**Deanship of Graduate Studies**
**Al – Quds University**

# Peripheral USB Interface Board

**Khaled Hasan Saleh Murad**

**M.Sc. Thesis**

**Jerusalem – Palestine**

**1427/2006**

# Peripheral USB Interface Board

**Prepared By :**
**Khaled Hasan Saleh Murad**

**B.Sc. of Electrical Engineering, Birzeit University, Palestine**

**Supervisor : Dr. Ahmad Al-Qutob**

**A thesis Submitted in Partial fulfillment of requirements for the degree of Master of Electronics and computer Engineering, Department of Electronics and Computer Engineering Program.**
**Faculty of Engineering / Alquds University**

**1427/2006**

**Al Quds University**
**Deanship of Graduate Studies**
**M.Sc. in Electronics and Computer Engineering**

## Thesis Approval

# Peripheral USB Interface Board

**Prepared By : Khaled Hasan Saleh Murad**
**Registration No. : 20310007**

**Supervisor : Dr. Ahmad Al-Qutob**

**Master thesis submitted and accepted, Date : 17 September 2006 .**
**The names and signature of the examining committee members are as follows :**

| | |
|---|---|
| **1- Head of committee : Dr. Ahmad Al-Qutob** | **Signature…….** |
| **2- Internal Examiner : Dr. Labib Arafeh** | **Signature…….** |
| **3- External Examiner : Dr. Adnan Yahya** | **Signature…….** |

**Jerusalem – Palestine**

**1427 / 2006**

# DECLARATION

I Certify that this thesis submitted for the degree of master is the result of my own research, except where otherwise acknowledged, and that this thesis (or any part of the same) has not been submitted for a higher degree to any other university or institution.

Signed ……

Khaled Hasan Saleh Murad

Date : 20 June 2006 .

# ACKNOWLEDGMENTS

Writing acknowledgment for all who have contributed to this work is practically impossible, but I would like to explicitly thank a few people who helped me along the way.

First, I thank my family especially my wife , my mother who have always supported and encouraged me in my educational endeavors, I also thank my advisor Dr. Ahmad Al-Qutob for his support, quick feedback and valuable advice, to Dr. Aysheh Al-Rifa'ee ,thank you for your help and assistance.

To all I have not mentioned, thank you. I thank my friends, I am thankful for Al Quds University, my many teachers and colleagues.

# ملخص


لقد تم تصميم ال USB (Universal Serial Bus) كبديل لمعظم مخارج الحاسوب القديمة (ports) والتي تشمل مخرج التوازي (parallel port) ومخرج التوالي (serial port) وغيرها، وبمقارنة ال USB بهذه المخارج نجد أن ال USB أكثر مرونة وأقل كلفة وأسهل في الاستخدام.

في هذه الرسالة تم بناء بطاقة USB لادخال و اخراج المعلومات (USB Input/Output Board)، كما تم تطوير كل من برنامج المتحكم الدقيق (microcontroller Firmware) وبرنامج الحاسب (PC Software)، ومن المتوقع ان تزودنا البطاقة وبرامجها بأساس سهل وقليل الكلفة لأداة عامة يمكن استخدامها للتحكم بأجهزة أخرى من خلال جهاز الحاسوب، علاوة على ذلك فان البرامج التي تم تطويرها يمكن استخدامها –مع أو بدون– بعض التعديلات من قبل اخرين لتطوير تطبيقات أخرى خاصه بهم.

بالأضافة لذلك تم تضمين دليل مستخدم (user manual) للبطاقة يبين كيفية بناء البطاقة وتعريفها على جهاز الحاسوب واستخدامه، ومن اجل توفير الوقت والجهد والمال تم استخدام (Human Interface Device -HID- Driver)، وهذا ال HID متضمن في برنامج Windows وقد استخدم ك (Device Driver) للبطاقة، ان استخدام ال (HID) يعفي المستخدم من تحميل (Drivers) خاصة قبل تشغيل الجهاز، وهذا ما يمنح الجهاز خاصية(plug-and-play).

لقد تم استخدام الدارة المتكامله TUSB3210 كمتحكم دقيق (microcontroller) والتي تشكل قلب البطاقة، كما تم استخدام برنامج (keil c51 compiler) لتطوير برنامج المتحكم الدقيق (microcontroller Firmware) ، بينما استخدم برنامج (Delphi 7) لتطوير برنامج الحاسب (PC Software) .

# ABSTRACT

USB (Universal Serial Bus) is designed to serve as a replacement for most of the old ports: Parallel ,Serial ,and so on , compared to these ports USB is more flexible, hot pluggable with low cost and ease of use.

In this thesis a USB Input/Output Board has been built, the board's Firmware and Host Application codes were also developed , the board and codes are expected to provide a platform for a simple, low cost and versatile instrument that can be used to control peripherals via Personal Computer, moreover the developed codes –with or without- small changes will hopefully be suitable to be used by other developers to develop their own applications.

