# Table of Contents

Table of Contents ................................................. ii
List of Tables ....................................................... iv
List of Figures ....................................................... v
Acknowledgements ................................................... vi

Introduction ......................................................... 1

1 USB ........................................................................ 3
1.1 Introduction ....................................................... 3
1.2 The Physical Layer ............................................... 5
  1.2.1 Signaling on the bus ........................................ 5
  1.2.2 HC - Host Controller ....................................... 6
  1.2.3 The USB System Software .................................. 7
1.3 The Protocol Engine Layer ....................................... 8
  1.3.1 The USB Logical Device ................................... 8
1.4 The Application Layer ........................................... 9
  1.4.1 The USB Protocol ........................................... 9
  1.4.2 Packet Field Formats ...................................... 11
  1.4.3 Packet Formats ............................................. 14
  1.4.4 Transaction Format ........................................ 16

2 Architecture ......................................................... 21
2.1 Communication Channels ........................................ 22

3 Operation .......................................................... 25
3.1 Main flow reset ................................................... 25
3.2 Sync Process ...................................................... 27
3.3 Start of Frame .................................................... 28
### Contents

3.4 CRC ........................................................................... 29  
3.4.1 CRC-5 ............................................................... 30  
3.4.2 CRC-16 ............................................................ 31  
3.5 Data Sending and Receiving ......................................... 32  
3.6 Conclusion ............................................................... 32  

4 Future Work and Critical Analysis ................................. 33  

Literature survey .......................................................... 34
List of Tables

3.1 Signal levels of the Bus ..................................... 28
List of Figures

1.1 Sync Packet ......................................................... 12
1.2 PID Packet ......................................................... 12
1.3 Address Packet ..................................................... 13
1.4 EndPoint Field ...................................................... 14
1.5 token Packet ....................................................... 15
1.6 SoF Packet ........................................................ 15
1.7 Data Packet ......................................................... 15
1.8 Handshake Packets ............................................... 16
1.9 Control Transfer ................................................. 18
1.10 Interrupt Transfer .............................................. 20
1.11 Isochronous Transfer ........................................... 20

2.1 arch ............................................................... 21
2.2 Communication Channels ...................................... 23
2.3 Typical List Structure .......................................... 24

3.1 Main flowchart ..................................................... 25
3.2 flownotmain ....................................................... 27
3.3 Start of Frame .................................................... 29
3.4 Data Field Format ............................................... 32
Acknowledgements

I would like to thank to professor Ph. D. Rajesh Gupta, for his many suggestions and support during this research. I am also thankful to all the colleagues that attended the ICS 213 class, for the all the kindness.
Introduction

In these days, in what concerns to embedded systems, one of the most important parts are the device drivers, that are the link between the software and the hardware. The main reason is due to the dropping of the prices of the hardware, causing an increasing of productivity and widely use of of systems on chip. Leaving a great work to do, to develop the interfacing of hardware with the software applications.

So, it’s one of the goals of the project to develop a library of device drivers, that could be used by another designers.

In this project, was determined to add a new feature to the Palm Vx, with the Palm OS. This new features will provide the Plam with an USB bus (using and controlling the bus).

This project is divided in tree study blocks. One is the development of the Host Controller, implemented in VHDL. The second is the Host Controller Driver, this piece of software is the one that will make the interface between the hardware and the application software and usually is developed using C language. The final parte will be to produce an application for a specific device.
The function of the Host controller is to control and exchange data between a set of devices that are sharing the same bus. The host controller is needed because neither the hosts can communicate between each others, neither the devices. So using the USB protocol we can only have in the same bus an Host and several devices.

The scope of this report will be about the implementation of the host controller, leaving the others issues open to be discuss in a possible update of this report.

In this document I wish to be able to explain all my work in the pass month and, to accomplish that I’ve divided this document in four parts. The first gives a simple explanation of the USB protocol. The second gives an overall explanation on the architecture. The third describes the operations and the flow of the code that was develop so far. The last part describes the work that should be done in order to finish this part and some ideas to accomplish that.

The documentation that was used, was provided by Intel corporation and Microsoft.
Chapter 1
USB

The Universal Serial Bus was invented and standardized by a group of computer and peripherals manufactures in 1995. The idea was to take the whole area of serial port and serial bus and update it with the twenty-first century technology.

It’s true that there were many standards of communication between host computers and peripherals, but the goal was to create a technology that combines low speed and high speed bus activity. The technology enables shared access for both speeds, a technology which provides robust protocol, automatic configuring of devices and a serial bus which is simplified and easy to plug into. All those requirements were met with the USB standards.

1.1 Introduction

It is important to remember the USB is not a serial port it is a serial bus, a fact that enables a single port on the computer to be a link for a myriad of devices, (up to 127 devices in a USB system). We can easily chain one device to another and use one port as a connecting point of many devices by using a hub. All these enables us to look at the USB system as a small network of devices.

