Bar-Ilan University
School of Engineering
VLSI Lab

USB2.0 Protocol Engine

Razi Hershenhoren
razi136@gmail.com

Omer Reznik
omer.big@gmail.com

Final Project – Forth Year
Computer Engineering

Team Instructor: Mr. Moshe Doron
Academic Supervisor: Prof. Wimer

October 2010

Table of Contents

Table of Contents **2**
Project Summary **5**
Project's Motivation **6**
Technical Background **7**
Introduction 7
Device's Endpoints 9
Descriptor Tables 10
Token Packets 12
Data Packets 13
Handshake Packets 13
Setup Packet 14
Enumeration Process 16
USB Communications Flow 16
Transaction Types 18
Data Toggle Synchronization 19
Protocol Engine Algorithms **20**
Development tools **22**
Modelsim 22
Quartus 22
Simvision 22
Altera DE3 FPGA Board 23
System Level Introduction **24**
The Protocol Engine 26
UTMI Interface Signals 26
DMA Interface Signals 28
Device Description **29**
Top Level Blocks 29
Enumeration FSM 30
Configured FSM ..... **Error! Bookmark not defined.**

Receive Data Packet 37

Send Data to DMA 38
Receive Data from DMA 39
SRAM 40
EP buffer OUT 41
EP Buffer IN 42
CRC5 43
CRC16 43
Detailed Functionality Description 44
OUT transaction 44
IN Transaction 47
Simulation 50
OUT transaction 50
IN transaction 51
Verification 53
Hurdles and Obstacles 54
High-Z pins 54
FPGA chip 54
USB Checker 55
Alternative Solution 56
Conclusion and Summary 57
A Look into the Future 59
ASIC implementation: 59
USB 3.0 59
Protocol Engine with DMA Controller 59
Bibliography 60
Appendix 62
Standard Device Descriptor Table 62
Configuration Descriptor Table 63
Interface Descriptor Table ..... **Error! Bookmark not defined.**
Endpoint Descriptor Table 63
PID Codes Table 65
Altera DE3 Stratix III FPGA board Top view 66

Pin mapping table for verification and demonstration 67
Schematic diagram of the 7-segment displays 68
Hexadecimal to 7-seg translation table 69
Bulk Interrupt IN Transfer state machine 70
Isochronous IN Transfer state machine 71
Bulk OUT Transfer state machine 72
Interrupt OUT Transfer state machine 73
Isochronous OUT Transfer state machine 74
Device Enumeration state machine 75

Project Summary

The Protocol Engine is a High Speed USB2.0 Communication Core, used for development and production of USB Classic and Vendor Specific Devices by processing USB2.0 packet-level Link-Layer Protocol tasks in hardware, thus offloading communication tasks from the µController.

The protocol engine performs, CRC check and generation, packet identifier decoding and verification, address recognition and handshake evaluation and response.

Acting on a received token and analyzing the token's PID, address and endpoint number fields, the protocol engine can handle USB packets and transactions based on data sequencing and state machine logic.

Protocol Engine compiles with High Speed USB 2.0 specification with a transfer rate of 480Mbps. The protocol engine also meets UTMI specification, generating control signals for UTMI transceiver interface according to the FSM states.

USB 2.0 is an industry-wide, host oriented protocol, utilizing physical serial bus. Protocol Engine performs transaction to/from host and Computational Cores End-Points, by managing a DMAC. Protocol Engine supports four types of transactions:

Control - used by the USB System Software to configure devices.

Bulk – a reliable transfer that includes the handshake phase.

Isochronous – real-time, saves overhead by excluding handshake phase.

Interrupt – a limited-latency transfer to or from a device.

The Protocol Engine is partitioned into several major blocks:

1. **Enumeration FSM –** handles the enumeration stages of the USB protocol.
2. **Configured FSM –**handles all IN/OUT transfers from end to end.
3. **Receive Data Packet –** handles data packets analysis & acceptance from UTMI.
4. **Send Data to DMA -** handles handshake with DMA interface in OUT transaction.
5. **Receive Data DMA -** handles handshake with DMA interface in IN transaction.
6. **EP Buffer OUT -** 8 bits width FIFO holds data packet received from the UTMI.
7. **EP Buffer IN -** 8 bits width FIFO holds data transferred from the DMA.
8. **CRC16/CRC5 –** cyclic redundancy check generator.

The Protocol Engine project was downloaded into Altera DE3 FPGA development board for simulation and verification.

Project's Motivation

Recent motivation for a separate USB 2.0 protocol engine stems from the fact that PCs have increasingly higher performance and are capable of processing vast amounts of data. At the same time, PC peripherals have added more performance and functionality. User applications such as multimedia applications demand a high performance connection between the PC and these highly sophisticated peripherals. The separate Protocol Engine addresses this need by working in parallel with the USB µController hence releasing the microprocessor from the burden of dealing with basic data transfer assignments such as token encoding and decoding, CRC check etc. This architecture will increase the parallel work of the USB device and increase its overall speed.

Today's High speed USB 2.0 compliant protocol engines are mostly firmware based and are not hardware based. In general firmware based solutions have much slower response rate and are less efficient than hardware based solutions, however in our case, since a high speed USB2.0 is required to deliver packets at a maximum speed rate of only 480Mbps this is not much of an issue. With an 8-bit bus, the protocol engine will have 16.66 ms to handle each word of data, which could be translated into a 60 Hz clock. Today's devices have clock rates of more than 100K times that.

In our case, the main motivation to use the hardware based solution is to reduce the power requirements for the device. This is a consideration especially for battery powered devices that consume more than 5V and cannot use the USB built in power supply.

Technical Background

Introduction

The Universal Serial Bus (USB) is a specification developed by Compaq, Intel, Microsoft and NEC, joined later by Hewlett-Packard, Lucent and Philips. These companies formed the USB Implementers Forum, incorporated as a non-profit corporation to publish the specifications and organize further development in USB. The USB is specified to be an industry-standard extension to the PC architecture with a focus on PC peripherals that enable consumer and business applications.

Main goal of the USB protocol was to replace the over growing number of different ports of PC connectivity. For example, parallel, ps/2 ports and serial could now be replaced by one simple connection. The USB will do the same role of all the replaced ports and suitable for all applications and peripherals.

USB 2.0 is an Industry-wide, host oriented protocol, employing serial bus, supporting up to 127 devices and hot insertion.

Several criteria were applied in defining the architecture for the USB: Ease-of-use for PC peripheral expansion. USB 2.0 represents a great advance in speed while keeping a low-cost solution that supports transfer rates of up to 480 Mbps. With full support for real-time data, voice, audio, and video it is the chosen protocol for most PC peripherals today. Comprehension of various PC configurations and form factors make the USB a multifunctional protocol capable of servicing various solutions. The USB is a generic protocol making its interface capable of quick diffusion into product. Augment the PC's capability by enabling new classes of devices giving the USB a capability to be implemented in new developed devices, advancing with technology. Fully backward compatibility of USB 2.0 for devices built to previous versions of the USB specification.

Robustness

The key advantage of the USB protocol is its robustness. The USB has signal integrity which enables it to use differential drivers, receivers, and shielding. CRC protection over control and data fields. Detection of attach and detach and system-level configuration of resources. Self-recovery in protocol, using timeouts for lost or corrupted packets. Flow control for streaming data to ensure isochrony and hardware buffer management. Data and control pipe constructs for ensuring

independence from adverse interactions between Functions.

Error Detection

The core bit error rate of the USB medium is expected to be close to that of a backplane and any glitches will very likely be transient in nature. To provide protection against such transients, each packet includes error protection fields. When data integrity is required, such as with lossless data devices, an error recovery procedure may be invoked in hardware or software.

The protocol includes separate CRCs for control and data fields of each packet. A failed CRC is considered to indicate a corrupted packet. The CRC gives 100% coverage on single- and double-bit errors.

Error Handling

The protocol allows for error handling in hardware or software. Hardware error handling includes reporting and retry of failed transfers. A USB Host Controller will try a transmission that encounters errors up to three times before informing the client software of the failure. The client software can recover in an implementation-specific way.

Data Speeds:

The USB specification defines three data speeds:

Name	Speed	Applications	Attributes
Low speed	1.5 Mbit/s	Mouse, Keyboard, Joysticks.	Lowest cost Ease of use Dynamic attach-detach Multiple peripherals
Full speed	12 Mbit/s	POTS, Audio, Broadband.	Lower cost Ease of use Dynamic attach- detach Multiple peripherals Guaranteed bandwidth Guaranteed latency
High speed	480 Mbit/s	Video, Storage, Imaging, Broadband.	Lower cost Ease of use Dynamic attach- detach Multiple peripherals Guaranteed bandwidth Guaranteed latency High bandwidth

Table 1 – USB data speeds

Low Speed

This was intended for cheap, low data rate devices like mice. The low speed captive cable is thinner and more flexible than that required for full and high speed.

Full Speed

This was originally specified for all other devices.

High Speed

The high speed additions to the specification were introduced in USB 2.0 as a response to the high speed of Firewire.

Device's Endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the final stop of a communication flow between the host and device. Each USB logical device is composed of up to 30 independent endpoints and a control endpoint. To reach an endpoint the host must send a packet to the correct device address assigned by the system at device attachment time, and the correct endpoint number given at the

design time. Each endpoint has a device-determined direction of data flow, up to 15 IN and 15 OUT endpoints are available and an additional control endpoint which is always referred to as endpoint zero. The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced.

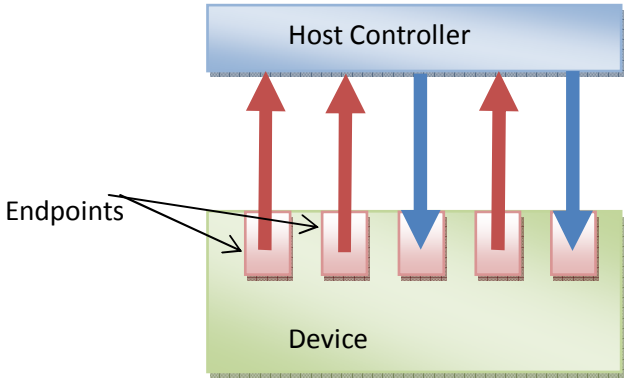

Figure 1– Each endpoint supports data flow in one direction: either input (from device to host) or output (from host to device).

Descriptor Tables

USB devices report their attributes using descriptors. A descriptor is a data structure with a defined format. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor followed by a byte-wide field that identifies the descriptor type.

Using descriptors allows concise storage of the attributes of individual configurations because each configuration may reuse descriptors or portions of descriptors from other configurations that have the same characteristics. In this manner, the descriptors resemble individual data records in a relational database.

Device Descriptor Table

A device descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device's configurations. A USB device has only one device descriptor.

All USB devices have a Default Control Pipe. The maximum packet size of a device's Default Control Pipe is described in the device descriptor. Endpoints specific to a configuration and its interface(s) are described in the configuration descriptor. A configuration and its interface(s) do not include an endpoint descriptor for the Default Control Pipe. Other than the maximum packet size, the characteristics of the

Default Control Pipe are defined by this specification and are the same for all USB devices.

Table 10 in the Appendix section shows the Standard Device Descriptor Table.

Configuration Descriptor Table

The configuration descriptor describes information about a specific device configuration. The descriptor contains a *bConfigurationValue* field with a value that, when used as a parameter to the *SetConfiguration*() request, causes the device to assume the described configuration. The descriptor *bNumInterfaces* field describes the number of interfaces provided by the configuration. Each interface may operate independently. For example, an ISDN device might be configured with two interfaces, each providing 64 Kb/s bi-directional channels that have separate data sources or sinks on the host. Another configuration might present the ISDN device as a single interface, bonding the two channels into one 128 Kb/s bi-directional channel. When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned.

A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among interfaces within a single configuration unless the endpoint is used by alternate settings of the same interface. Endpoints may be shared among interfaces that are part of different configurations without this restriction.

Table 11 – Configuration Descriptor Table in the Appendix section shows the Configuration Descriptor Table.

Interface Descriptor Table

The interface descriptor describes a specific interface within a configuration. A configuration provides up to four interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. When a configuration supports more than one interface, the endpoint descriptors for a particular interface follow the interface descriptor in the data returned by the *GetConfiguration*() request. The descriptor contains a *bInterfaceNumber* field that specifies the number of the interface.

An interface descriptor is always returned as part of a configuration descriptor. Interface descriptors cannot be directly accessed with a *GetDescriptor*() or *SetDescriptor*() request.

Error! Reference source not found. in the Appendix section shows the Interface Descriptor Table.

Endpoint Descriptor Table

Each endpoint used for an interface has its own descriptor. This descriptor contains the information required by the host to determine the bandwidth requirements of each endpoint. The descriptor contains a *bEndpointAddress* field that specifies the direction and endpoint number. *bmAttributes* field is used to determine the endpoint type (i.e Bulk, Isochronous, Control or Interrupt). wMaxPacketSize field contains the value of the Maximum packet size this endpoint is capable of sending or receiving. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-microframe data payloads. An endpoint descriptor is always returned as part of the configuration information returned by a *GetDescriptor*(Configuration) request. There is never an endpoint descriptor for endpoint zero.

Table 13 in the Appendix section shows the Endpoint Descriptor Table.

Token Packets

Field	PID	ADDR	ENDP	CRC5
Bits	8	7	4	5

Table 2 – Token packet bit distribution

A token packet consists of a PID, ADDR and ENDP fields. Table 2 shows the field formats and their respective number of bits. Packet ID specifies either IN, OUT or SETUP packet type. PID codes table is available in Table 14 in the Appendix. For OUT and SETUP transactions, the address and endpoint fields uniquely identify the endpoint that will receive the subsequent Data packet. For IN transactions, these fields uniquely identify which endpoint from a unique addressed device should transmit a Data packet. Only the host can issue token packets. An IN PID defines a data transaction from a function to the host. OUT and SETUP PIDs define data transactions from the host to a function.

The token's correctness is assured by the combination of two mechanisms. The 4 bits representing the PID field are duplicated and negated to become 8 bits long. The ADDR and ENPD fields are protected by the CRC5 field.

Data Packets

Field	PID	Data	CRC16
Bits	8	0-8192	16

Table 3 – Data packet bit distribution.

A data packet consists of a PID, data field containing zero or more bytes of data, and a CRC16 as shown in Table 4. There are four types of data packets, identified by different PIDs: DATA0, DATA1, DATA2 and MDATA. DATA0 and DATA1 are defined to support data toggle synchronization in bulk, setup and interrupt transactions. All four data PIDs are used in data PID sequencing for high bandwidth high-speed isochronous endpoints. Data must always be sent in integral even number of bytes. Similar to the token packet, the data CRC16 is computed over only the data field in the packet and does not include the PID, which has its own check field.

Handshake Packets

Field	PID
Bits	8

Table 4 – Data packet bit distribution.

Handshake packets, as shown in **Error! Reference source not found.**, consist of only a PID. Handshake packets are used to report the status of a data transaction.

Assuming successful token decode, a device, upon receiving a data packet, may return any one of the three handshake types:

- If the data packet was corrupted, the function returns no handshake.
- If the data packet was received error-free and the function's receiving endpoint is halted, the function returns STALL.
- If the transaction is maintaining sequence bit synchronization and a mismatch is detected, then the function returns ACK and discards the data.
- If the function can accept the data and has received the data error-free, it returns ACK.

- If the function cannot accept the data packet due to flow control reasons like out of space, it returns NAK.

Upon receiving a SETUP token, a device must accept the data. A device may not respond to a SETUP token with either STALL or NAK, and the receiving device must accept the data packet that follows the SETUP token. If a non-control endpoint receives a SETUP token, it must ignore the transaction and return no response.

Isochronous transactions have a token and data phase, but no handshake phase. The host issues an OUT token followed by the data phase in which the host transmits data. Isochronous transactions do not support a handshake phase or retry capability.

Setup Packet

Every USB device must respond to requests from the host on the device's endpoint zero. The setup packets are used for detection and configuration of the device and carry out common functions such as setting the USB device's address, requesting a device descriptor or checking the status of an endpoint. These requests are made using control transfers which will be discussed later on. The request and the request's parameters are sent to the device in the Setup packet. Every Setup packet has eight bytes with the following fields:

bmRequestType – One byte field which identifies the characteristics of the specific request. In particular, this field identifies the direction of data transfer in the second phase of the control transfer. The state of the Direction bit is ignored if the wLength field is zero, signifying there is no Data stage. The bitmap of bmRequestType field is specified in the table below.

Bits	7	6..5	4..0
Description	Data Phase Transfer Direction	Type	Recipient
Value	0 = Host to Device 1 = Device to Host	0 = Standard 1 = Class 2 = Vendor 3 = Reserved	0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved

Table 5 – bmRequest bitmap

bRequest – This 8 bit field specifies the particular request based on the Table 6 – bRequest codes below.

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved	2
SET_FEATURE	3
Reserved	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Table 6 – bRequest codes

wValue – Two byte field with variable content according to the request. It is used to pass a parameter to the device, specific to the request.

wIndex – Two byte field with variable content according to the request. It is used to pass a parameter to the device, specific to the request. The *wIndex* field is often used in requests to specify an endpoint or an interface.

wLength - Two byte field which specifies the length of the data transferred during the second phase of the control transfer. The direction of data transfer (host-to-

device or device-to-host) is indicated by the Direction bit of the *bmRequestType* field. If this field is zero, there is no data transfer phase.

Offset	Field	Size	Value	Description
0	bmRequestType	1	Bit-Map	See Table 5
1	bRequest	1	Value	Request
2	wValue	2	Value	Value
4	wIndex	2	Index or Offset	Index
6	wLength	2	Count	Number of bytes to transfer if there is a data phase

Table 7 – Format of setup data

Enumeration Process

After a device is powered on it must follow the enumeration process to receive an address from the host and configuration. When in *powered* state it must not respond to any bus transactions until it has received a reset from the bus. After receiving a reset, the device is then addressable at the default address. Then the system enters the *High Speed Handshake Detection* protocol which includes the detection of a series of at least six J-K-J-K-J-K chirps terminated by SE0 as described in Figure 2. The device then enters high speed mode and switch to a *Default State*.

