A Synthesizable Verilog Model of the Serial Protocol Engine for

USB 1.1 Device

Master thesis performed in Electronics Systems
By
Shankar Guna

LiTH-ISY-EX--07/3977--SE

Linköping 2007
A Synthesizable Verilog Model of the Serial Protocol Engine for

USB 1.1 Device

Master thesis in Electronics System division
at
Linköping Institute of Technology
by
Shankar Guna
LiTH-ISY-EX--07/3977--SE

Supervisor : Mr.Manigandan
Examiner : Mr. Kent Palmkvist
Linkoping 2007.
USB has become a popular interface for exchanging data between PC’s and peripherals. An increasing number of portable peripherals are using the USB interface to communicate with the PC. The design and implementation of a synthesizable model of the USB 1.1 protocol engine is presented in this report. The PHY is compatible with the USB 1.1 transceiver macrocell interface (UTMI) specification and the simulation test confirmed the successful operation of circuits for both full speed (12 Mbps) and low speed (1.5 Mbps) data transmission. The model is written completely in behavioral verilog with a top down approach and the model was verified and validated.

Number of pages: 71 Pages.

Keywords
Write about five to eight and do not use personal names like the tutor, teachers etc., names of countries, cities etc.
ABSTRACT

USB has become a popular interface for exchanging data between PC’s and peripherals. An increasing number of portable peripherals are using the USB interface to communicate with the PC. The design and implementation of a synthesizable model of the USB 1.1 protocol engine is presented in this report.

The PHY is compatible with the USB 1.1 transceiver macrocell interface (UTMI) specification and the simulation test confirmed the successful operation of circuits for both full speed (12 Mbps) and low speed (1.5 Mbps) data transmission. The model is written completely in behavioral verilog with a top down approach and the model was verified and validated.
LIST OF CONTENTS

CHAPTER 1 INTRODUCTION ................................................................. 1
  1.1 Motivation...................................................................................... 1
  1.2 Objective of the Thesis................................................................. 2
  1.3 Overview of the Task..................................................................... 2
  1.4 Design of the System and Tools Used......................................... 3
  1.5 Document Organization.............................................................. 3

CHAPTER 2 LITERATURE OVERVIEW................................................. 5
  2.1 Shortcoming of Existing PC IO Paradigm................................... 5
    2.1.1 Cable ...................................................................................... 5
    2.1.2 Installation and Configuration of Expansion Cards.............. 5
    2.1.3 No Hot Pluggability for Peripherals..................................... 6
    2.1.4 Cost ...................................................................................... 6
  2.2 Analysis of Potential Solution.................................................... 6
    2.2.1 Access Bus............................................................................ 8
    2.2.2 USB -- The Right Balance .................................................... 8
  2.3 USB Features............................................................................... 9
    2.3.1 Plug and Play Support......................................................... 9
    2.3.2 Hot Attachment ................................................................. 9
    2.3.3 Room for Growth............................................................... 9
    2.3.4 Low Cost............................................................................. 9

CHAPTER 3 USB SYSTEM OVERVIEW.............................................. 11
  3.1 USB System Component.............................................................. 11
    3.1.1 USB Device Drivers........................................................... 12
    3.1.2 USB Driver......................................................................... 12
    3.1.3 USB Host Controller Driver............................................... 12
    3.1.5 USB Devices...................................................................... 14
  3.2 USB Communication Model........................................................ 14
    3.2.1 Communications Flow....................................................... 15
    3.2.2 Transfers, IRPS, Frames, and Packets.................................. 16
      3.2.2.1 Transfers..................................................................... 16
      3.2.2.2 The USB Driver, IRPS and Frames.............................. 16
      3.2.2.3 The Host Controller Driver and Transaction.................. 17
3.3 Device Framework ................................................................. 17
  3.3.1 USB Bus Interface Layer .................................................. 18
  3.3.2 USB Device Layer ............................................................ 18
  3.3.3 Function layer ................................................................. 21
3.4 USB Peripheral Connection .................................................. 21
3.5 USB Signaling Environment .................................................. 22
  3.5.1 NRZI Encoding ............................................................... 22
  3.5.3 Differential Pair Signaling ................................................ 24
3.6 USB Transfer Types ............................................................. 24
  3.6.1 Interrupt Transfer ............................................................ 24
  3.6.2 Bulk Transfer ................................................................. 25
  3.6.3 Isochronous Transfer ...................................................... 25
  3.6.4 Control Transfer ............................................................. 25

CHAPTER 4 USB TRANSACTION .................................................... 27
4.1 Packets – The Basic Building Blocks of USB Transactions .......... 27
  4.1.1 Synchronization Sequence ................................................ 28
  4.1.2 Packet Identifier ............................................................. 29
  4.1.3 Packet-Specific Information ............................................. 31
  4.1.4 Cyclic Redundancy Checking (CRC) ................................. 31
  4.1.5 End of Packet (EOP) ......................................................... 31
4.2 Packet Types ........................................................................ 31
  4.2.1 Token Packets ............................................................... 31
    4.2.1.1 SOF Packet ............................................................... 32
    4.2.1.2 IN Packet ............................................................... 33
    4.2.1.3 OUT Packet ........................................................... 34
    4.2.1.4 SETUP Packet ......................................................... 35
  4.2.2 Data Packets --- Data0 and Data1 ..................................... 36
  4.2.3 Handshake Packet .......................................................... 38
  4.3.1 Packet Errors ............................................................... 40
    4.3.1.1 PID Checks ............................................................. 41
    4.3.1.2 CRC Errors ............................................................ 41
CHAPTER 5 IMPLEMENTATION .........................................................43
5.1 Functional Block Diagram.................................................................43
5.2 Implementation Overview .................................................................44
  5.2.1 Device Receiver SIE Interfacing Signals .....................................45
  5.2.2 Device Transmitter SIE Interfacing Signals ..................................50
  5.2.3 Digital Phase Locked Loop Interface ..........................................55
5.3 Implementation of Device Receiver SIE ..........................................57
  5.3.1 NRZI decoder .............................................................................57
  5.3.2 Bit Destuff ................................................................................57
  5.3.3 Shift Register ............................................................................57
  5.3.4 Byte Counter ............................................................................58
  5.3.5 CRC Checker ............................................................................58
  5.3.6 PID Decoder ............................................................................58
5.4 Implementation of Device Transmitter SIE ....................................58
  5.4.1 NRZI Encoder ............................................................................58
  5.4.2 Bit stuffer .................................................................................59
  5.4.3 Serial Shifter ............................................................................59
  5.4.4 Byte Counter ............................................................................59
  5.4.5 CRC Check Field Generator ....................................................59
5.5 Implementation of Digital Phase Locked Loop ..................................60
  5.5.1 Finite State Machine for End Of Packet Detection .....................60
  5.5.2 Finite State Machine for Generation of Synchronize Clock ...........61

CHAPTER 6 RESULTS .................................................................63
6.1 Simulation Results ..........................................................................63
6.2 Synthesis Report ............................................................................66

CHAPTER 7 CONCLUSION .........................................................67
  7.1 Conclusion ....................................................................................67
  7.2 Future Scope ................................................................................67

REFERENCES. ......................................................................................69
LIST OF TABLES

Table 2.1 Applications, Relative Performance and Desire Attributes ........................................... 7
Table 2.2 Various Solutions with Relative Performance and Complexity ..................................... 7
Table 2.3 Key USB Features ........................................................................................................... 10
Table 4.1 USB Token Packets ........................................................................................................ 32
Table 4.2 Direction of Data Packets ............................................................................................ 36
Table 4.3 Packet Type and CRC .................................................................................................... 41
Table 5.1 Interfacing signals for Device Receiver SIE ................................................................. 47
Table 5.2 Device Receiver SIE Error Format ............................................................................... 49
Table 5.3 Interfacing signals for Device Transmitter SIE ............................................................. 52
Table 5.4 Device Transmitter SIE Error Format .......................................................................... 54
Table 5.5 Interface signals of Digital Phase Locked Loop ............................................................. 55
# LIST OF FIGURES

| Figure 3.1 | Device Framework – Software’s View of Hardware | 19 |
| Figure 3.2 | NRZI Encoded Data | 22 |
| Figure 3.3 | Stuffed Bit | 23 |
| Figure 4.1 | Many USB Transaction Consists of Three Phases | 26 |
| Figure 4.2 | Packet Format | 27 |
| Figure 4.3 | Synchronization Sequence | 28 |
| Figure 4.4 | Packet Identifier Format | 29 |
| Figure 4.5 | Format of SOF Packet | 32 |
| Figure 4.6 | IN Token Packet Format | 33 |
| Figure 4.7 | OUT Token Packet Format | 34 |
| Figure 4.8 | SETUP Token Packet Format | 35 |
| Figure 4.9 | Data0 Packet Format | 36 |
| Figure 4.1.1 | Data1 Packet Format | 36 |
| Figure 4.1.2 | Handshake Packet Format | 38 |
| Figure 4.1.3 | PID Check | 39 |
| Figure 5.1 | Functional Block Diagram | 43 |
| Figure 5.2 | Receiver State Diagram | 45 |
| Figure 5.3 | Device Receiver Block | 46 |
| Figure 5.4 | Transmitter State Diagram | 50 |
| Figure 5.5 | Device Transmitter | 51 |
| Figure 5.6 | DPLL block | 55 |
| Figure 6.1 | Output signals for Digital Phase Locked Loop Module | 63 |
| Figure 6.2 | Output signals for Device Transmitter Module | 64 |
| Figure 6.3 | Output signals for Device Receiver Module | 65 |
CHAPTER 1 INTRODUCTION

Universal Serial Bus (USB) has emerged as a result of the difficulties associated with the cost, configuration, and attachment of the peripheral devices in the personal computer environment. In short, USB creates a method of attaching, and accessing peripheral devices that reduce overall cost, simplifies the attachment and configuration from the end-user perspective, and attempt to solve several technical issues associated with the old style peripherals.

1.1 Motivation

The motivation for the Universal Serial Bus (USB) comes from two interrelated considerations:

Ease-of-use

The lack of flexibility in reconfiguring the PC has been acknowledged as the Achilles’ heel to its further deployment. The combination of user-friendly graphical interfaces and the hardware and software mechanisms associated with new-generation bus architectures have made computers less confrontational and easier to reconfigure. However, from the end user's point of view, the I/O interfaces of PC’s, such as serial/parallel ports, keyboard/mouse/joystick interfaces, etc., do not have the attributes of plug-and-play.

Port expansion

The addition of external peripherals continues to be constrained by port availability. The lack of a bi-directional, low-cost, low-to-mid speed peripheral bus has held back the creative proliferation of peripherals such as telephone/fax/modem adapters, answering machines, scanners, PDA’s, keyboards, mice, etc. Existing interconnects are optimized to connect products for one or two ports. As each new function or capability is added to the PC, a new interface has been defined to address this need.

