```

Parameter Name	Parameter Type	Description
`ctrl_req`	`ZebuRequest *`	Request description
`setupCB`	`zusb_SetupCB`	Pointer to the setup callback
`context`	`void*`	Context to be passed to the callback function

7.3.8.2 `GetEndpoint()` Method

Gets an endpoint of the device. Returns a pointer to the endpoint; `NULL` otherwise.

```
DeviceEP *GetEndpoint (uint8_t num) const;
```

where `num` is the endpoint address that ranges from `0` to 15.U

USB OTG special feature

7.3.8.3 endOfSession () Method

generate a Usb End Of Session

```
void endOfSession ()
```

7.3.8.4 srpDetected () Method

This method return true when a SRP is detected

```
bool srpDetected()
```

7.3.8.5 srp_req ( ) Method

This method request a SRP request

```
void srp_req ()
```

7.3.8.6 setXtorType ( ) Method

This method set the type of Xtor (A-Device or B-Device)

```
void setXtorType    (usbType type)
```

7.3.8.7 disableHnpReq ( ) Method

This method disable the HNP Request

```
void disableHnpReq   ()
```

7.3.8.8 usbSuspend ( ) Method

This method suspend the USB BUS

```
void        usbSuspend  (   )
```

7.3.8.9 usbResume ( ) Method

This method resume the USB BUS

```
void usbResume  ( uint32_t duration )
```

7.4 DeviceEP Class

This class contains the methods to control an USB Device Endpoint and manage all the USB Device requests contained by `ZebuRequest` structures. This class is defined in the `UsbUrbDev.hh` file.

TABLE 24 DeviceEP Class Methods

Method Name	Description
`GetNumber`	Returns the endpoint address
`AllocRequest`	Creates a new request associated with the endpoint
`FreeRequest`	Frees a request
`EnableEP`	Enables an endpoint
`DisableEP`	Disables and endpoint
`QueueEP`	Queues a request to be sent to the device
`DequeueEP`	Dequeues a request
`SetHalt`	Stalls an endpoint
`ClearHalt`	Clears the stalling of an endpoint
`SetPrivateData`	Attaches a private data to be used by the testbench inside callbacks.
`GetPrivateData`	Gets the attached private data
`GetMaxPacket`	Gets the maxpacket size of the endpoint

7.4.1 Constructor/Destructor Methods

The `DeviceEP` constructor is private and then unavailable. It uses the `GetEndpoint()` method of the `UsbDev` class to get a pointer to the required endpoint.

7.4.2 `GetNumber()` Method

Returns the number of the endpoint, from `0` to `15`.

```
uint8_t GetNumber () const;
```

7.4.3 `AllocRequest()` Method

Creates and allocates a USB Device request for the endpoint. Using this method is mandatory; allocating statically a Device `ZebuRequest` is not allowed.

```
ZebuRequest* AllocRequest (uint32_t length);
```

where `length` is the length of the data to transfer.

This method returns a pointer to the request. It returns `0` in case of error.

7.4.4 `FreeRequest()` Method

Deletes a request of the current endpoint.

```
void FreeRequest (ZebuRequest *req);
```

where `req` is the request to delete.

7.4.5 `EnableEP()` Method

Enables the endpoint and sets its characteristics as defined in the EndPoint descriptor. This method indicates to the peripheral controller of the transactor that the endpoint is enabled and that the transactor prepares it for transfers.

```
zusb_status EnableEP (zusb_EndpointDescriptor *desc);
```

Parameter Name	Parameter Type	Description
`desc`	`zusb_EndpointDescriptor*`	Standard endpoint descriptor structure. This structure contains all the information needed to initialize the endpoint properly.

On success, it returns ZUSB_STATUS_SUCCESS; any other zusb_status enum type otherwise.

7.4.6 QueueEP( ) Method

Submits a request to the endpoint for transfer. This method queues a request for this endpoint. The peripheral controller then waits for any incoming request from the host on this endpoint. The endpoint must have been enabled with `EnableEP()`.

```
zusb_status QueueEP (ZebuRequest *req);
```

where `req` is the request to transfer to the host.

On success, it returns ZUSB_STATUS_SUCCESS; any other zusb_status enum type otherwise.

7.4.7 DequeueEP( ) Method

Removes a request from the transfer queue of the endpoint. This function is generally called when an error occurred and you want to cancel a request.

```
zusb_status DequeueEP (ZebuRequest *req);
```

where `req` is the pending request to abort.

On success, it returns ZUSB_STATUS_SUCCESS; any other zusb_status enum type otherwise.

7.4.8 SetHalt( ) Method

Use this method to stall an endpoint. The endpoint remains halted until the host clears this feature.

On success, this call sets the underlying hardware state that blocks data transfers.