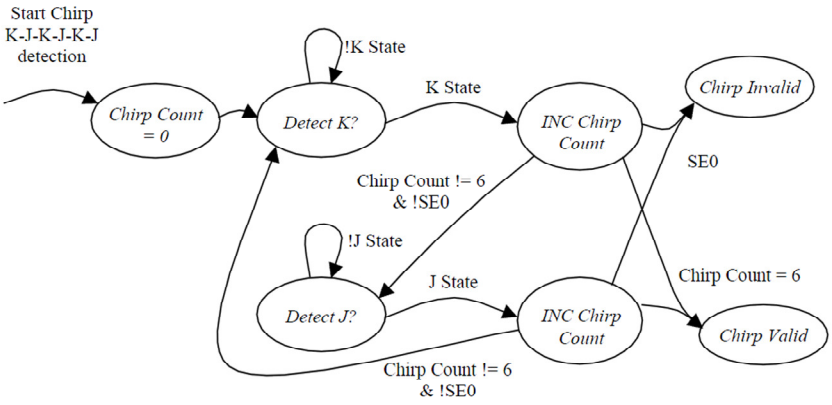

Figure 2 – High speed detection state machine

The device then waits for a *setup token* which precedes the *setup packet* for the *SET_ADDRESS* request. After the setup token has been received correctly, the host will send a *setup packet* to endpoint zero, with address zero and with packet ID *DATA0*. The *setup packet* contains the *SET_ADDRESS* request which contains the device new address assigned by the host. The new address is saved in the *wValue* field of the *setup packet* (see Table 7 for *wValue* description). After the device

changes its address it enters the *Addressed state.*

The device will enter its final state called *Configured State* which starts once the device receives the SET_CONFIGURATION request with a non-zero *wValue* field.

USB Communications Flow

All communications on the USB bus are initiated by the host. This means, for example, that there can be no communication directly between USB devices. A device cannot initiate a transfer, but must wait to be asked to transfer data by the host. In any USB system there is only one host and up to 127 peripherals.

The attached peripherals share USB bandwidth through a host scheduled, token-based protocol.

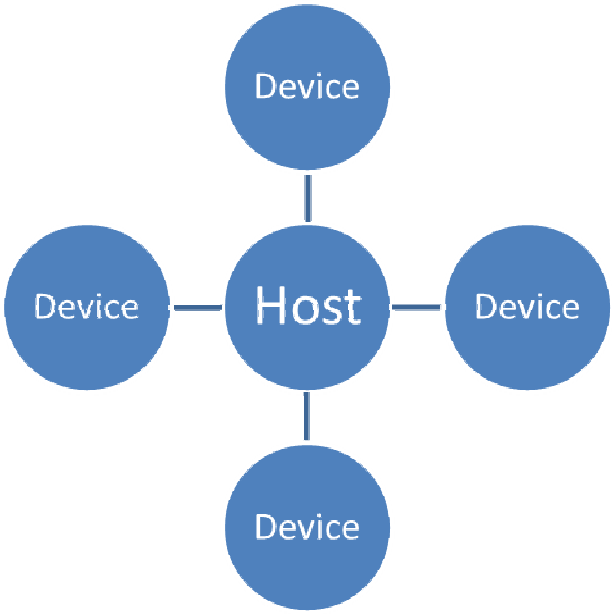

Figure 3 - The logical connection of a USB.

The USB is a polled bus. Each transaction begins when the host controller, on a scheduled basis, sends a USB packet describing the type and direction of transaction, the USB device address, and the endpoint number. This packet is referred to as the *token packet*. The USB device that is addressed selects itself by decoding the appropriate address fields from the *token packet*. In a given transaction, data is transferred either from the host to the device or from the device to the host. The source of the transaction then sends a data packet. If transfer was successful the destination, excluding isochronous type of transactions, responds with a handshake packet indicating the transfer was successful.

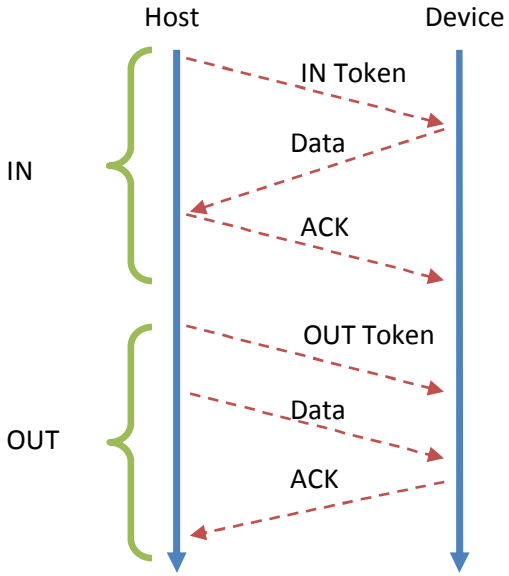

Figure 4 - A typical USB transaction involves a token packet followed by a data packet and ends with a handshake packet

Transaction Types

The USB architecture comprehends four basic types of data transfers:

1. Control Transfers - Control data is used by the USB System Software to configure devices when they are first attached. Mandatory using Endpoint 0 OUT and Endpoint 0 IN.
2. Bulk Transfers - Bulk data typically consists of larger amounts of data, such as that used for printers or scanners. Bulk data is sequential. Reliable exchange of data is ensured. Bulk transfers are designed to transfer large amounts of data with error-free delivery, but with no guarantee of bandwidth. The host will schedule bulk transfers after the other transfer types have been allocated.
3. Interrupt Transfers - A limited-latency transfer to or from a device is referred to as interrupt data. Such data may be presented for transfer by a device at any time and is delivered by the USB at a rate no slower than is specified by the device
4. Isochronous Transfers – Isochronous data is continuous and real-time in creation, delivery, and consumption. Timing-related information is implied by the steady rate at which isochronous data is received and transferred.

Isochronous data must be delivered at the rate received to maintain its timing. A typical example of isochronous data is voice. The timely delivery of isochronous data is ensured at the expense of potential transient losses in the data stream, where it is important to maintain the data flow, but not so important if some data gets missed or corrupted. In other words, any error in electrical transmission is not corrected by hardware mechanisms such as retries.

Data Toggle Synchronization

The USB provides a mechanism to guarantee data sequence synchronization between data transmitter and receiver across multiple transactions. This mechanism provides a means of guaranteeing that the handshake phase of a transaction was interpreted correctly by both the transmitter and receiver. Synchronization is achieved via use of the DATA0 and DATA1 PIDs and separate data toggle sequence bits for the data transmitter and receiver. Receiver sequence bits toggle only when the receiver is able to accept data and receives an error-free data packet with the correct data PID. Transmitter sequence bits toggle only when the data transmitter receives a valid ACK handshake. The data transmitter and receiver must have their sequence bits synchronized at the start of a transaction. The synchronization mechanism used varies with the transaction type. Data toggle synchronization is not supported for isochronous transfers.

Protocol Engine Algorithms

When the device is first powered on enumeration stage is initialized. When the enumeration stage is complete the device enters a standby mode and is ready to receive packets from the host. When a packet is received from the host, the device reacts to the received packet, and interacts with the device via the corresponding endpoint.

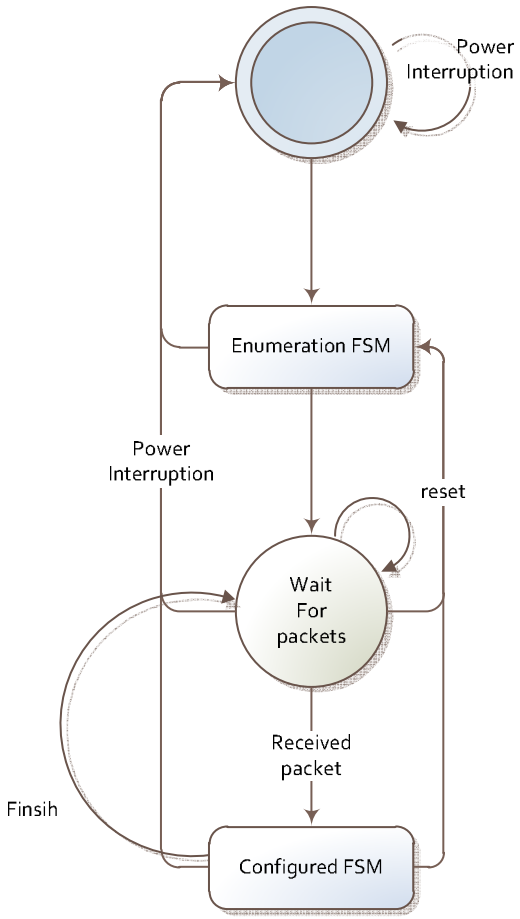

Figure 5 – Device Top level

Figure 6 describes the algorithm for the Configured device state. First, a token packet is received. The packet is checked for correct PID, CRC5 and, if the checks succeed, the device checks if the token is addressed to our device. If the token is addressed to our device the endpoint field is checked. If the endpoint is zero and the endpoint state is *stall*, then a *stall* handshake is sent back to the host.

At this point, the module analyzes the PID to determine if it is of type IN, OUT or PING. The device will act upon the transfer type set by the PID.

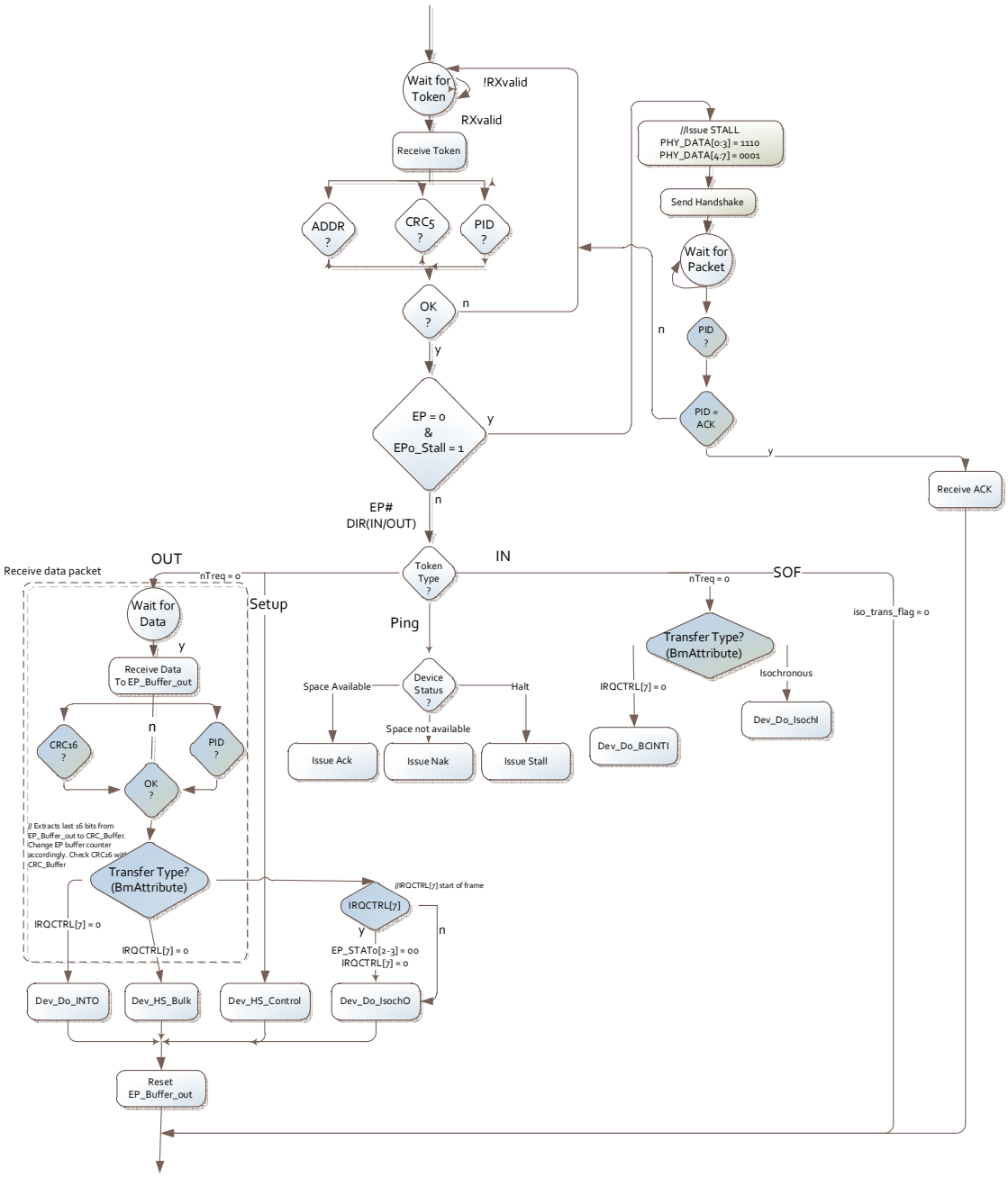

Figure 6 – Configured Final State Machine

Development tools

We used several development tools for compiling, simulations, debugging and verifications:

Modelsim

Modelsim is an integrated development environment (IDE) used by electronic designers to develop, debug, simulate and test electronic designs. We used Modelsim for comprehensive simulating and debugging the Protocol Engine.

Quartus

Quartus II version 9.0 software offers a seamless development flow for the design of digital hardware, allowing you to enter, compile, and simulate a design.

Quatrtus II provides all the necessary steps for downloading our design into the FPGA board, starting with analysis and synthesis through fitter and Assembler and ending with timing analysis.

MegaWizard Plug-in Manager was used to create a ROM element which was part of our demonstration. The ROM element contained the packet and data that will be sent through the PHY and DMA bus during our demonstration. Using the MegaWizard Plug-in Manager we could set the values of the ROM to the content of a pre written mif binary file.

Simvision

Simvision is Development software for simulation and debugging a HDL design. This tool was rarely used due to the often network problems that prevented us from using this tool.

Altera DE3 FPGA Board

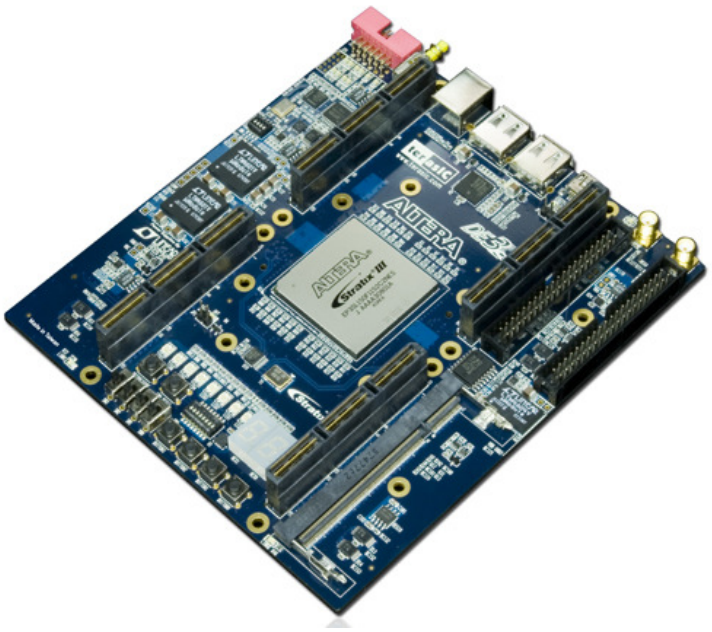

Figure 7 – Altera DE3 FPGA development board

The Altera DE3 FPGA development board is a powerful Programmable Gate Array. Using the DE3 FPGA board we could download the Protocol Engine project into hardware for simulation and verification of the design in Real Time conditions. For the simulation and verification of the Protocol Engine we used all the available board's switches and 7-seg displays and some of the LEDs and push buttons.

DE3 board possess the EP3SL340 Stratix III FPGA has 338K logic elements, compared to the 68K logic elements of the EP2C70F896C6 Cyclone II FPGA that can be found on the DE2-70 board. Although the DE2-70 has more leds, 7-seg displays and switches we could not use it. Since the complexity of the USB 2.0 protocol required many logic elements for implementation, then the design of the Protocol Engine resulted in using more logic elements. EP2C70F896C6 Cyclone II FPGA on the DE2-70 board cannot supply so many elements.

System Level Introduction

The USB 2.0 Protocol Engine will be integrated into a USB 2.0 device as a peripheral device, thus offloading communication tasks from the μController, by processing USB 2.0 packet-level Link-Layer Protocol tasks in Hardware.

The protocol engine interacts with the Transceiver chip (PHY) and with the DMAC. Data received through the USB cable will enter the device via the USB connector and will then be transferred to the PHY chip which serves as a Serializer-Deserializer (SERDES), bit stuffer/un-stuffer and the NRZI encoder/decoder, which also handles the low level USB protocol and the signaling task. The system will use a commercial mixed-signal USB 2.0 Transceiver chip (PHY).

The PHY's output is sent to the Protocol Engine. The interface between the USB 2.0 Transceiver chip (PHY) and the Protocol Engine, is defined by the UTMI (USB2.0 Transceiver Macrocell Interface) Standard. Both Transceiver chip and Protocol Engine support high-speed (480 Mbps) signaling bit rates.

The Protocol Engine's output is connected to the device system bus. The system bus is controlled by the μController and the DMAC. USB protocol-aware DMA engine maximizes data throughput while minimizing demands on the system bus. DMAC has the priority in getting access (via arbitration) to the bus. The DMAC performs communication data packet transfers between the Protocol Engine packet buffers and Device Endpoints, responding to Protocol Engine requests.

