

# USB Device IP Core User Guide

Revised May 18, 2020; Author Tudor Gherman

!!\*Under development\*!!

Please note that this version of the IP core is not a final one, but still currently under development. This notice will be removed once the IP core is in a stable/final state.

## 1 Introduction

This user guide describes the Digilent USB Device Controller Intellectual Property. This IP is designed to provide communication between an AXI Microblaze system and a USB 2.0 Host.

## 2 Features

- Capable to operate at 480Mb/s (USB High Speed)
- ULPI Interface
- tested with 2 endpoints
- Integrated DMA engine

## 3 Overview

The design is divided in two clock domains. The logic on the ULPI clock side is responsible with the UPLI bus decoding, speed negotiation and packet decoding. The logic on the AXI side is responsible for transferring data between the transmit buffers, receive buffers, context and the main memory.

IP quick facts	
Supported device families	Zynq®-7000, 7 series
Supported user interfaces	Xilinx®: AXI4, AXI4 Lite, ULPI
Provided with core	
Design files	VHDL
Simulation model	-
Constraints file	XDC
Software driver	N/A
Tested design flows	
Design entry	Vivado™ Design Suite 2015.4
Synthesis	Vivado Synthesis 2015.4

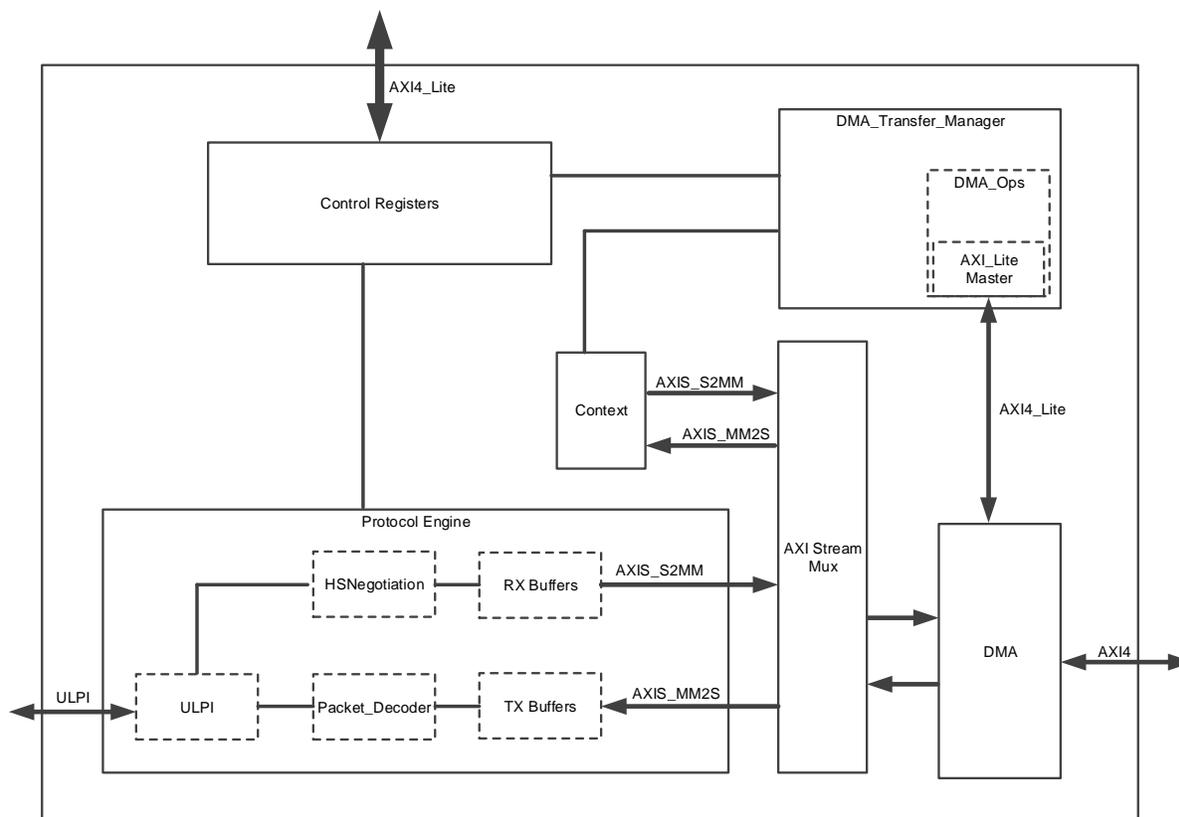


Figure 1. IP top-level diagram.