```
zusb_status SetHalt ();
```

On success, it returns ZUSB_STATUS_SUCCESS; any other zusb_status enum type otherwise.

7.4.9 ClearHalt( ) Method

Use this method when responding to the standard `SET_INTERFACE` request, for endpoints that are not configured after clearing any other state in the endpoint queue.

On success, this call clears the underlying hardware state reflecting endpoint halt and data toggle.

```
zusb_status ClearHalt ();
```

On success, it returns ZUSB_STATUS_SUCCESS; any other zusb_status enum type otherwise .

7.4.10 SetPrivateData() Method

Add a pointer to a user data to get private information inside completion functions.

```
void SetPrivateData (void *data);
```

where `data` is the pointer to user data.

7.4.11 GetPrivateData() Method

Gets the pointer to user data. This method is used inside completion function.

```
void* GetPrivateData ();
```

7.4.12 GetMaxPacket() Method

Returns the maximum packet size of the endpoint in bytes.

```
uint32_t GetMaxPacket () const;
```

7.5 ZeBuRequest Structure

The dedicated C++ ZebuRequest structure is supplied with the USB Device transactor to create and manage USB requests that must be processed by your testbench on the different endpoints.

This structure is defined in the UsbUrbDev.hh file.

```
struct ZebuRequest {
uint8_t*buf;
uint32_tlength;
uint8_t zero;
uint8_t short_not_ok;
int32_t status;
uint32_t actual;
zusb_ReqComplete complete;
void **context;
};
```

TABLE 25 Content of the ZebuRequest Structure

Parameter	Description
buf	Buffer containing data for the USB transfer
length	Data size of the USB transfer
zero	If true when writing data, indicates that the last packet must be a short packet by adding a 0 packet as needed. You must set this field when queuing an IN request.
short_not_ok	Reserved, must be set to 0.
status	Request status: one of the zusb_transfer_status values.
actual	Gives the actual size of the transfer. Must be used to: • Get the actual size of an OUT transfer (may be smaller than length field) • Set the actual size of an • IN transfer (may be smaller than length field)

TABLE 25 Content of the ZebuRequest Structure

Parameter	Description
`complete`	Pointer to a completion function. This callback is called each time a new request comes from the Host.
`context`	Context to be passed to the completion function

7.6 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)
status	Current status of the ZebuUrb block. The status returned is one of the zusb_transfer_status enum.

Parameter	Description
`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
`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.

7.7 URB API Enum Definitions

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

7.7.1 `zusb_status` enum

Statuses returned by the API methods.

TABLE 26 zusb_status Enum Description

Parameter Name	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

7.7.2 `zusb_transfer_status` enum

Statuses of transfers using `ZebuRequest`.

TABLE 27 zusb_transfer_status Enum Description

Parameter Name	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 Name	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

7.8 Managing USB Host URBs

This chapter discusses the way to send or receive USB blocks to/from the device.

7.8.1 Creating and Destroying ZeBu URBs

Allocating and destroying ZeBu URBs must be done using `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

7.8.1.1 `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.

7.8.1.2 `FreeUrb()` Method

Frees a `ZeBuUrb` structure.

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

where `zurb` is the pointer to the `ZebuUrb` to delete.

7.8.1.3 `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.

7.8.1.4 `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.

7.8.2 Filling a `ZebuUrb` Structure

The following methods must be used to fill a `ZebuUrb` structure. There is one version per type of transfer:

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.

7.8.2.1 `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

7.8.2.2 `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

7.8.2.3 `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.

7.8.2.4 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
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`.

7.8.2.5 `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
`offset`	`uint32_t`	Offset in transfer buffer
`length`	`uint32_t`	Length of the packet

7.8.3 Submitting ZeBu URBs

Once the ZebuUrb structure is properly filled using the methods described in the *Filling a ZebuUrb Structure* section, 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.

7.8.4 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 30 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

7.8.4.1 `SendControlMessage()` Method

Sends a control message and waits for the message to be complete. Function parameters correspond to the zusb_ControlSetup structure fields (refer to 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 which have actually been transferred.

On error, it returns one of the statuses of the zusb_status enum.

7.8.4.2 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 which have actually been transferred.

On error, it returns one of the statuses of the zusb_status enum.

7.8.5 Examples

7.8.5.1 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);
```

7.8.5.2 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");
}
```

7.8.5.3 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;
  }
}
```

7.9 Watchdogs and Timeout Detection

7.9.1 Description

To avoid deadlocks during USB Device transfers, the USB 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, described hereafter.

TABLE 31 Methods for Watchdogs and Timeout Detection

Method Name	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.

7.9.1.1 `EnableWatchdog()` Method

This 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.

7.9.1.2 `SetTimeOut()` Method

This method sets the watchdog timeout values in milliseconds.

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

where `msec` is the timeout value in milliseconds. By default, this value is 180 000 ms.

7.9.1.3 `RegisterTimeOutCB()` Method

This 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 the callback function
context	void *	Pointer to the data passed to the callback

If the deadlock cannot be fixed from the callback, true can be returned 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.

