Verification Continuum^{TM}

ZeBu® USB Device User Guide

Version Q-2020.06, August 2020

SYNOPSYS®

Contents

Feedback

List of Tables

List of Figures

About This Manual

Overview

This manual describes the ZeBu USB Device synthesizable transactor which is compliant with the USB 2.0 standard protocol.

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.

For further details on the API interfaces of the USB Device transactor, please refer to the ZeBu USB Device Transactor - USB API Reference Manual and ZeBu USB Device Transactor - URB API Reference Manual Reference (PDF and HTML formats).

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

1 Introduction

1.1 Overview

1.1.1 Description

The USB bus (Universal Serial Bus) is used for data transfers from a host to devices with bandwidths ranging from 1.5 to 480 Mb/s (Low-, Full-, and Hi-Speed devices). USB protocol has been 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.

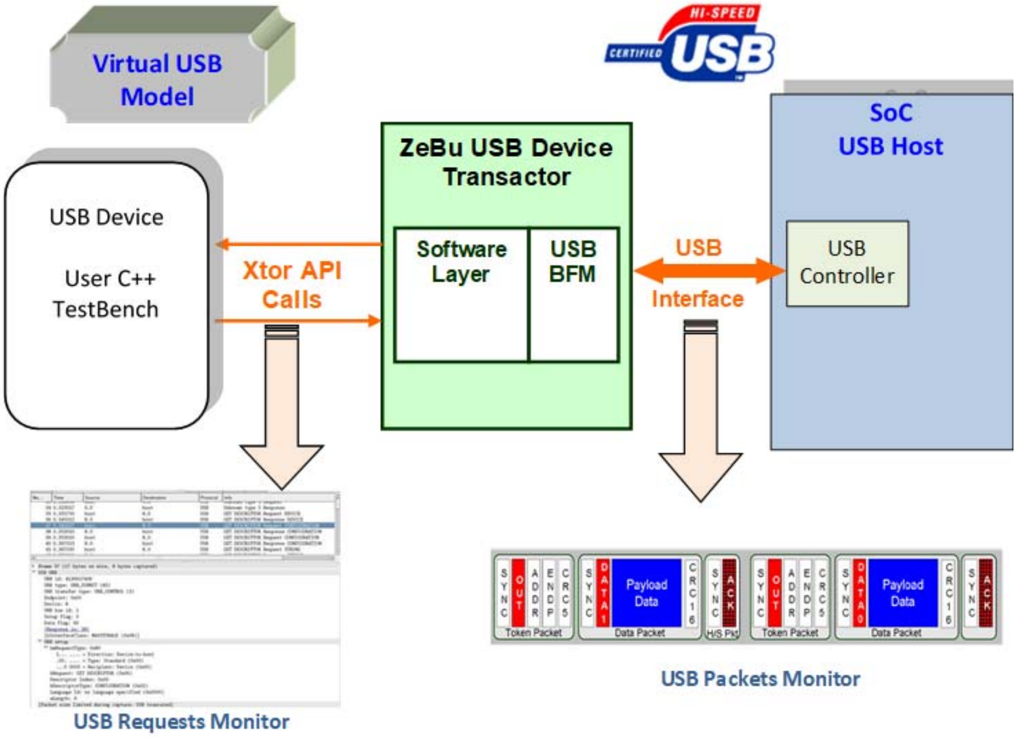


FIGURE 1. ZeBu USB Device Transactor application

1.1.2 USB Interfaces Compliance

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

- UTMI Level 3 interface (USB Transceiver Macrocell Interface).
  This is a UTMI parallel interface, coming in 8- and 16-bit data interfaces.
- ULPI interface (UTMI Low Pin Interface).
- USB differential links, also known as DP/DM Serial Interface.

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

- PHY UTMI2UTMI
- PHY UTMI2ULPI
- PHY CABLE

For more information on DUT-Transactor integration, please refer to Chapter 3.

For a description of each cable model, please refer to Section 4.3.

1.1.3 API Layers

The USB Device transactor offers 2 levels of C++ Application Program Interface (API):

- USB Endpoint level for USB packets transfer for Low-level USB software stack integration.
- USB Request Block (URB) Level for USB driver integration at kernel structure level

1.2 Features

1.2.1 List of Features

The ZeBu USB Device synthesizable transactor has the following features:

- USB 2.0 compliant.
- UTMI USB PHY DUT-side interface is UTMI+ Level 3 compliant.
- ULPI USB PHY DUT-side interface is compliant with ULPI revision 1.1.
- USB DP/DM cable DUT-side interface is compliant with USB 2.0 signaling.
- Controllable USB disconnection.
- Supports both USB Full-Speed (FS) and USB Hi-Speed (HS).
- Support of PING control.
- Supports BULK, Interrupt and Isochronous USB transfers.
- Supports up-to-1024-byte USB packets.
- 16 USB Endpoints are available.

1.2.2 List of Integration Components

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

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

1.2.3 APIs

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

- Endpoint 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.

1.2.4 Serial Interface

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

USB Full-Speed Mode Transfer Type	**USB Device transactor**
Bulk OUT	252 kbit/s
Bulk IN	252 kbit/s

1.2.5 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.

USB Hi-Speed Mode Transfer Type	USB Device transactor
Bulk OUT	12.62 Mbit/s
Bulk IN	2.0 Mbit/s

1.3 Limitations

USB Features

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

USB Transactor API

- Isochronous and Interrupts transfers are not supported by the USB Endpoint API
- Only one USB Device transactor per Linux process is allowed.

1.4 FLEXnet License Feature

You need the following FLEXnet license features:

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

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

2 Installation

2.1 Installing the ZeBu USB Device Transactor Package

To install the USB Device transactor, proceed as follows:

1. Make sure you have WRITE permissions on the IP directory and on the current directory.
2. Download the transactor compressed shell archive (`.sh`).
3. Install the USB Viewer as follows:

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

where:

- `[options]` defines the working environment:

For:	...with Linux OS:	...Specify:
ZeBu Server-1 environment	32- or 64-bit	nothing
Any ZeBu machine	32-bit only	32b

- `[ZEBU_IP_ROOT]` is the path to your ZeBu IP root 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 installation process is complete and successful when the following message is displayed:

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

If an error occurred during the installation, a message is displayed to point out the error. Here is an error message example:

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

2.2 Package Description

After the ZeBu Device Transactor has been correctly installed, it comes with the following elements:

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

2.3 File Tree

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

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

```
$ZEBU_IP_ROOT
 `-- XTOR
     `--USB.<version>
        |-- components
        |   |-- usb_driver_FS_Serial_Device.v
        |   `-- usb_driver_Utmi_Device.v
        |-- uc_xtor
        |   |-- usb_driver_FS_Serial_Device.v
        |   `-- usb_driver_Utmi_Device.v
        |-- doc
        |-- drivers
        |   |-- dve_templates
        |   |   |-- usb_driver_FS_Serial_Device_dve.help
        |   |   `-- usb_driver_Utmi_Device_dve.help
        |   |-- usb_driver_FS_Serial_Device.<version>.install
        |   |-- usb_driver_FS_Serial_Device.install
        |   |-- usb_driver_Utmi_Device.<version>.install
        |   `-- usb_driver_Utmi_Device.install
        |-- drivers.zse
        |   |-- usb_driver_FS_Serial_Device.<version>.install
        |   |-- usb_driver_FS_Serial_Device.install
        |   |-- usb_driver_Utmi_Device.<version>.install
        |   `-- usb_driver_Utmi_Device.install
        |-- example
        |   |-- phy_cable
        |   `-- phy_utmi
        |-- gate
        |   |-- USB_OTG_DUAL.ngo
        |   |-- usb_driver_FS_Serial_Device.<version>.edf
        |   `-- usb_driver_Utmi_Device.<version>.edf
        |-- include
        |   |-- DevEndpoint.<version>.hh
        |   |-- Usb.<version>.hh
        |   |-- UsbDev.<version>.hh
        |   |-- UsbStruct.<version>.hh
        |   |-- UsbUrbCommon.<version>.hh
        |   `-- UsbUrbDev.<version>.hh
        |-- lib32
        |   |-- libUsb.<version>_6.so
        |   |-- libUsbUL.<version>_6.so
        |   `-- libUsbUrb.<version>_6.so
        |-- lib64
        |   |-- libUsb.<version>.so
        |   |-- libUsbUL.<version>.so
        |   `-- libUsbUrb.<version>.so
        `-- misc
            |-- monitor
            |   `-- usb_monitor.sv
            |-- phy_cable
            |   |-- phy_cable_DeviceToHost.edf
```

```
                |    |-- phy_cable_DeviceToHost_blackbox.v
                |    |-- phy_cable_DeviceToHost_blackbox.vhd
                |    `-- src
                |-- phy_utmi2ulpi
                |    |-- phy_utmi2ulpi_DeviceToHost.edf
                |    `-- phy_utmi2ulpi_DeviceToHost_blackbox.v
                `-- phy_utmi2utmi
                     |-- phy_checker.edf
                     |-- phy_checker_blackbox.v
                     |-- phy_utmi2utmi_DeviceToHost.edf
                     `-- phy_utmi2utmi_DeviceToHost_blackbox.v
```

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

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

3 DUT-Transactor Integration with Cable Models

To properly integrate the transactor with the DUT, you should implement the appropriate cable model.