### 3.1 Protocol Engine

This block is responsible with managing communication on the ULPI bus, with speed negotiation and with packet encoding/decoding. Transmit packet data is passed to the Protocol Engine block through the TX Buffers, while received packets are placed in the RX Buffers.

#### 3.1.1 ULPI

The ULPI block decodes the data received over the ULPI bus and implements the required frameworks for NOPID, Transmit packet (PID), Register write (REGW) and register read (REGR) commands. According to the ULPI specifications, the USB PHY can abort packets that are being transmitted. This requirement is not implemented yet.

#### 3.1.2 High Speed Negotiation

This block implements the logic required for carrying out the High Speed Negotiation, Reset, Suspend and Resume as described by Chapter 7 of the USB Specifications. Suspend and Resume have not been tested yet.

### 3.1.3 Protocol Engine

This block implements Chapter 8 (Protocol Layer) of the USB specifications. The implemented and tested state machines are Dev\_Do\_BCINTI, Dev\_HS\_BCO. Dev\_HS\_Ping, Dev\_Do\_BCINTO are not tested. Isochronous transfers are not supported. Handshake responses are generated automatically.

### 3.1.4 Transmit Buffers

Data is passed from the DMA engine to the Protocol Engine in two stages. Packets are first stored into an AXI Stream interface FIFO. The second stage is implemented into a dual port BRAM block. Each endpoint has 1024 bytes reserved and the internal logic is responsible with distributing data from the TX FIFO to the correct address in BRAM. The transmit buffers are implemented in the Transmit\_Path module.

### 3.1.5 Receive Buffers

The received packet bytes are stored as they arrive in an input buffer. If the packet is received without errors it is passed to an AXI Stream interface FIFO. If errors are detected, the packet is discarded. There is no need for individual buffers for each endpoint since the DMA engine will distribute the packets to the endpoint buffers allocated in system memory. The receive buffer is placed in the top module.

## 3.2 Context

There are two data structures that provide the information required to transfer data between the ULPI interface and the system memory. These two structures are the Queue Heads (dQH) and the Transfer Descriptors (dTd). The software is responsible for creating both Queue Heads and Transfer Descriptors in system memory and the device controller will fetch a “copy” of each. For more information see 5.3 Device Data Structures.