This feature can be disabled by registering a NULL pointer.

7.9.2 USB Device Transactor Default Behavior

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

7.10 Logging USB Transfers Processed by the Transactor

7.10.1 USB Request Processing Logs

The USB Device transactor allows logging USB transfer activity in a file or on the standard output. Each log type is chosen independently.

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

7.10.2 Log Types

7.10.2.1 Main Info Log

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

Use the `logInfo` option to activate it.

7.10.2.2 USB Request Logs

This log reports URB activity.

Use the `logUrb` option to activate it.

7.10.2.3 Controller Core Logs

This log reports the device controller core activity (i.e. core initialization phase with endpoint management).

Uses the `logCore` option to activate it.

7.10.2.4 Data Buffer Log

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

Use the `logBuf` option to activate it.

7.10.2.5 Reference Clock Time Log

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

Use the `logTime` option to activate it.

7.10.3 USB Requests Log Example

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

```
Device XTOR   Info   : Enabling Info logs
Device XTOR   Info   : Enabling URB logs
Device XTOR   Info   : Enabling Controller logs
                                                                 logInfo
Device XTOR   Info    : Enabling Data Buffer transfer logs
Device XTOR   Info    : Enabling Reference clock time logs
Device CTRL   Info    : Starting controller core initialization
Device CTRL   Info    : Core initialization done
Device CTRL   Info    : Starting PCD initialization
Device CTRL   Info    : PCD initialization done
Device CTRL   DEBUG   : EP0 DataPID:D0 Frame:14
Device CTRL   DEBUG   : Setup pkt: ReqType: 0x80
Device CTRL   DEBUG   :     Req: GET_DESCRIPTOR (0x06)
Device CTRL   DEBUG   :     Value: 0x0100                       logCore
Device CTRL   DEBUG   :     Index: 0x0000
Device CTRL   DEBUG   :     Length: 0x0040
Device CTRL   DEBUG   : EP0 DataPID:D0 Frame:14
```

```
Device CTRL   DEBUG   : Setup Complete
Device CTRL   DEBUG   : EP0 dir:OUT type:CONTROL, maxpacket:64
Device CTRL   DEBUG   : EP0 Setup Phase Done
HOST TIME DEBUG       : clk_48m cycle - 5256487          logTime
```

7.11 Using the USB OTG Transactor when B-Device

The USB Device transactor behaves like a peripheral controller. The testbench is written like a Linux gadget driver.

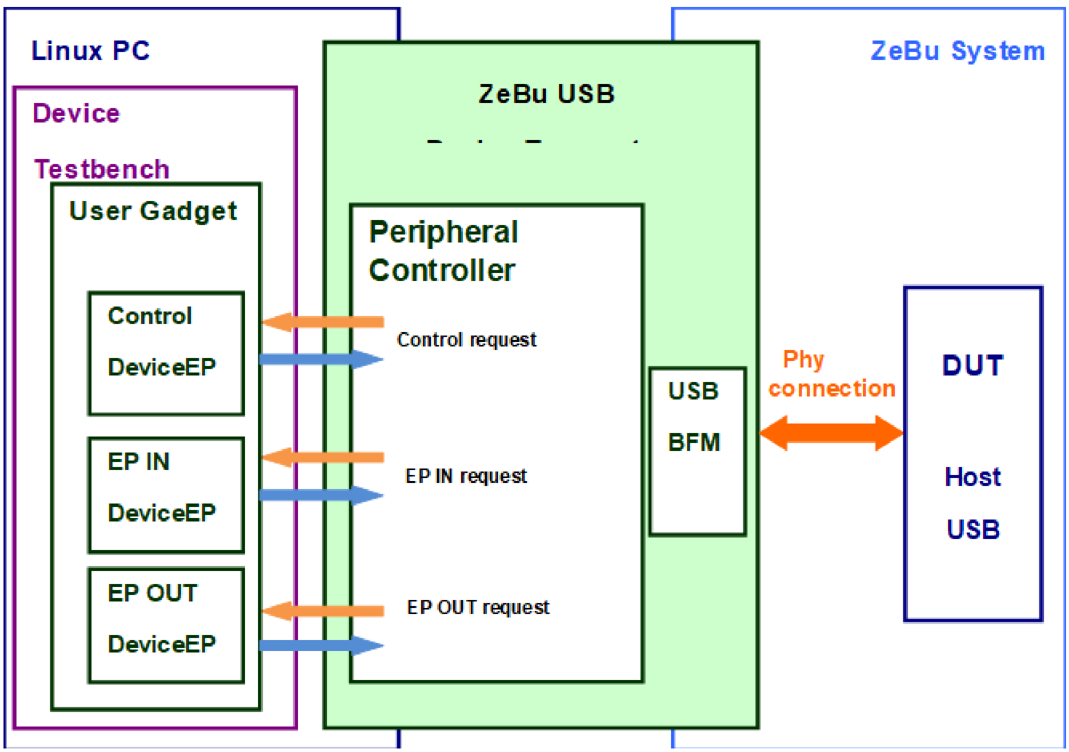

FIGURE 16. Device Transactor Overview

This section details the sequence to properly initialize and control the USB device transactor.