3.1 Architecture with the UTMI DUT Interface

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

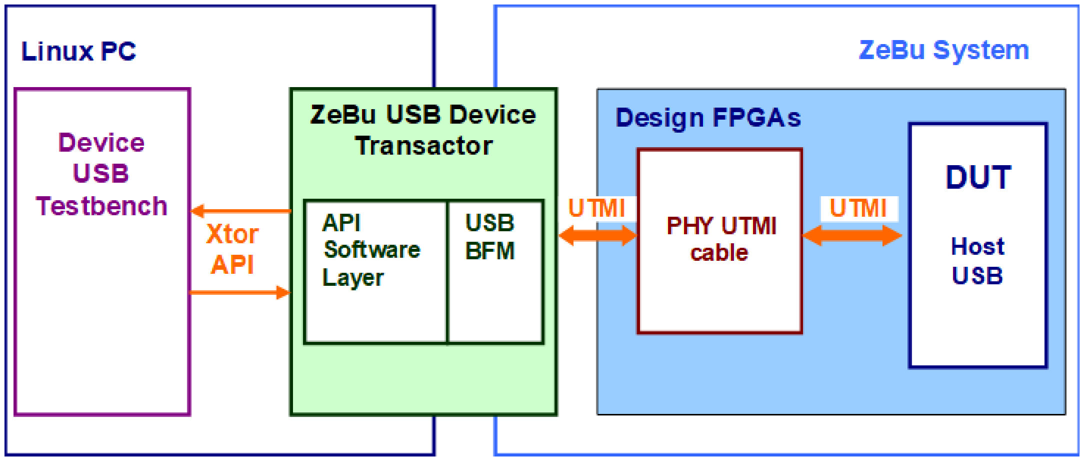


FIGURE 2. USB Device transactor with the UTMI DUT interface

Note *The misc/phy_utmi2utmi directory contains an example of this cable model architecture: phy_utmi2utmi_DeviceToHost_blackbox.v.*

3.2 Architecture with the ULPI DUT Interface

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

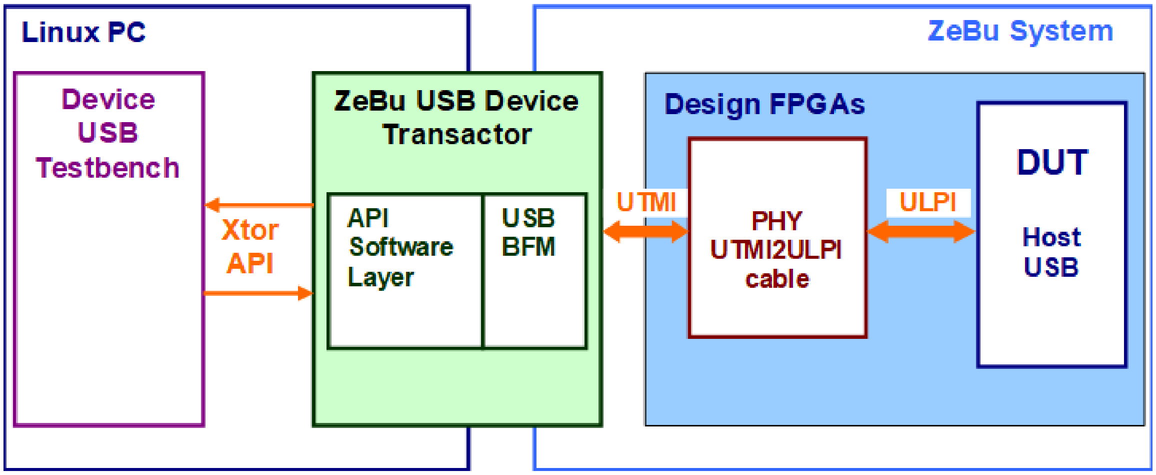


FIGURE 3. USB Device transactor with the ULPI DUT interface

Note *The misc/phy_utmi2ulpi directory contains an example of this cable model architecture: phy_utmi2ulpi_DeviceToHost_blackbox.v.*

3.3 Architecture with the USB Cable DUT Interface

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

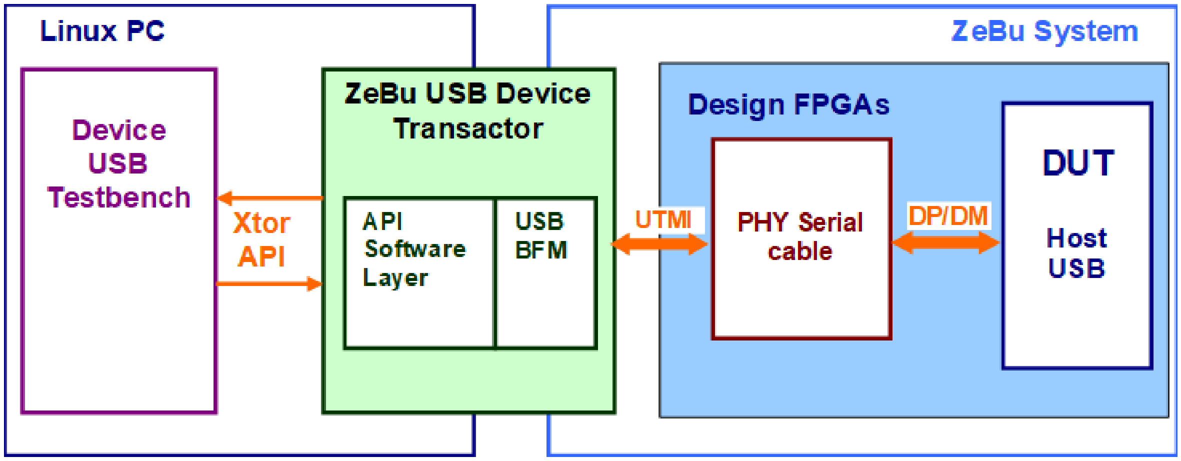


FIGURE 4. USB Device Transactor with the USB Cable DUT Interface

Note *The misc/phy_cable directory contains an example of this cable model architecture: phy_cable_DeviceToHost_blackbox.v.*

4 Using the URB-to-PCAP Converter

4.1 Interface Overview

The USB Device transactor has three types of interface:

- UTMI 8 or 16 bits
- DP/DM serial
- ULPI interface

4.1.1 UTMI Interface

TABLE 2 UTMI Hardware Interface

Symbol	**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

TABLE 2 UTMI Hardware Interface

Symbol	Size	Type (XTOR)	Type (Cable DUT)	Description
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)
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

TABLE 2 UTMI Hardware Interface

Symbol	**Size**	**Type (XTOR)**	**Type (Cable DUT)**	**Description**
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.
device_con	1	O	I	Simulate device connection/ disconnection
mon_config	2	O	I	Bus monitor 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 model version verification.

The following table describes the control signal of the transactor for UTMI mode.

TABLE 3 Control signals

Symbol	Size	Type (XTOR)	Type (B2B Cable)	Description
utmi_word_if	1	O	I	Data bus width: · 1'b0: 8-bit interface · 1'b1: 16-bit interface

TABLE 4 UTMI Size Interface Associated with Cable Model Type

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

4.1.2 DP/DM Serial Interface

TABLE 5 Signal List of the Serial interface

Symbol	Size	Type (XTOR)	Type (Cable DUT)	Description
xtor_clk	1	I	NA	Transactor clock
xtor_resetn	1	I	NA	Transactor reset (active low)
phy_clk	1	I	I	USB PHY clock
prst_n	1	I	I	PHY reset (active low)
DP_in	1	I	O	DP line

TABLE 5 Signal List of the Serial interface

Symbol	Size	Type (XTOR)	Type (Cable DUT)	Description
`DM_in`	1	I	O	DM line
`DP`	1	O	I	Full speed DP driver
`DP_en`	1	O	I	Full speed DP enable
`DM`	1	O	I	Full speed DM driver
`DM_en`	1	O	I	Full speed DM enable

4.1.3 ULPI Interface

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

4.2 Connecting the Transactor's Clocks

4.2.1 Overview

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

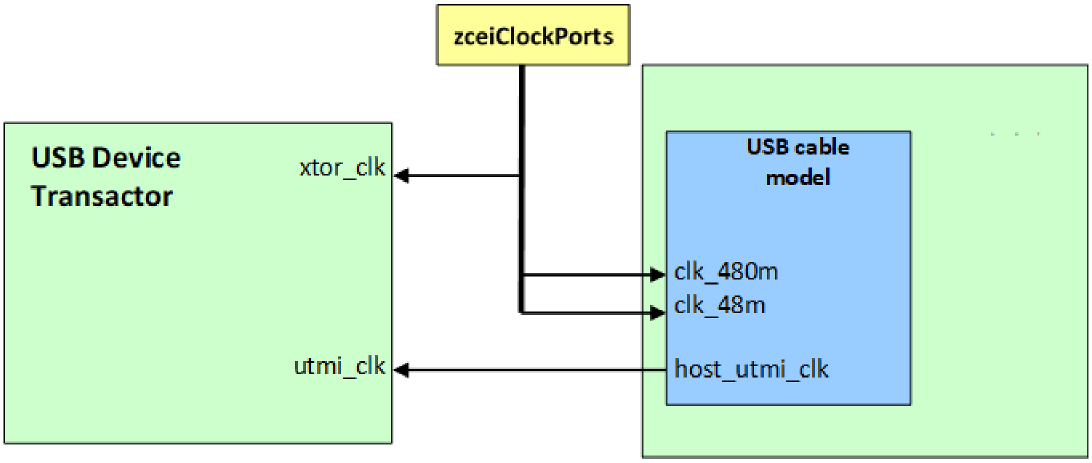


FIGURE 5. Connecting the USB clocks

These clocks are declared as `zceiClockPort` instances in the DVE file and their characteristics are defined for run-time in the `designFeatures` file.