In addition a detailed User Manual was produced, the manual describes how to build, install and use the board, in order to save time and money the Human Interface Device (HID) Driver included with Windows was used as the PC Device Driver, this eliminates the need for user to load specific drivers before running the device, thus furthering "plug-and play"-ability.

The TUSB3210 from Texas Instruments was chosen to be the microcontroller chip at the heart of the board, the Keil "C51" compiler was used to develop the Firmware , while the "Delphi 7" software package was used to develop the Host Application software.

# TABLE OF CONTENTS

# LIST OF TABLES

***Number*** ***Page***

# LIST OF FIGURES

***Number*** ***Page***

*Chapter 1*

# INTRODUCTION

This section is aimed to introduce the thesis, section 1.1 will give a short overview of the USB (Universal Serial Bus) development, section 1.2 will explain the goals of the thesis, section 1.3 will give a short description of the limitations of the project, and finally section 1.4 will list the outline of the thesis work.

## 1.1 USB Overview

The Universal Serial Bus (USB) was born from the need to connect an incredibly diverse range of peripheral to computers, it's based on an inexpensive high volume chipsets, with data transfer rates ranging from 1.5 Mbps to 480 Mbps, and a 25 meter distance capacity, it is designed to be a "plug and play " device with a 5 volt voltage and a current capacity of up to 500 mA, it is also compatible with many consumer products nowadays.

What makes USB communications preferable to other interfaces is the availability to design a device with any of the three speeds (1.5 ,12 ,480 Mbps) and any of the transfer types.

Once the device is attached to the USB port of the PC, it must respond to a series of requests, this is needed by the PC to learn about the device and communicate with it. In order to communicate with a device the PC should have a device driver for that device, this is needed to manage communications between the PC and the application.

To develop a USB device you have first of all to choose the right chip that suits your application , then you have to choose the device class in order to decide what device driver should be used , do you need to write the driver or use a one supplied by the chip's vendor or should you use the device drivers included with Windows ,such as Human Interface Device (HID) , then the Firmware code and the software that enables communication between the device and the PC should be developed , this requires knowing something about how USB works and how the operating system of the PC implements the interface , reading the USB specification would be of great help , also you need to purchase the tools needed to develop the codes and debug them.

Once you have the Firmware code , the Host Application software and the device (Interface Board) and you choose the device class (this decides what driver to use) , you can develop you application.

## 1.2 Thesis Goals and Design Trajectory

In this thesis the device (Interface board) has been built , the Firmware and host application codes were developed for a HID class device, choosing this class eliminates the need to write a Device Driver which requires an investment in tools , expertise in C++ and is time consuming, the developed Firmware and Host Application codes , the built board and the User Manual could be used by other developers to develop their own applications -such as Barcode reader, Flash Memory Reader and General Purpose Controller- with or without small modifications , this means that the introduced hardware and software are generic.

To accomplish this, the following design trajectory was followed :

1. The requirements of the needed controller chip were specified according to the size of data to be transferred, the transfer rate, the power supply requirements, the size of controller RAM, etc…

2. According to the above requirements a decision should be made on how the PC will communicate with the peripheral, we have three alternatives : using the Windows built in drivers, or a generic driver from another source or writing a custom driver.

3. The TUSB3210 controller chip from Texas Instruments was selected according to the needed requirements.

4. Building the Interface Board according to a generic purpose schematic.

5. Deciding upon the developing tools (software) needed to write the Firmware code and the Host Application software.

6. Developing the Firmware and the Host Application software simultaneously.

7. Verification and testing of the Interface Board, the Firmware and PC software.

8. Writing a user manual explaining how to build, install and operate the board and the accompanied codes.

## 1.3 Limitations

In the current design Windows' HID (Human Interface Device) driver was used to manage communication with the device, classic examples of HID devices are Keyboards, Mice and Joysticks, other examples include Remote Controls, Medical Instruments, Audio/Video devices and Vender Defined Functions, it also supports device primitives such as LED and a Button, and standard measurements such as time, temperature and distance.

The simplest and cheapest method for communication at moderate rates between a PC and the device is to use the HID Driver when it's feasible, because there is no need to write or install a driver and any Windows computer can access the device, in addition writing a custom Device Driver requires a big investment in tools and time, expertise in Visual C++ programming and a good knowledge about how windows communicates with the device and the host application, despite this advantage the HID class has the following main limitations (Axelson, 2005) :

1. The amount of data each transaction can carry ranges from 8 bytes per transaction for low-speed and 64 bytes per transaction for full speed devices, to 1024 bytes per transaction for high-speed devices, long reports can use multiple transactions.

2. The maximum transfer rate is limited to 800 Bytes per second (Bps) for low speed, 64,000 Bps for full speed ,and 24,576 KBps for high speed devices.

As explained above a trade-off exists between the transfer rate and writing a driver, writing a driver needs time, tools and big efforts, unfortunately the needed tools were not available for me, and time available was limited, this forces me to use the HID driver in spite of its limitations.