7.11.1 Device Transactor Control Overview

Here is an overview of the device transactor control in a testbench.

7.11.1.1 Typical Testbench Initialization Sequence

A typical testbench initialization sequence is as follows:

1. Initialization of the ZeBu board and the transactor.
2. Plug-in of the device to the system.
3. Initialization of the control endpoint.
4. Host chooses a configuration/interface for the device.
5. Initialization endpoints in the function of the chosen interface.

7.11.1.2 USB Transfers Control Flow

As soon as endpoints are initialized, the testbench starts. The Loop( ) method is called in a loop to make the USB clock run. Each time an event appears on an endpoint, a callback mechanism is used to process the USB transfer.

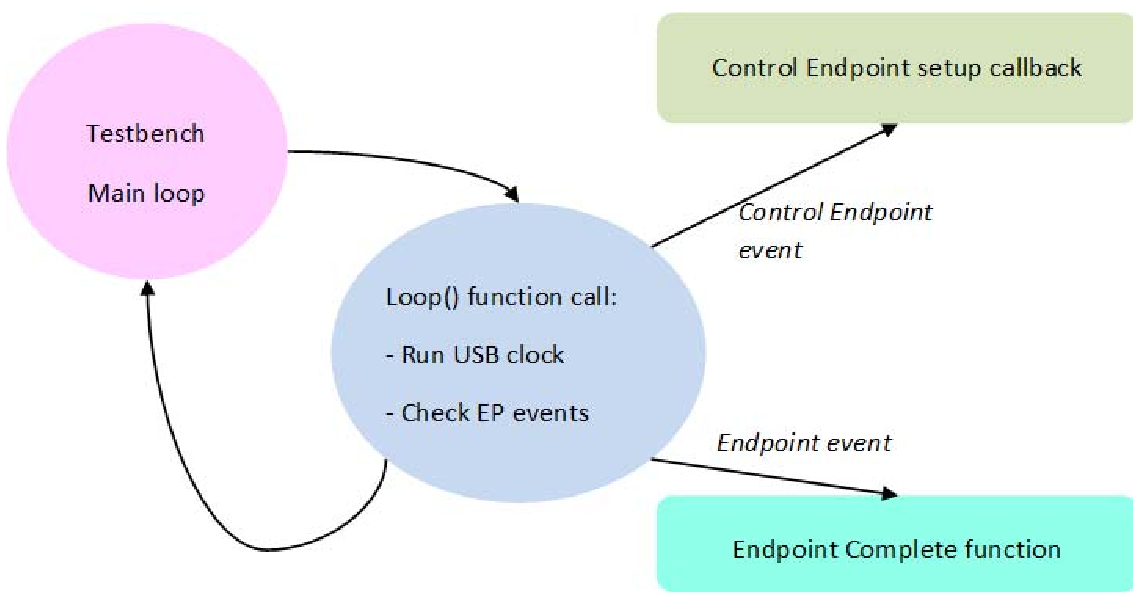

FIGURE 17. Main testbench loop transfer flow

7.11.2 Initializing the USB Device transactor

The device transactor initialization consists in:

1. Opening the ZeBu board by calling the `open()` method.
2. Calling the USB device transactor `Init()` method.
3. Calling the ZeBu board initialization using
4. `init()` method.
5. Setting transactor parameters.
6. Calling `InitBFM()`to finalize transactor initialization.

Here is an example of a typical initialization:

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

UsbDev usbDev;

// Initializing the USB transactor
  usbDev.Init(board, "usb_driver_dev_inst", "clk_48m");

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

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

// Initialize the USB transactor BFM
```

```
// Setting High speed devices
  if(usbDev.InitBFM(true)) { /* High speed device */
    cerr << "Initialization failed." << endl; return 1;
  }
```

At the end of this sequence, the transactor is ready to connect the device to the system.

7.11.3 Plugging the USB Device to the System

When the transactor is properly initialized, it is time to plug the device to the system by calling `USBPlug()`.

```
// Connecting the device
// Delay connection for 2 milliseconds
  if(usbDev.USBPlug(2) == ZUSB_STATUS_ERROR) {
    cerr << "Connection failed." << endl;
    return 1;
}
```

`USBPlug()` returns `ZUSB_STATUS_SUCCESS` if the device is properly connected.

This method physically connects the device to the hardware system in ZeBu and initializes the peripheral controller. The device is then ready to communicate with the Host.

7.11.4 Storing Device Endpoint/Request Representation

Device access is mostly done through callbacks called on endpoint events. It is recommended to store a device representation that is to be passed to these callbacks.

The device representation is design-dependent; the following code is an example of device structure used to store this information. You are free to have your own representation.