The plug and play capability of the USB is one of its advantages over other serial buses. This capability enables automatic detection of a new device, which is
attached into the system, an automatic configuration of it by the host, and an au-
tomatic detection of it’s detachment from the system. The flexible attachment and
detachment of devices to and from the system allows mobility on the bus and adjust-
ment of the system to new devices without the need of restart the whole system each
time a new device is detected. Another important aspect of the USB is it’s mid and
high speed flexibility. This feature refers to the ability of the USB to support simul-
taneously medium-speed devices, (which work in 1.5Mbps), and high-speed devices,
(which work in 12Mbps), this for the version 1.0 of the USB protocol.

The simultaneously work of the USB system finds expression also in the dual support
in both isochronous and asynchronous bandwidth allocation methods. Isochronous
means that the necessary bandwidth is guaranteed, whenever the device requires it
it will be available. Asynchronous on the other hand means that there is no guar-
antee the data will be sent whenever it will be possible to send it. Devices, such
as video and audio multimedia, that use stream transfer, will use the isochronous
method while devices that use bulk transfer, such as printers and scanners will use
the asynchronous method.

The USB is robust. Through all the different protocol layers there is an error de-
tection and recovery mechanism, which guarantees low error rate. The USB provides
detection of faulty devices and flow control mechanism, which is built in the protocol.

A typical USB system consists of:

**One host** → there is only one host in the USB system, which is responsible to the
whole complexity of the protocol (simplifies the designing of USB devices). The host
controls the media access no one can access the bus unless it got an approval required
from the host.

**Hub** → like the hubs used for computer network. The hub provides an intercon-
nect point, which enables many devices to connect to a single USB port. The logical
topology of the USB is a star structure, all the devices are connected (logically) di-
rectly to the host. It is totally transparent to the device what is its’ hub tier (the
number of hubs the data has to flow through). The hubs’ main functionality is the
responsibility of detecting an attachment and detachment of devices, handling the
power management for devices that are bus-powered (get power from the bus), and
responsibility for bus error detection and recovery. Another important role of the hub is to manage both full and low speed devices. When a device is attached to the system the hub detects the speed, which the device operates in, and through the whole communication on the bus prevents from full speed traffic to reach low seed device and vice versa prevent from low speed traffic to reach full speed device.

Device → everything in the USB system, which is not a host, is a device (including hubs). A device provides one or more USB functions. Most of the devices provide only one function but there may be some, which provides more than one and called compound devices. We refer to two kinds of devices - self powered or bus powered devices. A device that gets its power from the bus is called bus powered and on the other hand a device which supplies its own power is called self powered. As was mentioned before there are two kinds of devices: Full-speed devices operates in 12Mbps; Low-speed devices that work in 1.5Mbps.

The logic communication between the client software on the host and the function on the device is done through pipes. A pipe is the association between a specific endpoint on the device and the appropriate software in the host. An endpoint is the source or destination of the data that transmitted on the USB cable. An interface is composed of endpoints grouped together into a certain set. The client software wishes to transmit data between the buffers in the host and the endpoints in the device and by that manages the specific interface (which is associated with specific endpoints). Communication flow on the bus can be done is two directions: OUT - data flows from the host to the device or, IN - data follows from the device into the host.

1.2 The Physical Layer
1.2.1 Signaling on the bus

The physical layer is the physical interface to the USB cable. The main responsibility of the physical layer is to transmit 0 as 0 and 1 as 1 and to receive 0 as 0 and 1 as 1. The USB cable is 4 wire cable, signaling on the bus is done by signaling over two
wires (differential pair). There is a D+ wire and a D- wire, in a way that if we want to transmit 0 over the bus we will keep D+ low and D- high and vice versa to transmit 1 we need to keep D- low and D+ high. The other two cables are Vbus (+5v) and GND (5v) to deliver power to the device. Bits are send into the bus LSB first.

Data encoding and decoding is done using NRZI method. In NRZI coding if we want to transmit 1 we dont change the level of the signaling (if the differential pair represented logic 1, it will remain in this level also for the next clock), on the other hand if we want to transmit 0 we will flip the value of the differential pair. (There will be toggle in the level in a way that if the current value represented is differential 1 the next value will be differential 0).

One of the affects of the data encoding represented above, is that sending a string of ones will cause a continuous mode of transmission (the transmission lines will stay static with no change for that period). In order to prevent such a continuous state, bit staffing is performed before the NRZI decoding. The bit staffing is made by inserting a zero after six successor ones. In the decoder that zero is recognized as part of the bit stuffing and is ignored.

### 1.2.2 HC - Host Controller

The host is the "smartest" element in the USB system and plays a unique role in the system. The host initiates all the transactions, controls the media access and is the main engine for the protocol’s flow, as we will see later on. That is why the host controller, an additional hardware, is required to ensure that everything, which is transmitted on the bus, is correct and within the specifications.