The USB is the answer to connectivity for the PC architecture. It is a fast, bi-directional, isochronous, low-cost, dynamically attachable serial interface that is consistent with the requirements of the PC platform of today and tomorrow.
1.2 Objective of the Thesis

This thesis describes the protocol used in USB 1.1 system’s low-level interface. It also describes the type of transactions and errors in USB transactions.

The goal is to Designing of Protocol Engine (Serial Interface Engine) for the USB device. The design of SIE (Serial Interface Engine) is implemented in a Verilog Hardware Description Language.

1.3 Overview of the Task

The purpose of this thesis is to make a Verilog synthesizable model of USB 1.1 protocol engine.

Main Task:

- Obtaining a complete functional description of the full system, based on the details given in the USB 1.1 standard specification.
- Analyzing the variations of the verilog model from the actual USB 1.1 spec
- Modeling the complete system in verilog using the tool Model Sim.
- Simulation of the individual blocks and entire design with various test cases.

The guide for the design was the USB 1.1 standard specification. The implemented design has to be validated through testing. The behavior of the model was validated with the signals and timing details given with the specification.

1.4 Design of the System and Tools Used

The design was structured into different levels. It includes device transmitter and device receiver block, DPLL block, memory block, counter block. the total design was made easy with Mentor graphics HDL tool. A graphical user interface was provided to design the various functional blocks and their descending level hierarchy along with interconnecting the various blocks. so a great part of coding effort was reduced by the efficient usage of the tool to code the interface parts and the whole structure of the system.

The various blocks designed were simulated individually using the ModelSim, which is a part of Mentor Graphics. The synthesizability of the model was checked by ISE tool.
package.

1.5 Document Organization

Chapter 1 Introduces the motivation for the Universal Serial Bus and the objective of this thesis.

Chapter 2 Describes the rational for establishing USB as the solution to address the shortcoming of the existing PC I/O implementation. This chapter analyses the possible solutions that are based on existing or emerging standards, and introduces USB. It also describes the features of USB 1.1.

Chapter 3 Defines the USB environment by introducing and describing the key hardware and software elements and the interconnection between them. It also discusses the signaling environment used by USB. It also covers the nature of USB transfers. Frame based transfer are described along with the transfer types supported by USB system.

Chapter 4 Details individual transactions that comprise the transfer types discussed in Chapter 3. It defines the various packet types and how they are used to perform the requested transfer. It also discusses the errors in transactions.

Chapter 5 Describes the implementation of the Protocol Engine for USB 1.1 model it also describes the interfacing signals for Device Receiver SIE, Device Transmitter SIE and Digital Phase Locked Loop modules and their implementation in Verilog HDL.

Chapter 6 The results of the simulation are presented in this chapter.
CHAPTER 2 LITERATURE OVERVIEW

The USB (Universal Serial Bus) is a cable bus that supports data exchange between a host computer and a wide range of simultaneously accessible peripherals. The attached peripherals share USB bandwidth through a host scheduled, token-based protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation.

2.1 Shortcoming of Existing PC IO Paradigm

End users are faced with a variety of problem when connecting peripheral to their PCs as discussed in the forthcoming articles.

2.1.1 Cables

Dedicated cables are required for the mouse, keyboard, printer, external modem, Zip drive, plotter, etc. most of which are completely different. The variety of cables and connectors required to connect particular peripheral devices makes many users a little inconvenient and confusing.

2.1.2 Installation and Configuration of Expansion Cards

When peripherals are purchased many of them requires the installation of interface cards. Installations of such cards require reconfiguration of the system setup and installation of driving software through various internal/external devices, such as FDDs (Floppy Disk Drive), CDROMs, etc. This installation process needs shutting down and restarting the system. It becomes more frustrating if the device does not function, due to software and hardware conflicts.
2.1.3 No Hot Plug ability for Peripherals
In some cases while the system is running if we try to attach a new peripheral device then the system displays a message “Reboot the system”. Since the software checks for the presence of hardware and installs software for only those devices that it detects during the booting process, it cannot detect any new peripheral device. Hence we have to reboot the system so that software can detect the new device and load the necessary software to access it.

2.1.4 Cost
The cost of implementing system and peripheral devices based on the original PC design is fairly expensive due to the high cost of the standard peripheral connector and associated cables. Since most of the standard connector (serial and parallel ports) on the PC are used by wide variety of peripheral devices, if we want to attach a new peripheral device then it may require to purchase a expansion card which makes the system quite costly.

2.2 Analysis of Potential Solution
Numerous solutions exist that might satisfy the requirements of a new method for attaching and accessing peripheral devices. The following section briefly discusses various options that might be used. Presuming that all other features are supported by each solution and are more or less equal, a major factor in selecting a solution lies in evaluating cost performance ratio. The range of application that a new solution supports can be grouped as shown in Table 2.1. This table shows that some peripheral devices require little bandwidth, while others such as disk drivers and video applications require considerably more. Table 2.2 lists possible solutions and compares the performance verses the cost for each solution.
<table>
<thead>
<tr>
<th>Performance</th>
<th>Applications</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Speed:</td>
<td>Keyboard, mouse</td>
<td>Lower cost</td>
</tr>
<tr>
<td>Interactive Devices</td>
<td>Game peripherals</td>
<td>Hot plug-unplug</td>
</tr>
<tr>
<td>10 – 100 Kb/s</td>
<td>Monitor configuration</td>
<td>Easy of use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Multiple peripherals</td>
</tr>
<tr>
<td>Full Speed:</td>
<td>ISDN</td>
<td>Low cost Easy of use</td>
</tr>
<tr>
<td>Phone, Audio</td>
<td>PBX</td>
<td>Guaranteed Bandwidth</td>
</tr>
<tr>
<td>500 – 10,000 Kb/s</td>
<td>Digital Audio</td>
<td>Dynamic Attach/Detach</td>
</tr>
<tr>
<td></td>
<td>Scanner/Printer</td>
<td></td>
</tr>
<tr>
<td>High Speed:</td>
<td>Desktop Hard Disk Drive</td>
<td>High Bandwidth</td>
</tr>
<tr>
<td>Video, Disk, LAN</td>
<td>Video Conferencing</td>
<td>Easy of use</td>
</tr>
<tr>
<td>25 – 500 Mb/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.1** Applications, Relative Performance and Desire Attributes

<table>
<thead>
<tr>
<th>Bus Name</th>
<th>Data Rate</th>
<th>Host Complexity</th>
<th>Peripheral Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access bus</td>
<td>100 Kb/s</td>
<td>Simple HW or SW UART</td>
<td>Simple HW or SW UART</td>
</tr>
<tr>
<td>GeoPort</td>
<td>2.048 Mb/s</td>
<td>SCC USART</td>
<td>SCC USART</td>
</tr>
<tr>
<td>IEEE 1394</td>
<td>100 Mb/s</td>
<td>12000 – 20000 gates</td>
<td>5000 – 7000 gates</td>
</tr>
<tr>
<td>(Firewire)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>USB</td>
<td>12 Mb/s</td>
<td>10000 gates</td>
<td>1500 – 2000 gates</td>
</tr>
</tbody>
</table>

**Table 2.2** Various Solutions with Relative Performance and Complexity
2.2.1 Access Bus

The host and peripheral complexity are extremely low, making the cost very reasonable. However, the lower bandwidth of 100 Kb/s makes this solution too slow to support all the desired peripheral devices.

GeoPort

The GeoPort is a proprietary solution implemented in apple computers. GeoPort is focused solely on telecommunications and does not support the range of PC peripheral desired.

IEEE 1394

The solution, commonly referred to as Firewire, provides ample bandwidth to accommodate all peripheral applications. However, the associated complexity makes cost undesirable, particularly for the low performance/low cost peripherals.

2.2.2 USB -- The Right Balance

The Universal Serial Bus (USB) creates a solution for attaching PC peripherals that balances performance and cost. Devices attached to USB ports can incorporate additional connections for attaching other USB devices.
2.3 USB Features

2.3.1 Plug and Play Support
Automatic configuration is crucial for satisfying end user requirement. The USB eliminates the need of switches and jumpers to configure the device and eliminates the need to install the software when a new peripheral device is attached. The device should simply be attached by the user and be ready for immediate use.

2.3.2 Hot Attachment
When most of the legacy IO devices are attached to the system, they will not work till the system is restarted. Restarting of the system is required by the related software to detect the new peripheral device. USB provides a method of detecting that the new peripheral device has been attached to the system and automatically installs the relevant software needed to access the device.

2.3.3 Room for Growth
USB supports the whole new generation of peripheral devices. USB also provide flexibility for newer “smart” peripheral devices i.e. interactive peripherals (e.g. games), home automation, digital audio, telephony, compressed video, etc.

2.3.4 Low Cost
Universal Serial Bus (USB) provides a low cost solution for attaching peripheral devices to PCs. Table 2.3 lists the key Features of USB 1.1 system.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Cost</td>
<td>The USB provides a low cost solution for attaching peripheral devices to PCs.</td>
</tr>
<tr>
<td>Hot Pluggable</td>
<td>Device attachment is automatically detected by the USB and software automatically configure the device for immediate use, without user intervention.</td>
</tr>
<tr>
<td>Single Connector Type</td>
<td>The USB defines a single connector used to attach any USB device.</td>
</tr>
<tr>
<td>127 Devices</td>
<td>Supports the attachment of 127 devices per USB.</td>
</tr>
<tr>
<td>Low Speed or Full Speed Devices</td>
<td>The USB supports two-device speed: 1.5Mb/s and 12Mb/s. The lower speed makes it possible to implement low speed/low cost USB devices. This is due to reduced cost of the speed cables that can be design without shielding.</td>
</tr>
<tr>
<td>Cable Power</td>
<td>Peripheral can be powered directly from the cable. 5 volts DC power is available from the cable.</td>
</tr>
<tr>
<td>System Resource Requirement Eliminate</td>
<td>USB devices, unlike their ISA, EISA, and PCI cousins require no memory or IO address space and need no IRQ lines.</td>
</tr>
<tr>
<td>Error Detection and Recovery</td>
<td>USB transaction includes error detection mechanisms that are used to ensure that data is delivered without error. In the event of errors, transaction can be retried.</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>USB devices automatically enter a suspend state after 3ms of no bus activity. During suspend devices can consume no more than 500μA of current.</td>
</tr>
<tr>
<td>Supports for four type of transfers</td>
<td>The USB defines four different transfer types to support different characteristics required by the devices. Transfer types include: Bulk, Isochronous, Interrupt and Control transfers.</td>
</tr>
</tbody>
</table>

Table 2.3  Key USB Features
CHAPTER 3    USB SYSTEM OVERVIEW

This chapter defines the USB environment by introducing and describing the key hardware and software elements and the interconnection between them. It also discusses the signaling environment used by USB. It also covers the nature of USB transfers. Frame based transfer are described along with the transfer types supported by USB system.