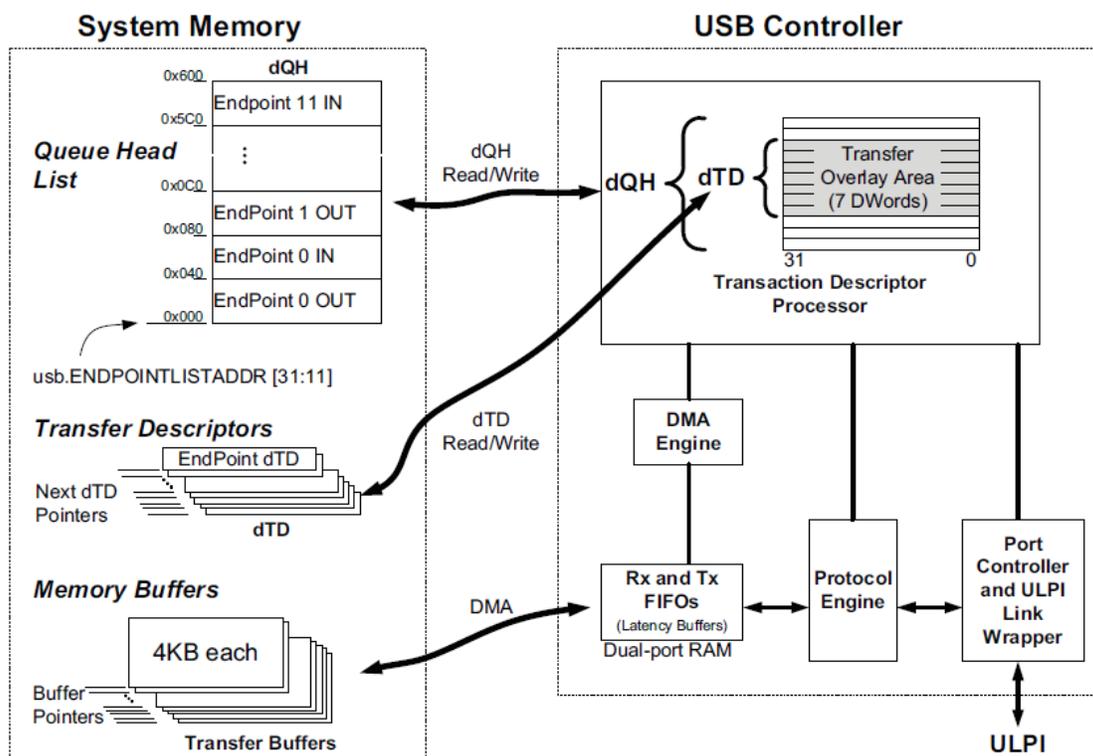


Figure 2. Device Descriptors.

### 3.3 DMA Transfer Manager

The DMA Transfer Manager block holds the state machines that control the DMA engine. There are four “frameworks” implemented in this block: one responsible for priming endpoints, one for control transfers, one for input packets and one for output packets. Both control information (Context memory) and packet data need to be transferred for each sequence. For a more detailed description refer to section 5.4 Device Frameworks

### 3.4 Control Registers

All control registers are implemented in this block. The processor can read and write registers through an AXI Lite interface.

## 4 Interface Descriptions

The interfaces of the Device Controller Core are listed and described in Table 1.

Interface Name	Interface	Signal Type	Init State	Description
AXI4 Lite	Slave	-	N/A	The AXI4 Lite interface is used to control register access.
AXI4	Master	-	N/A	The AXI4 master interface is used by the DMA engine to transfer data between the device controller and the system memory.
ULPI	Slave	-	N/A	See UTMI+ Low Pin Interface (ULPI) Specification, Revision 1.1, October 20th, 2004

Table 1. Port descriptions.

## 5 Hardware Description

### 5.1 Interrupts

The Status register (USBSTS) and the Interrupt Enable register (USBINTR) are responsible for triggering interrupts. There are 6 interrupt conditions currently supported by the controller:

1. UI: Set when a transfer is completed. (Bit 0 in USBSTS)
2. NAKI: Set when a device has generated a NAK. (Bit 16 in USBSTS)
3. SLI: Set when the device enters suspend state. (Bit 8 in USBSTS)
4. SRI: Set when a Start of Frame (SOF) packet is received. (Bit 7 in USBSTS)
5. URI: Set when Reset is detected. (Bit 6 in USBSTS)
6. PCI: Port Change Detect. (Bit 2 in USBSTS)

### 5.2 Endpoint Registers

**ENDPTSETUPSTAT:** Bits[11:0] are set when the corresponding endpoint receives a setup packet. The access type for this register is write-one-to-clear.

**ENDPTPRIME:** Bits [27:16] are relevant for IN endpoints while bits [11:0] are relevant for OUT endpoints. Software sets a bit to instruct the controller to fetch the corresponding queue head, transfer descriptor and packet data from the main memory. The access type for this register is write-one-to-set. Hardware automatically clears the corresponding bits when the prime operation is complete.

**ENDPTFLUSH:** Bits [27:16] are relevant for IN endpoints while bits [11:0] are relevant for OUT endpoints. Software sets a bit to instruct the hardware to flush the corresponding endpoint. The access type for this

register is write-one-to-set. Hardware automatically clears the corresponding bits when the flush operation is complete.

**ENDPTSTAT:** Bits [27:16] are relevant for IN endpoints while bits [11:0] are relevant for OUT endpoints. Hardware sets a bit when the corresponding endpoint has been primed. The access type for this register is read only.

**ENDPTCOMPLETE:** Bits [27:16] are relevant for IN endpoints while bits [11:0] are relevant for OUT endpoints. Hardware sets a bit to indicate that the corresponding endpoint has completed the transfer that was primed.