## 1.4 Outline of Thesis Work

In summary, USB devices have become very popular, many products such as digital cameras, and removal flash storage use USB. Nowadays PC's use USB to connect keyboards

and mice, USB is so popular because of its low cost, flexibility and its ease of use (plug and play), it is expected that many applications in home appliances, automotive and medical industries will also use USB interface.

What have been achieved in this thesis includes : building a USB interface board, developing both the Firmware and Host Application Software codes for a HID, the board and the software codes are expected to provide a platform for a simple, low cost, generic purpose instrument used to control peripherals via a host personal computer, developers are expected to use the board, Firmware and Host Application codes to design their own applications , this may or may not require small modifications to the codes, the amount of modifications depends on the application requirements .

Chapter 1 gives a short overview of USB development, explains the goals of this thesis, the design trajectory and a summary for the main limitations of the project.

Chapter 2 introduces and discuses the main USB concepts needed by the developer to understand the USB Communication Protocol, Human Interface Devices (HID's) are specifically introduced and discussed.

Chapter 3 introduces how to choose the suitable chip for a specific application; it also describes in details the Firmware and Host Application Software codes for the TUSB3210 full speed microcontroller.

Chapter 4 introduces the User Manual which describes how to build the hardware and develop the software codes for the interface board; it also explains the board's installation process.

Chapter 5 concludes this thesis and contains testing the board and codes, discussion, the main contributions and the related future work .

*Chapter 2*

# USB CONCEPTS OVERVIEW

## 2.1 USB Benefits

USB was first introduced in 1996, it was born out of the frustration of PC users who needed to communicate with an increasing number of peripherals without the limitations and frustrations of existing interfaces like Centronics Parallel Interface and the RS-232 Serial Port Interface.

In particular USB provides many benefits to both users and developers, it offers users simple connectivity, fast and reliable data transfers, flexibility, low cost and power conservation. Table (2.1) compares USB with other popular interfaces (Axelson, 2005).

For developers it offers flexibility built into the USB protocol, the technical support offered by the so many vendors especially in the controller chip and operating systems.

Despite the many advantages offered by the USB interface it is not perfect since its speed is limited to 480 Mbps for high speed devices and the maximum USB link can be as much as 30 meters when using links between five hubs and a device, the complexity of the USB protocol adds to the difficulties faced when using a USB interface.

## 2.2 Is USB the Right Choice for a Certain Project?

Before taking this decision, the developer should make sure that the PC that will use the device must have the minimum hardware and software requirements, in particular the PC should have a USB host controller and a root hub with at least one USB port, also its software should have an operating system that supports USB.

Table 2.1 : Comparison between USB and popular computer interfaces (Axelson, 2005).

| Interface | Format | Number of Devices (maximum) | Length (maximum, feet) | Speed (maximum, bits/sec.) | Typical Use |
|---|---|---|---|---|---|
| USB | asynchronous serial | 127 | 16 (or up to 96 ft. with 5 hubs) | 1.5M, 12M, 480M | Mouse, keyboard, disk drive, modem, audio |
| RS-232 (EIA/TIA-232) | asynchronous serial | 2 | 50-100 | 20k (115k with some hardware) | Modem, mouse, instrumentation |
| RS-485 (TIA/EIA-485) | asynchronous serial | 32 unit loads (up to 256 devices with some hardware) | 4000 | 10M | Data acquisition and control systems |
| IrDA | asynchronous serial infrared | 2 | 6 | 115k | Printers, hand-held computers |
| Microwire | synchronous serial | 8 | 10 | 2M | Microcontroller communications |
| SPI | synchronous serial | 8 | 10 | 2.1M | Microcontroller communications |
| I2C | synchronous serial | 40 | 18 | 3.4M | Microcontroller communications |
| IEEE-1394 (FireWire) | serial | 64 | 15 | 400M (increasing to 3.2G with IEEE-1394b | Video, mass storage |
| IEEE-488 (GPIB) | parallel | 15 | 60 | 8M | Instrumentation |
| Ethernet | serial | 1024 | 1600 | 10M/100M/ 1G | Networked PC |
| MIDI | serial current loop | 2 (more with flow-through mode) | 50 | 31.5k | Music, show control |
| Parallel Printer Port | parallel | 2 (8 with daisy-chain support) | 10–30 | 8M | Printers, scanners, disk drives |

### 2.2.1 The Host Controller

Since 1997 PC's have a hardware that supports USB, if it is not built in, an expansion card can be used , the host controller formats received and transmitted data and makes sure that the operating system components can understand the received data, it also performs other functions related to managing communications on the bus, the root hub and the host are responsible for the detection , the arrival and removal of a USB device, they are also responsible for the transmission of data between the host controller and the connected device. The device might be an additional hub or a peripheral that is connected to the USB bus.

### 2.2.2 The Operating System

Windows 95 had some USB support, this support was enhanced in Windows 98, 2000 and XP, Apple's iMac, Unix / Linux also support USB, Windows NT 4 does not support USB.