3.1 USB System Component

The primary hardware and software elements associated with a USB system includes.

USB Hardware
- USB Host Controller
- USB Devices

USB Software
- USB Device Drivers
- USB Driver
- USB Host Controller Driver

All USB transactions are initiated by USB software. These accesses are typically originated by a USB device driver that wants to communicate with its device. The USB driver provides the interface between USB device driver and the USB host controller. This software is responsible for translating client request into one or more transaction that are directed to or from a target USB device.

The following sections describe the role of each component involved in USB transfer.
3.1.1 USB Device Drivers

USB device drivers (or client drivers) issue request to the USB drivers via IO Request Packets (IRPs). These IRPs initiate a given transfer to or from a target USB device. For example, a USB keyboard driver must initiate an interrupt transfer by establishing a memory buffer into which data will be returned from the USB keyboard. The client driver has no knowledge of USB serial transfer mechanisms.

3.1.2 USB Driver

The USB driver knows the characteristics of the USB target device and how to communicate with the device via the USB. The USB characteristics are detected by the USB driver when it parses the device descriptors during device configuration. For example, some devices require a specific amount of throughput during each frame, while other may only require periodic access every Nth frame.

When an IRP is received from a USB client driver, the USB driver organizes the request into individual transactions that will be executed during the series of 1ms frames. The USB driver sets up the transactions based on its knowledge of the USB device requirements, the needs of the client drivers, and the limitation/capability of the USB.

Depending on the operating environment, the USB driver may be shipped alone with the operating system or added as an extension via a loadable device driver.

3.1.3 USB Host Controller Driver

The USB Host controller driver (HCD) schedule transactions to be broadcast over the USB. Transactions are scheduled by the host controller driver by building a series of transaction lists. Each list consists of pending transaction targeted for one or more of the USB devices attached to the bus. A transaction list, or a frame list, defines the sequence of transactions to be performed during each 1ms frame. The USB host controller executes these transactions list at 1ms intervals. A single block transfer requested by USB client may be performed as a series of transactions that are scheduled and executed during consecutive 1ms frames. The actual scheduling depends on a variety of factors including: the type of transaction, transfer requirements specified by the device and the transaction traffic of other USB devices.
The USB host controller initiates transactions. Each 1ms frame begins with a start of frame (SOF) transaction and is followed by the serial broadcast of all transaction contained within a current list. For example, if one of the requested transaction is a request to transfer data to a USB printer, the host controller would obtain the data to be sent from a memory buffer supplied by the client software and transmit the data over the USB.

3.1.4 USB Host Controller

All communication on USB originates at the host under software control. The host hardware consists the USB host controller, which initiates transactions over the USB system.

The host controller is responsible for generating the transactions that have been scheduled by the host software. The host controller driver, or HCD, software builds a linked list of data structures in memory that defines the transactions that are scheduled to be performed during a given frame. These data structure, called transfer descriptors, contain all the information the host controller needs to generate the transaction. This information includes:

- USB Device Address
- Type of Transfer
- Direction of Transfer
- Address of Device Driver’s Memory Buffer

The host controller performs writes to a target device by reading data from a memory buffer (supplied by the USB device driver) that is to be delivered to the target device. The host controller performs a parallel to serial conversion on the data, creates the USB transaction, and send over the USB bus.

If a read transfer is required, the host controller builds the read transaction. The target device recognizes that it is being addressed and that data is being requested. The device then transmit data back to the host controller the host controller then performs the serial to parallel conversion on the data and transfers the data to the device driver’s memory buffer. The USB host controller and target devices perform error checks during a transaction. Errors detected are recognized by the host controller to be logged and reported to the host software.
3.1.5 USB Devices

USB devices contain descriptors that specify a given device's attributes and characteristics. This information specifies to host software a variety of features and capabilities that are needed to configure the device and to locate the USB client software driver. The USB device driver may also use device descriptors to determine additional information needed to access the device in the proper fashion. This mechanism is referred to as the Device Framework and must be understood by software in order to configure and access the device correctly. USB devices can be implemented either as full-speed or low-speed devices.

High-Speed Device

High-speed devices see all transactions broadcast over the USB and can be implemented as full-feature devices. These devices accept and send serial data at the maximum 12Mb/s rate.

Low-Speed Device

Low-speed devices are limited in not only throughput (1.5Mb/s) but feature support. Furthermore, Low-speed devices only see USB transactions that follow a preamble packet. Low-speed ports remain disabled during full-speed transactions, preventing full-speed bus traffic from being sent over Low-speed cables. Preamble packets specify that the following transaction will be broadcast at low speed. Hubs enable their low-speed ports after detaching a preamble packet, permitting low-speed device to see the low-speed bus activity.

3.2 USB Communication Model

Unlike devices that reside on other common bus structure e.g. SCSI, Ethernet etc; USB devices do not consume system resources. That is, USB devices are not mapped into memory or IO address space. Further more all transactions are originated from the host system. The only systems resources are required by a USB system are the memory locations used by system software and the memory and or IO address space. This eliminates much of the difficulty encountered with standard peripheral implementation that required a considerable amount of IO space and large number of interrupt lines.
3.2.1 Communications Flow

The USB client initiates a transfer when it calls the USB system and requests a transfer. USB client driver supply a memory buffer used to store data when transferring data to or from the USB device. Each transfer between a given register (or endpoint) within a USB device and the client driver occurs via a communication pipe that USB system software establishes during device configuration. USB system software splits the client's request into individual transactions that are consistent with the bus bandwidth requirements of the device and the USB protocol mechanisms.

The requests are passed to the USB Host Controller Driver which in turn schedules the transaction to be performed over the USB. The host controller performs the transaction based on the contents of a transfer descriptor that is built by the HCD. The HCD knows all the information necessary to perform the required transaction via the USB. The key information contained within a transfer descriptor includes.

- Address of the target USB device
- Type of transfer to be performed
- Size of the data packet
- Location of the client’s memory buffer

The host controller may have registers that are mapped into the processor’s IO or memory address. These registers control the operation of the host controller and must be loaded with values by the HCD to ensure desired operation. For example, a register is loaded with an address pointer that specifies the memory location where the transfer descriptors reside.

The host controller fetches the transfer descriptors that have been built by the host controller driver. Each descriptor defines a given transaction that must be performed to satisfy a client’s transfer request. The host controller generates the USB transaction that is specified by each transfer descriptor. Each transaction results in data being transferred either from the client buffer to the USB device or from the device to the buffer depending on the direction of the transfer. When the entire transfer has completed, USB system software notifies the client driver.
3.2.2 Transfers, IRPS, Frames, and Packets

Transfers are initiated by the client driver when it issues a transfer request to the USB driver. Ultimately, the transaction is performed via the low-level packetized transaction over the USB. The following sections discuss each layer involved in completing a USB transfer.

3.2.2.1 Transfers

Each USB function is designed with a collection of endpoints, used by the client driver when accessing its function. Each endpoint has particular transfer characteristics that it supports. For example, when transferring information to a speaker the data transfer must continue at a constant data rate to prevent distortion of the audio. Other endpoints may have different characteristics and thus require a different transfer type. The transfer types supported by USB include:

- Isochronous Transfers
- Bulk Transfer
- Interrupt Transfers
- Control Transfer

Client drivers understand the nature of the transfer related to each endpoint associated with its function as does the USB driver. This information is determined by reading descriptors from the device.

3.2.2.2 The USB Driver, IRPS and Frames

When a client driver wishes to perform a transfer to or from a given endpoint, it calls the USB driver to initiate the transfer. The requested transfer is called an IO Request Packet (IRP). Some transfers consist of a large block of data. Since USB is a shared bus (i.e. many device use the same bus at the same time), a single device cannot typically perform an entire block transfer across USB at one time. Rather, a transfer is typically split up and performed in segment (called transactions) over a longer period of time. This ensures that a portion of the USB bandwidth can be allocated for the other USB devices.

USB communication is based on transferring data at regular (1ms) intervals called frames. Each USB device requires a portion of the USB bandwidth be allocated during these 1ms frames. Bandwidth allocation depends on the required throughput of the device (as specified by device descriptors) and the available USB bandwidth not
used by other USB devices. As each USB device is attached and configured, system software parses its device descriptors to determine the amount of bus bandwidth it requires. Software checks the remaining bandwidth and if the device’s requirements can be satisfied it is configured. If the bandwidth required by the device is not available, due to bus bandwidth already allocated to other devices previously attached, the device will not be configured and the user will be notified.

Not every USB device will necessarily transfer data during each frame. For example, host software will poll the keyboard every nth frame to check for keystrokes. Devices are allocated a portion of the overall bus bandwidth that they require during each frame. This will likely result in large bulk transfers, such as print jobs, being split over a fairly large number of 1ms frames. The actual number of frames required depends on the transfer capability of the printer’s USB interface, specified limitations placed on bulk transfers, and the amount of bus bandwidth being used by other devices currently installed on the USB.

3.2.2.3 The Host Controller Driver and Transaction
The host controller driver receives the packet requests from the USB driver and schedules them to be performed during a series of frames. The scheduling order is based on an algorithm defined by the host controller driver. The algorithm is based on USB transfer capabilities and limitations.

Scheduling is performed by building a series of data structures (called transfer descriptors) that define each sequential to be performed over the USB. The host controller reads and interprets these transfer descriptors and executes the USB transaction described in this thesis.

3.3 Device Framework
The device framework provides three logical layers that describe the relationship between the host hardware and software and a corresponding view of each USB device. Figure 3.3.1 illustrates these layers and the relationships between the host and a given USB device. The layered approach helps to explain the relationships between each piece of host software and the responsibilities each has in the USB system. The separate layers are provided to each understanding of the USB communication and are discussed in the following sections.
3.3.1 USB Bus Interface Layer

The USB bus interface layer provides the low-level transfer of data over the USB cables. This layer consists of the:

- Physical connection
- Electrical signaling environment
- Packet transfer mechanisms

This layer represents the actual transfer of data across the USB cable between the host system and the USB devices. The host side consists of the USB host controller, while the USB side consists of the USB interface within the device.

3.3.2 USB Device Layer

The USB device layer represents the portion of USB that comprehends the actual USB communication mechanism and the nature of the transfer required by a USB functional device. This layer consists of USB system software on the host side and logical view of the USB device on the device side. USB system software views a logical device on the device side as a collection of endpoints that comprise a given functional interface.
Figure 3.1 Device Framework – Software’s View of Hardware
USB system software provides the services needed to interface client software with its USB function. USB system software has specific knowledge of the USB transfer mechanisms and must allocate bus bandwidth for the community of USB devices. The logical USB device represents the collection of endpoints through which a client communicates with its function. USB system software views these endpoints via the device descriptors, which are parsed by the USB system software to obtain the transfer characteristics of a given device. These characteristics in conjunction with system software’s knowledge of the USB transfer mechanisms permit bus bandwidth to be reserved for each functional device as it is configured.