4.2.2 Signal List

TABLE 6 USB Device 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 DVE File Example

The following DVE file example connects the USB Device transactor using the serial interface:

```
zceiClockPort usb_device_zceiClockPort (
    .cclock  (device_xtor_clk)
    .cresetn (xtor_resetn)
);
zceiClockPort device_phy_clk (
    .cclock (device_phy_clk)
);
//Device driver part
usb_driver_FS_Serial_Device usb_driver_device_inst
     (
                .xtor_clk               (device_xtor_clk),
                .xtor_resetn            (xtor_resetn),
                .phy_clk                (device_phy_clk),
                .prst_n                    (cable_resetn),
                .DP_in                     (dp),
                .DM_in                     (dm),
                .DP                        (device_LSFS_DP_tx),
                .DP_en                     (device_LSFS_DP_en),
                .DM                        (device_LSFS_DM_tx),
                .DM_en                     (device_LSFS_DM_en)

      );
defparam usb_driver_device_inst.clkCtrl = device_xtor_clk;
defparam usb_driver_device_inst.process =
"usb_driver_inst_device";

assign cable_resetn = xtor_resetn;
```

4.2.3.2 Defining the Clocks in the `designFeatures` File

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

$B0.device_phy_clk.VirtualFrequency = 60;
$B0.device_phy_clk.GroupName = "phy_clk";
$B0.device_phy_clk.Waveform = "_-";
$B0.device_phy_clk.Mode = "controlled";
```

4.3 USB Cable Models

4.3.1 PHY UTMI Cable Model

The USB Device 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.

The USB cable model must be compiled on ZeBu within the DUT in order to be connected to the USB Device transactor at UTMI level.

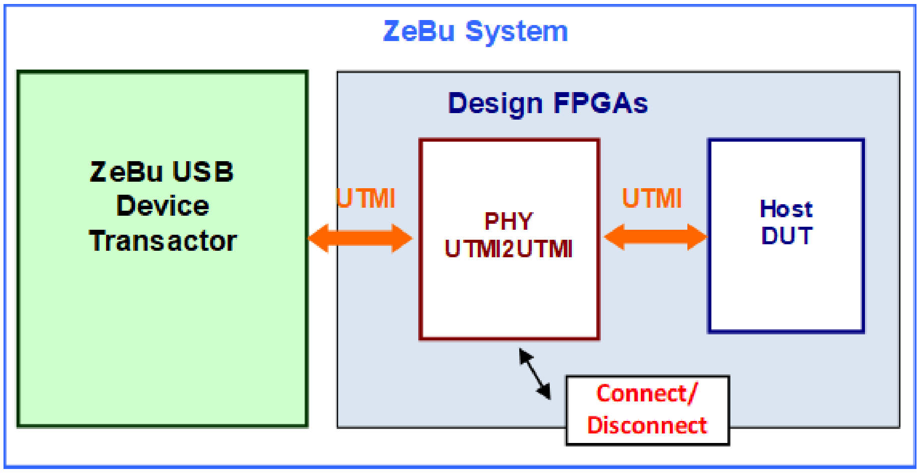


FIGURE 6. USB PHY UTMI2UTMI cable

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

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

TABLE 7 PHY UTMI2UTMI cable interface (DeviceToHost)

Symbol	Size	Type	Side	Description
cable_resetn	1	I	X	Cable reset (active low)
device_con	1	I	X	Device connection. Available on device side only. Please refer to Section 4.3.4 for further details.
clk_48m	1	I	X	48 MHz clock
clk_480m	1	I	X	480 MHz clock
UTMI Interface on both Transactor and DUT sides				
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 bytes read from the PHY Low/high bytes are asserted valid by utmi_rxvalid/utmi_rxvalidh respectively
utmi_dataout	16	I	X/D	8/16 bytes sent to the PHY Low/high bytes 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
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

TABLE 7 PHY UTMI2UTMI cable interface (DeviceToHost)

Symbol	Size	Type	Side	Description
`utmi_suspend_n`	1	I	X/D	PHY is in suspend mode: disables the clock
`utmi_termselect`	1	I	X/D	Selects HS/FS termination · `1'b0`: HS termination enabled · `1'b1`: FS termination enabled
`utmi_xcverselect`	2	I	X/D	Selects HS/FS transceiver · `2'b00`: HS transceiver enabled · `2'b01`: FS transceiver enabled
`utmi_word_if`	1	I	X/D	Selects 8/16 bits interface · `1'b0`: 8-bit interface · `1'b1`: 16-bit interface
`utmi_hostdisconnect`	1	O	X/D	Peripheral disconnect indicator to host.
Transactor Monitor				
`mon_config`	2	I	X	Monitor configuration
`rx_mon_trig`	1	O	D	Monitor internal signal
`rx_mon_sel`	4	O	D	Monitor internal bus
`rx_mon_data`	128	O	D	Monitor internal bus
`tx_mon_trig`	1	O	D	Monitor internal signal
`tx_mon_sel`	4	O	D	Monitor internal bus
`tx_mon_data`	128	O	D	Monitor internal bus

4.3.2 PHY ULPI Cable Model

The USB Device 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.

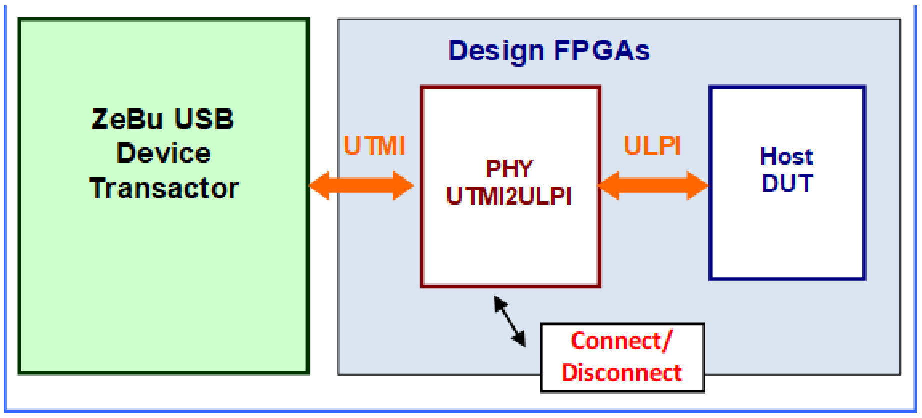


FIGURE 7. PHY UTMI2ULPI cable

ULPI interface connection with the DUT is described as follows:

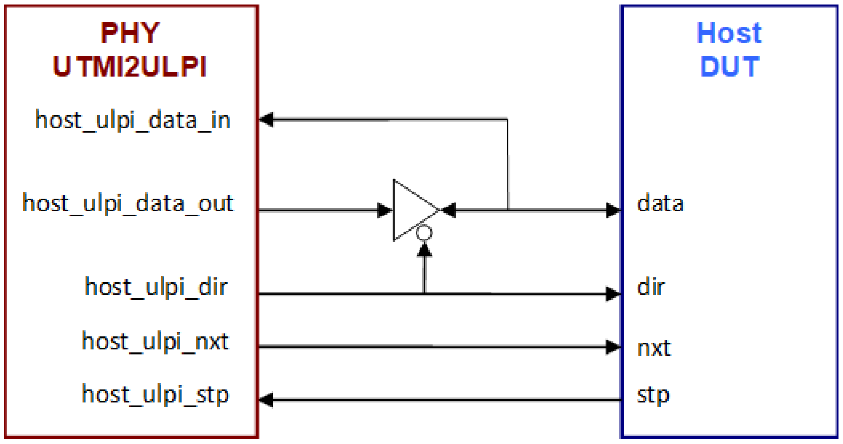


FIGURE 8. PHY UTMI2ULPI connection to standard Host DUT

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 DVE.
- When
- *Side* = D, the signal is driven by the DUT.

TABLE 8 PHY UTMI2ULPI cable interface

Symbol	Size	Type	Side	Description
cable_resetn	1	I	X	Cable reset (active low)
device_con	1	I	X	Device connection. Available on device side only. Please refer to Section for further details.
clk_48m	1	I	X	48 MHz clock
clk_480m	1	I	X	480 MHz clock
UTMI Interface on XTOR side				
utmi_clk	1	O	X	USB PHY clock
utmi_txready	1	O	X	PHY ready for next packet to transmit
utmi_datain	16	O	X	8/16 bytes read from the PHY Low/high bytes are asserted valid by `utmi_rxvalid/utmi_rxvalidh` respectively
utmi_dataout	16	I	X	8/16 bytes sent to the PHY Low/high bytes are asserted valid by `utmi_rxvalid/utmi_rxvalidh` 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

TABLE 8 PHY UTMI2ULPI cable interface

Symbol	Size	Type	Side	Description
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
utmi_termselect	1	I	X	Selects HS/FS termination 1'b0: HS termination enabled 1'b1: FS termination enabled
utmi_xcverselect	2	I	X	Selects HS/FS transceiver 2'b00: HS transceiver enabled 2'b01: FS transceiver enabled
utmi_word_if	1	I	X	Selects 8/16 bits interface 1'b0: 8-bit interface 1'b1: 16-bit interface
utmi_hostdisconnect	1	O	X	Peripheral disconnect indicator to host
utmi_fsls_low_power	1	I	D	PHY Low Power Clock Select (selects PHY clock mode for power saving)
utmi_fslsserialmode	1	I	D	PHY Interface Mode Select (tied low for parallel interface)
ULPI Interface on DUT side				
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 Monitor				

TABLE 8 PHY UTMI2ULPI cable interface

Symbol	Size	Type	Side	Description
mon_config	2	I	X	Monitor configuration.
rx_mon_trig	1	O	D	Monitor internal signal
rx_mon_sel	4	O	D	Monitor internal bus
rx_mon_data	128	O	D	Monitor internal bus
tx_mon_trig	1	O	D	Monitor internal signal
tx_mon_sel	4	O	D	Monitor internal bus
tx_mon_data	128	O	D	Monitor internal bus

4.3.3 PHY Serial Cable Model

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

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

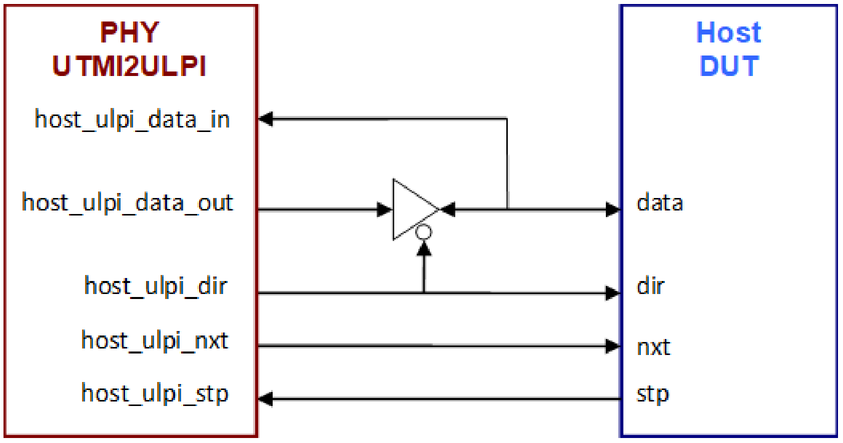


FIGURE 9. PHY serial cable

TABLE 9 PHY serial cable interface

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

TABLE 9 PHY serial cable interface

Symbol	Size	Type	Description
utmi_word_if	1	I	Selects 8/16 bits interface • 1'b0: 8-bit interface • 1'b1: 16-bit interface
utmi_hostdisconnect	1	O	Peripheral disconnect indicator to host.
Serial Interface on DUT side			
dm	1	I	DM line
dp	1	I	DP line
LSFS_DP_tx	1	O	Full-speed DP driver
LSFS_DM_tx	1	O	Full-speed DM driver
LSFS_DP_en	1	O	Full-speed DP enable
LSFS_DM_en	1	O	Full-speed DM enable
HS_DP_tx	1	O	High-speed DP driver
HS_DM_tx	1	O	High-speed DM driver
HS_DP_en	1	O	High-speed DP enable
HS_DM_en	1	O	High-speed DM enable
clk_48m	1	I	48 MHz clock
clk_480m	1	I	480 MHz clock

4.3.4 Connection/Disconnection of the USB Device from the DUT

From the Device transactor, connection/disconnection to the USB device is enabled via the device_con input symbol of the cable model. This input is driven by the transactor via the `USBPlug` and `USBUnplug` API methods.

See below a typical sequence of a device disconnection/connection:

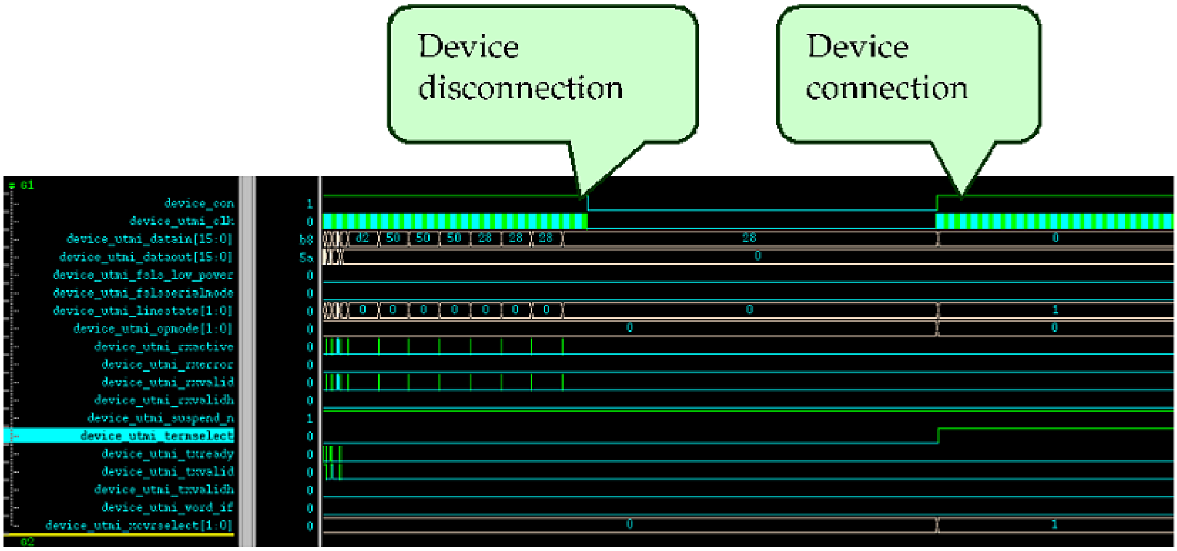
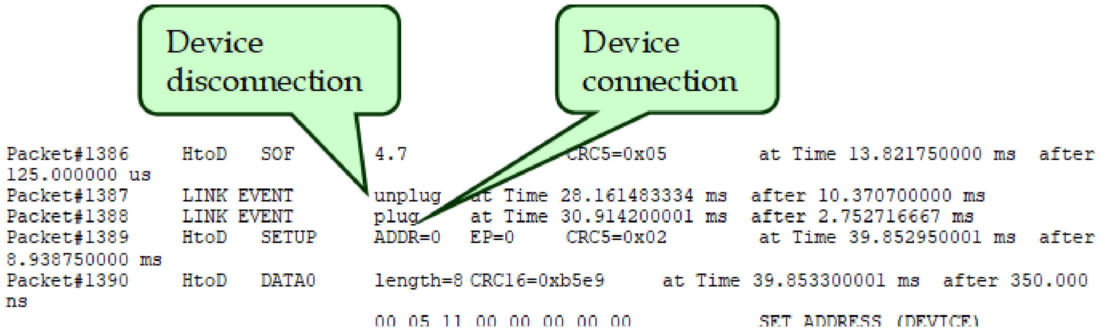


FIGURE 10.

The same sequence seen by the USB Bus Monitor:



```
Packet#1386     HtoD    SOF      4.7          CRC5=0x05        at Time 13.821750000 ms   after
125.000000 us
Packet#1387     LINK EVENT       unplug    at Time 28.161483334 ms   after 10.370700000 ms
Packet#1388     LINK EVENT       plug      at Time 30.914200001 ms   after 2.752716667 ms
Packet#1389     HtoD    SETUP    ADDR=0    EP=0       CRC5=0x02        at Time 39.852950001 ms   after
8.938750000 ms
Packet#1390     HtoD    DATA0    length=8 CRC16=0xb5e9      at Time 39.853300001 ms   after 350.000
ns
                                 00 05 11 00 00 00 00 00            SET ADDRESS (DEVICE)
```

4.3.5 PHY UTMI Interface Checker

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

4.3.5.1 Description

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

This additional checking module can be easily plugged to the USB cable interface and it reports any link or data error on the USB Host or USB Device side.

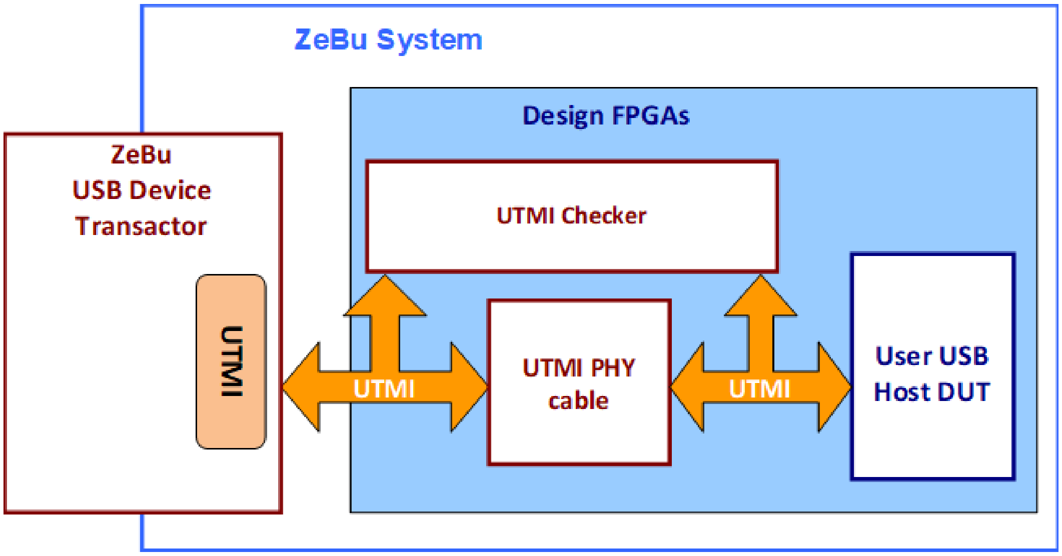


FIGURE 11. UTMI USB cable checker

4.3.5.2 Interface

TABLE 10 PHY serial cable interface

Symbol	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 bytes read from the PHY. Low/High bytes are asserted valid by utmi_rxvalid/ utmi_rxvalidh
hst_utmi_rxvalid	1	I	utmi_datain[7:0] contains valid data
hst_utmi_rxvalidh	1	I	utmi_datain[15:8] contains valid data
hst_utmi_rxactive	1	I	PHY is active
hst_utmi_data_out	16	I	Data transmitted to the PHY
hst_utmi_txvalid	1	I	utmi_dataout[7:0] contains valid data
hst_utmi_txvalidh	1	I	utmi_dataout[15:8] contains valid data
hst_utmi_opmode	2	I	PHY operating mode · 2'b00: normal operation · 2'b01: non-driving · 2'b10: disable bit stuffing and NRZI encoding
hst_utmi_termselect	1	I	Selects HS/FS termination · 1'b0: HS termination enabled · 1'b1: FS termination enabled
hst_utmi_word_if	1	I	Data bus width · 1'b0: 8-bit interface · 1'b1: 16-bit interface

TABLE 10 PHY serial cable interface

Symbol	Size	Type	Description
hst_link_error	1	O	Link error: · 1'b1: link error for host · 1'b0: no error
hst_data_error	32	O	Number of errors detected on USB Host side Synchronized on hst_utmi_clk
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 bytes read from the PHY. Low/High bytes asserted valid by utmi_rxvalid/utmi_rxvalidh
dev_utmi_rxvalid	1	I	utmi_datain[7:0] contains valid data
dev_utmi_rxvalidh	1	I	utmi_datain[15:8] contains valid data
dev_utmi_rxactive	1	I	PHY is active
dev_utmi_data_out	16	I	Data transmitted to the PHY
dev_utmi_txvalid	1	I	utmi_dataout[7:0] contains valid data
dev_utmi_txvalidh	1	I	utmi_dataout[15:8] contains valid data
dev_utmi_opmode	2	I	PHY operating mode · 2'b00: normal operation · 2'b01: non-driving · 2'b10: disable bit stuffing and NRZI encoding
dev_utmi_termselect	1	I	Selects HS/FS termination · 1'b0: HS termination enabled · 1'b1: FS termination enabled
dev_utmi_word_if	1	I	Data bus width · 1'b0: 8-bit interface · 1'b1: 16-bit interface

TABLE 10 PHY serial cable interface

Symbol	Size	Type	Description
dev_link_error	1	O	Link error: · 1'b1: link error for device · 1'b0: no error
dev_data_error	32	O	Number of errors detected on the device side. Synchronized on dev_utmi_clk

4.3.5.3 Advanced Debugging

The error detections of the PHY UTMI Interface Checker can be connected to the module for more efficient debugging.

Outputs of the checker module can be connected to an SRAM-trace driver, as shown in the DVE file example:

Note *SRAM-trace uses static-probes.*