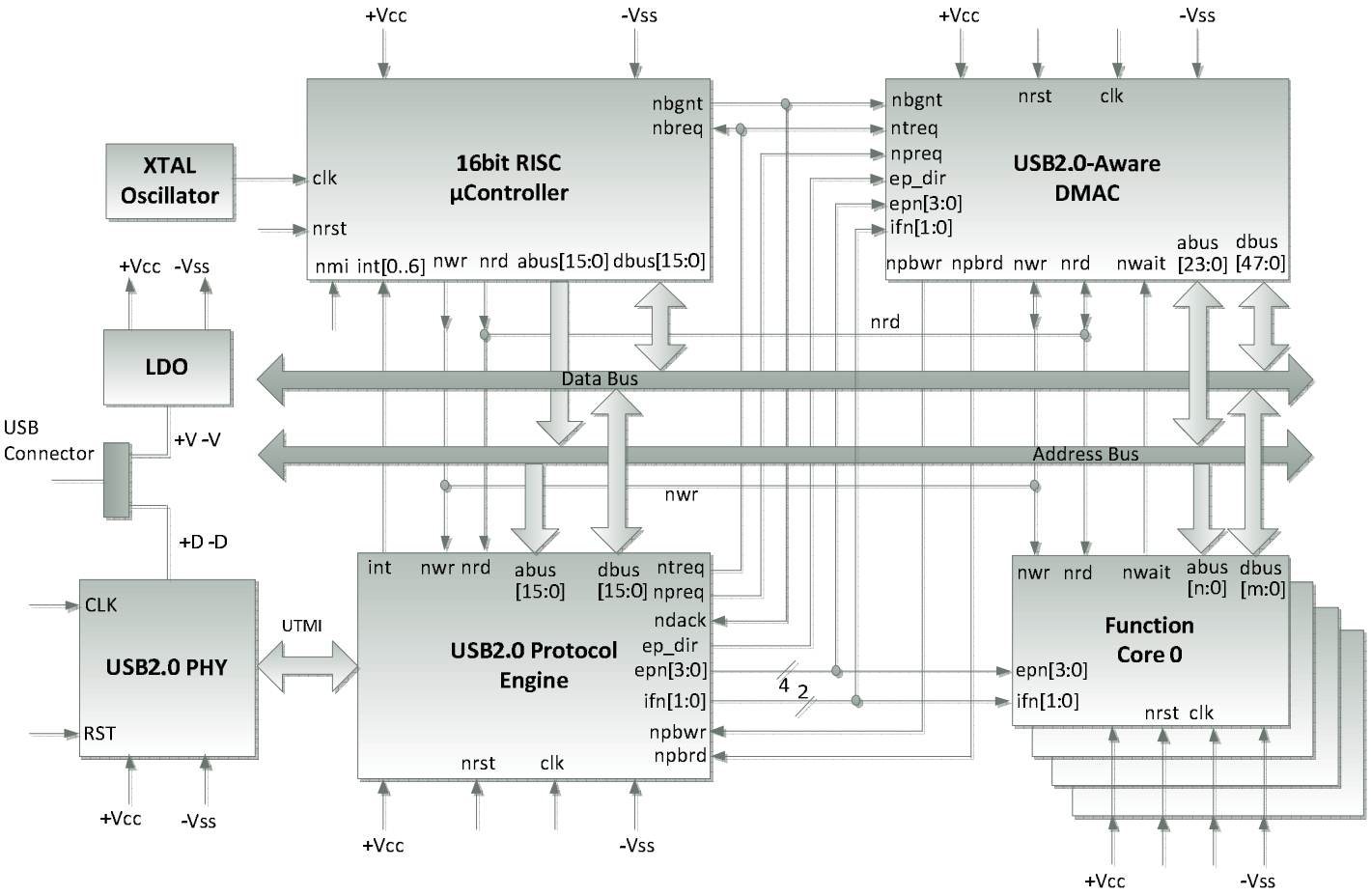

Figure 8 - µController-based Device Controller Block Diagram

The Protocol Engine

The protocol engine performs CRC check/generation, packet identifier (PID) decoding and verification, address recognition and handshake evaluation/response.

The protocol engine also meets UTMI specification. It generates control signals for UTMI transceiver interface according to the FSM states. Control signals are sent to the DMA interface and Function Core

Acting on its USB PID and address recognition logic, and other sequencing and state machine logic, the protocol engine can handle USB packets and transactions.

Protocol Engine supports up to 15 IN & 15 OUT Isochronous, Bulk, or Interrupt physical endpoints, and control endpoint 0. Information is sent to the endpoints via a DMAC and the System Bus.

UTMI Interface Signals

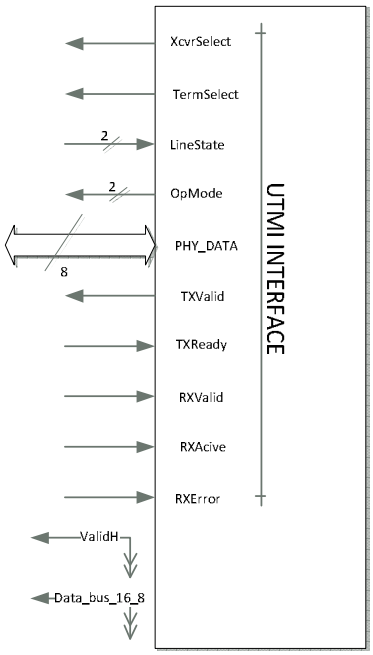

Figure 9 – UTMI Interface Block diagram.

The protocol engine and UTMI work closely together. The UTMI serves as a Serializer-Deserializer (SERDES), bit stuffer/un-stuffer and the NRZI encoder/decoder, which also handles the low level USB protocol and the signaling task.

Name	Direction	Active Level	Description
clk	Input	Rising-Edge	**Clock.** This output is used for clocking receive and transmit parallel data.
nrst	Output	Low	**Reset.** Reset all state machines in the PHY. Same pin as Device nrst.
XcvrSelect	Output	N/A	**Transceiver Select.** This signal selects between the FS and HS transceivers: 0: HS transceiver enabled 1: FS transceiver enabled
TermSelect	Output	N/A	**Termination Select.** This signal selects between the FS and HS terminations: 0: HS termination enabled 1: FS termination enabled This signal is set to 0 during enumeration.
LineState[1:0]	Input	N/A	**Line State.** These signals reflect the current state of the single ended receivers. They are combinatorial until a "usable" **CLK** is available then they are synchronized to **CLK**. They directly reflect the current state of the **DP** (LineState[0]) and **DM** (LineState[1]) signals: DM DP Description DP - USB data pin Data+ 0 0 0: SE0 DM - USB data pin Data– 0 1 1: 'J' State 1 0 2: 'K' State 1 1 3: SE1
OpMode[1:0]	Output	N/A	**Operational Mode.** These signals select between various operational modes: [1][0] Description 0 0 0: Normal Operation 0 1 1: Non-Driving 1 0 2: Disable Bit Stuffing & NRZI encoding 1 1 3: Reserved
RXValid	Input	High	**Receive Data Valid.** Indicates that the **Data bus** has valid data. The PHY RX Data Holding Register is full and ready to be unloaded. The PE is expected to latch the **Data bus** on the clock edge.
RXActive	Input	High	**Receive Active.** Indicates that the PHY detected SYNC and is active. **RXActive** is negated after a Bit Stuff Error or an EOP is detected.
RXError	Input	High	**Receive Error.** 0 - Indicates no error. 1 - Indicates that a receive error has been detected. This input is clocked with the same timing as the **Data out** lines and can occur at any time during a transfer. If asserted, it will force the negation of **RXValid** on the next rising edge of **CLK**.
TXReady (Wait Signal)	Input	High	**Transmit Data Ready.** If *TXValid* is asserted, the PE must always have data available for clocking in to the PHY TX Holding Register on the rising edge of **CLK**. **TXReady** is an acknowledgement to the PE that the PHY has clocked the data from the bus and is ready for the next transfer on the bus. If *TXValid* is negated, **TXReady** can be ignored by the PE.

TXValid	Output	High	**Transmit Valid.** Indicates that the **Data bus** is valid for transmit. The assertion of TXValid initiates PHY-generated SYNC transmission the USB. The negation of TXValid initiates PHY-generated EOP on the USB. Control inputs (OpMode[1:0], TermSelect,XcvrSelect) must not be changed on the de-assertion or assertion of **TXValid**. The PHY must be in a quiescent state when these inputs are changed.
ValidH	Bi-directional	N/A	If *ValidH* = 0, *PHY_DATA* is 8 bits. If *ValidH* = 1, *PHY_DATA* is 16 bits. In our system this bit is always set to 0.
DataBus16_8	Output	High	**Data Bus 16 - 8**. Selects between 8 and 16 bit data transfers. This bit is always set to 0 since our device supports an 8 bit PHY transfer.
PHY_DATA[15:0]	Bi-directional	N/A	**PHY bus.** These pins serve as the input and output data bus for the PHY device.

Table 8 – UTMI Interface Signals

DMA Interface Signals

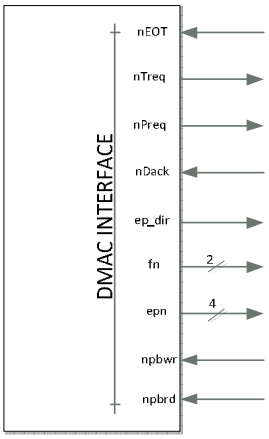

Figure 10 – DMA Interface Block diagram.

Name	Direction	Active Level	Description
nPreq	Output	Low	**Packet Request.** Data is ready to be sent or to be received.
nTreq	Output	Low	**Transfer Request.** Bus grant request.
nDack	Input	Low	**DMA Acknowledge.** Signal from DMA. DMA indicates that System Bus has been granted to him and data transfer begins.
nEOT	Input	Low	**End Of Transfer Indication.** DMA reports to PE that data transfer to the specific Endpoint, has been completed.
epn[3:0]	Output	N/A	**Endpoint number.** (0-15) for requested data transfer.
npbwr	Input	Low	**Packet buffer write.** Write signal to EP Buffer IN
npbrd	Input	Low	**Packet buffer read.** Read signal to EP Buffer OUT
ep_dir	Output	N/A	**Endpoint Direction.** IN (1) or OUT (0) for requested data transfer.
Ifn	Output	N/A	**Interface Number.** Selects the number of the current interface

Table 9 – DMA Interface Signals

Device Description

Top Level Blocks

The Protocol Engine is partitioned into several modules is described below and shown on the block diagram in Figure 11 – Top Level Block diagram.

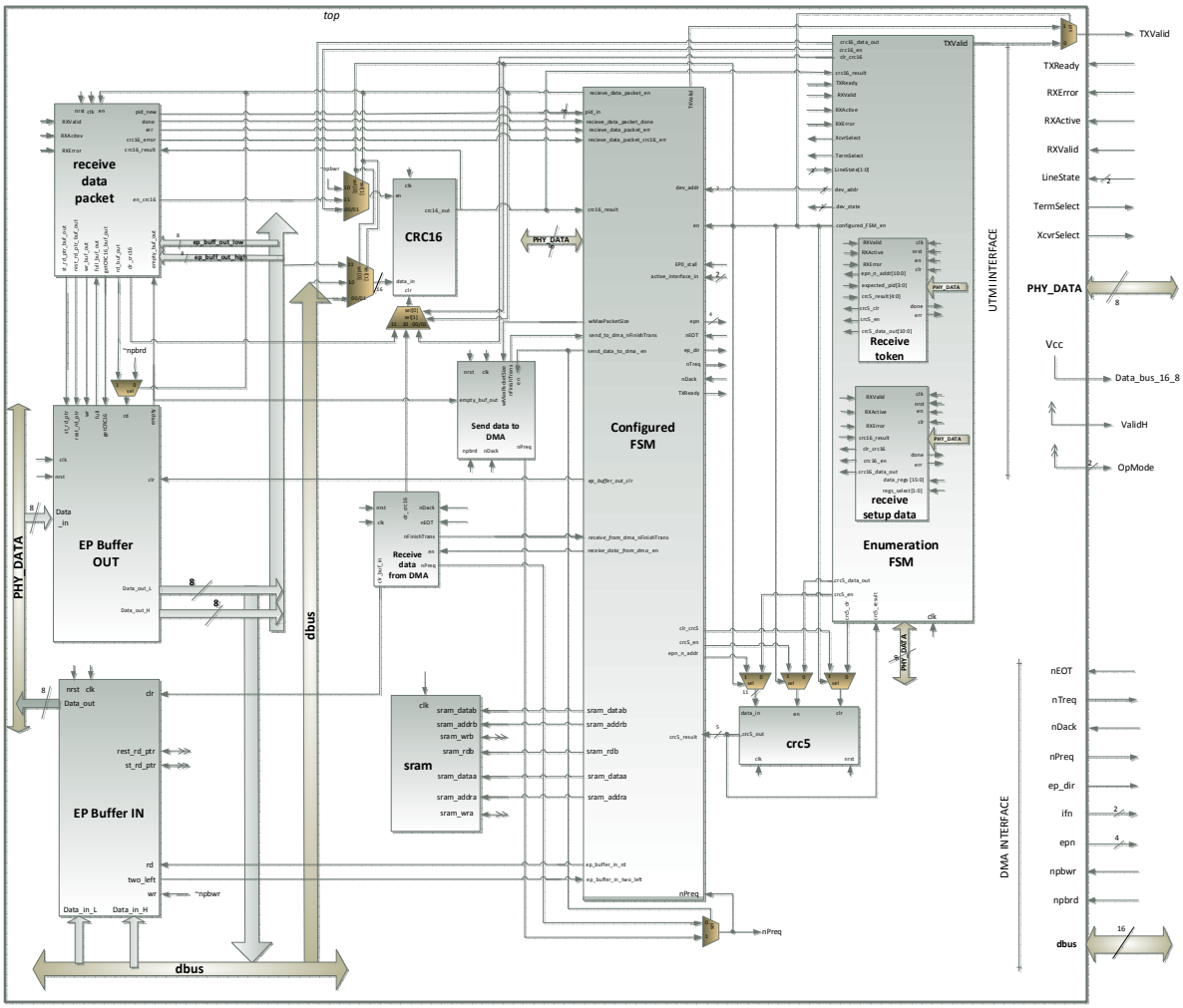

Figure 11 – Top Level Block diagram.

Enumeration FSM

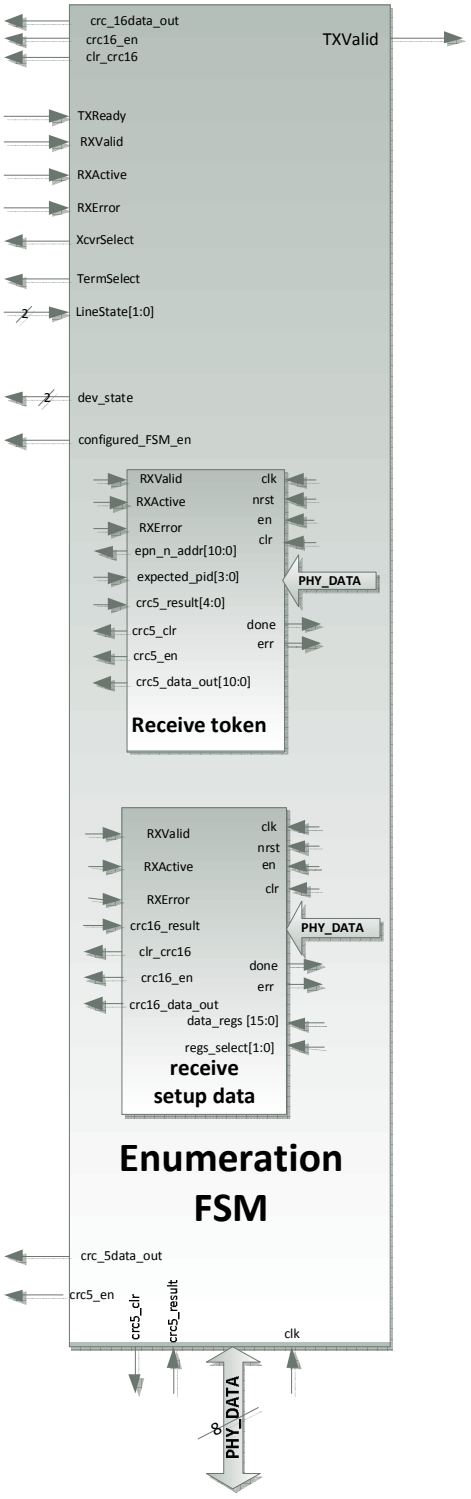

Figure 12 –Enumeration FSM block diagram.

The *enumeration FSM* module is in charge of the enumeration stages of the USB protocol as it is described in the *technical background* chapter. Once the device is

powered, this module is enabled and starts waiting for the reset signaling process. The reset process is composed from the following time constraints actions: *LineState* should be logic '0' for at least 2.5 ms, followed by that the module will assert Chirp K on the bus for at least 1 ms and not more than 7 ms after the reset. The *XcvrSelect* is then asserted which signals the high speed mode. The host will then send a series of at least six J-K-J-K-J-K chirps terminated by SE0, if the process ends successfully the *TermSelect* signal is asserted. The device will then change state to *default state*.

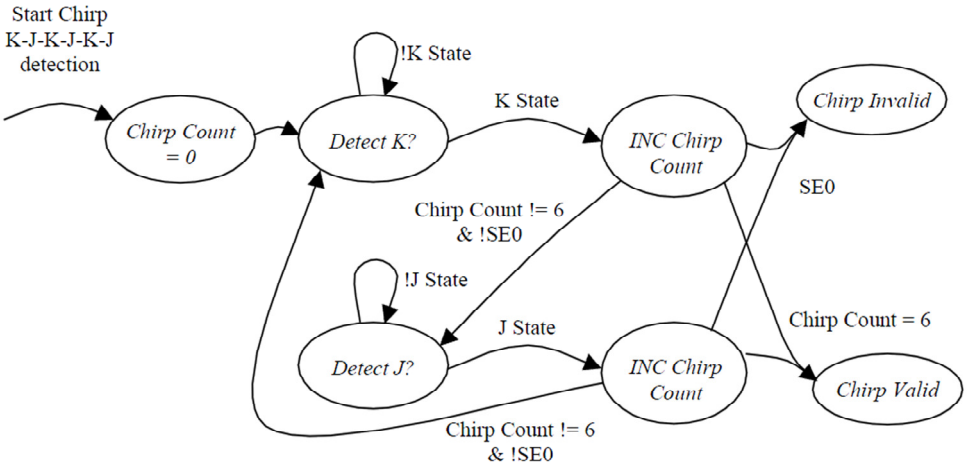

Figure 13 – High speed detection state machine

Default State - At this point the state machine waits for a setup token that precedes the SET_ADDRESS request. The token's address field should be zero. After a valid setup token is accepted, the state machine waits for a *setup packet* with PID DATA0, *wValue* contains the new device address, and the remaining fields should be zero. The data is saved and checked for CRC16 errors. The state machine then waits for an IN token so it could respond with zero length data packet indicating the previous request was accepted successfully. The state machine will wait for an ACK response from the host. Only after the ACK has been received the device address will change and the device state will switch to *addressed state*.