USB system software performs a variety of key functions as including:

- Device attachment/detachment detection
- Device configuration
- Bandwidth allocation
- Managing control flow between client and device
- Managing data flow between client and device
- Collecting status and transaction statistics
- Transaction scheduling
- Controlling the electrical interface

One set of USB system software exists in the system to manage accesses to all USB devices attached to USB cable. USB system software consists of the following entities:

**USB Driver (USBD)**
Provides interface and services for client software drivers, allocates bus bandwidth, and manages configuration process.

**USB Host Controller Driver**
Controls the operation of the host controller, schedules the transactions, and monitors completion status of transactions.
3.3.3 Function layer
This layer represents the relationship between client software and a given device’s functional interface. Each interface consists of a particular class of device that a matched class driver is designed to manipulate. USB client software cannot access their function directly as is typically done in other environments, since they are not mapped directly into memory and IO address space. Instead USB device drivers must use the USBD programming interface to access their devices. USB clients view their USB devices as consisting of a given interface, which they know how to manipulate. USB system software must report the interface type and other device characteristics to USB clients.

3.4 USB Peripheral Connection
USB provides a single type of connector for attaching peripheral to a system. USB 1.1 supports two different speeds of USB devices:
- Low-speed devices – 1.5 Mb/s (megabits per seconds).
- High-speed devices – 12 Mb/s (megabits per seconds).

Devices such as keyboards and mouse typically operate at low-speed, while other devices such as digital telephone must operate at full-speed. Each USB port must support the attachments of both low and full speed devices, unless the port has permanently attached device.

When a transaction is initiated by the host system, all full-speed devices will see the transaction. Each transaction contains an address field that identifies the targeted device. Low-speed devices only see low-speed transaction, which are always preceded by high-speed “preamble” transaction that direct to enable their low-speed ports.

USB uses differential signaling to perform serial transmission of information between host and the USB devices. Due to related EMI differences the cables used for low-speed verses full-speed devices are subject to different electrical characteristics.
3.5 USB Signaling Environment

USB serial data is NRZI (Non-Return to Zero, Inverted) encoded before being transferred via the USB cables using the differential signaling. NRZI encoding is performed first by the USB agent that is sending information. Next, the encoded data is then driven onto the USB cable by the differential driver. The receiver amplifies the incoming differential data and delivers the NRZI data to the decoder. Encoding and differential signaling are used to help ensure data integrity and eliminate noise problems.

3.5.1 NRZI Encoding

Data transferred via the USB is encoded using NRZI encoding to help ensure integrity of data delivery, without requiring a separate clock signal be delivered with the data. NRZI is by no means a new encoding scheme. It has been used for decades in a wide variety of applications. Figure 3.2 illustrates a serial data stream and the resulting NRZI data. Transition in the NRZI data stream represents 0s while no transition represents 1s. The NRZI encoder must maintain the synchronization with the incoming data stream to correctly sample the data. The NRZI data stream must be sampled within a data window to detect whether a transition has occurred since the previous bit time. The decoder samples the data stream during each bit time to check for transitions.

Transitions in the data stream permit the decoder to maintain synchronization with the incoming data, thereby eliminating the need for separate clock signal. However that a long string of consecutive ones results in no transitions, causing the receiver to eventually lose synchronization. The solution is to employ bit stuffing.

<table>
<thead>
<tr>
<th>Idle</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NRZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2 NRZI Encoded Data
### 3.5.2 Bit Stuffing

Bit stuffing forces transitions into the NRZI data stream in the event that 6 consecutive ones are transmitted. This ensures that the receiver detects a transition in the NRZI data stream at least every seventh bit time. This enables the receiver to maintain the synchronization with the incoming data. The transmitter of NRZI data is responsible for inserting a zero (stuffed bit) into the NRZI stream. The receiver must be designed to expect an automatic transition following six consecutive ones and discard the one bit that immediately follows the sixth consecutive ones.

![Diagram](image)

**Figure 3.3 Stuffed Bit**

The top line in Figure 3.3 illustrates raw data being delivered to the receiver. The data stream contains a string of eight consecutive ones. The second line represents the raw data with the stuffed bit added. A stuff bit is inserted between the sixth and seventh 1s in the raw data stream. Delivery of the seventh one is delayed by one data time so that the stuffed bit can be inserted. The receiver knows that the bit following the six consecutive ones will be a stuffed bit (0), causing it to be ignored. If the seventh bit in the raw data was 0, the stuffed bit would still be inserted in the same location.
3.5.3 Differential Pair Signaling
USB employs differential pair signaling to reduce signal noise. Differential drivers and receivers are used to help reduce the sources of signal noise listed below:

- **Amplifier noise** – the noise introduced when a signal is amplified by both the driver and receiver of a signal.
- **Cable noise** – the noise picked up by the cable due to electromagnetic fields.

3.6 USB Transfer Types
USB supports four transfer types. The transfer characteristics relate to the requirement of the application.

Each transfer type determines various characteristics of the communication flow including the following:

- Data format imposed by the USB
- Direction of communication flow
- Packet size constraints
- Bus access constraints
- Error handling.

The following four transfer types have been defined by the USB specification, each which reflects the nature of the transfers that may be required by the device endpoint.

3.6.1 Interrupt Transfer
An interrupt transfer is used for devices that are typically thought of as interrupt driven devices in legacy PC implementation. Since the USB does not support hardware interrupt, USB devices that are interrupt driven must be polled periodically to see if the device has data to transfer. For example, in legacy PC system a hardware interrupt is generated each time a key is pressed on the keyboard to notify the processor that it must execute a software interrupt routine to service the keyboard where as a USB keyboard is polled periodically to determine if keyboard data (e.g. resulting from a key being pressed) is ready to be transferred. The polling rate of course is critical; it must be enough to ensure that data is not lost but not so frequent that bus bandwidth is needlessly reduced.
3.6.2 Bulk Transfer
A bulk transfer is used for transferring large blocks of data that have no periodic or transfer rate requirement. An example of bulk transfer is a print job being transferred to a USB printer. While transfer speed is important for performance, a print job delivered slowly does not result in lost or corrupted data.

3.6.3 Isochronous Transfer
An isochronous transfer requires a constant delivery rate. Applications that use isochronous transfers must ensure that rate matching between the sender and the receiver can be accomplished. For example, a USB microphone and speaker would use isochronous transfers to ensure that no frequency distortion results from transferring data across the USB.

3.6.4 Control Transfer
Control transfers are used to transfer specific requests to USB devices and are most commonly used during device configuration. A special transfer sequence is used to pass requests (commands) to a device, sometimes followed by a data transfer, and concluded with completion status.
This chapter details individual transactions that comprise the transfer types discussed in Chapter 3. Each transfer broadcast over the USB consists of combinations of packets. These packets are combined to define individual transactions that are performed as a part of a larger transfer. Each transaction type is defined, along with the individual packets that comprise them. It also discusses the errors in transactions.

### 4.1 Packets – The Basic Building Blocks of USB Transactions

Transactions typically consist of three phases, or packets, as illustrated in Figure 4.1 However, a transaction may consist of one, two, or three phases depending on the type:

![Figure 4.1 Many USB Transactions Consist of Three Phases](image)

**Token Packet phase**

Each transaction begins with a token phase that defines the type of transaction. A device address is also included in this phase when the transaction targets a specific USB device. Some tokens stand-alone and thus are not followed by any additional packets, while others are always followed by either one or two additional packets.

**Data Packet phase**

Many transaction types include a data phase that carries the payload associated with the transfer. The data phase can carry a maximum payload of 1023 bytes of data during a single transaction; however, the maximum payload permitted depends on the transfer type being performed.
Handshake Packet phase

All USB transfers are implemented to guarantee data delivery, except isochronous transfers that have no handshake phase. The handshake phase provides feedback to the sender of data whether or not data was received without error. If errors are encountered during a transaction, retries are attempted.

<table>
<thead>
<tr>
<th>Synchronization Sequence</th>
<th>Packet ID</th>
<th>Packet-Specific Information</th>
<th>CRC Bits</th>
<th>EOP</th>
</tr>
</thead>
</table>

Packet

Figure 4.2 Packet Format

A packet is the mechanism used to perform all USB transactions. Figure 4.2 illustrates the basic format of USB packet. Immediately preceding each packet is a synchronization sequence that permits USB devices to synchronize to the data rate of the incoming bits within the packet. The type of the packet is defined by a bit pattern called a packet ID. Following the ID is packet-specific information (e.g. an address or data) that varies depending on the packet type. Finally, each packet ends with a sequence of Cyclic Redundancy Check (CRC) bits, used to verify correct delivery of the packet-specific information. The end of each packet is identified by an end of packet (EOP) state. Each type of packet is detailed in the following section.

4.1.1 Synchronization Sequence

Figure 4.3 illustrates the synchronization sequence. The synchronization sequence consists of eight bits starting with seven consecutive logic 0s and ending with logic 1. Since zeros are encoded with transitions of the differential data lines, the seven zeros each create a transition during each bit time, thus providing a clock that can be synchronized to. The synchronization sequence also alerts USB receivers that a packet is being sent, which will immediately follow the 8-bit synchronization sequence.
Packets can be broadcast over the USB at either full-speed (12 Mb/s) or low-speed (1.5 Mb/s) and the speed governs the rate at which bits within the packet are transferred. The USB receiver must detect the logic state of each bit value within the packet by sampling the data lines at the correct point during each bit time. The synchronization sequence is transmitted at the transfer speed being used, allowing the receiver to synchronize to either incoming data rate.

![Synchronization Sequence](image)

**Figure 4.3 Synchronization Sequence**

However, that low-speed devices cannot communicate at full-speed. As a result, low-speed cable segment carry only low-speed transactions. USB host keep low-speed ports disabled until a low-speed transaction is to be performed. Host must be given time to enable their low-speed ports prior to the start of the low-speed transfer.

### 4.1.2 Packet Identifier

Packet identifiers define the purpose and thus the content of a given packet. Packet are grouped into three major categories:

**Token Packet**

Token packets are sent at the beginning of a USB transaction to define the transfer type. (E.g. transfer to or from a USB device).
Data Packet

These packets follow token packets during transactions that require data payloads be transferred to or from USB devices.

Handshake Packet

Handshake packets are typically returned from the receiving agent back to the sender, thus providing feedback relative to the success or failure of the transaction. In some cases, the USB device being requested to send data to the system may send a handshake packet to indicate that it currently has no data to send.

Special Packet

Currently the only special packet defines is the preamble packet used to enable low-speed ports.

The format and length of a packet depends on its type. Token packets are all four bytes in length, but contain different information that describes some aspect of the transaction that it defines. Data packets are variable length depending on the transfer type associated with the transaction. For example, the data payload for bulk transfers is limited to 64 bytes during each transaction, while the data payload limit for isochronous transfers is 1024 bytes.