**ENDPTCTRL{11:0}:** Bits [19:18] are relevant for IN endpoints while bits [3:2] are relevant for OUT endpoints. This register is used to select the endpoint type. Isochronous endpoints are not supported. The access type for this register is read-write.

### 5.3 Device Data Structures

The processor does not directly instruct the DMA engine what data and where to/from to transfer it. Instead, it allocates space in system memory in order to define data structures (Queue Heads and Transfer Descriptors) that describe each individual transfer. Each endpoint has one corresponding Queue Head structure (Fig) and a linked list of Transfer Descriptors. The Device controller first fetches the Queue Head. Afterwards, based on the information specified in the Queue Head, the device controller fetches the first transfer descriptor and copies it into the Context memory in the Overlay Area of the corresponding endpoint.

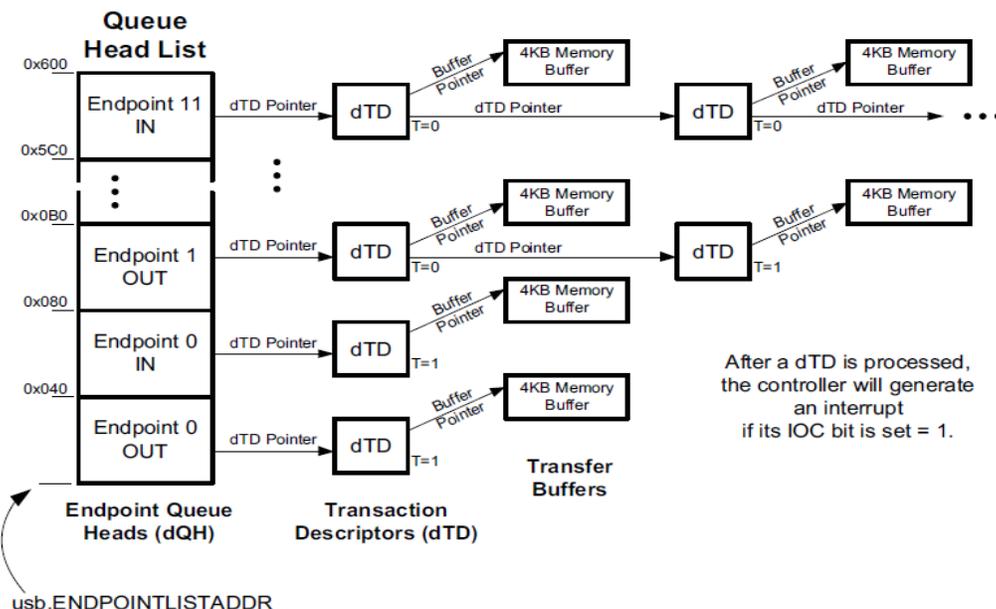


Figure 1. Device Link List example.

### 5.3.1 Queue Heads (dQH)

The **Maximum Packet Length** field stores the maximum packet length for each individual endpoint.

The **Setup Buffer Bytes 3...0** and **Setup Buffer Bytes 7...4** store setup packets received. Setup packets are always 8 byte long and reading this fields in the Queue head is the only mean by which software can access setup data.

Software is also responsible to write the **Next dTD Pointer** field with the address of the first Transfer Descriptor.