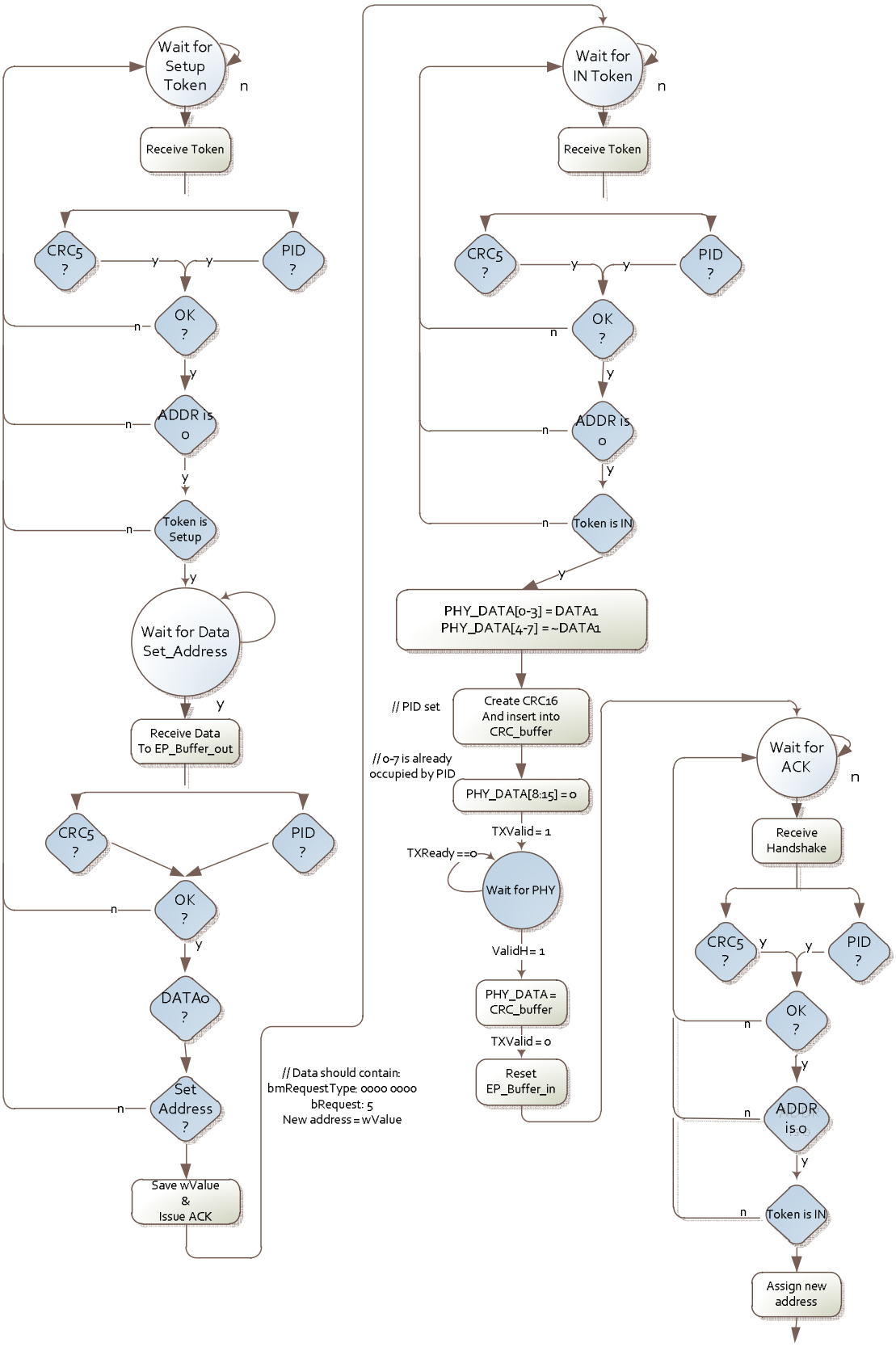

Figure 14 – Set Address state machine

Addressed State - at this point the state machine waits for another setup token that precedes the *SET_CONFIGURATION* request. After a valid setup token is accepted, the state machine waits for a *setup packet* with PID DATA0, *wValue* contains the device configuration number, the remaining fields should be zero. Since our device has only one configuration, it will only accept configuration requests that have their *wValue* field set to '1' and will ignore configuration requests which have any other value in the *wValue* field. The data is saved and checked for CRC16 errors. The state machine then waits for an IN token so it could respond with zero length data packet indicating the previous request was accepted successfully. The state machine will wait for an *ACK* response from the host. Finally after the *SET_CONFIGURATION* request the device state will change to *configured state.*

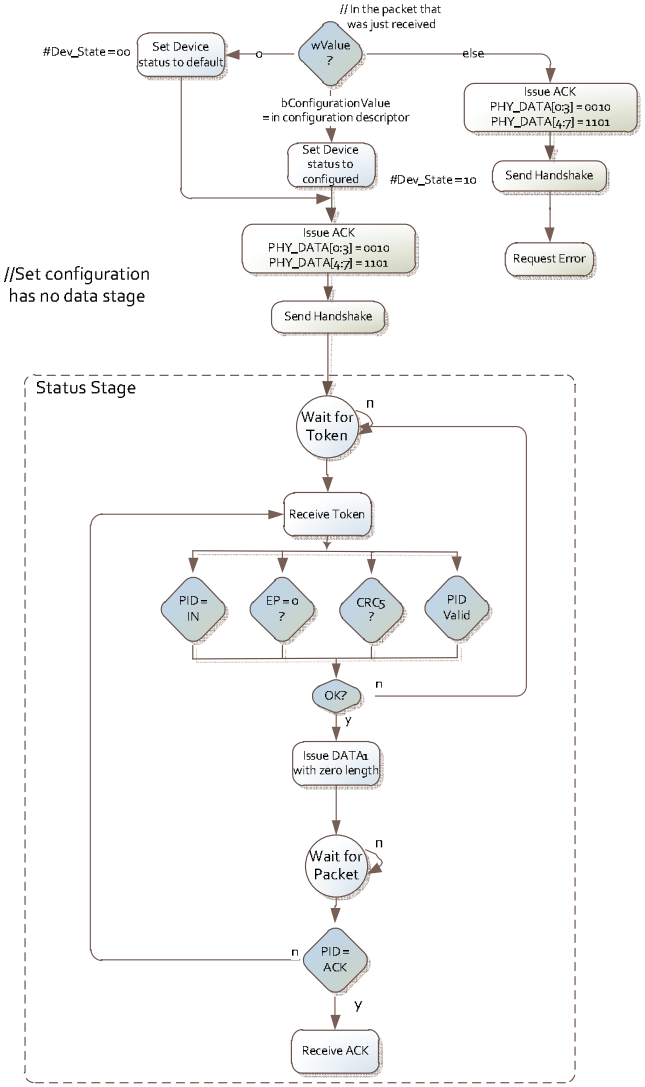

Figure 15 – Set Configuration state machine

Configured State - once the device is in configured state, the enumeration FSM raises the *enable* signal in the *configured FSM* module and enters an idle state. The enumeration FSM will wake up from its idle state only if a SET_CONFIGURATION request with configuration value set to zero will be received.

Enumeration FSM contains the sub modules *receive setup data* and *receive token*, which handle the UTMI handshake process, PID checks, CRC checks, and address check for received packets and tokens.

Configured FSM

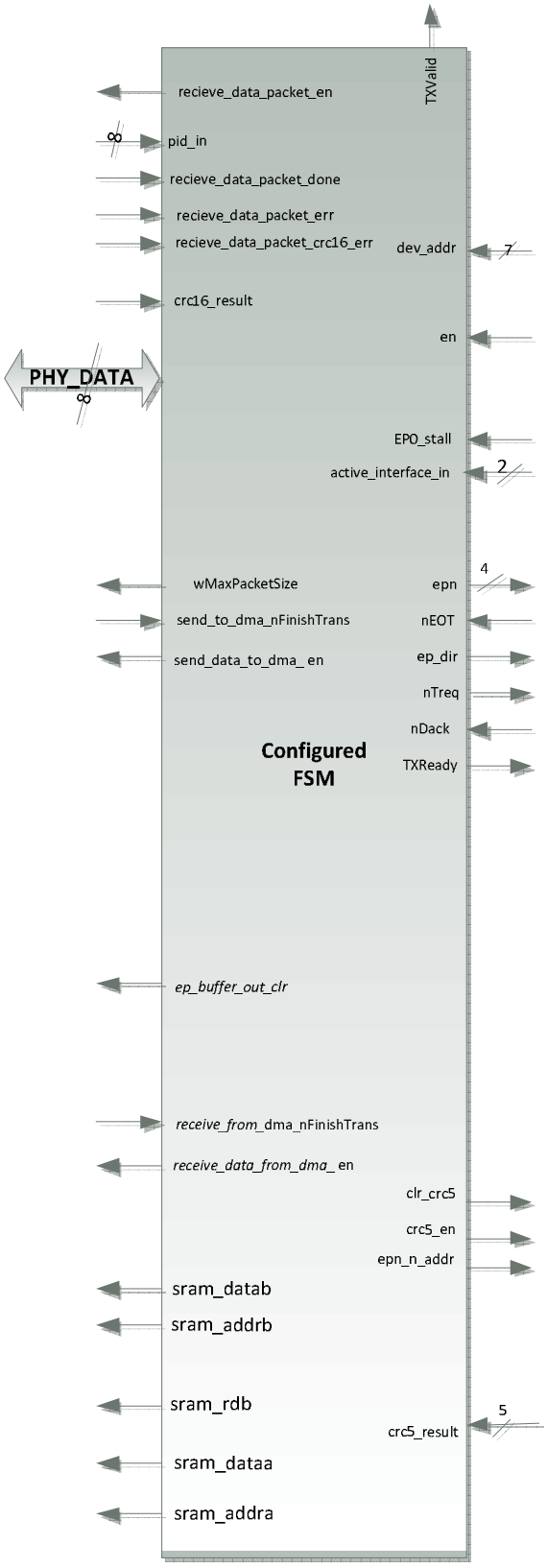

Figure 16 –Configured FSM block diagram.

The *configured FSM* module is the heart of the system while in *configured* state. It handles all transfers from end to end starting with the token stage through the data stage and ending with the handshake stage if required. It controls the following modules: *receive data packet*, *send data to DMA*, *receive data from DMA* and the *SRAM*.

configured FSM has access to the UTMI interface which enables it to receive tokens. Received tokens are checked for errors in PID, CRC5 (with the help of the *crc5* module) and having the device's address. Tokens with an error or of a different address are discarded. Valid tokens are analyzed for PID type. The PID determines the type of transfer the host requires: *OUT*, *IN*, *SETUP* or *PING*:

PING - The device will issue a handshake depending on its current state. If the endpoint which was addressed by the token is in a halt state (due to a previous request that was sent by the host) the device will issue a Stall handshake. If the device buffer is full, then a NAK handshake will be sent and if the device is ready to receive packets it will send ACK handshake.

SETUP - This feature is not yet supported in PE.

OUT - If an OUT PID is detected, *configured FSM* will enable the *receive data packet* module. It will than wait until its operation will finish and the data packet along with the CRC16 field will be received into *EP buffer OUT*. It will check the toggle synchronization bit and if its valid the *send to DMA* module will be enabled and *configured FSM* will wait for the *send_to_dma_nFinishTrans* signal to rise. Once the *send to DMA* finished successfully an ACK handshake will be sent.

IN - If and IN PID is detected, *configured FSM* will enable the *receive data from DMA* module. It will wait until its operation will finish and the data packet will be received into *EP buffer IN*. It will than operate its interface with the UTMI to send the data in the buffer with PID at the head and it's CRC16 at its tail.

Receive Data Packet

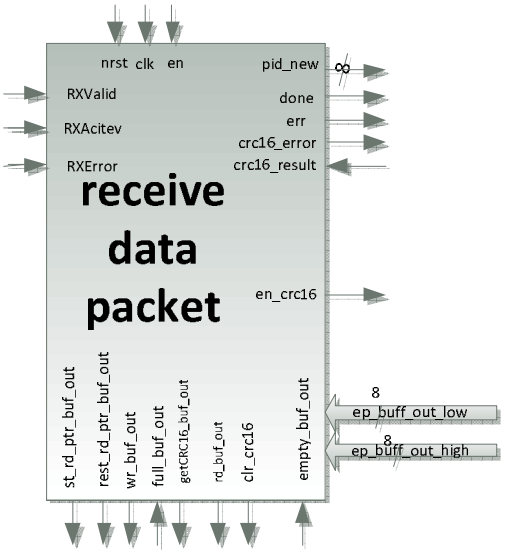

Figure 17 – Receive Data Packet block diagram.

This module contains a state machine which handles the acceptance of new data packets from the UTMI. When in initial state the block waits for *RXActive* signal to assert which signals beginning of the synchronization process between the host and the device and data is about to be sent from the host.

PE then waits for the *RXValid* signal to assert which signals that valid data is ready for reading from the *PHY_DATA* bus. The new received data is then stored in the *EP Buffer out* FIFO. The data packet transfer stage ends once the *RXValid* signal is deasserted.

After the *data packet* has been received, *Receive Data Packet* module will check the packet ID (PID) and toggle synchronization bit. If PID is valid, the module will assert the *st_rd_ptr_buf_out* signal which signals the *EP Buffer out* FIFO to store the current value of *its* read pointer. The state machine will then signal the *EP buffer OUT* to output the CRC16 field from the end of the FIFO and will save it for the CRC16 calculation check later on.

The state machine will then enable the *CRC16* module and assert the *rd_buf_out* signal, starting the data flow from the *EP buffer OUT* to the *CRC16* module. The CRC16 module reads the data from the FIFO and calculates the checksum. The output is then compared with the CRC16 field sent with the data packet. In the same time, the *Receive Data Packet* signals the FIFO to reset its read pointer to its

previous position so it would be ready for the DMA when it's time for it to read the data.

Send Data to DMA

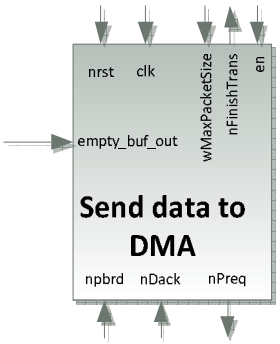

Figure 18 – Send Data to DMA block diagram.

This module handles the handshake process with the DMA in an OUT transaction. The handshake process starts when the PE asks the µCotroller for a grant to use the bus by asserting the *nTreq* signal. The request is granted when the *nDack* is asserted by the µCotroller. Then, the PE asserts the *nPreq* to signal that the data is ready to be read. When the *empty* signal in *EP Buffer OUT* is asserted, the *nPreq* signal is deasserted, signaling the end of the packet transfer. If the amount of data is of zero length or less than the maximum packet size mentioned in the specific endpoint descriptor table (wMaxPacketSize), then not only the packet transfer has ended rather the entire transaction to that endpoint has finished.

Receive Data from DMA

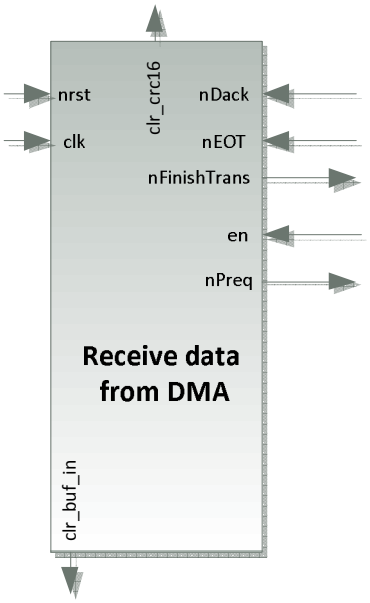

Figure 19 – Receive Data from DMA block diagram.

After an IN token, data has to be received from the DMA. This module uses the same signals as the *send to DMA* module to handle the handshake procedure in an IN transaction. Similarly to the sending process, the *nTreq* signal has to be asserted to request bus grant from µCotroller. When the *nDack* signal is asserted bus is granted. The DMA will then wait for *nPreq* from the PE indicating the PE is ready to receive data. The packet transfer will end once the *nDack* is deasserted, if in addition the *EOT* is asserted then not only the packet transfer has ended rather the entire transaction to that endpoint has finished.

SRAM

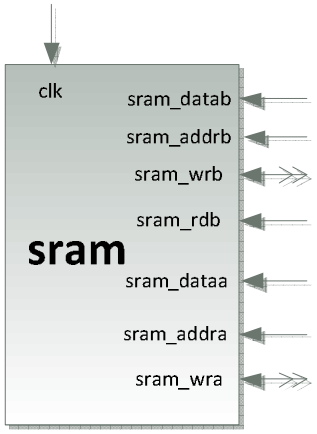

Figure 20 – SRAM block diagram.

The 16 bit width SRAM with 512 addresses holds the device's descriptor tables. The SRAM has two outputs which gives it the ability to read two different memory spots in a single clock cycle. The descriptor tables hold information such as endpoint's type (bulk, isochronous or interrupt), maximum packet size allowed etc. Once the chip is ready for production the SRAM will be replaced with a ROM memory and will be hard coded into the board in production.

EP buffer OUT

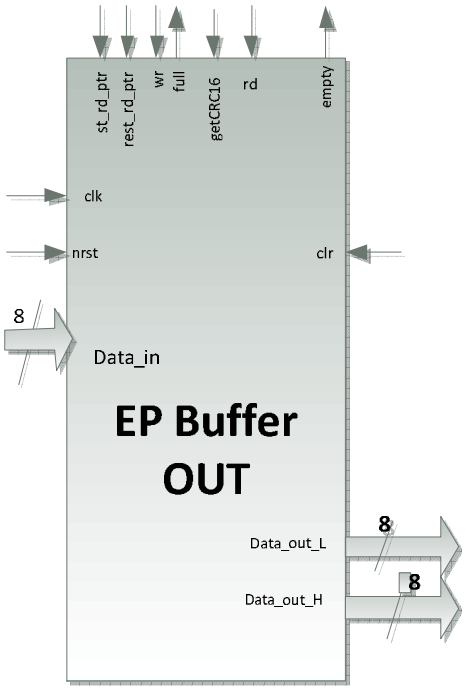

Figure 21 - EP Buffer OUT module block diagram

The *EP buffer out* module is a custom made 8 bits width FIFO that receives the data packet from the 8 bit *PHY_DATA* bus during an OUT transaction. The FIFO's 16 bit output is connected to the *crc16* module and the *dbus*. The FIFO is controlled mostly by the *receive data packet* module which has access to most of its control signals. The FIFO has a *read pointer* and a *write pointer*, which are used to keep track of the FIFO's status. The FIFO is said to be empty when the read and write pointers are equal and is full when the write pointer points to the highest address in the FIFO. The FIFO receives the data packet's raw data and CRC16 field, the PID field in the *receive data packet* module.