The host controller serves both the USB and the host and has the same functionality in every USB system:

- **Frame generation:** The host controller is responsible to partition the USB time into time units, in a way that each time unit is 1msec and is called a "frame". The host controller issues, periodically, SOF (Start Of Frame) packet every
1msec (after the transmission of the SOF the HC can transmit any other trans-
action for the rest of the frame period). The SOF contains the current frame
number, which is maintained by the HC.

- **Data Processing:** The HC handles the requests for data to and from the host.

- **Protocol Engine:** Handling the USB protocol level interface.

- **Error handling:** The HC handles errors such as:
  
  - Timeout - the function in the device is not responding
  
  - CRC error
  
  - Unexpected data payload

- **Remote wakeup:** The HC is able to enter the USB into a suspend state, and to
detect a remote wakeup signaling on the bus.

### 1.2.3 The USB System Software

The USB system is also responsible for the bandwidth allocation and bus power man-
agement in order to enable devices to access the bus. The USB System Software is
composed of the host software and two additional software interfaces:

**The Host Controller Driver (HCD):** is an interface to the host controller. The
purpose of using such an interface is to make it transparent for the host software
which host controller the device is connected to.

**The USB driver (USBD):** The client software (the top layer of the host communi-
cations layer) request data from the USBD in a form of IRPs (I/O Request Packets)
which consist of a request to send/receive data through a certain pipe. The USBD handle those requests. Another important role of the USBD is to supply the client software a general description of the device which the software is about to handle. The USBD is required to handle the enumeration process (a process which is activated the moment the device is attached to the bus and in the end of it the device is fully configured, is a part of the USB system and can response to the traffic on the bus), investigate the different configurations of the device and supply this knowledge to the client software. As a part of this role the USBD owns the default pipe, since when a device just enters the system, the only way to communicate with it is through the default pipe.

1.3 The Protocol Engine Layer

The middle layer in the communication layers model has an important role. The layer is responsible for the translating the data between the application layer (client software on the host and function on the device) and the USB transactions protocol. The layer wraps and unwraps the data according to the protocol.

The layer is referred differently in the USB host (is called the "USB system software" layer) and in the USB device (is called "USB logical device" layer), which is quite reasonable due to the different roles the two components perform in the system.

1.3.1 The USB Logical Device

The USB logical device is composed of a collection of independent endpoints. Each endpoint is given a unique address (endpoint number) at the design time, the USB logical device is also uniquely addressed at the end of the enumeration process. An endpoint is unidirectional (except from endpoint number zero), it may be IN type (supports data transfer from the device to the host) or OUT type (supports data transfer from the host to the device), this means that for bi-directional flow we need two endpoints each for a different direction. The combination of the USB logical device address, the endpoint number and the direction of the endpoint define uniquely a certain endpoint. An endpoint is characterized with a transfer type. As we will see later on, there are four types of transfers on the USB, each endpoint is associated
with only one transfer type, and by that characterized with its bandwidth allocation requirements.

All USB devices must support communication through the default pipe. The default pipe plays an important role in the enumeration process, and is the only communication channel to the device at attachment. The default pipe is associated with endpoint number zero (that is why endpoint number zero must be included as part of the device and must be of a control type). Endpoint number zero is and composed of two endpoints (one IN and one OUT) that share the same endpoint number and are referred as one.

Low speed devices can support two additional endpoints (beside endpoint number zero) which may be control or interrupt type. Full speed devices, on the other hand, can support up to maximum of 15 additional IN endpoints and 15 additional OUT endpoints. The additional endpoints can be used only after the devices has been fully configured.

1.4. The Application Layer

The application layer appears as the client software in the host and as the function in the device. The function in the device is composed of collection of interfaces and controls the functionality of the device. The client software manages the appropriate interface by transferring data from its buffers to the endpoints associated with the appropriate interfaces. The client software works with a specific device function, independent of the other device functions in the system.

1.4.1 The USB Protocol

The USB host handles most of the complexity of the USB protocol, which makes the peripherals design simple and low cost. Data flow can be from host to device and from device to host.
USB transactions are done through packets. Each transaction is composed usually from three phases:

- **Token phase** - the host initiates token indicating the future transaction type.

- **Data phase** - the actual data is transmitted through packet. The data direction matches the direction indicated by the token that was transmitted previously.

- **Handshake phase** - (optional) - handshake packet is sent, indicating the success or failure of the transaction.

The USB uses a polling protocol. Whenever the host wishes to receive data from the device it issues a token (a packet types that we will discuss later) addressed to that specific device. If the device has data to send it sends it after receiving the token and the host (if the handshake phase is included on the transfer) will respond with handshake packet. If the device doesn’t have anything to send the host issues the token to the next device. If, on the other hand, the host wishes to send data to the device, it will send the appropriate token and data packet following it. The device will response by a handshake packet.

The USB protocol is very robust. The protocol includes handshake mechanism, time-out rules (to prevent deadlock in the system), low control mechanism and very low physical bit error rate ($10^{-10}$). Each packet transmitted on the bus includes check bits and CRC protection.