### 2.2.3 The Components

The components of the USB include the electronic circuits, connectors and cables between the host and the connected device or devices, devices are required to contain circuits and code that knows how to communicate with the host.

### 2.2.4 Bus Topology

The only topology of the USB bus is the tiered star (Axelson, 2005), as shown in Figure (2.1) , the hub is the centre of each star while each point on a star is a device that is connected to the hub's port. Following are some facts concerning the USB bus:-

1. A typical hub can have two, four or seven ports.
2. The hubs arrange the communication between each other automatically, neither the host nor the device knows or cares about this process.
3. Only one device can communicate with the host at a time.
4. A maximum of 127 peripherals and hubs (including the root hub) can be connected to the bus.

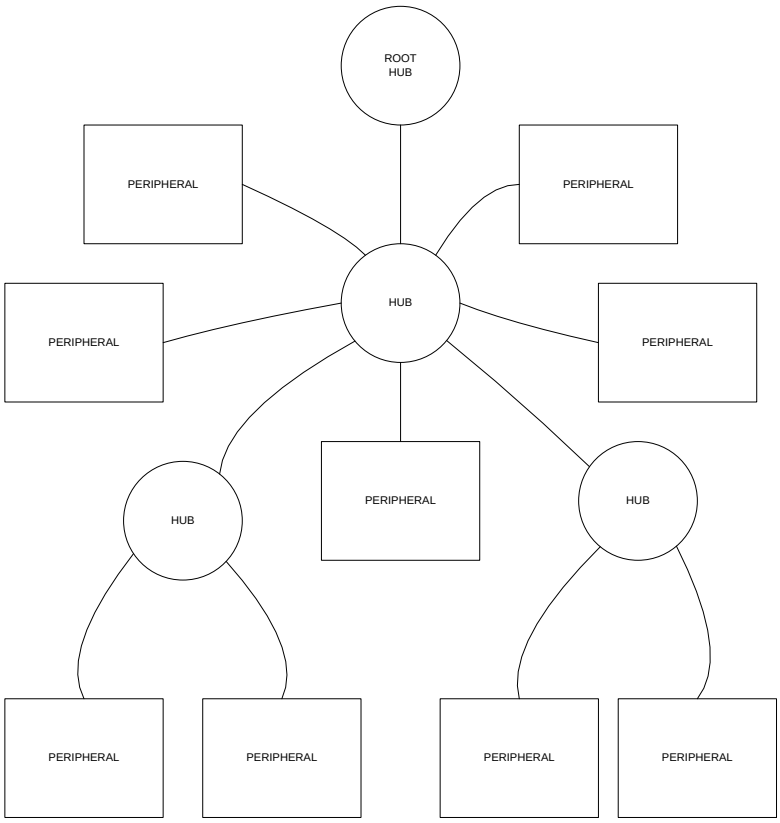



Figure 2.1 : USB uses a tiered star topology (Axelson, 2005).

## 2.3 Terms Definition:

When talking about the USB world the following words have specific meanings (Axelson, 2005) :-

- Host : It is defined as the computer that controls the interface.
- Function : A device that provides a capability to the host, examples are a mouse, a set of speakers or a data acquisition unit.
- Hub : A 1.x hub repeats received USB traffic in both directions, and also contains the intelligence to manage power, send and respond to status and control messages, and prevent full- speed data from transmitting to a low speed device .

- Device (peripheral) : It is something you attach to a USB Port on a PC or hub, or it is a function or a hub - except for the special case of the compound device, which contains a hub and one or more functions .

- Port : In general a computer Port is an addressable location that is available for attaching additional circuits (Axelson, 2005). USB ports differ from many other ports because all ports on the bus share a single path to the host, with RS 232 Serial Interface, each port is independent from the others.

- Firmware : A software code in the controller chip that enables it to communicate with the host and other circuits in the peripheral.

- Device driver: A software in the host to enable applications (programs that users run) to communicate with the peripheral.

## 2.4 Host and Device (peripheral) Duties

The host maintains the state of the bus and all the devices attached to it, only one host can be in a USB network, the host provides the following capabilities (Compaq, Intel, Microsoft, NEC, 1998) :-

a) Detecting the attachment / detachment of devices.

b) Managing data-flow control between the host and devices.

c) Collecting device status and activity statistics.

d) Error-checking.

e) Providing power to devices that request the power.

The device must respond wherever the host initiates communication. In particular the device has the following capabilities (Compaq, Intel, Microsoft, NEC, 1998) :-

a) Detecting and monitoring communications directed to the microcontroller chip.

b) Responding to all requests made by the host during enumeration and during data exchange with the host.

c) Error checking and power management.

d) Exchanging data with the host.

## 2.5 USB Standards

USB standard editions that have been released (Peacock, 2002) :

- USB 1.0, this is the first edition released in January 1996, it supports low speed (1.5 Megabits/second), the actual data transfer rate is less than 1.5 Mbps because a percentage of this rate is reserved for USB protocol overhead.