```
struct usb_device_example {
  // Control endpoint structures
```

```
  ZebuRequest* ctrl_req;         /* for control responses */
  DeviceEP*    ctrl_ep;         /* Controle EP0 */

  // Other endpoints structures
  DeviceEP*    in_ep;        /* IN BULK EP */
  DeviceEP*    out_ep;       /* OUT BULK EP */
  ZebuRequest* in_req;       /* IN BULK REQUEST */
  ZebuRequest* out_req;      /* OUT BULK REQUEST */

  // Pointer to the USB transactor
  UsbDev*      xtor;         /* Pointer to xtor */
};
```

where:

- `ctrl_req`: Pointer to the control endpoint request (`ZebuRequest`)
- `control_ep`: Pointer to the control endpoint (`DeviceEP`)
- `in_ep` and `out_ep`: Representation of endpoints used in the device; here, one in and one out (`DeviceEP` instances) are expected
- `in_req` and `out_req`: Representation of the device endpoint requests (`ZebuRequest`)
- `xtor`: Pointer to the USB Device transactor

7.11.5 Initializing the EP0 Control Endpoint

When USB plug-in is done successfully, the USB device transactor is ready to be used.

To be able to receive Host requests, control endpoint 0 of the device must be initialized. Since this endpoint is the control endpoint, it must be initialized before any other endpoints since it is used to configure the device.

7.11.5.1 Initialize EP0

The sequence to initialize EP0 consists in:

1. Getting the handle to the Control Endpoint from the USB transactor
2. Allocating a request structure for this endpoint, this structure is used to manage the USB control transfers during the run
3. Attaching a callback to be called at USB transfer completion stage

Here is an example of EP0 initialization using the example design.

```
static void control_ep_complete(DeviceEP*  ep,
                         ZebuRequest* req)
{
  // user code, request status check for example
}

// Instantiating and initializing an example device structure
usb_device_example dev;
dev.ctrl_ep = 0;
dev.in_ep = 0;
dev.out_ep = 0;
dev.ctrl_req = 0;
dev.in_req = 0;
dev.out_req = 0;
dev.xtor = usbDev;

// Initialize control endpoint
// Store control endpoint in example device structure

dev.control_ep = usbDev.GetEndpoint(0);

// Allocate request, the size depends on the maximum size expected
// for the device descriptor
```

```
dev.ctrl_req = dev.ctrl_ep->AllocRequest(MAX_SIZE);

// Set the complete callback function for control endpoint
dev.ctrl_req->complete = control_ep_complete;
```

7.11.5.2 EP0 Setup Control Callback

When control request from the Host appears on the control Endpoint. The USB control command received is handled in two places:

- Inside the transactor peripheral driver:
  The peripheral driver is responding to the USB host without propagating to the gadget driver on the following commands:
  - GET_STATUS
  - CLEAR_FEATURE
  - SET_FEATURE: Device and endpoint requests are processed by the peripheral controller, but the interface requests are passed to the user driver.
  - SET_ADDRESS
- In the user Gadget Driver through its Control setup callback:
  - GET_DESCRIPTOR
  - SET_DESCRIPTOR
  - SET_CONFIGURATION
  - GET_CONFIGURATION
  - SET_INTERFACE
  - GET_INTERFACE
  - SET_FEATURE: Interface requests only, the Device and Endpoint requests are processed by the peripheral driver.

The control setup callback attached to the control endpoint is used to read and process the commands dedicated to the gadget driver. The `RegisterControlSetupCB()` method is used to register this callback.

```
usbDev.RegisterControlSetupCB(dev.req, example_dev_setup_cb,
                              (void*)(&dev));
```

The control setup callback prototype is the following:

```
typedef int32_t (*zusb_SetupCB)(ZebuRequest*       ctrl_req,
                                zusb_CtrlRequest*  ctrl,
                                void*              context);
```

where:

- `ctrl_req`: pointer to the control endpoint request
- `ctrl`: control request command
- `context`: context passed to the callback, most probably the device description

Here is an example of control setup callback:

```
static int32_t
example_dev_setup_cb ( ZebuRequest* req,
                  zusb_CtrlRequest* ctrl,
                  void*             context)
{
  usb_device_example* dev =
static_cast<usb_device_example*>(context);
  int     value = -1;

  // The following code is an example of control request
  // interpretation.
  req->zero = 0;
  switch (ctrl->bRequest) {

 case ZUSB_REQ_GET_DESCRIPTOR:

    switch (ctrl->wValue >> 8) {

      case ZUSB_DT_DEVICE:
```

```
        // device_desc is a device descriptor of type
        // zusb_DeviceDescriptor (USB standard descriptor)
        value = min (ctrl->wLength, (uint16_t) sizeof device_desc);
        memcpy (req->buf, &device_desc, value);
        break;
    ...
   // user code