As was explained in the *receive data packet* module, the protocol engine has no way of knowing when the CRC16 field arrives from the PHY, but we do know that it is the last two bytes that were inserted into the FIFO. Since a regular FIFO does not allow reading the last inserted information, we added *getCRC16* signal which extracts the last two inserted bytes. The *st_rd_ptr* signal is used to store the position of the read pointer before the crc16 module reads the data in the FIFO. After extracting the last two bytes from the FIFO and after storing the read pointer, the *rd* signal is raised

and the data starts to flow through *crc16* module. At the end of the crc16 calculation process the FIFO *empty* signal is will assert and the rd pointer will point to the same memory address as the write pointer. At this point the *receive data packet* module asserts the *rest_rd_ptr* which resets the read pointer to its previous position before the crc16 calculations.

EP Buffer IN

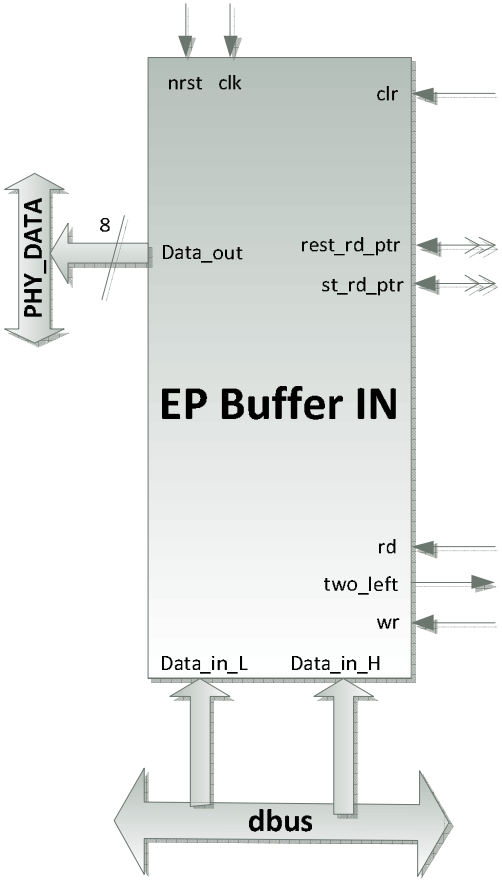

Figure 22 - EP Buffer IN module block diagram

The EP buffer IN module is a custom made 8 bits width FIFO that receives the data packet from the 16 bit dbus during an IN transaction. The FIFO's 8 bit output is connected to the *PHP_DATA* bus. During transfer from the DMA the write signal is controlled by DMA's *npbwr* signal. The EP Buffer IN receives only the raw data without the PID or CRC16 fields. The CRC16 is calculated simultaneously while the raw data is transferred from the DMA to the EP Buffer IN. The configured FSM is responsible for sending the PID first, the DATA from the FIFO second and the CRC16 last.

The FIFO has a *read pointer* and a *write pointer*, which are used to keep track of the FIFO's status. The FIFO is said to be empty when the read and write pointers point to the same memory block.

CRC5

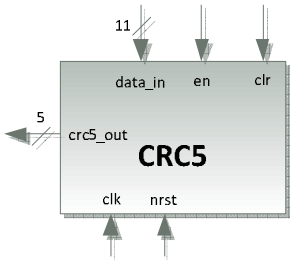

Figure 23 – CRC5 module block diagram

The CRC5 module is used whenever a token packet is received. A token packet has a crc5 field at its tail which has to be checked on arrival. The data part of the token is composed of the endpoint number and the device's address. This information is inserted into the *data_in* (11 bit) entrance which then produces the correct CRC5 after a single clock.

CRC16

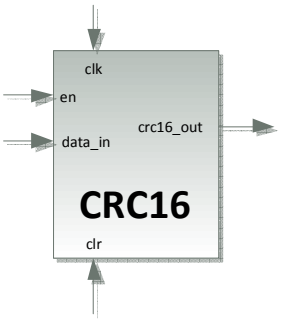

Figure 24 – CRC16 module block diagram

The CRC16 module is used whenever a data packet or a setup packet is received. Each data packet in a transfer contains crc16 field in its tail. Upon each clock cycle the data is inserted into the module's *data_in* (16 bits). After the last word is processed by the module we receive the correct crc16 in the module's output.

Detailed Functionality Description

OUT transaction

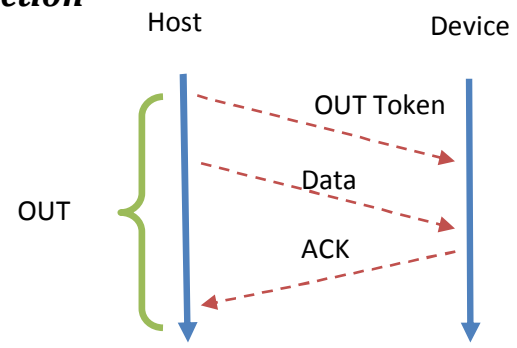

Figure 25 – OUT transaction.

The OUT transactions start, just like any other transaction in the system, with a token sent from the host. The state machine in *configured FSM* waits for the UTMI signals RXActive and RXValid and receives the OUT token, a detailed description of the state machine can be viewed in Figure 26.

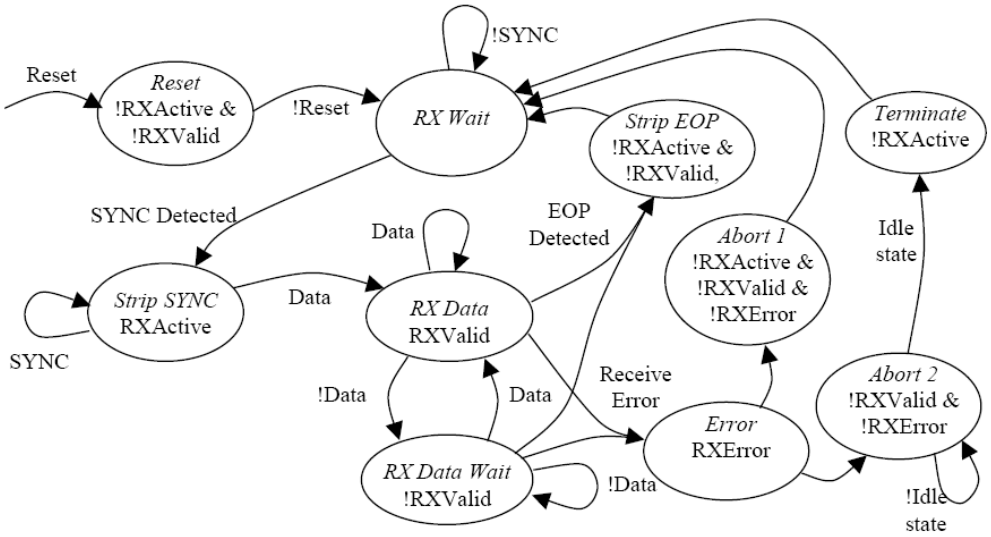

Figure 26– UTMI Receive State Diagram

The token's fields and size are discussed in the chapter "Technical Background". The packet is checked for correct PID by comparing the first four bits with the negation of the last four bits. The CRC5 field is checked with the help of the CRC5 module. If the checks succeed, it means the token was received correctly, the data load of the packet is without errors. The data load of the packet contains the endpoint number to which the data packet should be routed and the device address. First the state machine checks if the token is addressed to our device by comparing the device

address in the token packet with the device address saved in *dev_addr* register in the *enumeration FSM* module. If the token is addressed to our device the endpoint field is checked and saved in *epn* register. If the endpoint is zero and the endpoint state is *stall*, then a *stall* handshake is sent back to the host.

At this point, the module analyzes the PID to determine that its type is OUT (0001), asserts the *nTreq* signal to get bus access from the µController, sets the ep_dir signal to OUT and enables the *Receive Data Backet* block which waits for a *data packet* to be received from the host. The signals diagram in Figure 27- UTMI receive signals describes the transfer of a data packet from the UTMI.

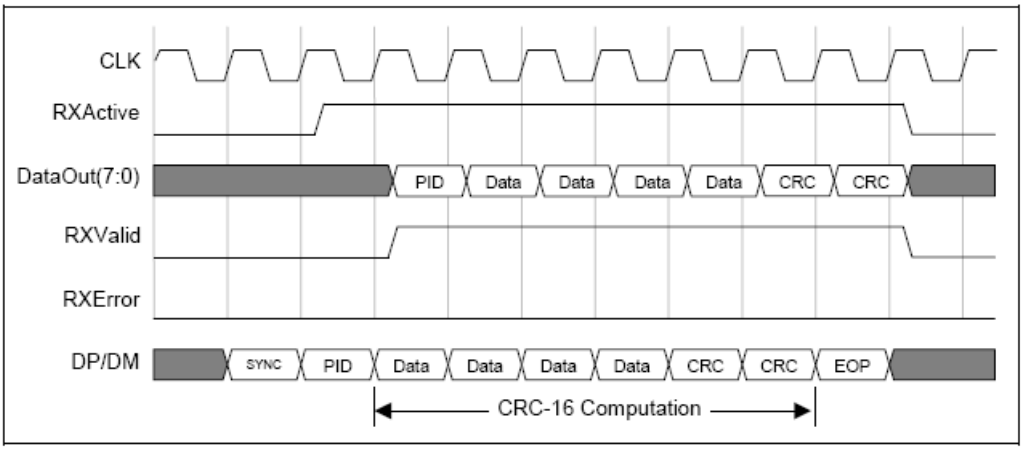

Figure 27- UTMI receive signals timing

The PID is saved in the *curr_pid* register and the Data and CRC16 are saved in the *EP Buffer OUT* FIFO. The FIFO is controlled mostly by the *Receive Data Packet* module which has access to most of its control signals. *Receive Data Packet* checks if the PID is of type DATA and if the PID is valid, the module will assert the *st_rd_ptr_buf_out* signal which stores the current value of the *EP buffer OUT* read pointer. The state machine will then signal the *EP buffer OUT* to output the CRC16 field from the end of the FIFO and will save it for the CRC16 calculation check later on. The state machine will then enable the *CRC16* module and will assert the *rd_buf_out* signal, starting the data flow from the *EP buffer OUT* to the *CRC16* module. The CRC16 module reads the data from the FIFO and calculates the checksum. The result is then compared with the CRC16 field which was sent with the data packet and was extracted earlier from the FIFO. At the same time, the FIFO's read pointer is reset to its previous position so it would be ready for the DMA when it's time for it to read the data.

If this operation ends successfully the *Receive Data Packet* module asserts the *done*

signal. If there was an error and the data was not received correctly *err* signal is asserted and if there was only an error with the CRC16 check then the *crc16_err* signal is asserted. In case of CRC16 error, *Bulk* transactions are retried and *isochronous* transactions are continued. At this point the *Receive Data Packet* is disabled and *configured FSM* checks the toggle synchronization bit of the PID. For *Bulk* transactions the PID should be DATA0 or DATA1 alternately. *Isochronous* transactions based on the endpoint's descriptor table have a sequence of two, one or none DATAM packets followed by DATA2, DATA1 or DATA0 respectively. See the Technical Background chapter for additional information about the toggle bit.

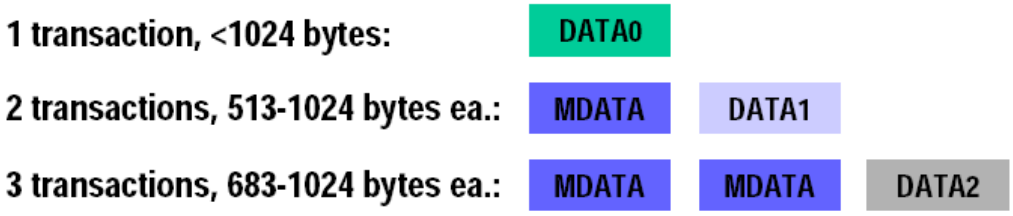

Figure 28 - Data Phase PID Sequence for Isochronous OUT endpoints

If the toggle check is valid the *send to DMA* module will be enabled and *configured FSM* will wait for the assertion of *send_to_dma_nFinishTrans* signal which means that the DMA has finished sending the information. The *send to DMA* module waits for the *nDack* signal to be asserted which means the bus request is granted. Then, the *send to DMA* asserts the *nPreq* to signal that the data is ready to be read. When the *empty* signal in *EP Buffer OUT* is asserted, the *nPreq* signal is deasserted, and the *send_to_dma_nFinishTrans* signal is asserted signaling the end of the packet transfer. If a packet is of zero length or less than the maximum packet size mentioned in the endpoint descriptor table (wMaxPacketSize), then the entire transaction to that endpoint has finished.

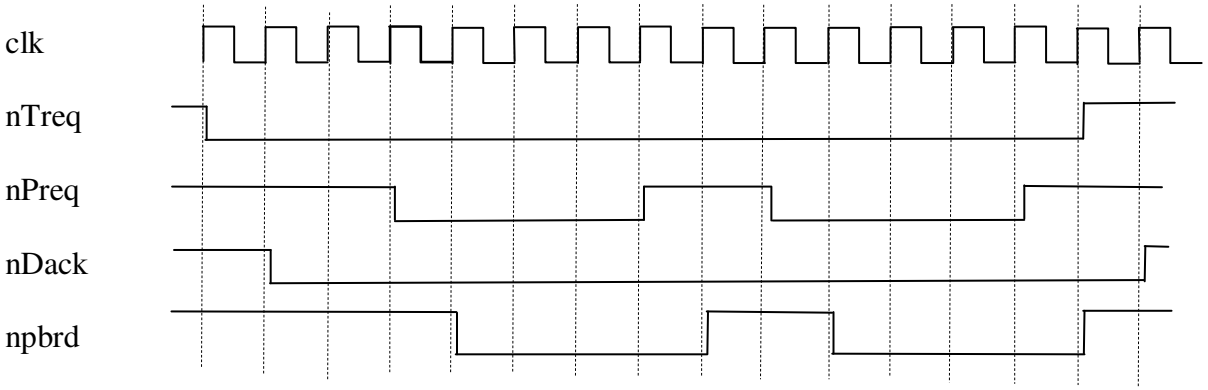

Figure 29– DMAC handshake signaling during an OUT transaction. Two packet transfers in one transaction.

After the successful data transfer, the *configured FSM* issues an ACK *Handshake packet* to the host.

IN Transaction

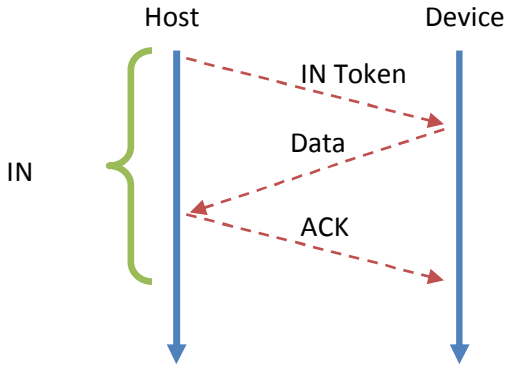

Figure 30 – OUT transaction.

The IN transactions start, just like any other transaction in the system, with a token sent from the host. The state machine in *configured FSM* waits for the UTMI signals *RXActive* and RXValid and receives the IN token. The token's fields and size are discussed in the chapter "Technical Background". The packet is checked for correct PID by comparing the first four bits with the negation of the last four bits. The CRC5 field is checked with the help of the CRC5 module. If the checks succeed, it means the token was received correctly, the data load of the packet is without errors. The data load of the packet contains the endpoint number from which the data packet would be routed from and the device address. First the state machine checks if the token is addressed to our device by comparing the device address in the token

packet with the device address saved in *dev_addr* register in the *enumeration FMS* block. If the token is addressed to our device the endpoint field is checked and saved in *epn* register. If the endpoint is zero and the endpoint state is *stall*, then a *stall* handshake is sent back to the host.

At this point, the module analyzes the PID to determine that its type is IN (1001), asserts the nTreq signal to get bus access from the µController and enables the *Receive Data from DMA* block which waits for a *data packet* to be received from the DMA. The signals diagram in Figure 31– DMAC handshake signaling during an IN transaction. Two packet transfers in one transaction. describes the transfer of a data packet from the DMA.

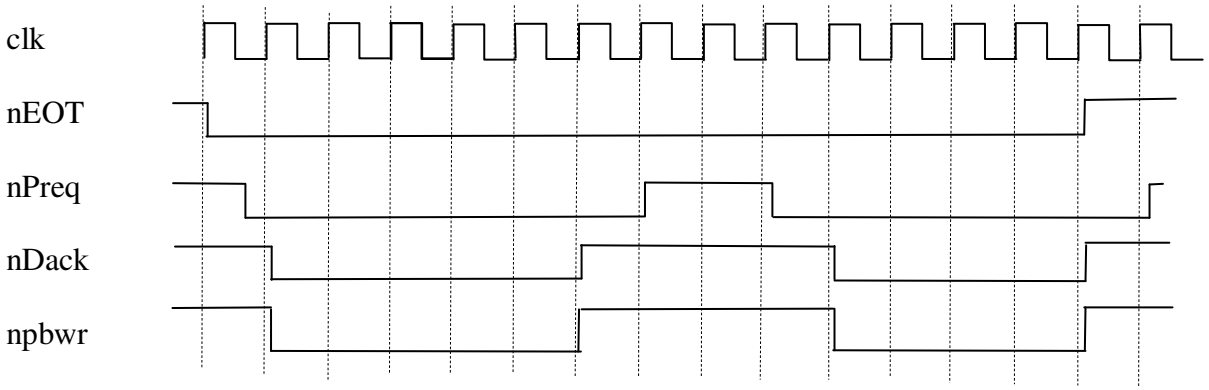

Figure 31– DMAC handshake signaling during an IN transaction. Two packet transfers in one transaction.

Once we are ready to receive data from the endpoint we assert the *nPreq* signal which requests a packet. The DMA will assert the *nDack* signal which acknowledges our request and asserts the *npbwr* signal which will enable the *EP Buffer IN* to receive the data. If an interrupt in the system will cause the µController to hand over bus control to a different peripherals in the system, then only the *npbwr* signal will deassert and reassert after the DMA will gain back its control over the bus. The *nDack* signal will deassert only after the competition of the packet transfer. If the entire transaction ended, meaning the endpoint had sent all the data it had, then the *nEOT* (End Of Transfer) signal will be asserted and bus control will be returned back to the µController. The *npbwr* signal also controls the *CRC16* enable signal. At the same time the data is transferred to the FIFO, it also enters the *data in* input of the CRC16 block. At the end of the transfer we get the CRC16 field ready for transmission.