- USB 1.1, released in September 1998, it supports full speed (12 Mb/S).

- USB 2.0, released in 2000, it supports high speed (480 Mb/S), and this version is compatible with USB 1.x .

## 2.6 Developing USB projects

Developing a USB project involves the design of the hardware and a code in the peripheral to manage communication with the host (Firmware) and enable the peripheral to run , also it includes designing the PC software needed to communicate with the peripheral , in particular the following elements are needed (Axelson, 2005):-

- A microcontroller chip with USB interface.

- Firmware on the chip to carry out communications with the host.

- Any additional hardware and code that is needed to process data read inputs and write to outputs.

- A PC that supports USB.

- Device driver software on the host.

- Application software on the host to enable users to access the device.

### 2.6.1 Development Tools

The following tools are needed to develop the USB device:-

- A compiler or assembler to create the Firmware code, the developer can use C or other high level language to create the code, to create an assembly code you will need a cross assembler that runs on a PC, the compiler or assembler should translate the written source code to the machine code (binary file).

- A development Kit to store the generated code in the controller's code memory.

- A programming language to write the host software, which may include a device driver and /or an application software, examples are C++ and Delphi.

### 2.6.2 Development Process Steps

This process includes initial decisions, enumeration and data exchange between the host and the device (Axelson, 2005) :

**a- Initial Decisions**

Before starting to develop any project the developer should gather data to enable him making some decisions:-

1. Specify the device specifications, data transfer type, transfer rate, power requirements, device driver needed …etc.

2. Choose the suitable chip that will enable you to meet the device specifications.

3. Decide whether to use a device driver supplied by the vendor, design a custom driver or use Windows - built - in drivers.

**b- Enumeration:**

The following steps should be accomplished to enable Windows to enumerate the device :

1. The Firmware should include the code the chip needs to be enumerated by the host, this can be done by the chip which should be able to send a series of descriptors to the host, these descriptors describe the chip's USB capabilities, when the host sends the enumeration request, the chip should respond to this request, this is done by hardware or by a program code stored in the chip.

2. Modify the (.INF) (information) file created by Windows to identify the device when enumerated or create the file using any text editor.

3. Build the necessary circuits needed to be connected to the host.

4. Download the Firmware into the device and plug it to the host's USB bus, once connected Windows should enumerate it.

**c- Data Exchange**

To enable the device to perform its intended functions, the following steps should be done:-

1. Design your Firmware such that it includes all the code needed to add the required abilities to the device.

2. Write the needed driver or use the drivers included with Windows.

3. Write the needed application software to enable the user to communicate with the device.

## 2.7 How USB Transfers Data:-

Understanding how the transfer of data works leads to deciding which transfer type is suitable for your project, it also helps in writing the Firmware for the chip and in debugging the circuits and codes.

The USB interface is difficult; trying to simplify it we will start from a big picture and work down to the details. If you want to know details about USB interface you should review the specification manual titled "Universal Serial Bus Specifications" (Compaq, Intel, Microsoft, NEC, 1998).

### 2.7.1 Communication Basics

USB communications are divided into two categories:-

**1. Configuration communications**

In this type the host learns about the device and gives it an address in order to exchange data with it, this is done when the device is connected to the host's USB bus when the host starts enumerating the device.

When the device is attached to the host, the host sends a series of standard requests to the device, the Firmware of the device should respond to these requests by returning the requested information.

**2. Application Communications**

In this type, the host exchanges data with the Device, and the user activates the functions of the device from the PC side.

Once the device is enumerated, the host can exchange data with it, this is done using standard API (Application Programmer’s Interface) functions at the host side, these functions enable the user to read from and write to the device, at the device side, exchanging data requires placing the data to be send in the transmit buffer and that to be read in the receive buffer, at the end of the transfer process, the device should acknowledge the host that it is ready for the next transfer.

### 2.7.2 USB Transfer Elements

A USB transfer is made up of transactions, each transaction is made up of packets, the packet contains the information to be exchanged, to understand USB transfers we need to know about endpoints and pipes (Compaq, Intel, Microsoft, NEC, 1998).

- **Peripheral Endpoint**

The Endpoint is a storage location (buffer) that stores multiple bytes, the endpoint stores data to be transmitted or received, the host has buffers but it does not have endpoints, the host is the starting point for communication with the device endpoints. According to specification an endpoint should have a unique address, this address consists of an endpoint number and direction, the number should be between 0 and 15 while the direction is from the host's perspective: IN is from the device towards the host and OUT is from the host towards the device.

Since a control endpoint is supposed to transfer data in both directions it consists of a pair of IN and OUT endpoints that share the same endpoint number, usually Endpoint 0 is configured as a control endpoint, the other transfer types send data in one direction only, each endpoint number supports both IN and OUT endpoint addresses.

- **Pipes**

A pipe is a logical communication channel between a device's endpoint and the host controllers' software, a USB pipe is not a physical object .

Establishing a pipe between the host and the device must be done before any transfer can occur, this process occurs directly after power -up or the attachment of the device when the host requests configuration information from the device, upon the removal of the device the host removes the pipes.