// If the Host requires a response from the device, it must
// queue a request
  if (value >= 0) {
    req->length = value;

// Forcing a zero packet if required
    req->zero = value < ctrl->wLength
        && (value % dev->control_ep->GetMaxPacket()) == 0;

// Queue the request
    value = dev->control_ep->QueueEP(req);
    if (value < 0) {
      printf("ep_queue error status : %d\n", value);
      req->status = 0;
    }
  }
  return value; }
```

The method must returns `0` on success or a negative value on error (the device will then stall).

7.11.6 Managing Endpoints

The management of endpoints 1 to 15 is different from the control endpoint. The endpoints are generally initialized in the setup control callback because of a set configuration or a set interface request from the host.

7.11.6.1 Initializing Endpoint

The user gadget device must contain a descriptor of each endpoint used through the `zusb_EndpointDescriptor` structure.

```
static const zusb_EndpointDescriptor z_bulkout_ep = {
  bLength : 0x07,
  bDescriptorType : ZUSB_DT_ENDPOINT,
  bEndpointAddress : 0x01,
  bmAttributes : 0x06,
  wMaxPacketSize : 0x0200,
  bInterval : 0x01
};
```

The descriptor above is used by the transactor to get information on the endpoint when it is enabled. This descriptor is defined by USB 2.0 standard specification.

The information extracted from this descriptor is the following:

- The maximum packet size of this endpoint
- The transfer type (Bulk, Interrupt, Isochronous)
- The endpoint address

The sequence to initialize an endpoint consists in:

1. Getting the handle to the endpoint from the USB transactor
2. Enabling the endpoint
3. Attaching private data to endpoint (used inside callbacks to pass user data)

The following example shows a typical sequence to initialize an endpoint. Note that the private data passed to the endpoint is a pointer to a `usb_device_example` structure. All device information is then available inside callbacks.

```
// Getting static endpoint descriptor
  d = &z_bulkin_ep;
  dev->in_ep = dev->xtor->GetEndpoint((d->bEndpointAddress) &
ZUSB_ENDPOINT_NUMBER_MASK);
  if(dev->in_ep == 0) {
    cerr << "Unable to get xtor endpoint"; return -1;
  }

// Enable the endpoint
  result = dev->in_ep->EnableEP (d);
  if (result != ZUSB_STATUS_SUCCESS) {
    dev->in_ep->DisableEP();
  }
  else {
    dev->in_ep->SetPrivateData(dev);
  }
```

7.11.6.2 Disabling an Endpoint

At any time during the run, an endpoint can be disabled by calling the `DisableEP()` method. It is generally called when an error occurred on endpoint or when the user changes the device interface and then modifies the used endpoints.

When calling this function, all requests attached to the endpoint are cancelled and structures a freed correctly.

7.11.6.3 Creating a Request Transfer

To receive requests from the host, the endpoint must initiate a request transfer by:

1. Allocating a request structure for this endpoint; this structure is used to manage the USB transfers during the run.
2. Attaching a callback to be called at USB transfer completion stage.
3. Queuing the request.

```
  // Endpoint is enabled and can start transfers

  ZebuRequest* req;

  req = dev->in_ep->AllocRequest(512);
  if (!req) return -1;

// Attach a completion function to the request
  req->complete = bulk_complete;
  req->length = 512;

  // user code to fill request buffer
  …
  result = dev->in_ep->QueueEP(req);
```

7.11.6.4 Canceling a Transfer

At any time, the request on an endpoint can be canceled by calling the `DequeueEP()` method.

7.11.6.5 OUT Endpoint Completion Function

The completion function is passed during the request creation and is called when the host accesses the endpoint. This function processes incoming data (`OUT` direction).

After request processing, this completion function is responsible for queuing a new request to activate the OUT endpoint request reception in the peripheral driver.

The following code is an example of completion function of an OUT endpoint and Bulk transfer:

```
static void
bulk_complete(DeviceEP*  ep,
              ZebuRequest* req)
{
  struct usb_device_example  *dev = (struct usb_device_example*)ep-
>GetPrivateData();
  int    status = req->status;
  int i, j;

  if(status == 0) {       /* normal completion? */
    printf("\nbulkout %d reading %d bytes\n", numTransfer, req-
>actual);
    // Reading data from the request
    for (i=0; i<req->actual; i++) {
      rbuf[0] = req->buf[i];
    }
  }
  else {
    if(status == ZUSB_TRANSFER_CANCELLED) {
      printf("\nRequest cancelled for endpoint number %u, ignoring
data\n", ep->GetNumber());
      return;
    }
    else {
      printf("\nEndpoint number %u stopped, status %d unexpected,
exit\n", ep->GetNumber(), status);
```