There are four main USB transfer types:

- **Isochronous transfer**: Isochronous transfer, is used for multimedia devices such as audio, video, etc. Important characteristic of the transfer is that bandwidth is guaranteed - the required bandwidth is reserved for the devices uses this transfer type. In isochronous transfers there is less attention to the success of the transfer (whether or not the whole data arrived on time) since the traffic included in this transfer type has a high tolerance for errors.
• **Bulk transfer**: Bulk transfer is consisted of massive amount of data and is used by devices requires it such as printers, scanners, etc. The bandwidth allocated in each transaction of the transfer varies according to the bus resources at the time. Bulk transfers are done in reliable mode - there is great deal of awareness to errors.

• **Interrupt transfer**: Interrupt transfer is a limited-latency transfer and used for devices such as mouse, joystick that needs to report short event notification, characters or coordinates. A USB device that works in an interrupt transfer mode defines, as part of its configuration, the time interval it wants to send or receive information. The host is responsible to turn to device at that specific rate, and then the device is allowed to send or receive the necessary data.

• **Control transfer**: Control transfers are used to configure a device. The configuration is done at the enumeration process but can be done also at any state of the communication process. When a device enters the system the host needs to learn about it and configure it at the appropriate configuration, all this communication is done using the control transfers. Control transfer can also includes special messages defined by the vendor.

### 1.4.2 Packet Field Formats

Before learning the different types of packets used by the protocol, let us view the different fields in the packets:

**SYNC Field**

The SYNC field appears at the start of each packet. It appears on the bus as idle followed by "KJKJKJKK" (encoded in NRZI encoding). The SYSC (synchronization) field allows the receiving peripheral synchronize its internal clock to the incoming data. The following packets description will ignore this field (for simplicity) but we must not forget its existence.
1.4. THE APPLICATION LAYER

Figure 1.1: Sync Packet

**PID - Packet Identifier Field**

The PID field contains the identity of the packet. Since there are many types of packets we need to indicate at the start of the packet, which packet it is. The PID field is composed of eight bits as is shown in the following diagram. The first four bits are used to notify the actual id of the packet, and the next four are used as check bits (are one’s complements of the first four bits) and used for error detecting.

Figure 1.2: PID Packet

- **Tokens**: Token packets can be OUT, IN, SOF and SETUP.
  - OUT token indicates that the following data will be transmitted from the host to the device.
  - IN token indicates that the following data will be transmitted from device to host.
  - SOF token indicates start of frame.
  - SETUP token indicates that the following packet will be sent from host to device and will contain setup command (used for configuration).

- **Data**: Data PID appears in data packets. Data PID can be either DATA0/DATA1, the different PID is used for data toggle synchronization.
• **Handshake**: Handshake PID is used in handshake packets, in order to indicate the success / failure of the transfer. Handshake PID can be either ACK, NAK, STALL.
  
  – **ACK**: The receiver received error free packet.
  – **NAK**: The receiver is unable to receive the data (for example due to overflow problem), or the sender is unable to send data (underflow problem for example).
  – **STALL**: The specific endpoint is halted or the specific SETUP command is not supported.

**Address Field**

The address field is divided into two fields:

• **Address field (ADDR)**: This field contains the actual address of the function (normally the device itself), assigned to it at the enumeration process. Each function in the system has its unique address and there can be up to 127 different addresses in the system (address zero is reserved and is used as an initial address of a function, it is not allowed to use address zero as a permanent address).

![Address Packet](image)

Figure 1.3: Address Packet

• **Endpoint Field (ENDP)**: The endpoint number field contains the number of the endpoint referred. Each endpoint in a specific function is identified uniquely with an endpoint number. In low-speed device there can be two additional endpoints (beside endpoint number zero) and for full-device there can be up to 16 endpoint at any type (including endpoint number zero). The endpoint field is used in OUT, IN and SETUP tokens.
1.4. THE APPLICATION LAYER

Chapter 1. USB

Figure 1.4: EndPoint Field

Frame Number Field

The frame number field is composed of 11 bits indicating the number of the current frame. The field is contained only in SOF token indicating the start of the frame.

Data Field

The data field contains the data transmitted in the transaction. The data field can contain up to 1023 bytes.

CRC field

The CRC (cyclic redundancy check) field is used to protect all the fields in a token packet (except of PID field) and to protect the data in the data packets. The CRC field in a token packet is composed of 5 bits while in data packets is composed of 16 bits.

1.4.3 Packet Formats

Let us view the different packets format:

Token Packet

As was mentioned before, each transaction begins by issuing a token by the host. The ADDR and ENDP field uniquely defines the endpoint that is about to receive the following data packet in SETUP or OUT transactions, and on the other hand specifies the endpoint that is about to send data in IN transactions.