```
SRAM_TRACE checker(
  .output_bin({
    host_data_error[31:0],
    host_link_error,
    device_data_error[31:0],
    device_link_error
}));
```

4.4 USB Bus Monitor

The USB Symbols and Packets monitoring feature allows to view/analyze USB low-level protocol transactions exchanged over the USB cable model between Host and Device.

It enables the monitoring of the USB Bus at UTMI level: you can dump 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

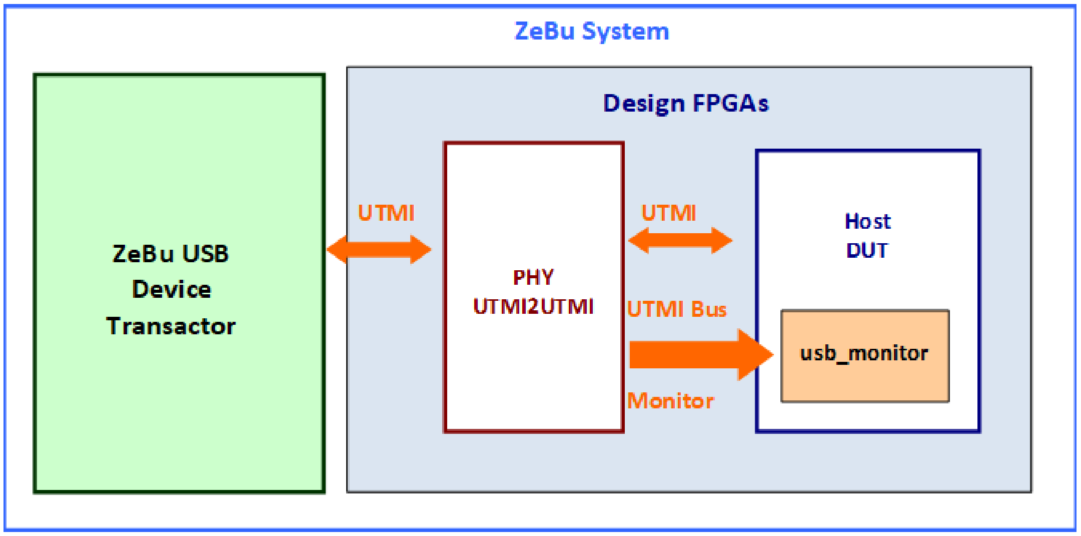


FIGURE 12. Verification environment for the USB Bus monitoring feature

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 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_DeviceToHost U_phy (
    .cable_resetn(cable_resetn),
    .device_utmi_clk(device_utmi_clk),
    …
    .mon_clk  (mon_clk),
    .mon_config  (mon_config[1:0]),
    .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_device(
    .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 monitor is controlled through the USB Device transactor software API (see Chapter 5 for details about the methods available in the API).

5 Software Interface

The USB Device transactor provides two APIs allowing two different abstraction layers for USB operations and transfer control.

5.1 USB Endpoint API

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 (splitting data into single operations, checking endpoint status, sending data again in case of NAK response, etc.).

All those methods are provided in the USB Device Endpoint API of the transactor, described in Chapter 6.

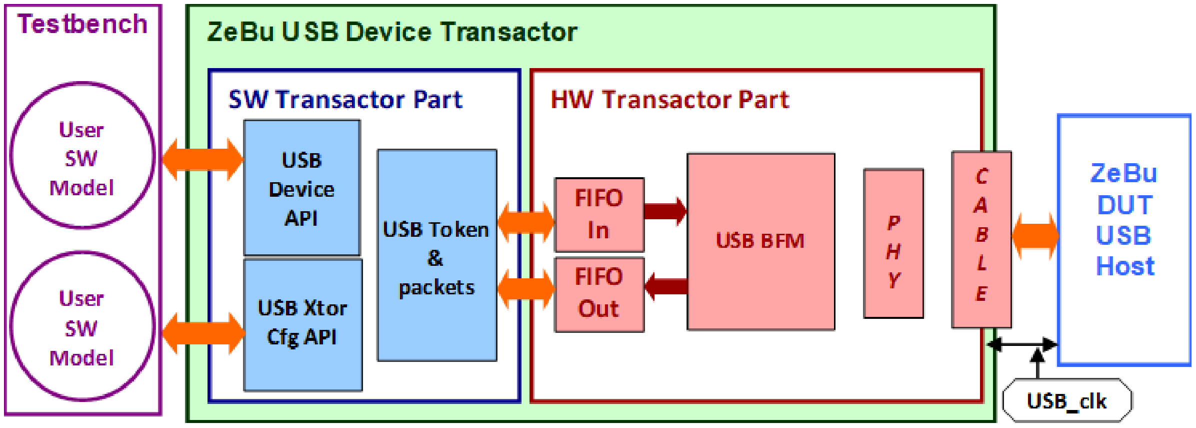


FIGURE 13. USB Device transactor Endpoint API

5.2 USB Request Block (URB) API

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

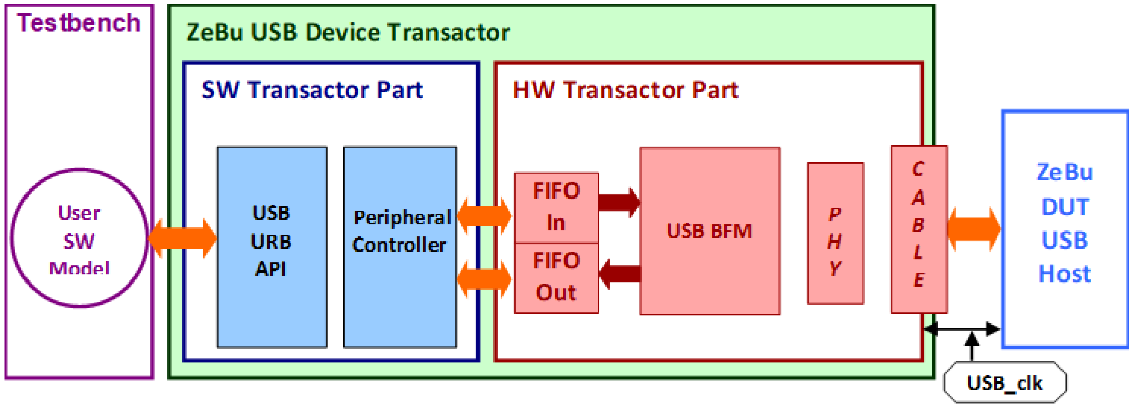


FIGURE 14. USB transactor URB API

6 USB Endpoint Software API

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

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

For complementary information on the USB Endpoint software API, please refer to the ***USB Device Transactor – USB API Reference Manual*** provided in the transactor package.

6.1 Using the USB Device Endpoint API

6.1.1 Libraries

The USB Device API is defined in `UsbDev.hh` and `UsbStruct.hh` header files.

The following libraries should be used:

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

6.1.2 Data Alignment

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

6.2 USB Device API Class Description

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

TABLE 11 USB Device API Class Description

Class	Description
UsbDevice	Represents the USB Device transactor. It manages directly USB status, USB packets and then requests to or from the DUT processed by the USB Device transactor.
DevEndpoint	Represents the data exchange over the USB Device transactor. It provides information on the current status of each USB Host channel.

6.3 UsbDevice Class

6.3.1 Description

UsbDevice class contains the methods to control the USB Device transactor and to send data packets to and get data from the USB link. It also manages the USB flow control mechanism.

The following tables give an overview of available types and methods for the USB API UsbDevice class.

TABLE 12 Types for UsbDevice Class

Type Name	Description
DevStatus_t	USB Device status
XferTyp_t	USB transfer type
CtrlReq_t	USB Request structure
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

TABLE 13 Methods for UsbDevice Class

Method Name	Description
Transactor Initialization and Control	
UsbDevice	Constructor
~UsbDevice	Destructor
init	ZeBu Board attachment

TABLE 13 Methods for UsbDevice Class

Method Name	Description
`config`	USB transactor configuration & initialization
`USBPlug`	Connects and attaches the USB Device connector to the USB cable
`USBUnplug`	Disconnects and detaches the USB Device connector from the USB cable
`loop`	Processes all the pending Tx/Rx USB packets, waits for BFM events or interrupts , and blocking methods
`delay`	Inserts a delay in milliseconds of USB clock
`registerCallBack`	Registers a service loop callback
`log`	Selects the logging mode
`setLogPrefix`	Sets a log prefix to be used in log information
`setAddress`	Sets the device address
`getVersion`	Returns either a string or a float corresponding to the transactor version
Endpoint Management	
`Endpoint`	Get specific endpoint handler (0 ≤ n ≤ 15)
Endpoint Operation	
`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 Operation	
`USBPlug`	Connects the USB Host to the cable
`USBUnplug`	Disconnects the USB Host from the cable

TABLE 13 Methods for UsbDevice Class

Method Name	Description
usbReset	Sends the USB Reset sequence to the USB Device
USB Cable Model Status	
isHostAttached	Returns true if a host is connected on the USB bus
USB Protocol Information	
portSpeed	Returns the port speed
maxPktSize	Returns the maximum packet size
General Purpose Vector Operations	
writeUserIO	Drives the general purpose bus_O vector
readUserIO	Reads the general purpose bus_I vector

6.3.2 Transactor Initialization and Control Methods

6.3.2.1 Constructor/Destructor Methods

Allocates/frees the software structures of the USB transactor.