```
        exit(1);
      }
    }

    // Queuing a new request
    status = ep->QueueEP(req);
    if (status != ZUSB_STATUS_SUCCESS) {
      exit(1);
    }
  }
```

7.11.6.6 `IN` Endpoint Completion Function

The completion method is passed during the request creation and is called when the host accesses the endpoint. This function has to send requested data (`IN` endpoint).

After request processing, this completion function is responsible for queuing a new request to activate the `IN` endpoint request reception in the peripheral driver.

The following code is an example of completion function of an `IN` endpoint and `Bulk` transfer:

```
static void
bulk_complete(DeviceEP*  ep,
              ZebuRequest* req)
{
  struct usb_device_example  *dev = (struct usb_device_example*)ep->GetPrivateData();
  int    status = req->status;
  int i, j;

  req->length = 512;
  req->actual = req->length;
```

```
  req->zero = 0;
  if(status == 0) {       /* normal completion? */
    printf("\nbulkin %d writing %d bytes\n", numTransfer, req-
>actual);
    // Writing data to the request structure
    for (i=0; i<req->actual; i++) {
      req->buf[i] = localbuf[i];
    }
  }
  else {
    if(status == ZUSB_TRANSFER_CANCELLED) {
      printf("\nRequest cancelled for endpoint number %u, ignoring
data\n", ep->GetNumber());
      return;
    }
    else {
      printf("\nEndpoint number %u stopped, status %d unexpected,
exit\n", ep->GetNumber(), status);
      exit(1);
    }
  }
  // Queuing a new request
  status = ep->QueueEP(req);
  if (status != ZUSB_STATUS_SUCCESS) {
    exit(1);
  }
}
```

7.11.6.7 Stalling an Endpoint

When the endpoint cannot handle the request, it can indicate it to the Host by stalling the endpoint. An endpoint is stalled by calling the `SetHalt()` method. Stalling an endpoint is not a fatal error but indicates that the device either cannot interpret a command or cannot handle the request for the moment.

The Host is responsible for clearing the halt condition by sending a `CLEAR_FEATURE` request on the endpoint. This clear halt condition is then managed internally by the transactor peripheral controller.

If the application decides to clear the halt condition by itself (during a `SET_INTERFACE` for example), it can be done by calling the `ClearHalt()` method.

7.11.7 USB Device Main Loop Description

As soon as control endpoint 0 is initialized, the testbench can start running the clock and wait for requests by calling the `Loop()` method.

```
while(1) {
  if((err = usbDev.Loop()) != ZUSB_STATUS_SUCCESS) {
    cerr << “Device error status : “ << err << endl;
  }
}
```

During this loop:

- Control requests from the Host are either processed in the peripheral driver or sent to the user driver through its control callback (see Section above for further details).

USB request on the other endpoints are sent to user driver as soon as it is enabled and a request is queued.

7.12 Using the USB OTG Transactor when A-Device

7.12.1 Initializing the USB Host Transactor

Please find below the sequence to properly initialize the USB Host transactor.

7.12.1.1 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"));
```

7.12.1.2 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;
  }
```

7.12.1.3 Connecting the USB Host Transactor to the Cable

`USBPlug()` connects the Host to the USB cable:

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; }
```

7.12.1.4 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 Section). Sends multiple GET_DESCRIPTOR control requests to discover the USB device (number of calls depends on the device).
10. Sends a SET_CONFIGURATION control request to select the configuration. Default is the first one available. Can be set to another value with *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
```

7.12.2 Managing USB Device Configuration and Interfaces

This section explains the following sections:

- *Managing the Device Address*
- *Managing the Device Configuration*
- *Managing the Device Interface*
- *Example of Configuration Parsing*
- *Disconnecting the USB Host Transactor from the Cable*

7.12.2.1 Managing the Device Address

You can manage the device address using the following methods:

- `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.

7.12.2.2 Managing the Device Configuration

You can manage the device configuration using the following methods:

- `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.

7.12.2.3 Managing the Device Interface

You can manage the device interface using the following methods:

- `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 <u>after</u> 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.

7.12.2.4 Example of Configuration Parsing

The following example explains the 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);
```

7.12.2.5 Disconnecting the USB Host Transactor from the Cable

Follow the below sequence to disconnect the USB Host transactor.

```
// Disonnects 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 OTG 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. For more information on the hardware setup, see *USB Bus Monitoring*.

8.1.2 USB OTG 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 32 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
ProgName	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.

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.

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 33 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.

```
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.
`CreateFSDB`	bool	Enable or disable the FSDB monitor for Protocol Analyzer.
`ProgName`	const char	Must be the name of testbench executable. Use the ProgName argument if the PATH variable is not pointing to the run directory. (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.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.

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

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.

9 Tutorial

The example directory of the transactor package contains the *phy_utmi* example. To run the example, see *Running the Examples*.