Start Of Frame Packet

On the USB, 1-ms frame times are used to transfer data. The Host Controller begins each frame by generating a Start Of Frame (SOF). If there is isochronous data to be
transferred, the Host Controller Driver schedules this data first. The Host Controller Driver ensures that there is enough time to complete all scheduled isochronous and interrupt transfers with some time remaining for control and bulk transfers. SOF token can be used as a trigger to process isochronous OUT endpoints.

Data Packets

Data packets are composed of PID (indicating that the packet is a data packet), data field, which contains the actual data to be transmitted and CRC16 to protect the data field.
Handshake Packets

Handshake packets are composed only of PID indicating the results of the previous stage.

- ACK, which indicates that the packet was received with no CRC or bit stuff errors, can be used at the handshake phase of a SETUP, OUT transfers (sent by the device) or in IN transfer (sent by the host).
- NAK, which is used for flow control, can be sent in the handshake phase of OUT or IN transfers.
- STALL, which indicates some problem in the transfer (endpoint halted or control command not supported), is not allowed to be used by the host.

![Figure 1.8: Handshake Packets](image)

### 1.4.4 Transaction Format

#### Control Transfer

A control transfer is composed of three or two phases: setup, data (optional) and status, each of those phases is composed of three phases (token, data, handshake).

The setup stage role is to indicate the device which setup command the host wishes to send. There are many kinds of SETUP commands such as:

- SET_ADDRESS : setting a permanent address to a function.
- GET_DEVICE_DESCRIPTOR: the host wishes to get the device descriptor, which contains details concerning the device - how many configurations, interfaces it has, is the device is self/ bus powered etc.
• **GET_CONFIGURATION_DESCRIPTOR**: the host wishes to learn about a specific configuration of a device.

• **GET_CONFIGURATION**: the host detects which configuration is active at the moment in the device.

• **SET_CONFIGURATION**: the host sets a specific configuration on the device.

At the beginning of a setup stage the host issues a SETUP token, followed by the setup command packet. The device must response with an ACK packet.

The data stage (if included) contains the flow of the data, which direction (from host to device of from device to host) indicated in the setup stage. The data stage is composed of one or more IN or OUT transactions (all the transactions in the data stage must be in the same direction - all IN or all OUT). Each transaction in the data stage begins with IN/OUT token issued by the host, afterwards data is sent (in the appropriate direction) and the transaction ended with a handshake packet.

The status stage reports the host the results of the previous stages: setup and data stages. The report is always from the device to the host. Important characteristic of this stage is that the data flow direction in it is opposite to the one in the data stage (if there wasn't data stage the direction will be IN).

If the direction of the status stage is IN then the report is done in the data phase of the transaction, if it is OUT then the report is done in the handshake phase.

Example of Control transfer: (Get_Device_Descriptor as SETUP command)

**Bulk Transfer**

Bulk transfers are composed of one or more three phases transactions. Each transaction starts with a token sent from the host indicating the direction of the data transfer in the following phase. In the next phase, data is transmitted according to the direction indicated by the token. If there was no detection of data error while receiving
the data, the last phase is the handshake phase, in which a report concerning the success of the transaction is being sent. If error was detected no handshake packet is sent.

There are two kinds of bulk transfers:

- **IN transfer** - in which the host asks for data from the device - data flow direction is from the device into the host.

- **OUT transfer** - in which the host wishes to send data to the device - data flows from the host out to the device.

Whenever the host wishes to receive data from the device, it initiates an IN token and sends it to the device, when the device receives the token, it sends data as response to the token and the host responds with an ACK packet if the data was received error free and doesn’t send any handshake in case of error detection. In case the device can’t send the required data (underflow - data needs to be sent but the transmit FIFO is empty, or any other function problem), the device won’t responds with a data packet but with NAK or STALL indicating its inability to answers the host demands. This situation results two phase transaction.

If, on the other hand, the host wishes to send data to the device, it initiates an OUT token and sends, in the next stage, the data it wished to send. The device,
after receiving the data response with a handshake packet. Bulk transfers are highly reliable due to the handshake and timeout (which was mentioned earlier) mechanisms. If there is any problem in the USB system, the host will detect it and prevent deadlocks in the system.

**Interrupt Transfer**

Interrupt transfers are very similar to bulk transfers. As in bulk transfers data can be sent from host to device and from device to host. If the host wishes to know which interrupt is pending on the device, it initiates an IN token to the appropriate endpoint. If there is a pending interrupt, the function will send details concerning the interrupt, as a data packet in the following stage. If the information was received error free by the host, it will initiate an ACK packet in the handshake phase. In case of error detection no handshake will be transmitted.

If, on the other hand, the host has initiated an IN packet but there are no pending interrupts and the endpoint has no information to send, the function will return a NAK packet. In case of an error condition in the function a STALL packet will be sent. The host will initiate an OUT token in case it wishes to transmit data to the device (data to serve the interrupt for example), following the token a data packet will be sent. The device, upon receiving the data, detects for errors, if the data is error free the device will response with ACK, NAK or STALL (as in bulk transfer). If the data was corrupted no handshake will be sent.