```
UsbDevice (void)
~UsbDevice (void)
```

6.3.2.2 init() Method

Connects to the ZeBu board.

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

TABLE 14 Init Parameters

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

Note *When the Bus Monitoring (see Section 4.4) is used, 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.*

6.3.2.3 config() Method

Configures the USB Device transactor and allocates software structures accordingly.

```
uint32_t config (speed_conf_type_t spdSupport, utmi_type_t phyIf);
```

TABLE 15 Confi Method Parameters

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.
`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 needs to be called after an `init()` and before performing any operation on the USB Device 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.

6.3.2.4 `loop()` Method

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

```
DeviceStatus_t loop (void);
```

6.3.2.5 `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);
```

TABLE 16 delay Method Parameters

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.

6.3.2.6 registerCallBack() Method

Registers a callback to be called during service loop.

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

TABLE 17 registerCallBack Parameters

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

6.3.2.7 Log() Method

Configures the log mechanism.

```
void Log (logMask_t mask);
```

where mask is the log level as defined hereafter:

TABLE 18 Log level for Log method

Parameter Name	Description
logOff	No information.
logInfo	Information about USB host 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.

Please refer to Section 7.8.2 for further details on logs.

Example: Transactor Log File for a 512-byte Bulk IN Transfer (Device Side)

```
-- DEV -- Starting 512 bytes BULK IN transfer
Device DEBUG           : clk_48m cycle - 2253155
Device DEBUG           : Starting 512 bytes transfer :
Device DEBUG           : ep2-IN xfer_len=512 xfer_cnt=0
xfer_buff=0x80f4238 start_xfer_buff=0x80f4238
Device DEBUG           : clk_48m cycle - 2253264
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2253449
Device DEBUG           : clk_48m cycle - 2255909
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2256093
Device DEBUG           : clk_48m cycle - 2260357
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2260541
Device DEBUG           : clk_48m cycle - 2263001
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2263185
Device DEBUG           : clk_48m cycle - 2265728
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2265912
Device DEBUG           : clk_48m cycle - 2268454
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2268638
Device DEBUG           : clk_48m cycle - 2271180
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2271364
Device DEBUG           : clk_48m cycle - 2273906
Device DEBUG           :  * * Sending 64 bytes packet
Device DEBUG           : clk_48m cycle - 2274091
Device DEBUG           : clk_48m cycle - 2276550
Device DEBUG          : EP2 IN Xfer Complete
 -- DEV -- Handling BULK IN Endpoint event
 -- DEV -- Xfer completion on IN EndpPoint
```

6.3.2.8 `setLogPrefix()` Method

Sets a new log prefix for log information.

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

where `prefix` is the prefix to use in log information.

6.3.2.9 `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);
```

where `addr` is the device address.

6.3.2.10 `getVersion()` Method

Two different prototypes exist for this method:

- Returns a string corresponding to the transactor version.

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

- Returns a float corresponding to the transactor version.

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

where `version_num` is the transactor version number.

6.3.3 Endpoint Management Methods

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

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

where n is the endpoint number.

6.3.4 Endpoint Operation Methods

6.3.4.1 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);
```

TABLE 19 epEnable Parameters

Parameter Name	Parameter Type	Description
ep	DevEndpoint *	Device endpoint handler
typ	XferTyp_t	Transmission type
ep_is_in	Bool	Transmission direction
max_pkt_size	uint16_t	Maximum packet size

6.3.4.2 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);
```

TABLE 20 enableReception Parameters

Parameter Name	Parameter Type	Description
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.

6.3.4.3 `sendData()` Method

Starts sending data over a specified endpoint.

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

TABLE 21 sendData Parameters

Parameter Name	Parameter Type	Description
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

6.3.4.4 `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

6.3.4.5 `sendStatusPkt()` Method

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

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

TABLE 22 sendStatusPkt Parameters

Parameter Name	Parameter Type	Description
ep	DevEndpoint *	Device endpoint descriptor

6.3.4.6 `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);
```

TABLE 23 rcvSetup() Parameters

Parameter Name	Parameter Type	Description
ep	DevEndpoint *	Device endpoint descriptor
req	CtrlReq *	Request structure pointer

6.3.5 USB Operations

6.3.5.1 `USBPlug()` Method

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

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

TABLE 24 USBPlug() Parameters

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)

6.3.5.2 `USBUnplug()` Method

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

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

TABLE 25 USBUnplug Parameters

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

6.3.6 USB Cable Model Status Method

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

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

6.3.7 USB Protocol Information

6.3.7.1 `portSpeed()` Method

Returns the device speed.

```
HCSpeed_t portSpeed (void)
```

6.3.7.2 `maxPktSize()` Method

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

```
uint32_t maxPktSize (void)
```

6.3.8 General Purpose Vectors Operations

6.3.8.1 `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`.

6.3.8.2 `readUserIO()` Method

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

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

6.4 DevEndpoint Class

6.4.1 Description

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

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

TABLE 26 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 27 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

6.4.2 Detailed Methods

6.4.2.1 `getEpNumber()` Method

Returns the endpoint number.

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

6.4.2.2 `dataPresent()` Method

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

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

6.4.2.3 `xferComplete()` Method

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

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

6.4.2.4 `setupReceived()` Method

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

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

6.4.2.5 `setupDone()` Method

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

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

6.4.2.6 `isActive()` Method

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

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

6.4.2.7 `isDisabled()` Method

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

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

6.4.2.8 `display()` Method

Displays the endpoint status for debug purposes.

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

6.5 Example

This example uses the USB Device transactor as well as the PHY UTMI cable model.

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

6.5.1 Device Process of the Testbench

The Device process is as follows:

1. Initializes the core.
2. Plugs device
3. Waits for events.
4. When an event is detected:
   - On endpoint 0: performs 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.

6.5.2 USB Device Testbench Example

Here is an example of testbench using an USB Device transactor. This example is an extract of the `example/phy_utmi/src/bench/tb_dev.cc` file:

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


DevEndpoint* ctrlep;
DevEndpoint* bulkinep;
DevEndpoint* bulkoutep;
DevStatus_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.d32 != 0 || (!bulkin_started && dataQueueLength > 
0)) {
          if (stat.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.d32 & (0x1<<bulkin_epnum)) {
            if(verbose)
              printf(" -- DEV -- Handling BULK IN Endpoint event
\n");
            handle_bulkinep(&usbDev, bulkinep);
          }
          if (stat.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 Device 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/phy_cable/src/bench` directory, also named `tb_dev.cc`.

7 URB Software API

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

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

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

For complementary information on the URB software API, please refer to the ***USB Device Transactor - URB API Reference Manual*** provided in the transactor package.

7.1 Using the URB API

7.1.1 Libraries

The DeviceEP class and the ZebuRequest structure used in the USB Device transactor are equivalent to the Linux usb_ep and usb_request structures.

The URB API is defined in the UsbUrbDev.hh and UsbUrbCommon.hh header files. The available library files are:

- libUsbUrb.so64-bit gcc 3.4 library
- libUsbUrb_6.so32-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.so64-bit gcc 3.4 library
- libUsbUL_6.so32-bit gcc 3.4 library

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

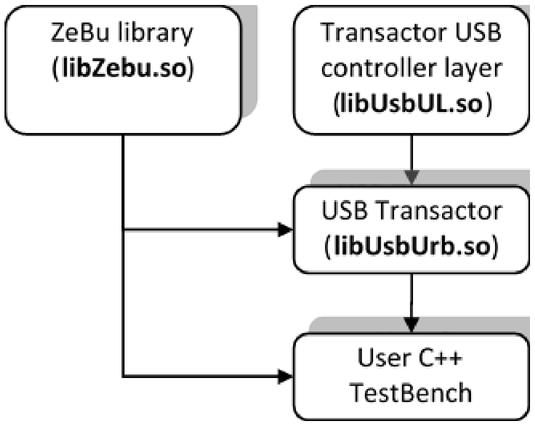


FIGURE 15. Libraries hierarchy overview

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

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

TABLE 28 18: URB API Class description

Class	Description
UsbDev	Represents the USB Device transactor
DeviceEP	Represents an Endpoint of the device

TABLE 29 URB API Structure description

Structure	Description
ZebuRequest	Represents the request of a device endpoint

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

TABLE 30 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

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

TABLE 31 USB Device 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 UsbDev Class

7.3.1 Description

The methods associated with the `UsbDev` 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 `UsbDev` class is defined in the `UsbUrbDev.hh` file.