Every device has a default control pipe that uses endpoint 0, which supports generic USB status and configuration protocol.

### 2.7.3 USB Transfer Types[1]

USB supports four data transfer types: control, interrupt, bulk and isochronous.

- Control mode : this mode is initiated by the host, in this mode data travels in both directions, but only in one direction at a time, this mode is used for initialization of the device since it enables the host to read information about the device, set the device address and other settings. Control transfers may be used to transfer small amounts of data; all USB devices must support control transfers.

- Interrupt mode : in this mode the host has to initiate the transfer of data, and queer devices to see if they need to be serviced, peripherals exchanging small amounts of data that need immediate attention (such as mice and keyboards) use this type of transfer.

- Bulk mode : this mode is used when data accuracy is of prime importance, and the rate of data transfer is not guaranteed, typical applications include printers and scanners.

- Isochronous mode : this mode sacrifices data accuracy in favor of guaranteed timing of data delivery, typical applications are audio and video devices, in this type data received with errors is not automatically re-transmitted as in the other modes, so occasional errors must be expected.

### 2.7.4 Transfer Initiation

The USB specification defines the transfer as the process of making and carrying out a communication request.

"Windows application opens communication with a device using a handle retrieved using standard API functions" (Axelson, 2005), Applications can request the device to send data like reading the contents of report, they also can provide data to be send to the device like sending the contents of a text file. The operating system passes the request of the application

[1] For more information on Transfer Types ,see Appendix A.

to the appropriate device driver which passes this request to other system-level drivers until the request is received by the host controller, which interne initiates the transfer on the bus.

### 2.7.5 Transactions

Any USB transfer consists of transactions, and each USB transaction consist of

- Token packet.
- Optional data packet.
- Handshake packet: used to acknowledge transactions and to provide a means of error correction.

Figure (2.2) shows the elements of a typical transfer.

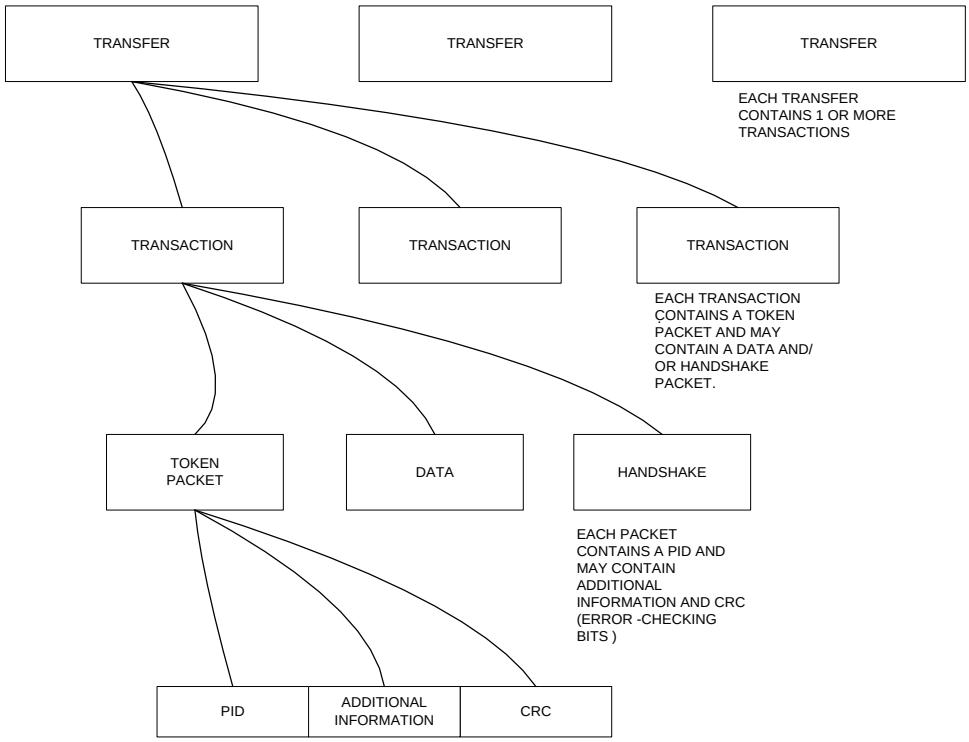



Figure 2.2 : USB transfer elements, each transfer consists of a transaction, the transaction contains packets, each packet contains a packet identifier (PID), CRC and sometimes additional information (Axelson, 2005).

USB is a host centric bus, which means that the host initiates all transactions, the first packet (token packet) is generated by the host, it describes what is to follow and if the data transfer is a read or write, it also contains the address of the device and the target endpoint.

The next packet is usually a data packet which is followed by a handshaking packet to report the status of the transaction.

### 2.7.6 USB Transaction Parts.

A transaction consists of three parts (phases) that occur in sequence: token, data and handshake, a phase consists of one or two packets, the packet is a block of information which conform to a special format, all packets begin with a packet ID (PID) that contains information that identify the transaction (Compaq, Intel, Microsoft, NEC, 1998) .