After the transfer from the DMA ends, the *Configured FSM* uses the UTMI interface signals to send the packet to the host. Figure 32 shows the relationship between the signals during transmission to the UTMI.

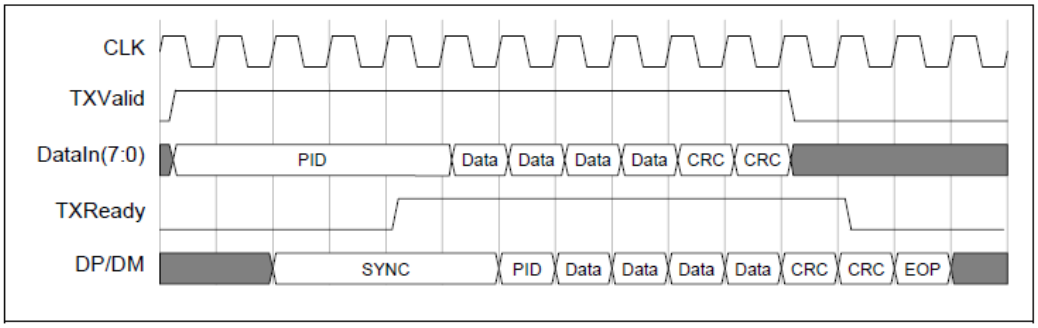

Figure 32 - UTMI transmit signals timing

Finally the packet transaction to the host starts. First, *Configured FSM* sends PID, *DATA0, DATA1, MDATA* or *DATA2*, based on the current toggle state and transaction type as described in the "Technical Background" chapter. Then the *rd* signal of the *EP Buffer IN* is asserted and the data is sent, when the FIFO's empty signal asserts, the CRC16 field is sent.

Simulation

To simulate the Protocol Engine we used Mentor Graphics Modelsim and Cadence SimVision Simulator. Unfortunately due to repeatedly computer network problems in the engineering building SimVision was rarely used.

Following are sample print screens of various transaction states of the Protocol Engine and their examination.

OUT transaction

This transaction will simulate a *data packet* sent from the host to the Bulk type endpoint number one, in the USB device at address 0001010.

Figure 33 and Figure 34 shows the simulation of the *OUT* Transaction. First the *RXActive* and *RXValid* signals are asserted by the UTMI to signal that a new transfer is about to start. Then the *PHY_DATA* bus is mounted with the *OUT token*. A *token* is composed of three bytes as described in the chapter Technical Background The first byte is *PID* and in this example it is 0xe1 or *1110 0001* which is the code for an *OUT token*. The next two bytes (0x8a and 0xf0) are device address, endpoint number and CRC5. If we break it down we can see that for 0x8a (1 0001010) the seven LSBs are the device's address and the one MSB is the LSB of the endpoint number. For 0xf0 (11110 000) the three LSBs are the MSBs of the endpoint number and the five MSBs are the CRC5.

After the token has been received the state machine checks it's validity by analyzing the PID, CRC5 and Address fields. While the *Configure FSM* block checks the validity of the *token*, the *Receive data packet* module starts to receive the *data packet* which was sent by the UTMI immediately after the token. Notice that the *nTreq* signal is asserted at this point to request bus grant from the µController.

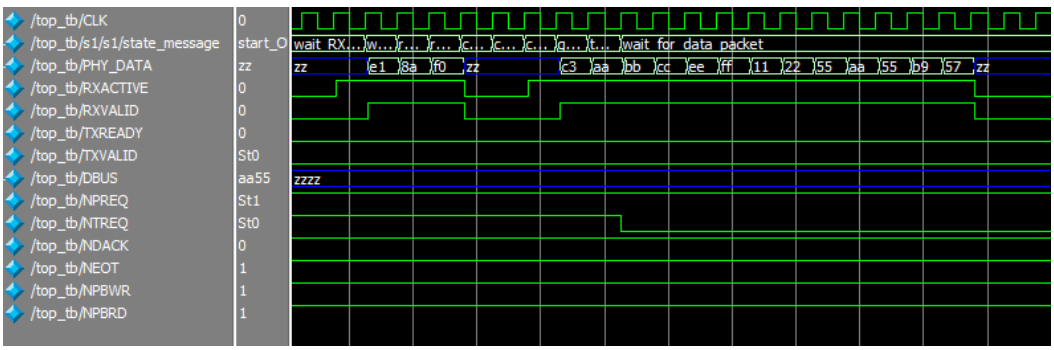

Figure 33 - Packet received from the UTMI in an OUT transaction.

The received *data packet* is composed of PID, data and CRC16 fields. In our simulation, the PID field is 0xc3 (1100 0011) which according to the PID codes table in the appendix it is the PID for DATA0. The data field of the packet is composed of ten bytes which is less than 512 bytes, the maximum packet size allowed for this endpoint. The last two bytes (0xb9 and 0x57) are the CRC16 field.

After *Receive data packet* module accepts the *data packet* and checks the CRC16 field for errors the Protocol Engine has to send the data to the endpoint through the *dbus.* The *nTreq* signal was already asserted after the token has been received correctly, and now µController asserts the *nDack* signal which grants the Protocol Engine access to the *dbus.* The Protocol Engine will signal the DMA that the data is ready to be read by asserting the *nPreq* signal. The DMA will read the data from the Protocol Engine's FIFO by asserting the *npbrd* signal.

In this simulation we also simulate an interrupt in the system that causes the µController to hand over bus control to a different peripheral in the system. When the interrupt occurs, the DMA deasserts the *npbrd* signal and the FIFO will not read more data. *npbrd* will reassert after the DMA will gain back its control over the bus.

When the FIFO empties out, the *nPreq* signal is deasserted. If the packet is less than the maximum packet size allowed for transmission to this endpoint as it is in this case, then the *nTreq* signal also deasserts and the control over the bus is handed over back to the µController.

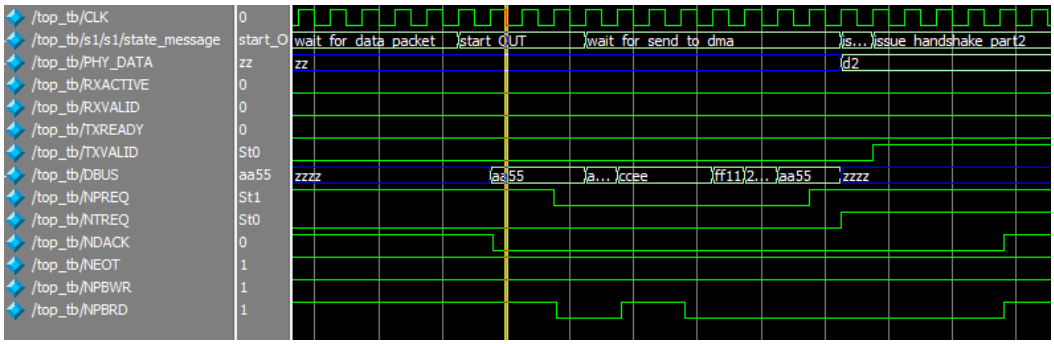

Figure 34 – Packet is sent to the DMA in an OUT transaction.

IN transaction

This transaction will simulate a *data packet* sent from the Bulk-type, endpoint number one, in the USB device at address 0001010, to the host.

Figure 33 Figure 35 shows the simulation of the IN Transaction. First the Protocol

Engine has to accept and analyze a token packet. *RXAcitve* and *RXValid* signals are asserted by the UTMI to signal that a new transfer is about to start. Then the *PHY_DATA* bus is mounted with the *IN token*. A *token* is composed of three bytes as described in the chapter Technical Background. The first byte is *PID* and in this example it is 0x69 or *0110 1001* which is the code for an *IN token* according to the PID codes table in the appendix. The next two bytes (0x8a and 0xf0) are device address, endpoint number and CRC5. If we break it down we can see that for 0x8a (1 0001010) the seven LSBs are the device's address and the one MSB is the LSB of the endpoint number. For 0xf0 (11110 000) the three LSBs are the MSBs of the endpoint number and the five MSBs are the CRC5.

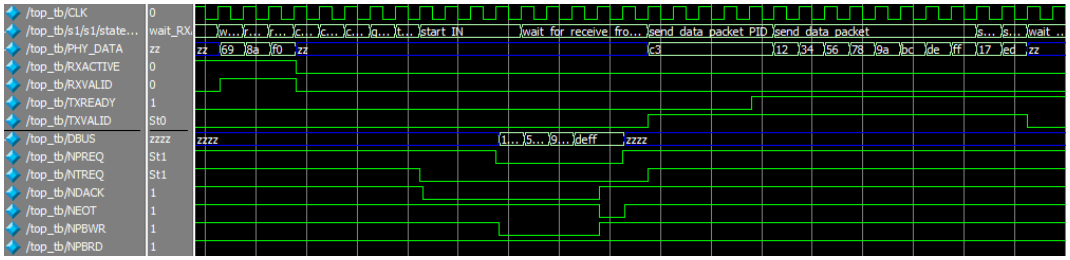

Figure 35 – IN transaction.

After the token has been received the state machine checks it's validity by analyzing the PID, CRC5 and Address fields. Notice that the *nTreq* signal is asserted at this point to request bus grant from the µController. *nDack* is asserted which means the bus is granted by the µController. The Protocol Engine then asserts *nPreq* to signal that the data is ready to be received. *npbwr* is then asserted by the DMA and the data is written to the Protocol Engine's FIFO via the *dbus*. At the same time the data is also sent to the CRC16 module which will output the CRC16 field afterwards.

Now, the Protocol Engine will send the *data packet* to the host via the *PHY_DATA*. TXReady signal is asserted to signal the UTMI that valid data is ready for transfer. The UTMI will then assert the TXValid signal and the packet would be mounted on the bus. In our simulation, the PID field is 0xc3 (1100 0011), which according to the PID codes table in the appendix is the PID for DATA0. Following the PID we send the data and then the CRC16 created by the CRC16 module.

Verification

To verify the Protocol Engine we designed, we used the Altera DE3 board with a Stratix III FPGA chip. The Stratix III is much powerful than the Cyclone FPGA chip we intended to use in the beginning. Since Cyclone II have only 68K logic elements, and since the complexity of the USB 2.0 protocol required many logic elements for implementation, the Protocol Engine could not fit into it. Therefore we switched to the Stratix III FPGA which has 338K logic elements.

Compiling and downloading the designed to the FPGA was done by the Altera Quartus II version 9.0 development software. The design was tested on the three stages of enumeration: default, addressed and configured, and for IN and OUT transactions.

For testing the Protocol Engine we created a module called *phy_dma_demo* that simulates the host and DMAC by stimulating the appropriate signals of the DMA and UTMI interfaces. Using the MegaWizard Plug In we created a static ROM within the *phy_dma_demo* module that was used to store the demonstrated token and data packets. The ROM was content was set by a pre defined MIF binary file. To connect the interface signals between the *phy_dma_demo* module and the *top* module of the Protocol Engine, we created an additional module called *demo*.

To connect the tri-state buffer of the DMA and PHY busses we assigned the 40-pin expansion header #2 GPIO1 that resides on the DE3 board. Besides connecting the interface signals further signals were output for debugging, these output signals were assigned to the leds, 7-seg displays, switches, push-buttons and some were assigned to the 40-pin expansion header #1 GPIO0.

To verify the signals that were assigned to the GPIO0 bank, we use Agilent's InfiiVision Mixed Signal Oscilloscope. To translate 4 binary digits to hexadecimal digit for use with the 7-segment display, we created a module which input is 4 bit wide and its output is connected to the 7-seg display, asserting the appropriate pins for displaying the input in hexadecimal.

By verifying the Protocol Engine on the FPGA board, we created a demonstration that emphasizes the capabilities and feasibility of the protocol engine.

Hurdles and Obstacles

High-Z pins

In a project of this magnitude and scale there were quite a few design problems and hurdles we had to overcome. One of the major issues we faced was having different output signals derive the same input signal. Since two or more output signals cannot derive the same input signal simultaneously we had to come up with a way to disconnect signals to avoid contention. The solution we used was to set the unused signals to high-z state and the remaining signal maintain connected.

This solution worked fine in the simulation we run, but caused critical warnings when we synthesized the project in Quartus II. The warning said that Quartus is substituting each high-z to a MUX. It turns out that the FPGA chip does not support high-z states and only allow them for I/O pins.

Since we did not trust Quartus to make the substitution correctly, we had to go back and dive into the code to replace all of the instances where we used high-z states, with MUXES. The only signals we left with high-z are the *PHY_DATA* and *dbus* which we connected to the device's 40-pins Expansion Header which support high-z state by the FPGA.

This solution led to a different problem in the demonstration. Since our demonstration is actually composed of a different module that drives data packets to our buses, the buses no longer use the I/O pins. To solve that, we did use the I/O pins for the buses and we also connected the output signals of the demonstration module to the same pins. In this way the PE was able to receive the data via the pins and was able to use the high-z state for bus signals.

FPGA chip

Another obstacle we faced was when we wanted to download our project to the FPGA board. The FPGA chip we chose was Cyclone II FPGA that can be found on the DE2-70 board, because of its many switches and 7-segment display units. After setting all of the signals to the switches, buttons and displays in the system exactly as the way we wanted, we compiled our project to fit to the DE2-70 FPGA board and discovered that our design could not fit into our chosen Cyclone II FPGA chip.

Apparently the Cyclone II FPGA chip on the DE2-70 board has only 68K logic elements which are simply not enough for our project. This revelation forced us to switch to the DE3 board with EP3SL340 Stratix III FPGA chip. With 338K logic elements, it has more than five times as many logic elements as the DE2-70 board. Unfortunately, the DE3 is missing many of the I/O capabilities of the DE2. DE3 board only has two 7-segment displays compared with eight of the DE2, and only 4 switches compared with sixteen of the DE2.

At this point, changing boards meant to go back and re-declare the signals and change our demonstration scheme.

USB Checker

After downloading our project to the DE3 FPGA board we had to verify our design with the USB 2.0 protocol. Checking the integrity of the data that transferred through the UTMI and DMA busses was also necessary.

USB checker which does automatic tests and provide a detailed report would be the ideal solution for verifying the Protocol Engine compliance with USB protocol.

Since USB checker wasn't at our disposal, an alternative solution had to be found. For checking the PE's interface signals with the DMAC and UTMI, we output those signals to the 40-pins Expansion Header on the DE3 board. By using the switches on the DE3 board we could check step by step that all the interface signals are asserted and the right order and time. Connecting the UTMI and DMA busses also to the 40-pins Expansion Header allowed us to check the integrity of the data being transferred, and to detect and fix bugs.

Alternative Solution

An alternative solution for a hardware based protocol engine is a firmware or software based protocol engine. The advantage in firmware base solution is that it is easier to develop, faster to implement, inexpensive and more flexible to changes. Firmware solutions can be easily modified and upgrade. With that in mind, firmware solutions also have some drawbacks. Firmware base solutions take more space than hardware solutions. In general firmware based solutions have much slower response rate than hardware based solutions, however in our case, since a high speed USB2.0 is required to deliver packets at a maximum speed rate of only 480Mbps this is not much of an issue. With an 8-bit bus, the protocol engine will have 16.66 ns to handle each word of data, which could be translated into a 60 MHz clock. Today's devices have clock rates of more than 1000 times that.

USB2.0 protocol is a well-established protocol which was adopted widely within the computer industry. Once the protocol has been set to be a standard it does not change, hence modifying the solution would not be an issue thus a firmware solution does not have a greater advantage over the hardware solution in this case.

Another disadvantage in using firmware based solution is the device's power consumption. Firmware based solutions generally consume more power than hardware based solutions, since firmware implementation uses hardware which is not designed for a specific task, resulting in less inefficient use of the hardware.

Our implementation is hardware based solution which reduces the power requirements for the device. This is a consideration especially for battery powered devices that consume more than 5V hence unable to use the USB built in power supply.

Conclusion and Summary

Work on the project was divided into several steps. The first step was to read and learn the USB2.0 specifications. We read the relevant chapters dealing with the protocol, mainly chapter three, five, eight and nine, and also other smaller parts of the specification. We also found quite a few websites online that tried to simplify the protocol.

The second step was to write a specification file for our device based on what we read from the USB specifications.

The next step was to translate everything we learnt from the specification into finite state machines using Visio drawings. This part made us dive even deeper into the specification and understand it to its fool extent. The protocol deals with many situations and has a lot of features which can be used by the devices and the host. We had to sift through all of these features, and find the ones that are related to the normal operation of the device.

The fourth step was drawing a top level block diagram that describes all the modules and signals to that will be used in the system.

The fifth step was using Verilog to write the different module of the system. We first had to learn the language since it was the first time we ever used it. Also, each module we created was tested and simulated using a test bench we wrote.

The next step was to test the entire system using a test bench in Modelsim or Simvision. We created a test bench with many different packet IN and OUT transactions and tested our code. We found all kinds of problems and bugs that were related to the synergy between the different modules and interfaces. At this point we also inserted the signals we needed for the demonstration on the FPGA board.

Our next step was to use Quartus to synthesize the project and get it ready for download to the FPGA board. At this step we found a major flaw in our design and it is described in the chapter Hurdles and Obstacles.

Finally we tested our project on the board itself and wrote the project's book.

It was interesting to work on a project related to something so common that we see and use every day.

The benefits of having a USB2.0 protocol engine implemented in hardware are clear. These benefits, already discussed in the chapter Project's Motivation, will surly stand out in any commercial protocol engine.

This project hasn't denied the feasibility of hardware based USB2.0 protocol engine in principle. There are still more advanced functions which were not implemented and are required for a commercial protocol engine. Also, there are quite a few important steps before this project could be implemented in ASIC. With this in mind, it is still early to defiantly say that a hardware based USB2.0 protocol engine chip is in hand's reach.

A Look into the Future

ASIC implementation:

Since the USB2.0 standard is widely used in many peripheral devices. It is well established and has already proved itself to be reliable and efficient. The protocol's strict restrictions assure that different devices are compatible in every machine. This assures that the Protocol Engine we designed can be used in various products that uses USB2.0 standard as their chosen protocol. The programmable *SRAM* is designed to be able to hold any kind of descriptor tables for any product and can support up to 15 IN and 15 OUT endpoints with each of four different interfaces.