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

TABLE 32 Methods for UsbDev Class

Method Name	Description
Transactor Initialization and Control	
`UsbDev`	Constructor
`~UsbDev`	Destructor
`Init`	ZeBu Board attachment
`InitBFM`	USB transactor configuration & initialization
`USBPlug`	Connects and attaches USB Device connector to the USB cable
`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
`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
`getVersion`	Returns either a string or a float corresponding to the transactor version
USB Device Setup	

TABLE 32 Methods for UsbDev Class

Method Name	Description
`RegisterControlSetupCB`	Register callback called when a control setup access comes from Host
`GetEndpoint`	Returns an endpoint of the connected device
General Purpose Vector Operations	
`writeUserIO`	Drives the general purpose bus_O vector with the specified value
`readUserIO`	Reads the general purpose bus_I vector

7.3.2 Transactor Initialization and Control Methods

7.3.2.1 Constructor/Destructor Methods

Allocates/frees the structures for the USB Device transactor.

```
UsbDev (void)
~UsbDev (void)
```

7.3.2.2 `Init()` Method

Connects to the ZeBu board.

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

Parameter Name	Parameter Type	Description
`zebu`	`Board *`	Pointer to 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 *NoteWhen Bus Monitoring is used (see Section 4.4), the last argument (clkName) must be specified and must have the same name as the transactor clock. In the examples shown in Section 4.2.3, 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 (see Section 7.6.1).

7.3.2.4 `USBPlug()` Method

Connects the device 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 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 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 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 `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.

7.3.2.8 `RegisterCallback()` Method

Registers a callback to be called during service loop.

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

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

7.3.2.9 `SetDebugLevel()` Method

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

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

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

TABLE 33 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.

Please refer to Section 7.8 for further details on the available levels.

7.3.2.10 `SetLogPrefix()` Method

Sets a new log prefix for log.

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

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

7.3.2.11 `SetLog()` Method

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.

7.3.2.12 `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 Setup Methods

7.3.3.1 `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

7.3.3.2 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.

7.3.4 General Purpose Vectors Operations

7.3.4.1 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.

7.3.4.2 readUserIO() Method

Reads the 4-bit general purpose bus_I vector.

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

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 34 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 (see Section 7.6.1).

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 (see Section 7.6.1).

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 (see Section 7.6.1).

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 (see Section 7.6.1).

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 (see Section 7.6.1).

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_tzero;
uint8_tshort_not_ok;
int32_tstatus;
uint32_tactual;
zusb_ReqCompletecomplete;
void **context;
};
```

TABLE 35 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.

TABLE 35 Content of the ZebuRequest Structure

Parameter	Description
`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)
`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 URB API Enum Definitions

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

7.6.1 `zusb_status` enum

Statuses returned by the API methods.

TABLE 36 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

7.6.2 `zusb_transfer_status` enum

Statuses of transfers using `ZebuRequest`.

TABLE 37 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
ZUSB_TRANSFER_CANCELLED	ZeBu USB transfer is cancelled

TABLE 37 zusb_transfer_status Enum Description

Parameter Name	Description
`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

7.7 Watchdogs and Timeout Detection

7.7.1 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 38 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.

7.7.1.1 `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.

7.7.1.2 `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.

7.7.1.3 `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 described in Section 7.7.2 hereafter.

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

7.7.2 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.

7.8 Logging USB Transfers Processed by the Transactor

7.8.1 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.

7.8.2 Log Types

7.8.2.1 Main Info Log

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

Use the `logInfo` option to activate it.

7.8.2.2 USB Request Logs

This log reports URB activity.

Use the `logUrb` option to activate it.

7.8.2.3 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.

7.8.2.4 Data Buffer Log

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

Use the `logBuf` option to activate it.

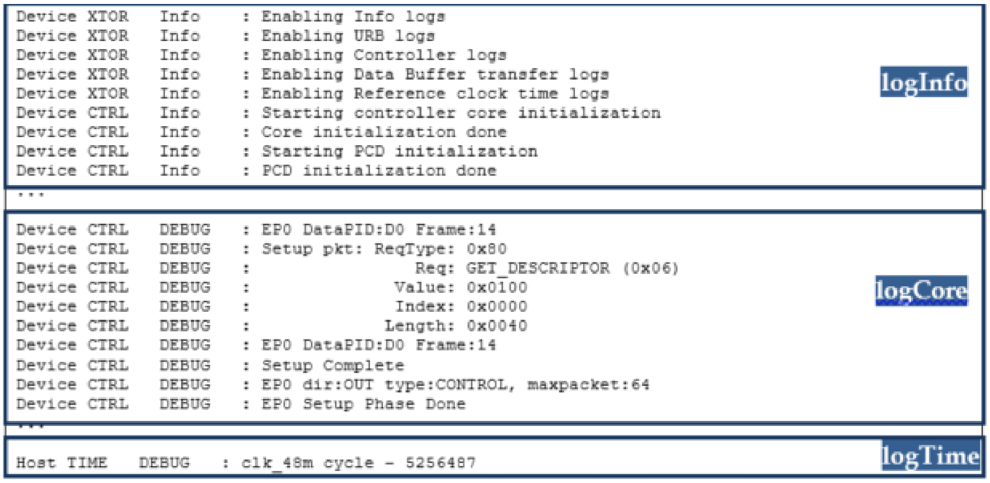
7.8.2.5 Reference Clock Time Log

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

Use the `logTime` option to activate it.

7.8.3 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
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
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
```

7.9 Using the USB Device Transactor

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

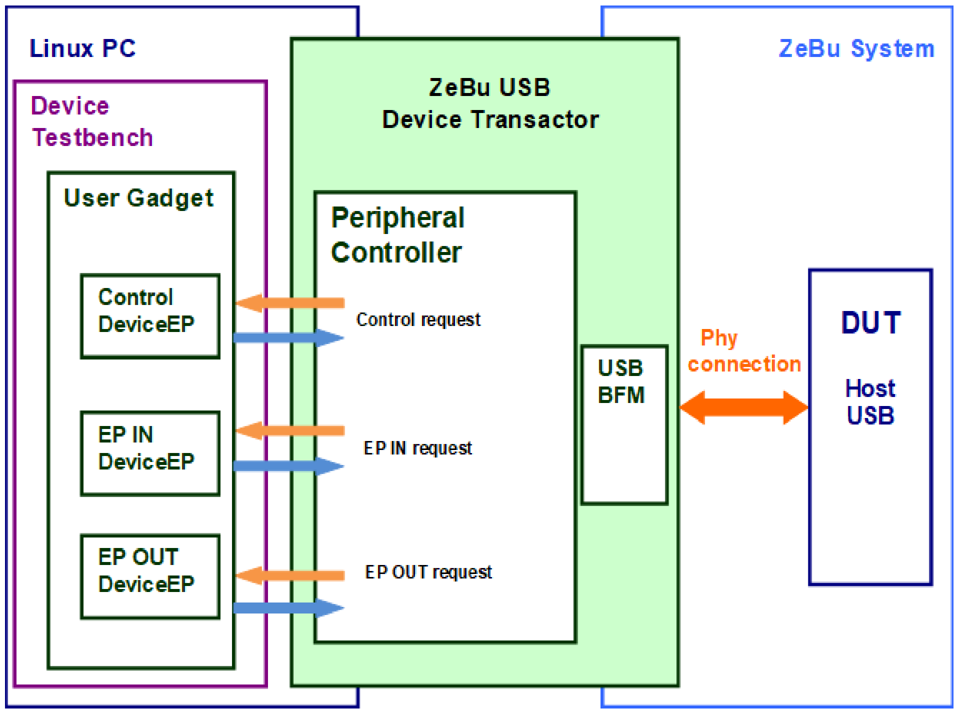


FIGURE 16. Device Transactor Overview

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

7.9.1 Device Transactor Control Overview

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

7.9.1.1 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.

7.9.1.2 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.

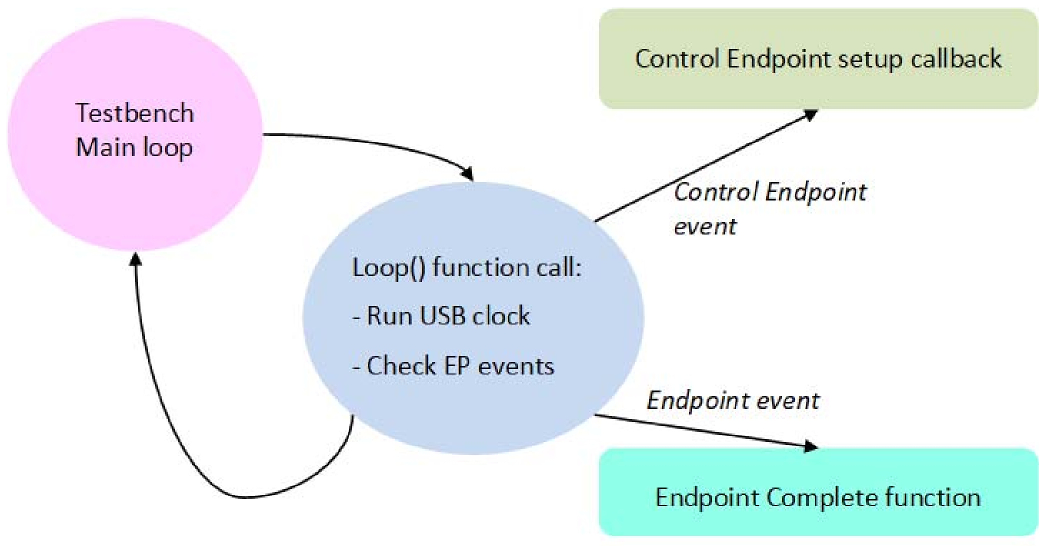


FIGURE 17. Main testbench loop transfer flow

7.9.2 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
7. 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.