![Data, Synchronization Pattern, Packet Type I.D. (PID), Type Field, Check Field](image)

**Figure 4.4 Packet Identifier Format**
Refer to Figure 4.4 Packets IDs consists of a four-bit identifier field followed by a four-bit check field. The check field contains the inverted value (1’s complement) of the packet ID value. Immediately following the eight-bit packet ID is the packet-specific information.

4.1.3 Packet-Specific Information
Each packet contains information that is related to the job it performs. The information may consist of a USB device address, a frame number, data to be transferred to or from the USB device, etc. This information is crucial to the success of a given transaction and is verified at the end of the packet with CRC bits.

4.1.4 Cyclic Redundancy Checking (CRC)
The USB agent that sends a given packet computes either a 5-bit or 16-bit CRC depending on its type. Data packets use a 16-bit CRC, while all other packets type use the 5-bit CRC. The CRC is generated and checked for the packet-specific data only, since the packet ID has its own check bits.

4.1.5 End of Packet (EOP)
The end of each packet is signaled by the sending agent by driving both differential data lines low for two bit times followed by an idle for 1-bit time. The agent receiving the packet recognized EOP when it detects the differential data lines both low for one bit time.

4.2 Packet Types

4.2.1 Token Packets
Token packets define the type of transaction that is to be broadcast over the USB. All transaction begins with a token packet. Four types of Token packets are defined by the USB 1.1 specification:

SOF (Start of Frame) --- Indicates start of the next 1ms frame.
IN --- Specifies USB transaction used to transfer data from a target USB device to the system.
OUT --- Specifies a USB transaction used to transfer data from the system to a target USB device.

SETUP --- Indicates the start of a control transfer. SETUP is the first stage of a control transfer and is used to send a request from the system to the target USB device.

<table>
<thead>
<tr>
<th>PID Type</th>
<th>PID Name</th>
<th>PID [3:0]</th>
<th>Description of Token Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>SOF</td>
<td>0101b</td>
<td>Contains start of frame (SOF) marker and frame number. The SOF token is used by isochronous endpoints to synchronize its transfers.</td>
</tr>
<tr>
<td>Token</td>
<td>SETUP</td>
<td>1101b</td>
<td>Contains USB device address + endpoint number. Transfer is from host to device for setting up a control endpoint (e.g. configuration)</td>
</tr>
<tr>
<td>Token</td>
<td>OUT</td>
<td>0001b</td>
<td>Contains the USB device address + endpoint number. The transfer is from host to device.</td>
</tr>
<tr>
<td>Token</td>
<td>IN</td>
<td>1001b</td>
<td>Contains the USB device address + endpoint number. The transfer is device to host.</td>
</tr>
</tbody>
</table>

Table 4.1 USB Token Packets

Table 4.1 lists the Token Types supported and describe the token-specific information embedded within the packet and the action specified. Each token is identified by its packet ID, as shown in column three.

4.2.1.1 SOF Packet

SOF provides a way for target devices to recognize the beginning of a frame. As an example, isochronous application can use SOF to trigger and synchronize the start of the transfer at the beginning of a specified 1ms frame. The SOF packet is broadcast to all full-speed devices at the beginning of each frame. Embedded within the SOF packet is an 11-bit frame number as illustrated in Figure 4.5 The frame number is verified by the receiver with the five CRC bits. The SOF packet defines a transaction consisting
Solely of the token packet. No data or handshake packet is associated with an SOF packet; therefore delivery is not guaranteed. However, USB targets must perform the checks and take the appropriate action as follows:

- **PID check error** -- ignore the packet
- **Frame CRC error** – ignore the frame number

When software wishes to read information from a given device an IN token is used. The IN packet notifies the target USB device that data is being requested by the system. IN transaction may be used in a variety of USB transfers types including:

- Interrupt transfers
- Bulk transfers
- Data phase of control transfers
- Isochronous transfers

As illustrated in Figure 4.6 an IN token packet consists of the ID type field, the ID check field, the USB device and endpoint addresses, and five CRC bits. An IN transaction starts with an IN packet broadcast by the host followed by the data packet.
returned from the target USB device, and in some cases concluded with a handshake packet sent from the host back to the target device to confirm receipt of the data. IN transactions used during isochronous transfers do not include a handshake packet.

The amount of data that can be transferred during an IN transfer depends on the transfer type being performed.

4.2.1.3 OUT Packet
System software specifies an OUT transaction when data is to be transferred to a target USB device. Three types of transfers employ OUT transactions

- Bulk transfers
- Data phase of control transfers
- Isochronous transfers

Figure 4.7 illustrates the contents of an OUT token packet. An OUT packet consists of the packet ID or type field, the type check field, the USB target device and endpoint ID, and a 5-bit CRC. The OUT token packet is followed by a data packet, and a handshake packet (for bulk transfers only). The data payload size is governed by the transfer employing the OUT transaction.
SETUP packets are used only during the setup stage of control transfers. The SETUP transaction starts a control transfer, and is defined as the setup stage. A SETUP transaction is similar in format to an OUT transaction: the SETUP packet is followed by DATA0 packet, and an acknowledge packet. The SETUP packet transfers a request to be performed by the target device. A variety of requests are supported by the host and other USB devices. Depending on the request, the SETUP transaction may be followed by one or more IN or OUT transactions (data stage), or may be followed only by a status stage consisting of a final data packet transferred from the endpoint to the host system. Figure 4.8 illustrates the format of SETUP packet.

![OUT Token Packet Format](image)

**Figure 4.7 OUT Token Packet Format**

**4.2.1.4 SETUP Packet**

SETUP packets are used only during the setup stage of control transfers. The SETUP transaction starts a control transfer, and is defined as the setup stage. A SETUP transaction is similar in format to an OUT transaction: the SETUP packet is followed by DATA0 packet, and an acknowledge packet. The SETUP packet transfers a request to be performed by the target device. A variety of requests are supported by the host and other USB devices. Depending on the request, the SETUP transaction may be followed by one or more IN or OUT transactions (data stage), or may be followed only by a status stage consisting of a final data packet transferred from the endpoint to the host system. Figure 4.8 illustrates the format of SETUP packet.
Data packets carry the data payload associated with a given transaction. Direction of a data packet transfer is specified by the transaction type and may be used to transfers data either to or from a target USB device as listed in Table 4.2

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Direction of Data Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN transaction</td>
<td>From USB device</td>
</tr>
<tr>
<td>OUT transaction</td>
<td>To USB device</td>
</tr>
<tr>
<td>SETUP transaction</td>
<td>To USB device</td>
</tr>
</tbody>
</table>

Two types of data packets (Data0 and Data1) are defined to support synchronization of long transfer between sender and receiver. For example, if a long transfer is being sent from the host to a printer, the transfer will be performed in small blocks via a relatively large number of frames. To verify that a data transaction is not missed during a long transfer, a technique called data toggle can be employed.
The format of Data0 packet is illustrated in Figure 4.9 and the format of Data1 is illustrated in Figure 4.1.1.
4.2.3 Handshake Packet

USB devices use handshake packets to report the completion status of a given transaction. The receiver of the data payload (the target device) is responsible for sending a handshake packet back to the sender. Three possible results can be reported via different handshake packets:

**Acknowledge packet (ACK)**

This acknowledges error free receipt of the data packet.

**No Acknowledge packet (NAK)**

Reports to the host that the target device is temporarily unable to accept or return data. During interrupt transactions NAK signifies that no data is currently available to return to the host (i.e. no interrupt request is currently pending).

**Stall packet (STALL)**

Used by the target to report that it is unable to complete the transfer and that software intervention will be required for the device to recover from the stall condition.
The Handshake Packets are illustrates in Figure 4.1.2

**Figure 4.1.2 Handshake Packet Format**
4.3 Error in Transaction

4.3.1 Packet Errors

The USB devices detect three types of packet errors:

- Packet ID (PID) checks
- Cyclic Redundancy Checks (CRC)
- Bit stuff errors

If any of these error conditions exists, the receiver of the packet must ignore the packet and not respond to it in any manner. The receiver consequently never sends a packet back to the transmitter if the packet just received contains an error. The type of packet error detected is not significant to the USB devices or Host as it relates to error recovery. However, the host system may capture statistics regarding the nature of packet failures.

![Packet Identifier Diagram](image)

**Type**
- Token
- Data
- Handshake
- Special

**Check**
- 1's compliment of Type

*Figure 4.1.3 PID Check*
4.3.1.1 PID Checks

Each packet broadcast over the USB starts with a Packet ID (PID) consisting of four bits and is followed by a PID check field as illustrated in Figure 4.1.3. The check field is the PID inverted (1’s complement). All potential USB target devices must perform the PID check and ignore the packet if an error is detected, since the definition of the packet is unknown.

4.3.1.2 CRC Errors

Each packet contains CRC bits used to validate the information send following the Packet ID field. The nature of this information varies depending on the packet type. Each packet contains either 5 or 16 CRC bits determine by the packet’s potential size. Furthermore, the packet type the fields covered by the CRC vary according to the packet type. Table 4.3 gives the packet type and number of crc bit required.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Fields</th>
<th>Max. Size of fields</th>
<th>Numbers of CRC bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start of Frame</td>
<td>Frame number</td>
<td>11-bits</td>
<td>5</td>
</tr>
<tr>
<td>IN</td>
<td>Device and endpoint address</td>
<td>11-bits</td>
<td>5</td>
</tr>
<tr>
<td>OUT</td>
<td>Device and endpoint address</td>
<td>11-bits</td>
<td>5</td>
</tr>
<tr>
<td>SETUP</td>
<td>Device and endpoint address</td>
<td>11-bits</td>
<td>5</td>
</tr>
<tr>
<td>DATA0</td>
<td>Data payload</td>
<td>1023 bytes</td>
<td>16</td>
</tr>
<tr>
<td>DATA1</td>
<td>Data payload</td>
<td>1023 bytes</td>
<td>16</td>
</tr>
<tr>
<td>ACK</td>
<td>NA — packet ID only</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>NAK</td>
<td>NA — packet ID only</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>STALL</td>
<td>NA — packet ID only</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>PEEAMBLE</td>
<td>NA — packet ID only</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 4.3 Packet Type and CRC
The 5-bit CRC field for the token packets is based on a generator polynomial:
\[ G (X) = X^5 + X^2 + 1 \]

The bit pattern representing this polynomial is 00101b. The five-bit residual at the receiver will be 01100b if all bits are received correctly.

The 16-bit CRC field for the data packets is based on the generating polynomial:
\[ G (X) = X^{16} + X^{15} + X^2 + 1 \]

The bit pattern representing this polynomial is 100000000000101b. If the data is received without errors then the 16-bit residual will be 1000000000001101b.

The CRC bit stream will contain stuffed bits if the CRC contains six consecutive ones.

4.3.1.3 Bit Stuff Errors

Bit stuffing ensures that the sender and receiver of NRZI data maintain synchronization by forcing a transition into the data stream after detecting six consecutive 1s.