Having a well established protocol allows the Protocol Engine to be implemented in ASIC, Cadence Encounter Synthesis tool, Low-Power Kit, System-On-Chip Functional Verification Kit and VirageLogic 65nm Libraries.

The chip would have many advantages over other software based commercial products. Unlike other Protocol Engines out in the market, this Protocol Engine is hardware based and thus consumes less power. This feature is a great advantage especially in portable machines that use USB devices such as a graphical engine "on key".

USB 3.0

Furthermore, USB3.0 spec has been published and not long from now the computer industry will adopt this standard and move forward to the more advance protocol. A future improvement for the Protocol Engine is to support USB3.0 features and speed.

Protocol Engine with DMA Controller

An advance version of the Protocol Engine would be an assimilation of the DMA Contorller into it. Including the DMA Controller would simplify the interface with the device’s bus and reduce chances of bus contention by upgrading the synchronization capabilities of the Protocol Engine in the USB device.

Bibliography

USB Manuals, Tutorials and Specs:

USB Made Simple:

http://www.usbmadesimple.co.uk/index.html

USB in a NutShell:

http://www.beyondlogic.org/usbnutshell/usb1.shtml

USB 2.0 Specifications:

http://www.usb.org/developers/docs/usb_20_081810.zip

NET2272 - USB 2.0 Peripheral Controller

http://vital-ic.ru/chip/data/NET2272F.pdf

UTMI

USB 2.0 Transceiver Macrocell Interface (UTMI):

http://www.intel.com/technology/usb/download/2_0_xcvr_macrocell_1_05.pdf

CRC

A Fast CRC Update Implementation:

http://ce.et.tudelft.nl/publicationfiles/805_404_lu.pdf

Algorithms for Cyclic Redundancy Code (CRC) Computation:

http://ishaksuleiman.tripod.com/00000.pdf

A Practical Parallel CRC Generation Method:

http://outputlogic.com/my-stuff/circuit-cellar-january-2010-crc.pdf

Altera Quartus and FPGA manuals

Altera Configuration Handbook Device Configuration Options:

http://www.altera.com/literature/hb/cfg/cfg_cf52006.pdf

DE2-70 Manual - DE2 Development and Education Board User Manual:

http://csg.csail.mit.edu/6.375/6_375_2010_www/handouts/other/DE2Manual.pdf

DE3 Manual - DE3 Development and Education Board User Manual:

http://www.terasic.com.tw/attachment/archive/260/DE3_User_manual_v1.2.3.pdf

FPGA Incremental Compilation—Divide and Conquer:

http://www.altera.com/literature/cp/cp-01001.pdf

Stratix III Device Handbook:

http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf

Altera Quartus Handbook Recommended HDL Coding Styles:

http://www.altera.com/literature/hb/qts/qts_qii51007.pdf

Verilog Tutorials and Manuals

USB Complete – Everything You Need to Develop USB Peripherals by Jan Axelson.

Verilog HDL - A Guide to Digital Design and Synthesis Samir Palnitkar.

Verilog Tutorial:

http://www.asic-world.com/verilog/intro1.html

Appendix

Standard Device Descriptor Table

Offset	Field	Size	Value	Description
0	bLength	1	0x12	Size of this descriptor, in bytes.
1	bDescriptorType	1	Constant	DEVICE Descriptor Type
2	bcdUSB	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	bDeviceClass	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	bDeviceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB-IF.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID (assigned by the USB-IF)
10	idProduct	2	ID	Product ID (assigned by the manufacturer)
12	bcdDevice	2	BCD	Device release number in binary-coded decimal
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer
15	iProduct	1	Index	Index of string descriptor describing product
16	iSerialNumber	1	Index	Index of string descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

Table 10 – Standard Device Descriptor Table

Configuration Descriptor Table

Offset	Field	Size	Value	Description
0	bLength	1	0x0A	Size of this descriptor, in bytes.
1	bDescriptorType	1	0x02	CONFIGURATION descriptor
2	wTotalLength	2	0x006D	Length of the total configuration block, including this descriptor, in bytes
4	bNumInterfaces	1	0x04	This device has four interfaces
5	bConfigurationValue	1	0x01	ID of this configuration
6	iConfiguration	1	0x00	Unused, no string descriptor
7	bmAttributes	1	0x80	Bus-powered device, no remote wakeup capability
8	bMaxPower	1	0xFA	500mA maximum power consumption (in 2mA units)

Table 11 – Configuration Descriptor Table

Interface Descriptor Table

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor, in bytes.
1	bDescriptorType	1	0x04	INTERFACE descriptor type
2	bInterfaceNumber	1	Number	Number of this interface. (zero-base value)
3	bAlternateSetting	1	0x00	Default setting
4	bNumEndpoints	1	Number	Number of endpoints used by this interface (excluding endpoint 0)
5	bInterfaceClass	1	0x00	Unused
6	bInterfaceSubClass	1	0x00	Unused
7	bInterfaceProtocol	1	0x00	Device does not use class-specific protocols on a device basis.
8	iInterface	1	0x00	Unused, no string descriptor

Table 12 – Interface Descriptor Table

Endpoint Descriptor Table

Offset	Field	Size	Value	Description
0	bLength	1	0x07	Size of this descriptor, in bytes.
1	bDescriptorType	1	TBD	ENDPOINT descriptor type (5- Endpoint)
2	bEndpointAddress	1	TBD	Bit 7 Direction 0 = Out, 1 = In. Ignored for control endpoints. Bits 0..3 Endpoint number = 1
3	bmAttributes	1	TBD	Bits 0..1 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronization Type (Iso Mode) 00 = No Synchronization 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4 = Usage Type (Iso Mode) 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = Unused
4	wMaxPacketSize	2	TBD	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-(micro)frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For all endpoints, bits 10..0 specify the maximum packet size (in bytes). For high-speed isochronous and interrupt endpoints: Bits 12..11 specify the number of additional transaction opportunities per microframe: 00 = None (1 transaction per microframe) 01 = 1 additional (2 per microframe) 10 = 2 additional (3 per microframe) 11 = Reserved Bits 15..13 are reserved and must be set to zero.
6	interval	1	TBD	Interval for polling high BW endpoint for data. Period = $2^{bInterval-1}$ NAK/ACK per 1µFrame (125µSec)

Table 13 – Endpoint Descriptor Table

PID Codes Table

PID	Code (bits)
OUT	0001
IN	1001
SOF	0101
SETUP	1101
PING	0100
DATA0	0011
DATA1	1011
DATA2	0111
MDATA	1111
ACK	0010
NACK	1010
STALL	1110
NYET	0110

Table 14 – PID Codes Table

Altera DE3 Stratix III FPGA board Top view

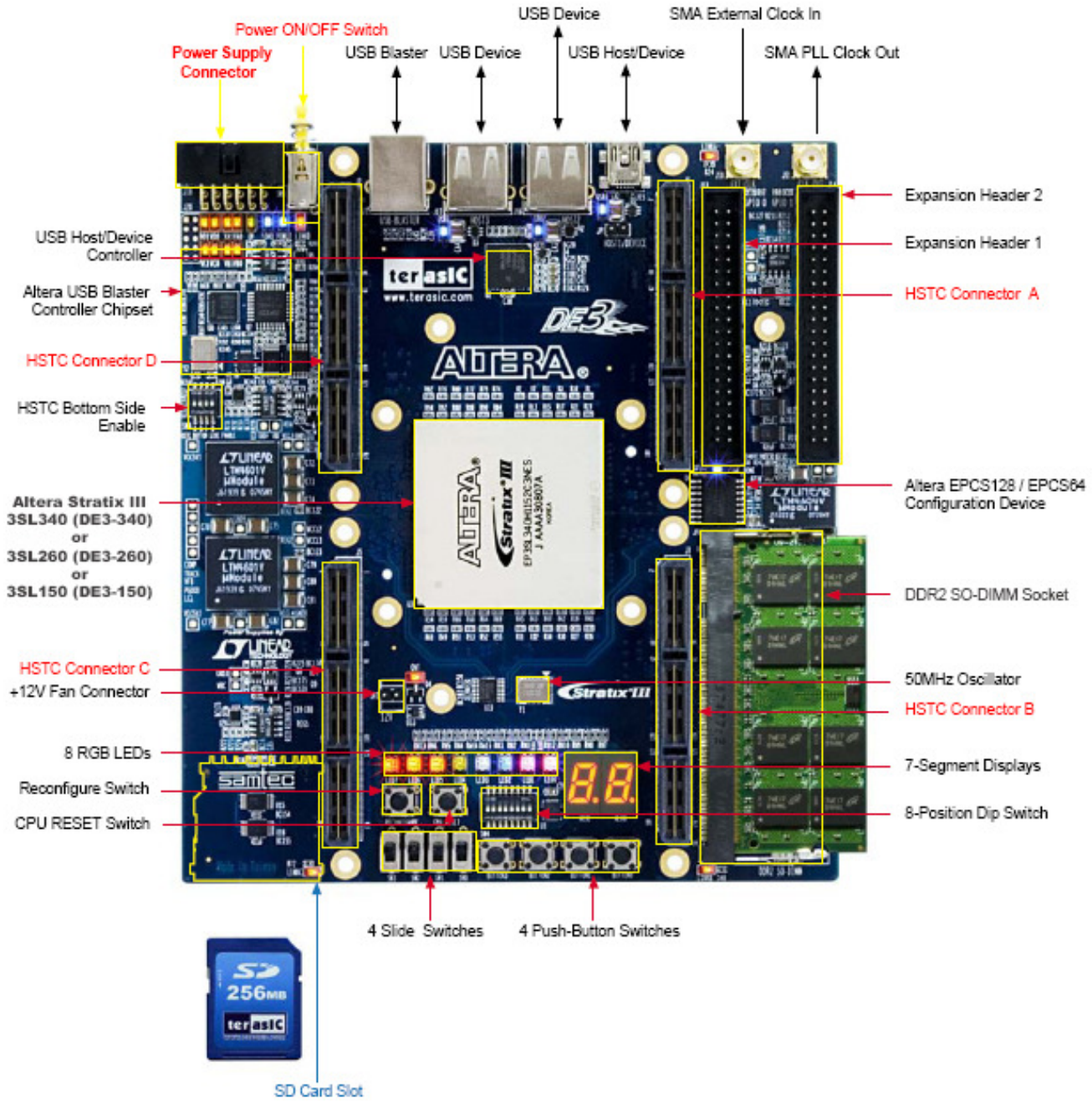

Figure 36 – Altera DE3 Stratix III FPGA board Top view.

Pin mapping table for verification and demonstration

Signal Name	FPGA Pin No.	Verilog name
SW[0]	W5	rom_addr[0]
SW[1]	W6	rom_addr[1]
SW[2]	W9	rom_addr[2]
SW[3]	W11	rom_addr[3]
CPU RESET	U31	nrst
OSC2_50	W2	clk
BUTTON[0]	K1	go0
BUTTON[1]	K2	go1
LED [0] (Blue color)	AB5	led0
LED [1] (Blue color)	AB6	led1
LED [4] (Green color)	Y3	led4
LED [5] (Green color)	W3	led5
LED [6] (Green color)	AA4	led6
LED [7] (Green color)	Y4	led7
HEX0_D[0]	W12	HEX0_D[0]
HEX0_D[1]	Y11	HEX0_D[1]
HEX0_D[2]	W10	HEX0_D[2]
HEX0_D[3]	W8	HEX0_D[3]
HEX0_D[4]	W7	HEX0_D[4]
HEX0_D[5]	Y5	HEX0_D[5]
HEX0_D[6]	Y6	HEX0_D[6]
HEX0_DP	V3	HEX0_DP

Signal Name	FPGA Pin No.	Verilog name
HEX1_D[0]	P3	HEX1_D[0]
HEX1_D[1]	N4	HEX1_D[1]
HEX1_D[2]	N3	HEX1_D[2]
HEX1_D[3]	N1	HEX1_D[3]
HEX1_D[4]	M1	HEX1_D[4]
HEX1_D[5]	L1	HEX1_D[5]
HEX1_D[6]	L2	HEX1_D[6]
HEX1_DP	V4	HEX1_DP
GPIO1_D0	AC26	PHY_DATA[0]
GPIO1_D1	AC25	PHY_DATA[1]
GPIO1_D2	AE28	PHY_DATA[2]
GPIO1_D3	AD27	PHY_DATA[3]
GPIO1_D4	AE27	PHY_DATA[4]
GPIO1_D5	AD26	PHY_DATA[5]
GPIO1_D6	AD29	PHY_DATA[6]
GPIO1_D7	AF29	PHY_DATA[7]
GPIO1_D8	AD28	dbus[0]
GPIO1_D9	AF28	dbus[1]
GPIO1_D10	AB27	dbus[2]
GPIO1_D11	AE30	dbus[3]
GPIO1_D12	AB26	dbus[4]
GPIO1_D13	AE29	dbus[5]
GPIO1_D14	AB25	dbus[6]
GPIO1_D15	AB24	dbus[7]
GPIO1_D16	AM21	dbus[8]
GPIO1_D17	AD31	dbus[9]
GPIO1_D18	AP20	dbus[10]
GPIO1_D19	AD30	dbus[11]
GPIO1_D20	AL22	dbus[12]
GPIO1_D21]	AJ20	dbus[13]
GPIO1_D22	AM22	dbus[14]
GPIO1_D23	AJ21	dbus[15]

Table 15 – Pin mapping table for verification and demonstration

Schematic diagram of the 7-segment displays

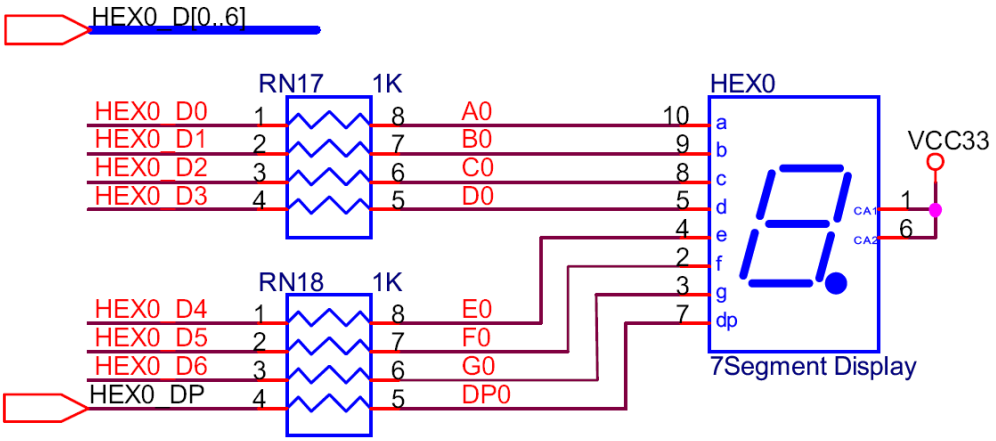

Figure 37 – Schematic diagram of the 7-segment displays

Hexadecimal to 7-seg translation table

All pins are Active Low

	segments numbers								
Hex Number	6	5	4	3	2	1	0	Drive	Note
0	1	0	0	0	0	0	0	7'h40	
1	1	1	1	1	0	0	1	7'h79	
2	0	1	0	0	1	0	0	7'h24	
3	0	1	1	0	0	0	0	7'h30	
4	0	0	1	1	0	0	1	7'h19	
5	0	0	1	0	0	1	0	7'h12	
6	0	0	0	0	0	1	0	7'h02	
7	1	1	1	1	0	0	0	7'h78	
8	0	0	0	0	0	0	0	7'h00	
9	0	0	1	0	0	0	0	7'h10	
A	0	0	0	1	0	0	0	7'h08	
B	0	0	0	0	0	1	1	7'h03	small b
C	1	0	0	0	1	1	0	7'h46	
D	0	1	0	0	0	0	1	7'h21	small d
E	0	0	0	0	1	1	0	7'h06	
F	0	0	0	1	1	1	0	7'h0E	