Example of Interrupt transfer:

**Isochronous Transfer**

Isochronous transfers are composed of one or more two phases transactions. As was mentioned earlier, there is no handshake phase in isochronous transfers. The host initiates either an IN token, in order to receive data from the device, or an OUT token to send data. In the next stage data is transmitted in the direction indicated by the token which was sent before.

Example of ISO transfers:
1.4. THE APPLICATION LAYER

Figure 1.10: Interrupt Transfer

Figure 1.11: Isochronous Transfer
Chapter 2

Architecture

The figure below illustrates the overall architecture of the core of the host controller. The host interface provides a bridge between the internal data memory and control registers to the function controller. The data memory and control registers interface to the Protocol Layer. The protocol layer interfaces to interface block. Each of the blocks will be described in detail below in this section.

Figure 2.1: arch
2.1 Communication Channels

SRAM
The SSRAM is a single ported Synchronous SRAM block that is used to buffer the input and output data.

Protocol Layer
The protocol layer is responsible for all USB data IO and control communications.

Memory Interface and Arbiter
The memory interface and arbiter arbitrates between the USB core and host interface for memory access. This block allows the usage of standard single port Synchronous SRAM. Besides arbitration it performs data steering and flow control.

Clocks
Time is present in the USB system via clocks. In fact, there are multiple clocks in a USB system that must be understood:

- Sample Clock: This clock determines the natural data rate of samples moving between client software on the host and the function. This clock does not need to be different between non-USB and USB implementations.

- Bus Clock: This clock runs at a 1.000ms period (1kHz frequency) and is indicated by the rate of SOF packets on the bus.

- Service Clock: This clock is determined by the rate at which client software runs to service IRPs that may have accumulated between executions. In most existing operating systems, it is not possible to support a broad range of isochronous communication flows if each device driver must be interrupted for each sample for fast sample rates. Therefore, multiple samples, if not multiple packets, will be processed by client software and then given to the Host Controller to sequence over the bus according to the pre-negotiated bus access requirements.

2.1 Communication Channels
There are two communication channels between the Host Controller and the Host Controller Driver. The first channel uses a set of operational registers located on the
HC. The Host Controller is the target for all communication on this channel. The operational registers contain control, status, and list pointer registers. Within the operational register set is a pointer to a location in shared memory named the Host Controller Communications Area (HCCA). The HCCA is the second communication channel. The Host Controller is the master for all communication on this channel. The HCCA contains the head pointers to the interrupt Endpoint Descriptor lists, the head pointer to the done queue, and status information associated with start-of-frame processing.

The basic building blocks for communication across the interface are the Endpoint Descriptor (ED) and Transfer Descriptor (TD).

The Host Controller Driver assigns an Endpoint Descriptor to each endpoint in the system. The Endpoint Descriptor contains the information necessary for the Host Controller to communicate with the endpoint. The fields include the maximum packet size, the endpoint address, the speed of the endpoint, and the direction of data flow.
Endpoint Descriptors are linked in a list. A queue of Transfer Descriptors is linked to the Endpoint Descriptor for the specific endpoint. The Transfer Descriptor contains the information necessary to describe the data packets to be transferred. The fields include data toggle information, shared memory buffer location, and completion status codes. Each Transfer Descriptor contains information that describes one or more data packets. The data buffer for each Transfer Descriptor ranges in size from 0 to 8192 bytes with a maximum of one physical page crossing. Transfer Descriptors are linked in a queue: the first one queued is the first one processed.

Each data transfer type has its own linked list of Endpoint Descriptors to be processed.

Figure 2.3: Typical List Structure
Chapter 3

Operation

This section describes the USB function controller of the host controller and, will be explained how some of the processes that are implemented so far in VHDL, to help in that explanation I’ll put some pieces of the VHDL code.

3.1 Main flow reset

The above picture represents the main flowchart of the protocol layer, the state machine here represented is implemented in the file "Control_Flow.vhd", with the
PL_ControlFlow entity. This file has the top entity in the protocol layer architecture. This is the entity that enables some processes, such as Start Sync, Start of frame, Token Out, Send Data etc. The way that this is done, is by using this concurrent statement:

\[ \text{Start}_\text{SDAT} <= '1' \text{ when (CS} = \text{Send}_\text{Data}) \text{ else '0'}; \]

Where the \text{Start}_\text{Sync} is the signal that enables the Send Data process, and it will be 1 when the NS "next State" is \text{Send}_\text{Data}. At this state the Control_Flow process will be waiting until the \text{Sdat}_\text{Ended} signal is 1. The signal \text{Start}_\text{Sync} can only be written by the Control_Flow process and, the \text{Sdat}_\text{Ended} only by the Send Data process.