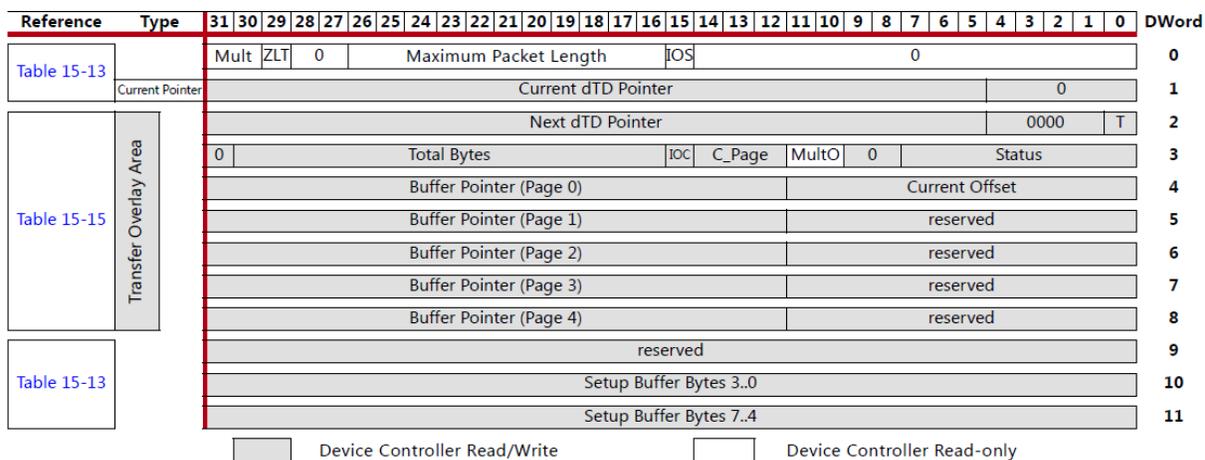


Figure 4. Device Queue Head.

### 5.3.2 Transfer Descriptors (dQH)

Transfer Descriptors hold information that effectively describe the DMA transfers that need to take place in order to move packets between the system memory and TX/RX Buffers.

The **Total Bytes** field specifies the number of bytes that the device controller is supposed to transmit or receive in order to consider a transfer completed.

The **Status** field is written back by the device controller once a transfer is completed. Only bit 7 is currently used, which represents the active status.

The **Buffer Pointer** (The device controller currently uses only **Page 0**) field and the **Current Offset** field are used to determine the memory address where packet data is stored.

The **Terminate Transfer (T)** bit is set by software if the dTD is the last in the linked list. If T is cleared, the device controller will fetch the next dTD using the information in the **Next dTD Pointer** field to determine its address.

The **Next dTD Pointer** field specifies the address of the following dTD in the linked list.

Reference	Type	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	DWord
Table 15-15	Transfer Overlay Area	Next dTD Pointer																												0000	T	0		
		0	Total Bytes										IOC	C_Page	MultO	0	Status										1							
		Buffer Pointer (Page 0)										Current Offset										2												
		Buffer Pointer (Page 1)										R	Frame Number										3											
		Buffer Pointer (Page 2)										reserved										4												
		Buffer Pointer (Page 3)										reserved										5												
		Buffer Pointer (Page 4)										reserved										6												

Device Controller Read/Write       Device Controller Read-only

Figure 5. Device Transfer Descriptor

## 5.4 Device Frameworks

### 5.4.1 Priming

Because of the strict timing requirements on the USB bus, data is loaded in the transmit buffers in advance. This procedure is called priming and it is different between transmit endpoints and receive endpoints.

#### 5.4.1.1 Transmit Endpoints

1. The device controller fetches the queue head (dQH) from main memory based on the address written by software in ENPOINTLISTADDR register.
2. The device controller fetches the transfer descriptor from system memory from the address specified in the Next dTD Pointer field of the previously transferred dQH.
3. The device controller fetches packet data from main memory and fills the transmit buffer based on the address specified in the Buffer Pointer and Current Offset fields of the previously transmitted dTD.
4. The device controller updates its “copy” of the Next dTD Pointer field so that it points to the remaining packet data (if any). When an IN packet will be sent by the host data will be fetched from that address.
5. The device controller will update the ENPTSTAT and ENDPTPRIME registers.

#### 5.4.1.2 Receive Endpoints

1. The device controller fetches the queue head (dQH) from main memory based on the address written by software in ENPOINTLISTADDR register.
2. The device controller fetches the transfer descriptor from main memory from the address specified in the Next dTD Pointer field of the previously transferred dQH.

3. When an OUT packet will be sent by the host, data from the receive buffer will be transferred by the device controller into main memory at the address specified in the Buffer Pointer and Current Offset fields.
4. The device controller will update the ENPTSTAT and ENDPTPRIME registers

## 5.4.2 Setup Packet Framework

Unless the SLOM bit in USBMODE register is set, the device controller copies the setup packet bytes into the corresponding dQH in system memory through a DMA transfer. If the interrupts are enabled and the UI (Bit 0 in USBINTR register) condition is not masked the core will trigger an interrupt.