Table 16 – Hexadecimal to 7-seg translation table

Bulk Interrupt IN Transfer state machine

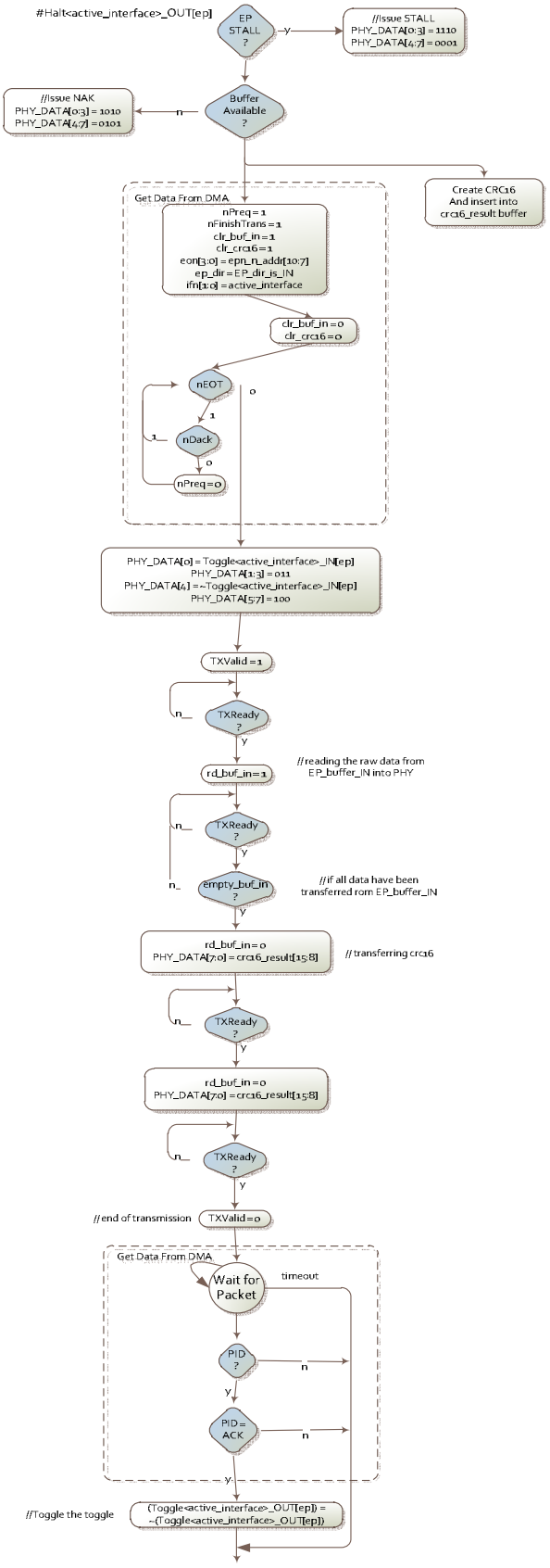

Figure 38 – Bulk Interrupt IN Transfer state machine

Isochronous IN Transfer state machine

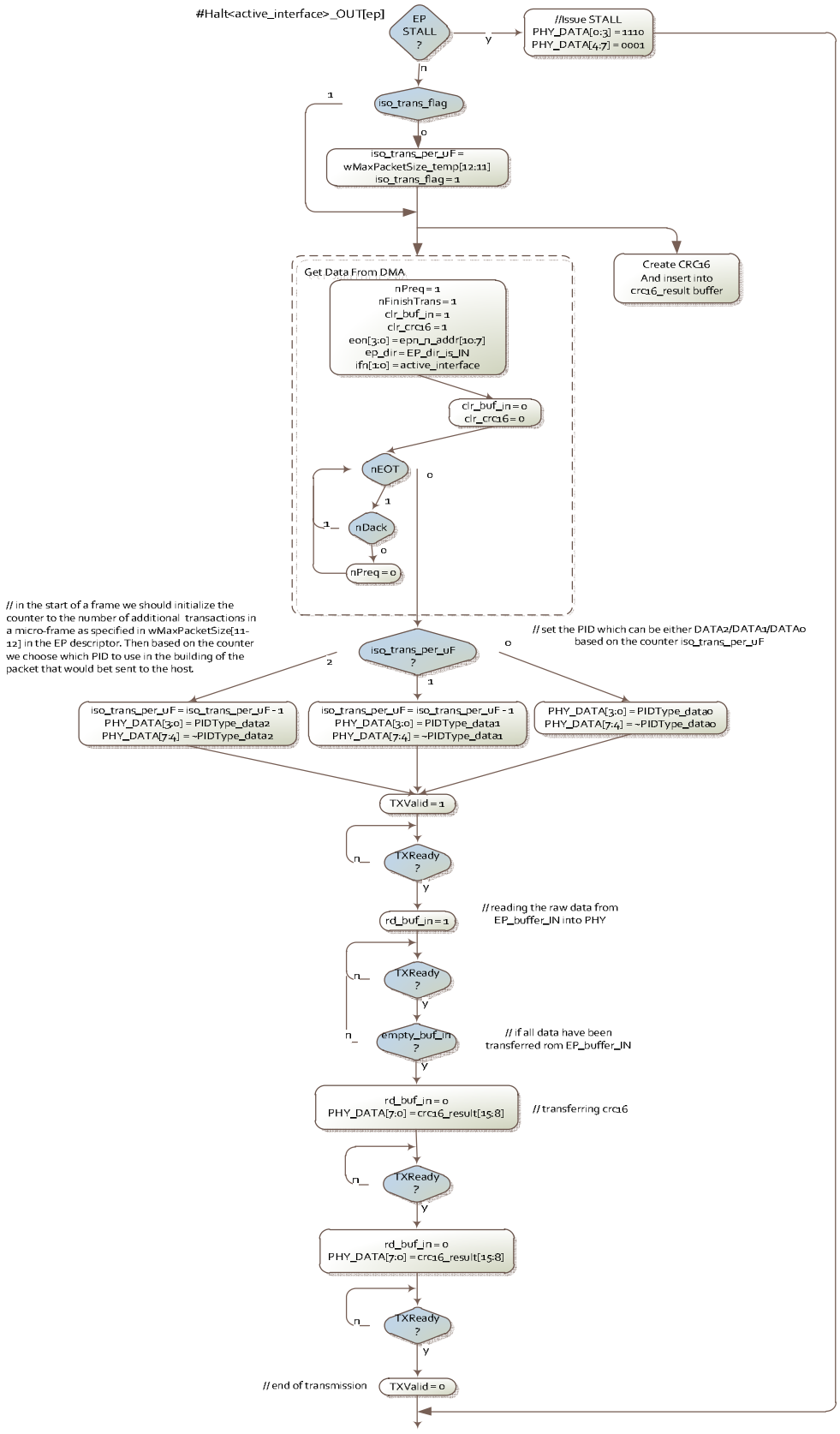

Figure 39 – Isochronous IN Transfer state machine

Bulk OUT Transfer state machine

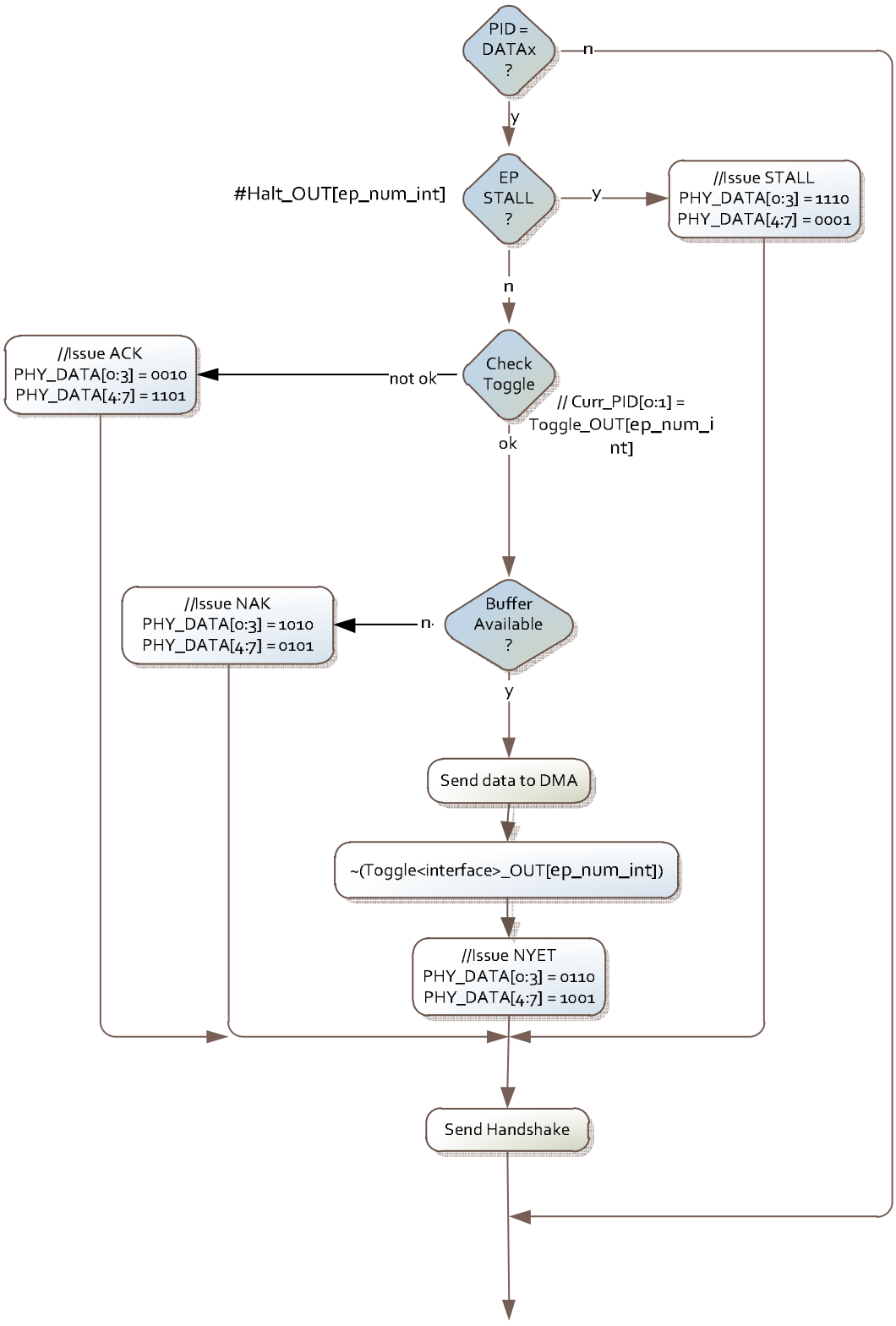

Figure 40 – Bulk OUT Transfer state machine

Interrupt OUT Transfer state machine

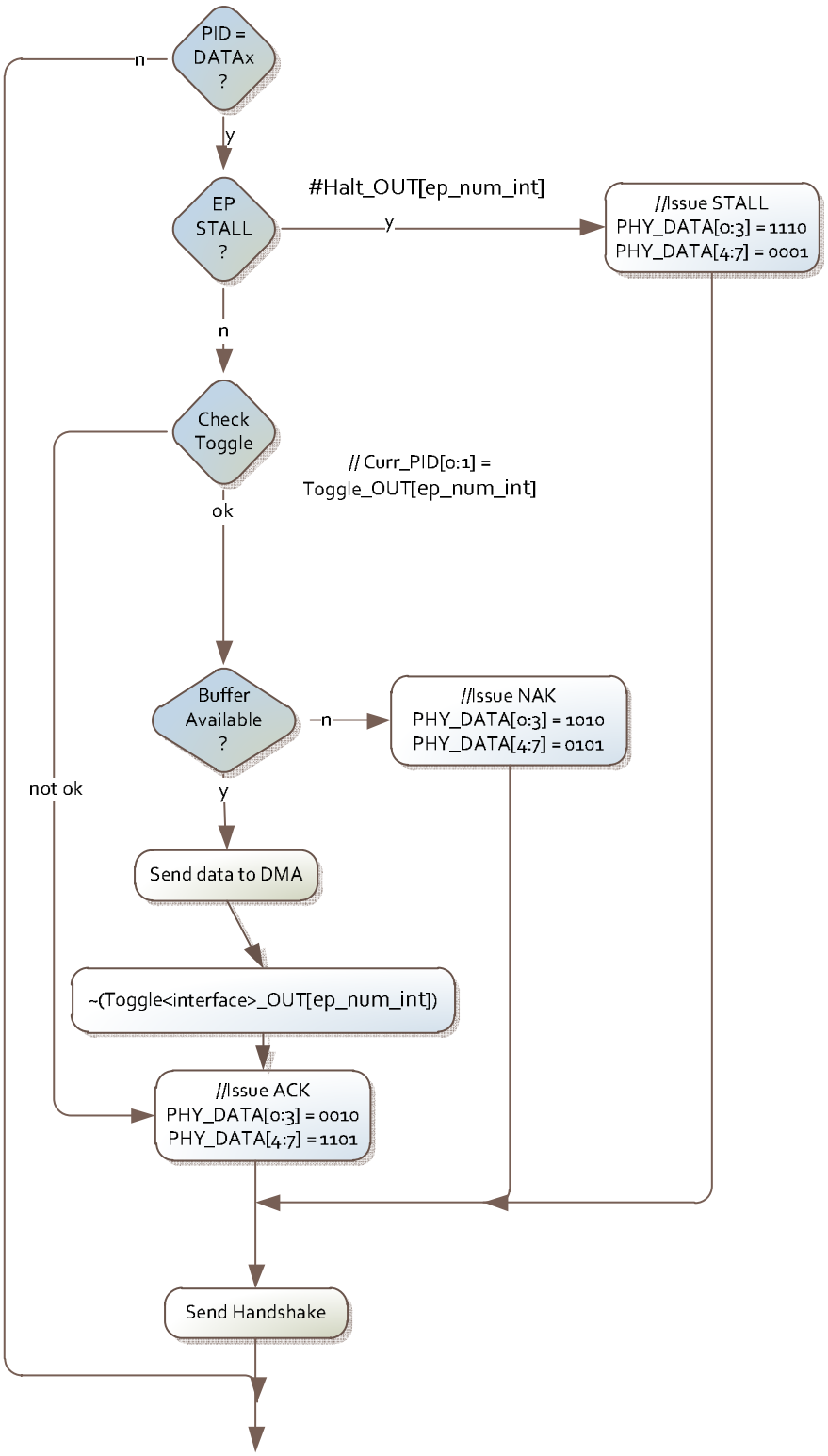

Figure 41 – Interrupt OUT Transfer state machine

Isochronous OUT Transfer state machine

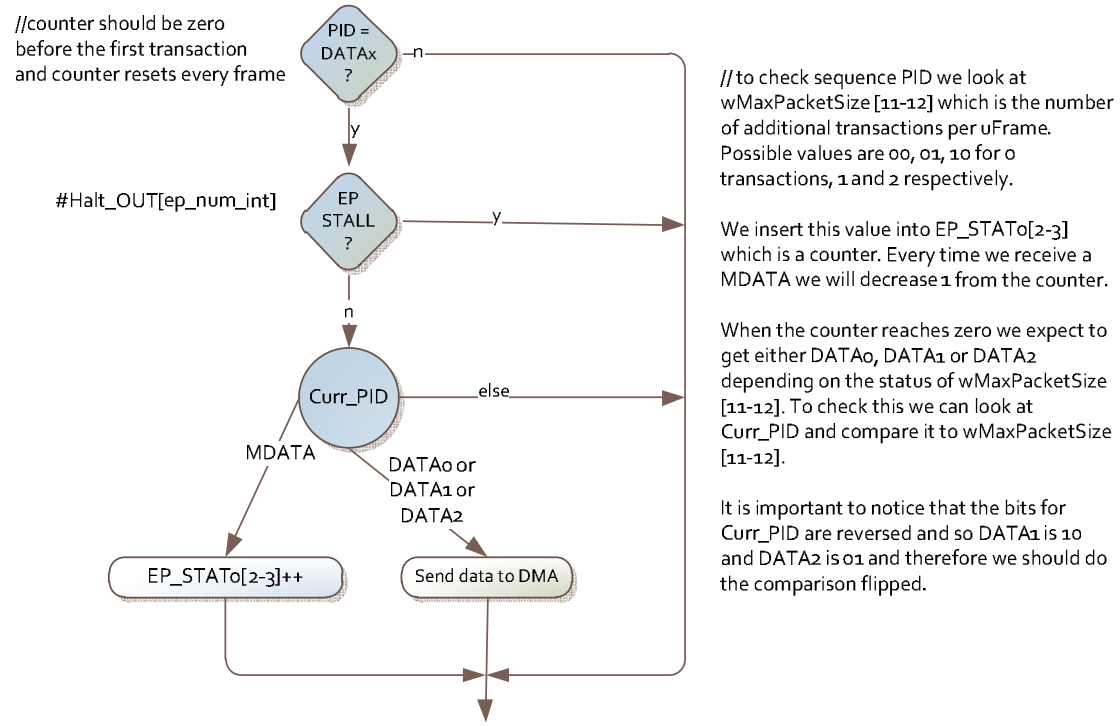

Figure 42 – Isochronous OUT Transfer state machine

Device Enumeration state machine

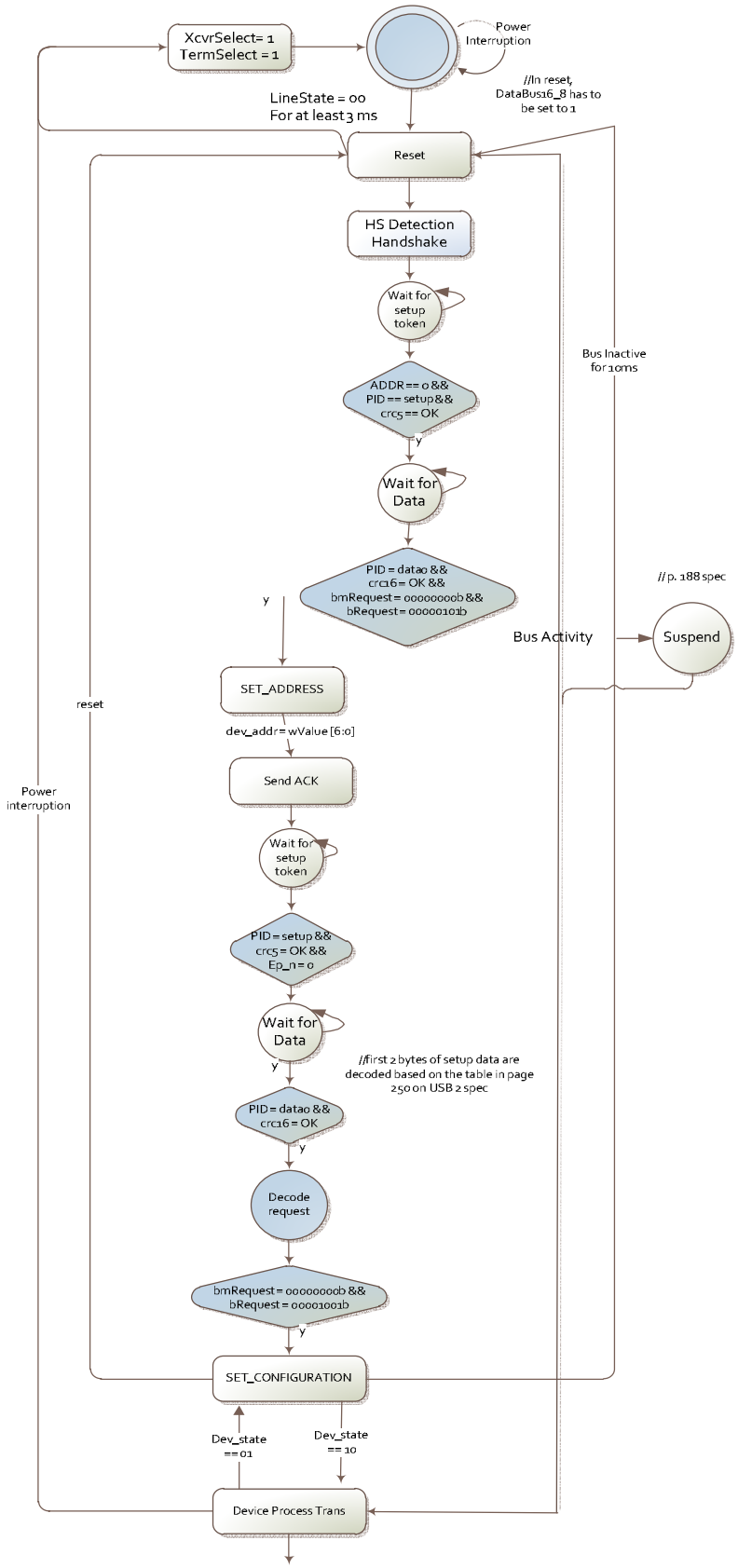

Figure 43 – Device Enumeration state machine