7.9.3 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.

7.9.4 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

7.9.5 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.

7.9.5.1 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;
```

7.9.5.2 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).

7.9.6 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.

7.9.6.1 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);
  }
```

7.9.6.2 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.

7.9.6.3 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);
```

7.9.6.4 Cancelling a Transfer

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

7.9.6.5 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);
  }
}
```

7.9.6.6 `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);
  }
}
```

7.9.6.7 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.

7.9.7 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 7.9.6 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.

8 USB Bus Monitoring Feature

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

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

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

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

8.1 Prerequisites

8.1.1 USB Bus Monitor Hardware Instantiation

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

8.1.2 USB Device Transactor Initialization Constraint

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

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

8.2 Using the USB Monitoring Feature with USB Channel API

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

TABLE 39 Methods for Bus Monitoring using Channel API

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

8.2.1 Starting and Stopping Log

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

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

8.2.2 Defining the Monitor File Name

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

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

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

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

8.2.3.1 `setBusMonFileName()` Method

Sets the log file name.

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

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

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

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

8.2.3.2 startBusMonitor() Method

Opens a log file and starts activity logging.

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

where type activates the monitor:

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

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

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

8.2.3.3 stopBusMonitor() Method

Stops bus activity logging and closes the current binary file.

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

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

8.3 Using the USB Monitoring Feature with URB API

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

TABLE 40 Bus Monitoring methods using URB API

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

8.3.1 Starting and Stopping Logging

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

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

8.3.2 Defining the Log File Name

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

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

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

8.3.3 Description of the URB API Interface for USB Bus Logging

8.3.3.1 SetBusMonFileName() Method

Sets the log file name.

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

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

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

8.3.3.2 `StartBusMonitor()` Method

Opens a log file and starts activity logging.

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

where type activates the log:

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

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

8.3.3.3 `StopBusMonitor()` Method

Stops bus activity logging and closes the current binary file.

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

This method returns 0 on success, -1 otherwise.

9 URB Monitoring Feature

9.1 Description

The USB Device transactor includes a URB monitor mechanism which allows dumping of USB bus activity at a URB level of abstraction.

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

TABLE 41 ZeBu URB Monitoring API Methods

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

9.2 Prerequisite

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

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

9.3 Using the URB Monitoring Feature

9.3.1 Starting and Stopping Monitoring

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

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

9.3.2 Defining the Monitor File Name

The URB Monitoring API allows defining a filename for the monitor file generated by the transactor.

The `SetUrbMonFileName()` method sets a file name and controls filename indexing. If the `SetUrbMonFileName()` method is not used, the transactor uses the transactor instance name (defined in the DVE file) as file name each time the `StartUrbMonitor` method is called.

The monitor file generated is a binary file (`.umn`) which can be read with the ZeBu USB Viewer (please refer to the ***ZeBu USB Viewer User Manual***).

9.3.3 Description of the URB API Interface for URB Monitoring Feature

9.3.3.1 `SetUrbMonFileName()` Method

Sets the monitor file name.

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

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

This method returns 0 on success, -1 otherwise.

9.3.3.2 StartUrbMonitor() Method

Opens a new monitor file and starts URB monitoring.

```
int32_t StartUrbMonitor ();
```

This method returns 0 upon success, -1 otherwise.

9.3.3.3 StopUrbMonitor() Method

Stops URB monitoring and closes the current monitor file.

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

This method returns 0 upon success, -1 otherwise.

10 Tutorial

Two examples are provided in the example directory of the transactor package: `phy_cable` and `phy_utmi`. They are detailed hereafter.

10.1 phy_utmi Example

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

10.1.1 File Tree

The following files are provided for this example:

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

10.1.2 Overview

The phy_utmi example consists of two testbench processes representing respectively a USB Host and a USB Device connected by a cable:

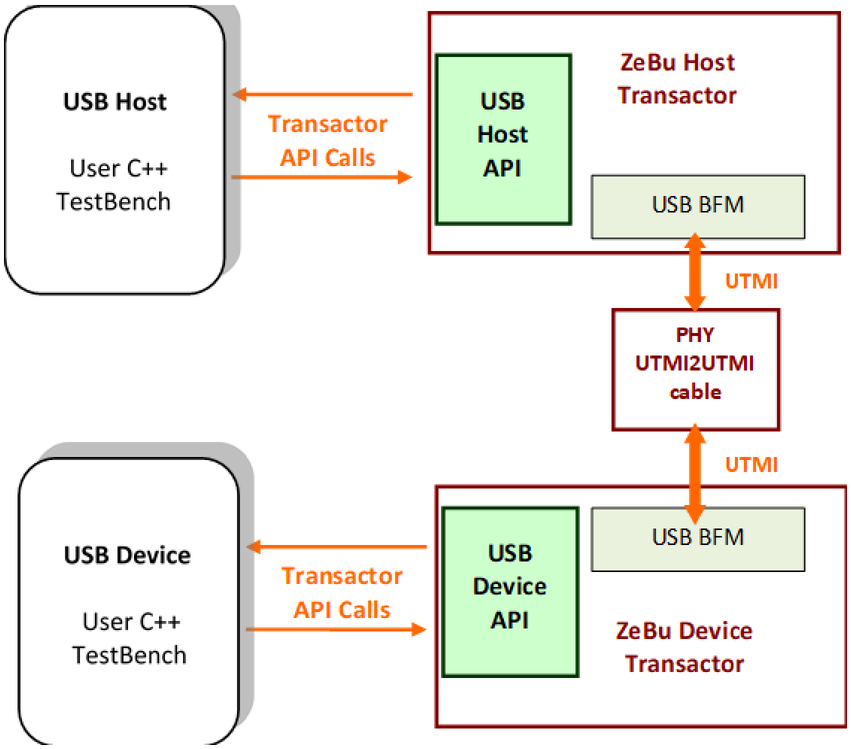


FIGURE 18. phy_utmi example overview

10.1.3 ZeBu Design

The ZeBu design consists of:

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

10.1.4 Software Testbench

This example can be run using either the Channel API or the URB 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 host starts reading an input file (`test.in`) and sends data to the device using `Bulk OUT` transfers.
2. When all the data has been transferred, the Host starts reading back the data from the Device using
3. `Bulk IN` transfers, and puts the data in an output file (`test.out`).
4. The files are then compared to check that all worked fine.

10.2 phy_cable Example

10.2.1 File Tree

The following files are provided for this example:

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

10.2.2 Overview

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

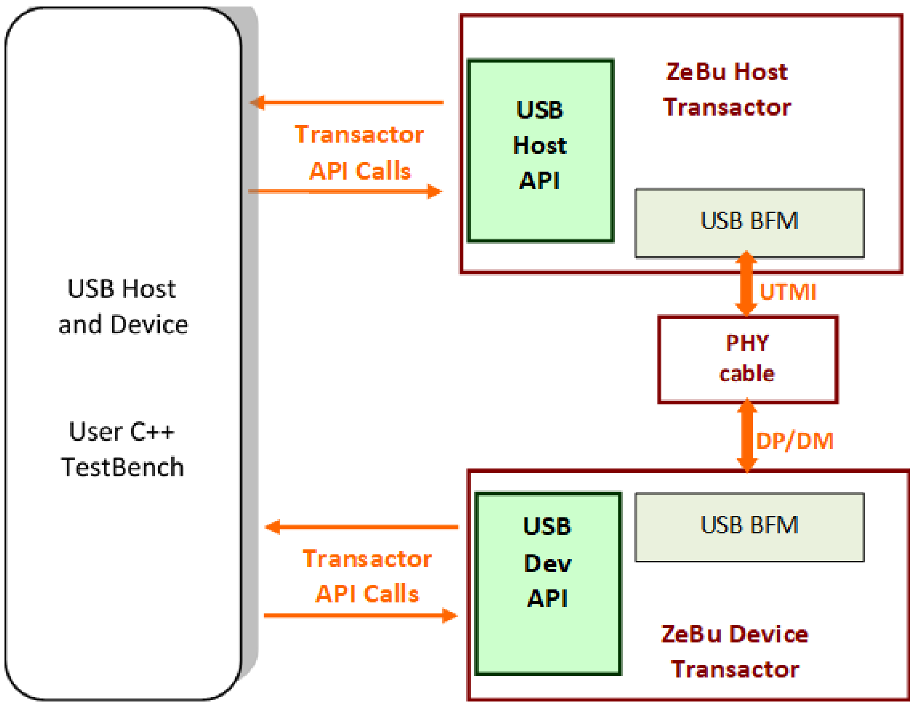


FIGURE 19. phy_cable example overview

10.2.3 ZeBu Design

The ZeBu design consists of:

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

10.2.4 Software Testbench

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

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

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

10.3 Running the Examples

10.3.1 Setting the ZeBu Environment

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

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

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

10.3.2 Compiling the Designs

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

```
@> make compil
```

10.3.3 Running the phy_utmi Example

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

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

10.3.4 Running phy_cable Example

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

```
@> make run
```