## 5.4.3 IN Packet Framework

This sequence of events is triggered when the device successfully responds with data to an IN token packet.

1. The device controller checks if there are any bytes left to transmit for the current dTD.
2. If all bytes specified by the dTD have been transmitted, the dTD is copied back in the system memory with its **Status** field updated. Otherwise, the state machine jumps to step 4.
3. If the **Terminate** bit is set, the controller considers the transfer specified by the processed dTD completed and takes no further action. If **Terminate** bit is cleared, the next dTD in the linked list is fetched from main memory from the address specified by the **Next dTD Pointer** field.
4. If there are bytes left to transmit for the current dTD, the device controller computes the address in the data buffer where the remaining data resides and triggers a new DMA transfer. When the data has been transferred to the TX Buffer, the controller will be ready to respond to a new IN token sent by the host.

# 6 Mouse Emulation

The functionality of the core is demonstrated by emulating a mouse on the Genesys2 board. The pointer can be moved with the push-buttons on the board.

The enumeration process on the device side is carried out by endpoint0. After the speed negotiation is done, the host will request several descriptors from the device. These requests are encoded in setup packets. The sequence of events is as follows:

1. The host sends a Token packet with its PID indicating a SETUP token.
2. The host sends an OUT packet with the 8 bytes that make up the setup information.
3. The device controller will copy the setup packet in main memory, more specifically in the corresponding dQH in the Setup Buffer Bytes fields and triggers an interrupt.
4. The device driver is responsible with preparing the requested response and priming endpoint0.

5. Meanwhile, the device controller will issue NAK handshake packets in response to IN packets received from the host. Once the endpoint is primed, the device controller will issue a DATA1 packet in response to the IN sent by the host.
6. If an ACK handshake token is received from the host, the device controller will set the corresponding bit in ENPTCOMPLETE register.
7. The device controller driver is responsible to check if a zero length packet (the status phase of the control transfer) was received from the host.

This sequence will repeat for all the following control transfers initiated by the host. The descriptors required are: Device Descriptor, Configuration Descriptor, Device Qualifier Descriptor, String Descriptor0, String Descriptor1, String Descriptor2, String Descriptor3, HID Report Descriptor. Once the host gathers all this information will start sending IN tokens on endpoint1. The length and format of the information coded in the HID reports is specified in the HID Report Descriptor.

## 7 Limitations

1. ULPI: Extended Register Write (EXTW) and Extended Register Read are not verified (not needed for the current application). Packet abort is not supported yet.
2. HS\_Negotiation (Chapter7): Suspend and Resume are not tested. Remote Wakeup not implemented. Full Speed and Low Speed not tested.
3. Packet\_Decoder(Chapter8): Dev\_HS\_Ping, Dev\_Do\_BCINTO are not tested. Isochronous transfers are not supported. OUT transactions are not working, need to be fixed. OUT packets are not used with input devices by the HID protocol.
4. Device Data Structures (Context): **IOC** is ignored. Interrupts are generated when a dTD is complete regardless of IOC value. **Status** field of the dTD is not checked by software and is not correctly written back by hardware. **MultO** is ignored (it should only be used with isochronous transfers which are not supported yet). **Mult** (used only with isochronous packets) field ignored. **ZLT** field ignored. Zero Length transfers are dictated by software (dTDs with Total Bytes field set to 0) only.
5. DMA\_Transfer\_Manager: **Active Status** issue. The ZYNQ USB core seems to periodically read dTDs from main memory and, if the software modifies the active bit after the dTD has been copied in the Device Controller memory, the Device Controller will still execute the transfer described by the dTD. This issue was noticed with circular buffers. In the current implementation, if the dTD is inactive, the USB Device controller will stall. This bit is only used to indicate that the Device controller has released a buffer.
6. Control Registers: Only registers mentioned in section 5 have been tested. Not all ZYNQ USB core registers have been implemented.

## 8 References

The following documents provide additional information on the subjects discussed:

1. Xilinx Inc., UG585: Zynq-7000 AP SoC Technical Reference Manual, v1.10, February 23, 2015.
2. UTMI+ Low Pin Interface (ULPI) Specification, Revision 1.1, October 20th, 2004.

3. Universal Serial Bus Specification Revision 2.0, April 27, 2000