9.1 phy_utmi

This example shows how the transactor is used for an integration with the USB DUT device using the UTMI interface, under the following topics:

- *Overview*
- *ZeBu Design*
- *Software Testbench*

9.1.1 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:

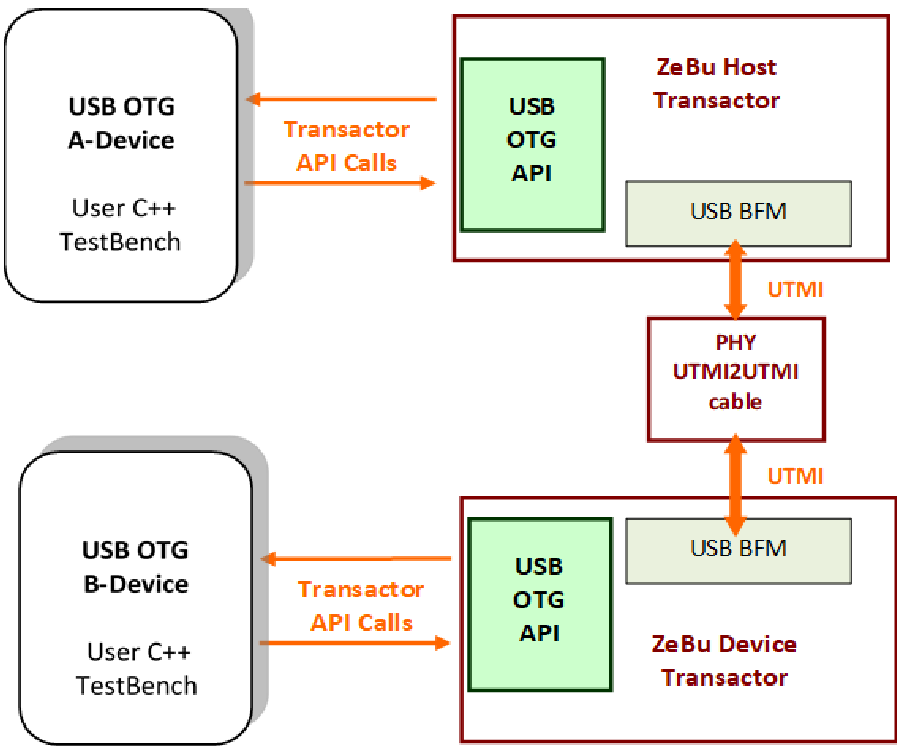

FIGURE 18. phy_utmi example overview

9.1.2 ZeBu Design

The ZeBu design consists of a:

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

9.1.3 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 OTG 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).

9.2 Running the Examples

Running the examples consists of the following steps:

- *Setting the ZeBu Environment*
- *Compiling the Designs*
- *Running the phy_utmi Example*
- *SRP-HNP Example*

9.2.1 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>
```

9.2.2 Compiling the Designs

1. Go to the example/otg/zebu directory.
2. Specify the following make command:

```
@> make compile
```

9.2.3 Running the phy_utmi Example

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

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

9.2.4 SRP-HNP Example

This example can be run using either the Channel API. The initialization sequence (Setting device address, etc.) is quite different between Channel and URB API testbenches but the demo sequence is the same:

1. The USB OTG 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.

9.2.4.1 HNP testbench example

```
usbHst->enableHNP();

//usb Host Transfer
….

// usb Host Suspend
  usbHst->delay(3,true);
  usbHst->usbSuspend();

  // check Hnp Detection flag of OTGStatus
    OTGStatus_t  stat = usbHst->loop();
  while (stat.HnpDetected == 0){
    stat = usbHst->loop();
  }
//Check the Transactor Type
  printf("#TB -- HST HNP -- HNP Detected\n" );  fflush(stdout);
  if(stat.IsDevice) printf("#TB -- HST HNP -- Host is Detected as a
Device\n" );  fflush(stdout);
```

```
  if(stat.IsHost  ) printf("#TB -- HST HNP -- Host is Detected as a Host
\n" );  fflush(stdout);


      //-- request a HNP
      printf("#TB -- HST -- Start HNP \n"); fflush(stdout);
      bool hnp_req = usbHst->hnp_req();

//Wait HNP success
      while(stat.HnpSuccess ==0)
        stat = usbHst->loop();
```

9.2.4.2 SRP testbench example

Host Side :

```
//usb Host Transfer



//-- Start End od seccion request
printf("#TB  -- HST SRP -- Starting Host End Of
Session\n");fflush(stdout);
usbHst->endOfSession();




 while (!usbHst->srpDetected())
      usbHst->loop();
  printf("#TB  -- HST SRP -- SRP Detected\n");fflush(stdout);



 usbHst->usbReset(10);
```

Device Side :

```
//-- Check End Of Session
         do {
          stat = usbDev->loop();
          SessionEnd = stat.SesEndDetected;
          printf("#TB -- DEV SRP -- SessionEnd Actif  %d\n" ,
SessionEnd);fflush(stdout);
        } while (!SessionEnd) ;

//-- Apply SRP request
        usbDev->delay(10,true);
          usbDev->srp_req();
```