## 2.8 USB Functions

USB Functions are USB devices which provide a capability or function such as a Printer, Zip Drive, Scanner, Modem or other peripheral, figure (2.3) below shows how the host is connected to a USB devices (Peacock, 2002).

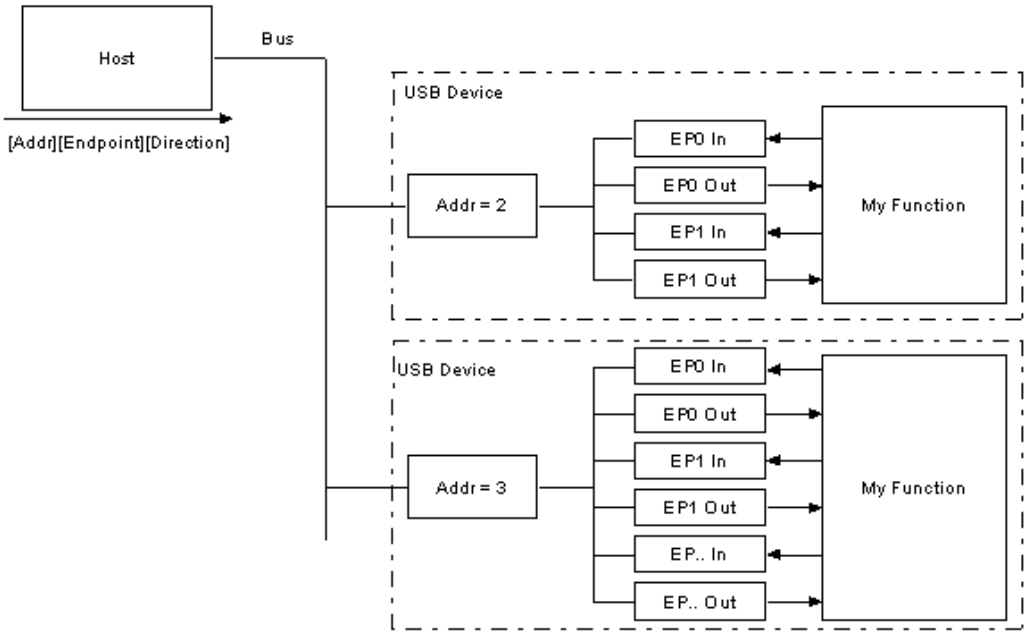



Figure 2.3 : Connection of host PC to a USB device (Peacock, 2002) .

## 2.9 Bandwidth management

The host is responsible for managing the bandwidth, during enumeration and when configuring Isochronous and Interrupt Endpoints , the host is always watching the operation of the bus in order to keep the bandwidth within the allowed limits, the specification puts limits on the bus such that no more than 90% of any frame to be allocated in Interrupt and Isochronous transfers on a full speed bus, for high speed bus, this limitation is reduced to 80% of a microframe, the remaining (10% or 20%) is left for control and bulk transfers (Compaq, Intel, Microsoft, NEC, 1998).

## 2.10 USB Descriptors[2]

Descriptors are data structures or formatted blocks of information, that enable the host to learn about a device (Compaq, Intel, Microsoft, NEC, 1998).

All USB devices must store the information needed in descriptors and when requested by the host, must respond (send the descriptors) in the expected format, these descriptors describe to the host what the device is, what is the USB version supported, the number of endpoints and their type…etc, the most common USB descriptor types are:-

1. Device Descriptor.
2. Configuration Descriptors.
3. Interface Descriptors.
4. Endpoint Descriptors.
5. String Descriptors.

Each device has only one device descriptor which contains information about the device such as the USB revision of the device, product and vender ID's and the number of configurations the device is allowed to have.

[2] For more information see Appendix B.1

Each device also has one or more configuration descriptors, this type of descriptors contains information about the power consumption of the device, whether the device is bus or self powered and the number of interfaces supported by the configuration, during enumeration the host reads the device descriptor and decides which configuration should be enabled, one configuration can be enabled at a time, it should be mentioned that few devices have more than one configuration.

Each interface descriptor has zero or more endpoint descriptors which contains the information needed to communicate with the endpoint, if the interface has zero endpoints, it can use the control endpoint (endpoint 0) for communications, a device can have one or more interface descriptors enabled at a time.

Each endpoint descriptor contains the transfer type, transfer direction and endpoint maximum packet size; all endpoints except endpoint "0" are supposed to have endpoint descriptor.

A string descriptor is used to store text such as the devices name, and version, the developer's name, etc.

Each descriptor consists of a series of fields, some fields use prefixes to indicate the format or the contents of the data in that field, examples are:-

b= byte, w=word, bm = bit map, bcd= binary –coded decimal.

i= index, id= identifier.

Table (2.2) shows values defined by USB and HID specification.