USB receivers expect to see a guaranteed transition (stuffed bit) in the data stream after six consecutive 1s. If a stuffed bit is not present, this indicates that the packet has been corrupted or that the sender is not properly generating stuffed bits, or that the receiver is not decoding the NRZI data correctly.
CHAPTER 5 IMPLEMENTATION

This chapter describes the implementation of the Protocol Engine for USB 1.1 Device model. It also describes the interfacing signals for Device Receiver SIE, Device Transmitter SIE and Digital Phase Locked Loop modules and their implementation in Verilog HDL.

5.1 Functional Block Diagram

![Functional Block Diagram]

Figure 5.1 Functional Block Diagram
5.2 Implementation Overview

The Implementation of Protocol Engine for USB 1.1 Device model is carried out in a VERILOG Hardware Description Language (VERILOG HDL). The design is divided into three modules:

- Device Receiver SIE
- Device Transmitter SIE
- Digital Phase Locked Loop

The USB 1.1 ATX (Analog Transceiver) receives the bi-directional D+ and D- lines of the USB bus. It transfers this data in the unidirectional interface between ATX (Analog Transceiver) and SIE. The Device Receiver SIE (Serial Interface Engine) receives this serial data and converts it into a parallel data stream. The parallel data is saved in the local buffer.

The Device Transmitter SIE receives the parallel data, serialize this parallel data and transmit this data on the USB bus. All the physical layer protocol are implemented in SIE block.

The Digital Phase Locked Loop generates the 12 MHz clock if the device is full speed and 1.5 MHz clock if the device is Low speed. The generated clock is synchronized with the incoming data line from the ATX. This module is also generate the signal for end of packet (EOP) detection. If the (EOP) condition is detected then \( SE0\_detected \) signal goes high which indicates the end of packet.
5.2.1 Device Receiver SIE Interfacing Signals

The assertion of Reset will force the Receive State Machine into the *Reset* state. The *Reset* state negates Active and RXValid. When the Reset signal is negated the Receive State Machine enters the *RX Wait* state and starts looking for a SYNC pattern on the USB. When a SYNC pattern is detected the state machine will enter the *Strip SYNC* state and assert RXActive. The length of the received SYNC pattern varies and can be up to 32 bits long. As a result, the state machine may remain in the *Strip SYNC* state for several byte times before capturing the first byte of data and entering the *RX Data* state. After 8 bits of valid serial data is received the state machine enters the *RX Data* state, where the data is loaded into the RX Holding Register on the rising edge of CLK and RXValid is asserted. The SIE must clock the data off the DataOut bus on the next rising edge of CLK.

![Figure 5.2 Receiver State machine](image)
This is the USB clock, which is generated by DPLL module. This clock is synchronized with the incoming bit stream. All the transitions are take place at the positive edge of this clock.

**RX**

**USB_CLK**
This is the USB clock, which is generated by DPLL module. This clock is synchronized with the incoming bit stream. All the transitions are take place at the positive edge of this clock.

**RX_RCV**
This signal is the input to device SIE. It carries the data bits transmitted by the Host Transmitter. The transmitted bits are in NRZI (Non Return to Zero Inverted) format.

**Pll_reset_n**
This is the incoming signal from the DPLL module. When it goes low, it reset the device SIE. All intermediate signals take their default value.

**RX_ENABLE**
This is the input signal from device controller. When it becomes high then the Device Receiver SIE goes in receive state i.e. Device Receiver SIE is expecting some data bytes transmitted by Host.

**SE0_detected**
This is the incoming signal from the DPLL module. When it becomes high then it indicates the end of packet.
**RX_DEV_ADD [6:0]**

Every device has the 7-bit unique address. This 7-bit address is used to correct reception of the data bits by that device.

**RX_DEV_ENDP [3:0]**

USB 1.1 specification supports 16 endpoint numbers. This is the address line for a particular endpoint number.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USB_CLK</strong></td>
<td>In</td>
<td>DPLL generated clock</td>
</tr>
<tr>
<td><strong>RX_RCV</strong></td>
<td>In</td>
<td>Differential line from ATX</td>
</tr>
<tr>
<td><strong>Pll_reset_n</strong></td>
<td>In</td>
<td>Reset signal to Device Receiver SIE</td>
</tr>
<tr>
<td><strong>RX_ENABLE</strong></td>
<td>In</td>
<td>When high, SIE goes in receiving mode</td>
</tr>
<tr>
<td><strong>SE0_detected</strong></td>
<td>In</td>
<td>When high, indicates End of packet</td>
</tr>
<tr>
<td><strong>RX_DEV_ADD [6 : 0]</strong></td>
<td>Out</td>
<td>Received device address</td>
</tr>
<tr>
<td><strong>RX_DEV_ENDP [3 : 0]</strong></td>
<td>Out</td>
<td>Received endpoint number</td>
</tr>
<tr>
<td><strong>RX_PID [3 :0]</strong></td>
<td>Out</td>
<td>Received packet identification code</td>
</tr>
<tr>
<td><strong>RX_LENGTH [11: 0]</strong></td>
<td>Out</td>
<td>Total bytes received in the packet</td>
</tr>
<tr>
<td><strong>RX_COMPLETE</strong></td>
<td>Out</td>
<td>Indicates reception of a packet</td>
</tr>
<tr>
<td><strong>RX_sof_no [10: 0]</strong></td>
<td>Out</td>
<td>Received frame number</td>
</tr>
<tr>
<td><strong>RX_buf_wr</strong></td>
<td>Out</td>
<td>Write enable for memory buffer</td>
</tr>
<tr>
<td><strong>RX_buf_add [11: 0]</strong></td>
<td>Out</td>
<td>Memory buffer address</td>
</tr>
<tr>
<td><strong>RX_data_bus [7: 0]</strong></td>
<td>Out</td>
<td>Data bus for memory buffer</td>
</tr>
<tr>
<td><strong>RX_sie_error [5:0]</strong></td>
<td>Out</td>
<td>Indicates error occurred during receiving data</td>
</tr>
</tbody>
</table>

Table 5.1 interfacing signals for Device Receiver SIE
**RX_PID [3:0]**
In USB transmission, the data transfer occurs through packets only. There are four types of packets. Each type has two or more sub type. These 4 bits indicates the specific packet.

**RX_LENGTH [11:0]**
These are the output lines from device SIE. These lines indicate the number of bytes received in a data packet.

**RX_COMPLETE**
This is the outgoing signal. When all the data bytes of a packet are received then this signal goes high. All the outgoing signals are valid after this signal become high.

**RX_sof_no [10:0]**
In USB transmission, the whole transmission is divided into frames. Each frame is of 1ms. These lines give the frame number. And it is used to check the transmission is continuous or not.

**RX_buf_wr**
This is the outgoing signal from device receiver SIE which goes to memory buffer. On every 8th bit received it becomes high so that every byte can be save in memory buffer.

**RX_buf_add [11:0]**
These are the address lines for the memory buffer. For every data bytes received the RX_buf_add is incremented by one and data byte is stored at this location.

**RX_data_bus [7:0]**
These are the data line, which carries the one data byte. When the RX_buf_wr is high then the data on these lines is stored in the memory buffer.

**Bus timeout**
This is the outgoing signal. If sync is not detected (syncOK = 1) within 18 clock cycles after the RX_ENABLE becomes high then this signal become high.
**RX_sie_error [5:0]**

This is a 6-bit register. When any one of the following errors has occurred then corresponding bit of RX_sie_error become high.

Table 5.2 list the RX_sie_error register.

<table>
<thead>
<tr>
<th></th>
<th>Bus timeout</th>
<th>Bitstuff error</th>
<th>PID error</th>
<th>CRC 16 error</th>
<th>NA</th>
<th>SE0 error</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.2 Device Receiver SIE Error Format**

1. **Bus timeout**

   If sync is not detected (syncOK = 1) within 18 clock cycles after the RX_ENABLE becomes high then the corresponding bit RX_sie_error [5] become high.

2. **Bit stuff error**

   After six consecutive ones if the next received bit is not zero then RX_sie_error[4] become high.

3. **PID error**

   A PID error exists if the four PID check bits are not complements of their respective packet identifier bits. When the higher 4 bit (RX_PID [7:4]) of received PID register are not the complement of lower 4 bit (RX_PID [3:0]) then RX_sie_error [3] become high.

4. **CRC error**

   The USB specification uses the Cyclic Redundancy Checksums (CRC) to protect all non-PID fields in token and data packets from errors during transmission.

   If the generated CRC field is not equal to ‘01100b’ for token packet and ‘1000000000001101b’ for data packet then it set the second bit of RX_sie_error [2].

5. **SE0 error**

   If the SE 0 condition is detected before the last bit in a data packet received then RX_sie_error [0] become high.
5.2.2 Device Transmitter SIE Interfacing Signals

The behavior of the Transmit State Machine is described below and illustrated in Figure 12. The Reset signal forces the state machine into the Reset state which negates TXReady. When Reset is negated the transmit state machine will enter the TX Wait state.

In the TX Wait state, the transmit state machine looks for the assertion of TXValid. When TXValid is detected, the state machine will enter the Send SYNC state and begin transmission of the SYNC pattern. When the transmitter is ready for the first byte of the packet (PID), it will enter the TX Data Load state, assert TXReady and load the TX Holding Register. The state machine may enter the TX Data Wait state while the SYNC pattern transmission is completed.

TXReady is used to throttle transmit data. The state machine will remain in the TX Data Wait state until the TX Data Holding register is available for more data. In the TX Data Load state, the state machine loads the Transmit Holding register. The state machine will remain in the TX Data Load state as long as the transmit state machine can empty the TX Holding Register before the next rising edge of CLK.
USB 1.1 devices operate at either low speed (1.5 Mb/s) or full speed (12 Mb/s). If this signal is low its mean that the device is operating at low speed and if it is high then device is operating at full speed.

**TX_CLK**

This clock is used for transmission of data on USB bus. All the transitions are occurs at the positive edge of this clock.

**TX_ENABLE**

This is the input signal from device controller. When it becomes high then the device transmitter SIE goes in transmit state i.e. Device Transmitter SIE start transmitting data packet or handshake packet.

**reset_N**

This is the incoming signal. When it goes low, it reset the Device Transmitter SIE. All intermediate signals take their default value.
**TX_LENGTH [11:0]**

These are the incoming lines from device controller. These lines indicate the number of bytes to be transmitted in a data packet.

**TX_handshake_pid [7:0]**

In USB transmission, data transfer occurs through packets only. In USB specification there are 10 packet types. Each packet has 4-bit address. When the device is required to send the handshake packet then it send the four bit of that handshake packet through these lines to Device Transmitter SIE. And the other four bits are complement of PID address.