This is the same method that is used to enable and disable processes. If a process needs to reset another process he can do it at any time. At this moment any time that an ended signal is issue the process that receives the signal will reset the process that issued the ended signal, this syntax represents the reset method:

\[ \text{RSTTTOK} <= '1' \text{ when (Reset} = '1' \text{ or CS} = \text{Send}_\text{Data} \text{ or clk}_\text{1m} = '1') \text{ else '0'}; \]

So every time that the global reset "Reset" is set to 1, or CS is \text{Send}_\text{Data} or \text{clk}_\text{1m} is 1, the token process will be reset.

The figure 3.2 represents the first states of the main flowchart. this states are not represented in the previous figure, just to turn easier to understand the previous flow.

So, when the global reset or the signal from the process that counts the 1ms is 1, the main state machine, that is the one represented in this figure, will return to the idle state, restarting the state machine and all other processes that depend on this process.
3.2 Sync Process

This process, that is a simple state machine and is in file ”sync.vhd”, implements the send of a predefined string. The SYNC field appears on the bus as IDLE followed by the binary string KJKJKJKK, in its NRZI encoding. It is used by the input circuitry to align incoming data with the local clock and is defined to be eight bits in length. SYNC serves only as a synchronization mechanism. The last two bits in the SYNC field are a marker that is used to identify the first bit of the PID. In the table 3.1 we can see the different signal states.

The signals Start_Sync and Sync_Ended are used to communicate between processes.
### 3.3 Start of Frame

Start of Frame (SOF) packets are issued by the host at a nominal rate of once every 1.00 ms 0.05. This figure represents the start of frame. Just to remember that Low-speed devices support control and interrupt transfers while full-speed devices support all of the transfers described in the figure.
Besides the PID, and the CRC fields, it also includes the frame number field, this field is generated by another process that counts and gives the the initial time of the start of frame, by another words is the process that puts the clk_1m signal to 1. The CRC field will be explained in the next sub-section.

## 3.4 CRC

One of the most popular methods of error detection for digital signals is the Cyclic Redundancy Check (CRC), and is used in the USB protocol. The basic idea behind CRCs is to treat the message string as a single binary word $M$, and divide it by a key word $k$ that is known to both the transmitter and the receiver. The remainder $r$ left after dividing $M$ by $k$ constitutes the "check word" for the given message. The transmitter sends both the message string $M$ and the check word $r$, and the receiver can then check the data by repeating the calculation, dividing $M$ by the key word $k$, and verifying that the remainder is $r$. The only novel aspect of the CRC process is that it uses a simplified form of arithmetic, which we’ll explain below, in order to perform the division.

The USB protocol uses two remainders for the crc-5 and the crc-16, the remainders are 100101 and 1100000000000101 respectively. It’s worth noting that that the remainder of any word divided by a 6-bit word will contain no more than 5 bits, so our CRC words based on the polynomial 100101 will always fit into 5 bits, and the same for the crc-16.In general, a polynomial with $k$ bits leads to a "$k$-1 bit CRC".
3.4. CRC

CHAPTER 3. OPERATION

3.4.1 CRC-5

The algorithm implemented to calculate the CRC-5 is very simple and is shown here:

\[
\begin{align*}
\text{Msg} & \leftarrow \text{Message } \& \text{ div}._0; \text{c1} \leftarrow \text{Msg(0 to 5) } \text{xor div}._1 \text{ when } \text{Msg}(0) = '1' \text{ else } \\
& \quad \text{Msg(0 to 5) } \text{xor div}._0; \\
\text{c2} & \leftarrow (\text{c1 (1 to 5) } \& \text{Msg(6)}) \text{ xor div}._1 \text{ when } \text{c1 (1)} = '1' \text{ else } \\
& \quad (\text{c1 (1 to 5) } \& \text{Msg(6)}) \text{ xor div}._0; \\
\text{c3} & \leftarrow (\text{c2 (1 to 5) } \& \text{Msg(7)}) \text{ xor div}._1 \text{ when } \text{c2 (1)} = '1' \text{ else } \\
& \quad (\text{c2 (1 to 5) } \& \text{Msg(7)}) \text{ xor div}._0; \\
\text{c4} & \leftarrow (\text{c3 (1 to 5) } \& \text{Msg(8)}) \text{ xor div}._1 \text{ when } \text{c3 (1)} = '1' \text{ else } \\
& \quad (\text{c3 (1 to 5) } \& \text{Msg(8)}) \text{ xor div}._0; \\
\text{c5} & \leftarrow (\text{c4 (1 to 5) } \& \text{Msg(9)}) \text{ xor div}._1 \text{ when } \text{c4 (1)} = '1' \text{ else } \\
& \quad (\text{c4 (1 to 5) } \& \text{Msg(9)}) \text{ xor div}._0; \\
\text{c6} & \leftarrow (\text{c5 (1 to 5) } \& \text{Msg(10)}) \text{ xor div}._1 \text{ when } \text{c5 (1)} = '1' \text{ else } \\
& \quad (\text{c5 (1 to 5) } \& \text{Msg(10)}) \text{ xor div}._0; \\
\text{c7} & \leftarrow (\text{c6 (1 to 5) } \& \text{Msg(11)}) \text{ xor div}._1 \text{ when } \text{c6 (1)} = '1' \text{ else } \\
& \quad (\text{c6 (1 to 5) } \& \text{Msg(11)}) \text{ xor div}._0; \\
\text{c8} & \leftarrow (\text{c7 (1 to 5) } \& \text{Msg(12)}) \text{ xor div}._1 \text{ when } \text{c7 (1)} = '1' \text{ else } \\
& \quad (\text{c7 (1 to 5) } \& \text{Msg(12)}) \text{ xor div}._0; \\
\text{c9} & \leftarrow (\text{c8 (1 to 5) } \& \text{Msg(13)}) \text{ xor div}._1 \text{ when } \text{c8 (1)} = '1' \text{ else } \\
& \quad (\text{c8 (1 to 5) } \& \text{Msg(13)}) \text{ xor div}._0; \\
\text{c10} & \leftarrow (\text{c9 (1 to 5) } \& \text{Msg(14)}) \text{ xor div}._1 \text{ when } \text{c9 (1)} = '1' \text{ else } \\
& \quad (\text{c9 (1 to 5) } \& \text{Msg(14)}) \text{ xor div}._0; \\
\text{c11} & \leftarrow (\text{c10 (1 to 5) } \& \text{Msg(15)}) \text{ xor div}._1 \text{ when } \text{c10(1)} = '1' \text{ else } \\
& \quad (\text{c10(1 to 5) } \& \text{Msg(15)}) \text{ xor div}._0; \\
\text{crc} & \leftarrow \text{c11(1 to 5)};
\end{align*}
\]

This is a concurrent statement, so it means that the crc value will be always available, to obtain the right value you only have to pass the value of the Msg and in the same cycle you can obtain it. But it’s advisable, if possible to wait for the next clock cycle.

The div._1 as the value of the remainder and div._0 is a string of all zeros.
3.4.2 CRC-16

This is very similar with the crc-5 algorithm, the difference is that the values of the remainder and the div.0 are different. And the other difference is that is used an auxiliary process that makes the xor of the different bytes that are put in the Msg.

tmpcrc16 <= c9(1 to 15) When x = 1 else c9(1 to 15) xor acc;
crc <= tmpcrc16;

Process(LoadNewByte, Reset, x, tmpcrc16)
begin
if Reset = '1' then
Div.1 <= "11000000000000101";
Div.0 <= (Others => '0');
x <= 0;
acc <= (Others => '0');
elsif (LoadNewByte event and LoadNewByte = '1') then
if x = 0 then
x <= x + 1;
acc <= tmpcrc16;
elsif x = 1 then
x <= x + 1;
acc <= tmpcrc16;
else
x <= x;
acc <= tmpcrc16;
end if;
end if;
end process;

The process is here with the proposal of retain the last value of the signal tmpcrc16, into the signal acc. This is possible because the process has a ”delta” delay. The rest of the code is just alike to the crc-5 using the concurrent statement but just for 8 bits.
The reason of implementing the algorithm this way is because the CRC-16 is always
3.5 Data Sending and Receiving

The data field may range from zero to 1,023 bytes and must be an integral number of bytes. Figure 3.4 shows the format for multiple bytes. Data bits within each byte are shifted out LSb first.

![Figure 3.4: Data Field Format](image)

At this moment the algorithm that is implemented is to send data, and he tests if the byte that he is sending is the last one, otherwise he will send the next byte available on the Byte signal and sends a alert to the crc-16 to continue xoring all the receiving bytes.

3.6 Conclusion

There are some entities that are not explained here just because there are very alike, they all are a state machine that run with the same clock as the top entity. One of the entity is the test bench. At this moment I use these code to give all the stimulus to the top entity of the design, and to see all the results of the tests is used the program ModelSim.
Chapter 4

Future Work and Critical Analysis

There is many work that isn’t done. At this moment is needed to implement the interface between the SRAM, the Host controller driver and the protocol layer. There is a need to construct and implement the control registers and the block that does the serialization and deserialization of all data. All this to finish the host controller.

The best way to implement this interface is to create a synchronous FIFO because both parts uses different clocks, and an example of using a FIFO is when a generator of data use a FIFO to even out its generation of data, because the generator might be slower or faster than the consumer depending on the complexity of the producer consumer algorithm for different cases.

To finish the overall project is needed to finish all the drivers. The drivers that make the interface between the host controller, in this case the FPGA, and the drivers of the specific device that the user wishes to use.

The work was not finish because the time that was available for this project was not enough. The problem was due to all the changes that I’ve included. During of this developing work process, the discussions was very helpful, and we were aware that the time will not be enough for improvements that we would like to reach.
Literature survey