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEED</strong></td>
<td>In</td>
<td>Indicates whether low speed or Full speed</td>
</tr>
<tr>
<td><strong>TX_CLK</strong></td>
<td>In</td>
<td>USB transmission clock</td>
</tr>
<tr>
<td><strong>TX_ENABLE</strong></td>
<td>In</td>
<td>When high, transmit SIE goes in transmit state.</td>
</tr>
<tr>
<td><strong>reset_N</strong></td>
<td>In</td>
<td>Reset signal to Device Transmitter SIE</td>
</tr>
<tr>
<td><strong>TX_LENGTH [11: 0]</strong></td>
<td>In</td>
<td>Total bytes to be transmitted from the packet</td>
</tr>
<tr>
<td><strong>TX_handshake_pid [7: 0]</strong></td>
<td>In</td>
<td>Handshake PID</td>
</tr>
<tr>
<td><strong>TX_data_pid [7 :0]</strong></td>
<td>In</td>
<td>Data PID</td>
</tr>
<tr>
<td><strong>send_handshake</strong></td>
<td>In</td>
<td>If high then sending packet is handshake</td>
</tr>
<tr>
<td><strong>send_data</strong></td>
<td>In</td>
<td>If high then sending packet is data</td>
</tr>
<tr>
<td><strong>TX_buf_data [7: 0]</strong></td>
<td>In</td>
<td>Data bus from memory buffer</td>
</tr>
<tr>
<td><strong>TX_sie_error_control [5:0]</strong></td>
<td>In</td>
<td>Deliberately introduces error.</td>
</tr>
<tr>
<td><strong>TX_Dplus</strong></td>
<td>Out</td>
<td>Differential D+ line</td>
</tr>
<tr>
<td><strong>TX_Dminus</strong></td>
<td>Out</td>
<td>Differential D- line</td>
</tr>
</tbody>
</table>
### Table 5.3 Interfacing signals for Device Transmitter SIE

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_COMPLETE</td>
<td>Out</td>
<td>Indicates transmission of a packet is over.</td>
</tr>
<tr>
<td>TX_buf_add[11:0]</td>
<td>Out</td>
<td>Memory buffer address</td>
</tr>
</tbody>
</table>

**TX_data_pid[7:0]**

In USB transmission, data transfer occurs through packets only. In USB specification there are 10 packet types. Each packet has 4-bit address. When the device is required to send the data packet then it send the four bit of that data packet through these lines to Device Transmitter SIE. And the other four bits are inversion of PID address.

**Send_handshake**

This is the incoming signal from device controller. When this signal become high then the Device Transmitter SIE will transmit the handshake packet.

**Send_data**

This is the incoming signal from device controller. When this signal is high then the Device Transmitter SIE will transmit the data packet.

**TX_buf_data[7:0]**

These are the incoming lines from the memory buffer. These lines carry the data byte to be transmitted to HOST.

**TX_sie_error_control[5:0]**

This is a 6-bit register. When any one bit of this register is high then the Device Transmitter SIE introduced the error deliberately corresponding to that bit.
Table 5.4 list the $TX\_sie\_error\_control$ register.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>NA</td>
<td>Bitstuff error</td>
<td>PID error</td>
<td>CRC 16 error</td>
<td>NA</td>
<td>SE0 error</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.4 Device Transmitter SIE Error Format**

1. **Bitstuff error**
   
   When the bit stuff error bit is high then instead of transmitting zero after six consecutive ones, it will transmit the one. i.e. when $TX\_sie\_error\_control$ [4] = 1 then one is transmitted after six consecutive ones.

2. **PID error**
   
   When the PID error bit is high then instead of transmitting the PID for DATA0 or DATA1 packet, the Device Transmitter SIE transmits 11110011 patterns, this will give the PID error while receiving the PID.

3. **CRC error**
   
   If the crc error bit is high then it will not generate the correct crc bits, instead of EX\_OR operation it will perform ANDing operation upon generator polynomial ‘100000000001101b’.

**TX\_Dplus**

This is the outgoing signal. This is the differential D+ line.

**TX\_Dminus**

This is the outgoing signal. This is the differential D- line.

**TX\_COMPLETE**

This is the outgoing signal. When the transmission of packet is over then this signal goes high.

**TX\_buf\_add [11:0]**

These are the address lines to the memory buffer. For every data byte transmitted $TX\_buf\_add$ is incremented by one and the contents of that address location are loaded into the 8-bit shift register.
5.2.3 Digital Phase Locked Loop Interface

![Diagram of DPLL](image)

**Figure 5.6 DPLL block**

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reset_N</td>
<td>In</td>
<td>When goes low, it reset the DPLL</td>
</tr>
<tr>
<td>clk_4x</td>
<td>In</td>
<td>Multiplied by 4 clock from oscillator</td>
</tr>
<tr>
<td>rx_dp</td>
<td>In</td>
<td>Incoming differential D+ line from ATX</td>
</tr>
<tr>
<td>rx_dm</td>
<td>In</td>
<td>Incoming differential D- line from ATX</td>
</tr>
<tr>
<td>RX_recv</td>
<td>In</td>
<td>Differential line from ATX. Incoming data</td>
</tr>
<tr>
<td>usb_clk</td>
<td>Out</td>
<td>Synchronized multiplied by 1 clock generated from DPLL</td>
</tr>
<tr>
<td>SE0_detected</td>
<td>Out</td>
<td>When high, indicates end of packet</td>
</tr>
<tr>
<td>pll_reset_n</td>
<td>Out</td>
<td>Reset signal generated for device receiver</td>
</tr>
</tbody>
</table>

**Table 5.5 Interface signals of Digital Phase Locked Loop**
reset_N
This is the incoming signal. When it goes low, it reset the DPLL. All intermediate
signals take their default value. This signal is used to generate the pll_reset_n signal for
the Device Receiver SIE

 clk_4x
This is the raw clock of multiplied by 4 type, which is used to generate the 12
MHz clock if the device is full speed and 1.5 MHz clock if the device is low speed. The
generated clock is synchronized with the incoming data line from the ATX

 rx_dp
This is the incoming signal. This is the D+ data line from the ATX (Analog
transceiver)

 rx_dm
This is the incoming signal. This is the D- data line from the ATX (Analog
transceiver)

 RX_rcv
This is the differential data line from the ATX. This line is input to the DPLL
module.

 usb_clk
This is the outgoing clock from the DPLL. This clock is generated by DPLL
module. And it is synchronized with the incoming data line. This clock is 12 MHz if the
device is full speed and 1.5 MHz if the device is low speed.

 SE0_detected
This is the outgoing signal from the DPLL module. In USB transmission, the
end of packet is represented by two logic 0’s followed by logic 1. When this condition is
detected by DPLL then SE0_detected signal goes high, which indicates the end of
packet.

 Pll_reset_n
This is the outgoing signal from the DPLL module. If reset_N goes low then at
the next positive edge of the clk_4x, this signal goes low. This signal is used to reset the
Device Receiver SIE.

5.3 Implementation of Device Receiver SIE

The implementation of Device Receiver SIE is carried out in a various blocks. The details of blocks is as follows

5.3.1 NRZI decoder

The USB 1.1 devices employ NRZI data encoding for transmission of data packet. In NRZI encoding, logic 1 is represented by no change in level and logic 0 is represented by a change in level. So there is need for conversion of NRZI to NRZ. This block samples the incoming bit stream at the positive edge of USB_CLK and converts this bit stream into NRZ format. The Incoming data stream is RX_NRZI and is converted into NRZ format. The signal NRZ carries the NRZ data.

5.3.2 Bit Destuffer

In order to ensure adequate signal transitions, bit stuffing is employed by the transmitting device when sending a packet on USB. A zero is inserted after every six consecutive ones in the data stream before the data is NRZI encoded, to force a transition in the NRZI data stream.

This block detects that seventh (stuff) bit and doesn’t allow to enter into shift register. The input to this block is NRZdel bit stream and the output is bitdestuffenable, which goes low when there are six consecutive ones.

This block also checks for the bitstuff error. After six consecutive ones if the next bit is not zero then RX_sie_error [3] become high.

5.3.3 Shift Register

This is the 8-bit shift register. The input to this block is USB_CLK and the NRZdel bit stream. If the bitdestuffenable signal is high then at the positive edge of the USB_CLK it shifts the bit stream to right.
5.3.4 Byte Counter
This block counts the number of bit received. The input to this block is \( \text{NRZdel} \) bit stream and after every 8th bit received it drives the signal \( \text{cout} \) to high, which is used to count the length of the total bytes received. If \( \text{rcvDATA} \) is high then it drives the \( \text{RX_buf_wr} \) to high, which is used to write enable for memory buffer.

5.3.5 CRC Checker
The USB specification uses the Cyclic Redundancy Checksums (CRC) to protect all non-PID fields in token and data packets from errors during transmission.

The input to this block is bit stream \( \text{RX_data_bus}[7:0] \), depending on the data packet or token packet it generate the CRC 16-bit or 5-bit field. If the generated CRC field is not equal to ‘01100b’ for token packet and ‘1000000000001101b’ for data packet then it set the second bit of \( \text{RX_sie_error} \) [2].

5.3.6 PID Decoder
This block decodes the specific packet from received PID bit. Depending on the \( \text{RX_PID}[3:0] \) it sets the corresponding signal.

For example if \( \text{RX_PID}[3:0] = '0011b' \) then it drives the \( \text{data0} \) signal high.

5.4 Implementation of Device Transmitter SIE
The implementation of Device Transmitter SIE is carried out in various blocks
The details of block is as follows

5.4.1 NRZI Encoder
The USB 1.1 devices employ NRZI data encoding for transmission of data packet. In NRZI encoding, logic 1 is represented by no change in level and logic 0 is represented by a change in level.

The input to this block is \( \text{TX_nrzi}, \text{TX_start}, \text{tx_enable} \) and \( \text{SPEED} \). If the \( \text{TX_nrzi} \) is 1 then \( \text{TX_Dplus} \) and \( \text{TX_Dminus} \) drives as its previous value. If the \( \text{TX_nrzi} \) is 0 then \( \text{TX_Dplus} \) and \( \text{TX_Dminus} \) drives as inverted of their previous value. If the \( \text{SPEED} \) is 0 then \( \text{TX_Dplus} \) and \( \text{TX_Dminus} \) are exchange.
5.4.2 Bit stuffer
In order to ensure adequate signal transitions, bit stuffing is employed by the transmitting device when sending a packet on USB. A zero is inserted after every six consecutive ones in the data stream before the data is NRZI encoded, to force a transition in the NRZI data stream. This block adds the stuff bit if six consecutive ones are found. For every one transmitted the bitstuffcnt is incremented by one if bitstuffcnt becomes 6 then the hold signal goes high for one cycle. And at the same time one zero is assign as a transmit bit.

5.4.3 Serial Shifter
It is 8-bit serial shift register. The input to this block is hold and tx_enable. As long as tx_enable is true and the hold signal is zero then at the positive edge of the TX_CLK the bit stream shifted to right by one bit position.

5.4.4 Byte Counter
The input to this block is hold & tx_enable. For every positive edge of the TX_CLK, TX_bitcnt is incremented by one. If the hold signal is 0 then TX_bitcnt will not incremented. After every 8th bit transmitted the byte_out goes to high and TX_bitcnt reset to 0.

5.4.5 CRC Check Field Generator
The USB specification uses the Cyclic Redundancy Checksums (CRC) to protect all non-PID fields in token and data packets from errors during transmission. This block will generate these crc bit and store them in 16 bits register TX_crc16bit.

The input to this block is hold signal & crc16start. When crc16start goes high and hold is zero then this block works as follows.

First TX_crc16bit [16:0] is loaded with all ones. If the incoming bit TX_shift_reg [0] is same as the MSB of TX_crc16bit then TX_crc16bit [14:0] is EX-ORed with generator polynomial (1000000000000101) and if it is not same then 0 is appended as MSB of TX_crc16bit. Before appending as crc field, all the bits of TX_crc16bit are inverted.
5.5 Implementation of Digital Phase Locked Loop

The 12Mhz clocks in the host and the device are asynchronous since they are derived from different crystals, which can have a tolerance of +/-0.25%. So bit wise synchronization is needed and achieved with the help of a PLL. A typical implementation would use a digital PLL with 4x over sampling to derive the received clock. At any bit, the derived clock period can be +/-25% of the nominal period.

The Digital Phase Locked Loop generates the 12 MHz clock if the device is full speed and 1.5 MHz clock if the device is Low speed. The generated clock is synchronized with the incoming data line from the ATX (Analog Transceiver). This module is also generating the signal for end of packet detection. If the EOP condition is detected then SE0_detected signal goes high which indicates the end of packet.

5.5.1 Finite State Machine for End Of Packet Detection

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input conditions</th>
<th>Outputs</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE0_rst</td>
<td>Reset_N = 0</td>
<td></td>
<td>SE0_rst</td>
</tr>
<tr>
<td></td>
<td>rx_dp = 0 &amp; rx_dm = 0</td>
<td></td>
<td>SE_1</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>SE0_rst</td>
</tr>
<tr>
<td>SE0_1</td>
<td>rx_dp = 0 &amp; rx_dm = 0</td>
<td></td>
<td>SE0_2</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>SE0_rst</td>
</tr>
<tr>
<td>SE0_2</td>
<td>rx_dp = 0 &amp; rx_dm = 0</td>
<td></td>
<td>SE0_3</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>SE0_rst</td>
</tr>
<tr>
<td>SE0_3</td>
<td>rx_dp = 0 &amp; rx_dm = 0</td>
<td></td>
<td>SE0_3</td>
</tr>
<tr>
<td></td>
<td>(rx_dp = 1 &amp;</td>
<td></td>
<td>SE0_dead</td>
</tr>
<tr>
<td></td>
<td>rx_dm = 1) or</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(rx_dp = 0 &amp;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rx_dm =01)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SE0_dead</td>
<td>rx_dp = 1 &amp; rx_dm = 0</td>
<td></td>
<td>SE0_detect</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>SE0_rst</td>
</tr>
<tr>
<td></td>
<td>Rx_dp=1 &amp; rx_dm =0</td>
<td></td>
<td>SE0_detect</td>
</tr>
</tbody>
</table>
5.5.2 Finite State Machine for Generation of Synchronize Clock

* Signal dpll_A is synchronized with the positive edge of the raw clock (clk_4x)
* Signal dpll_B is synchronized with the negative edge of the raw clock (clk_4x)

<table>
<thead>
<tr>
<th>Present state</th>
<th>Input conditions</th>
<th>Outputs</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPLL_C</td>
<td>Reset_N = 0</td>
<td></td>
<td>DPLL_C</td>
</tr>
<tr>
<td></td>
<td>Dpll_B = 0</td>
<td>usb_clk = 0</td>
<td>DPLL_D</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_C</td>
</tr>
<tr>
<td>DPLL_D</td>
<td>dpll_B = 1</td>
<td>usb_clk = 0</td>
<td>DPLL_5</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_D</td>
</tr>
<tr>
<td>DPLL_5</td>
<td>usb_clk = 0</td>
<td></td>
<td>DPLL_C</td>
</tr>
<tr>
<td>DPLL_7</td>
<td>dpll_A = 1</td>
<td>usb_clk = 1</td>
<td>DPLL_6</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_B</td>
</tr>
<tr>
<td>DPLL_6</td>
<td>dpll_B = 1</td>
<td>usb_clk = 1</td>
<td>DPLL_4</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_1</td>
</tr>
<tr>
<td>DPLL_4</td>
<td>dpll_B = 1</td>
<td>usb_clk = 0</td>
<td>DPLL_5</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_1</td>
</tr>
<tr>
<td>DPLL_1</td>
<td>usb_clk = 0</td>
<td></td>
<td>DPLL_3</td>
</tr>
<tr>
<td>DPLL_3</td>
<td>dpll_A = 0</td>
<td>usb_clk = 0</td>
<td>DPLL_2</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_F</td>
</tr>
<tr>
<td>DPLL_2</td>
<td>dpll_B = 0</td>
<td>usb_clk = 0</td>
<td>DPLL_0</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_5</td>
</tr>
<tr>
<td>DPLL_0</td>
<td>dpll_B = 0</td>
<td>usb_clk = 0</td>
<td>DPLL_1</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td></td>
<td>DPLL_5</td>
</tr>
<tr>
<td>DPLL_B</td>
<td>usb_clk = 0</td>
<td></td>
<td>DPLL_2</td>
</tr>
<tr>
<td>DPLL_F</td>
<td>usb_clk = 0</td>
<td></td>
<td>DPLL_6</td>
</tr>
</tbody>
</table>
CHAPTER 6  RESULTS

This chapter presents the results of the simulation.

6.1 Results

Figure 6.1 describes the input and output signals of Digital Phase Locked Loop module. \( clk_{4x} \) is raw clock. All the transitions are take place at the positive edge of the clock. DPLL module generates the 12 MHz clock if the speed is full and 1.5 MHz clock if the speed is low. The generated clock is \( usb_{\_clk} \), which is used by the Device Receiver module. The signal dpll_A is synchronized with the positive edge of the clk_4x line while the signal dpll_B is synchronized with the negative edge of the clk_4x line. When the rx_dp and rx_dm signals go low for two-bit time duration then SE0_detected signal goes high.

![Figure 6.1 Output waveform of Digital Phase Locked Loop](image)

Figure 6.1 Output waveform of Digital Phase Locked Loop
Figure 6.2 describes the input and output signals for the Device transmitter SIE module. \( TX_{\text{clk}} \) is 12 MHz clock if the speed is full and 1.5 MHz if the speed is low. When the \( tx_{\text{enable}} \) signal goes high the \( TX_{\text{state}} \) enters into the sync state. All the protocols specified by the Universal Serial Bus 1.1 specification are followed when the device transmit the data to host system. Transmitter transmits the number of byte specified by the \( tx_{\text{length}} \). The data is transmitted by the \( TX_{\text{Oplus}} \) and \( TX_{\text{Ominus}} \) signals and it is in NRZI format. When the transmission is completed the \( TX_{\text{complete}} \) signal goes high.

![Output waveform of Device Transmitter module](image.png)
Figure 6.3 describes the input and output signal of Device Receiver SIE module. This module uses the $usb\_clk$ generated by the DPLL module. All the transitions are take place at the positive edge of the clock. The input to this module is $RX\_rcv$ line. Which carries the data transmitted by the host system. This data is in NRZI format. The Device Receiver SIE converts this data into NRZ format. The stuffed bit is removed when six consecutive ones are found followed by the stuffed bit. If there is no stuffed bit then $RX\_sie\_error$ [4] bit set as high. The data is saved into receiver memory. When the $SE0\_detected$ signal goes high, it indicates the end of packet. When all the data bytes are received then $RX\_COMPLETE$ signal goes high.

![Figure 6.3 Output waveform of Device Receiver](image-url)
6.2 Synthesis Report

<table>
<thead>
<tr>
<th>Resource</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLBs</td>
<td>190</td>
</tr>
<tr>
<td>CLB flip flops</td>
<td>203</td>
</tr>
<tr>
<td>4 input LUTs</td>
<td>337</td>
</tr>
<tr>
<td>3 input LUTs</td>
<td>53</td>
</tr>
<tr>
<td>Bonded IOBs</td>
<td>42</td>
</tr>
<tr>
<td>IOB flops</td>
<td>12</td>
</tr>
<tr>
<td>IOB latches</td>
<td>0</td>
</tr>
<tr>
<td>Clock IOB pads</td>
<td>3</td>
</tr>
<tr>
<td>TBUFs</td>
<td>24</td>
</tr>
<tr>
<td>Total Equivalent Gate Count</td>
<td>4103</td>
</tr>
</tbody>
</table>

Other Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Device</td>
<td>xc4vlx25</td>
</tr>
<tr>
<td>Optimization Goal</td>
<td>Speed</td>
</tr>
<tr>
<td>Target Family</td>
<td>Virtex -4</td>
</tr>
</tbody>
</table>
CHAPTER 7  CONCLUSION

7.1 Conclusion
An attempt is being made to design the Protocol Engine for USB 1.1 Device Model, which consists of three modules i.e. Device Receiver SIE, Device Transmitter SIE and Digital Phase Locked Loop.

This protocol engine uses NRZI (Non Return to Zero, Inverted) encoding scheme, which maintains the synchronization with the incoming data and eliminates the need of separate clock to be sent over the bus. This protocol engine also checks for any error occurs during a data transmission. The error checking mechanisms uses CRC (Cyclic Redundancy Checksum) error code, and if any error occurs that will be notified to the device controller.

All the protocols that are specified in Universal Serial Bus Specification 1.1 (such as encoding scheme is NRZI with Bit stuffing, generation of CRC bits used for error checking, packet identification, end of packet detection) and used for transmission and reception of data over the USB bus are implemented in this protocol engine.

This protocol engine is implemented in verilog HDL, and the simulator used for simulation is ModelSim.

7.2 Future Scope
The protocol engine presented here supports only for USB 1.1 low speed devices, which operate at 1.5 Mb/s and full speed devices, which operates at 12 Mb/s. But the application such as data transfer between hard disk and CDROM require high data transfer rate about 25-400 Mb/s.

With the development of high speeds/high data transfer rate HDD, CDROMs and DVDs, it can be envisaged that a protocol engine with faster data rate could be developed, particularly with the newer Universal Serial Bus (USB 2.0).
References

   Http://www.usb.org/developers/docs/

    Mindshare, Inc 1997.

    Pearson Education Asia, 2001


    Http://www.usb.org/developers/docs/whitepapers

[6] “Cyclic Redundancy Checks in USB”
    Http://www.usb.org/developers/docs/whitepapers
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår. Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för  ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances. The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Författarens för- och efternamn]