Table 2.2: Descriptor and the corresponding value and type of that descriptor (Axelson, 2005).

| type | Value (hexadecimal) | descriptor |
|---|---|---|
| Standard | 01 | device |
| | 02 | configuration |
| | 03 | string |
| | 04 | interface |
| | 05 | endpoint |
| | 06 | device-qualifier |
| | 07 | other-speed-configuration |
| | 08 | interface-power |
| Class | 21 | HID |
| | 29 | hub |
| Specific to the HID class | 22 | report |
| | 23 | physical |

## 2.11 USB Standard Requests[3]

There are three Types :

**1. Device Requests :** All devices must respond to standard requests.

**2. Interface Requests :** According to USB specification, there are five standard interface requests : GET_FEATURE, CLEAR_FEATURE, SET_FEATURE, GET_INTERFACE and SET_INTERFACE.

**3. Endpoint requests :** According to USB specification, there are four standard endpoint requests : GET_FEATURE, CLEAR_FEATURE, SET_FEATURE, and SYNCH_FRAME.

## 2.12 Enumeration

It is a process by which the host determines what device has just been connected to the bus and what are the needs of the device such as power consumption, number and type of

[3] For more details see Appendix B.2

endpoints. When a new device is connected to an active host, the host sends a series of requests to the device hub in order to establish communication with the device, the host then starts the enumeration process by sending control transfers containing standard USB requests to endpoint 0 , if the device responds by sending the requested information to the host the enumeration is successful, otherwise it is failed, when enumeration is complete, Windows add the new detected device to the Device Manager display in the control panel (Axelson, 2005).

## 2.13 Human Interface Devices (HIDs)[4]

A human interface device (HID) is a device that communicates with the host computer using structured reports, this type of device interacts directly with people, in this type applications communicate with HIDs using the drivers built into the operating system, typical examples of HID class devices include keyboards, mice, joysticks and control found on some devices like VCR remote controls, games, other devices that may not interact with humans include bar-code readers, thermometers, or voltmeters (Universal Serial Bus, 2001).

### 2.13.1 Abilities and Limitations of HID Class Devices.

A HID may not have interface with humans, instead it has to be able to function within the limits and specifications of the HID class. Following are the abilities and limitations of a HID (Axelson, 2005).

- HIDs use structured reports to exchange data with the host, the device's Firmware must conform to the HID report format, the host exchanges data with the device by sending and receiving these reports, it can use Control or Interrupt transfers only.

- For full speed devices the maximum report size is 64 bytes per transaction, while its 8 bytes for low speed and 1024 bytes for high-speed, a report may use multiple transactions.

- The device may send data to the host at any time, this requires the host driver to poll the device periodically to obtain new data.

[4] For more details on HID Descriptors and HID Specific Requests see Appendix c

- Regarding the transfer speed, the speed is limited for low and full-speed devices, for low-speed HIDs such as Mice and Keyboards, the maximum transfer rate is 800 bytes per second, while it reaches 64,000 bytes per second for full-speed devices such as Barcode Readers and UPS Controller, for high – speed HIDs such as Compact Flash Card Reader and Oscilloscopes, it is 24.576 Megabytes per second.

- There is no guaranteed rate of transfer, if a device is configured for 10 ms interval, the time between transactions may be equal to or less than this period.

- Under Windows 98, host to device data transfers may use control transfers only.

### 2.13.2 HIDs Requirements

The USB specification defines the requirements that should be met by a device to be classified as a HID. Following is a brief description of some of these requirements (Universal Serial Bus, 2001) :-

**1- Endpoints**

For a HID device to send data to the host, it must have an interrupt IN endpoint, while an interrupt OUT endpoint is optional. According to specification all HIDs should use the default control pipe or interrupt pipe to exchange data with the host, table (2,3) below shows details.

Table 2.3 : Transfer types and their uses in HIDs (Axelson, 2005).

| Transfer Type | Source of Data | Type of Data | Required Pipe? | Windows Support |
|---|---|---|---|---|
| Control | Device (IN transfer) | Data that doesn't have critical timing requirements. | Yes | Windows 98 and later |
| | Host (OUT transfer) | Data that doesn't have critical timing requirements, or any data if there is no OUT interrupt pipe. | Yes | |
| Interrupt | Device (IN transfer) | Periodic or low-latency data. | Yes | |
| | Host(OUT transfer) | Periodic or low-latency data. | no | Windows 98 SE and later |

### 2- Control Pipe

The control pipe for a HID is used for (Universal Serial Bus, 2001) :

- Receiving and responding to requests for USB control and class data.
- Transmitting data when polled by the HID class driver (using the Get-Report request).
- Receiving data from the host.

All USB devices must support Endpoint zero, this means that only an Interrupt IN pipe is included in the interface descriptor using an Endpoint Descriptor.

### 3- Interrupt Transfers

If the host is expected to receive data from the device quickly or periodically, the interrupt pipe should be used to exchange data instead of the control pipe, an Interrupt IN pipe sends data to the host and an interrupt OUT pipe sends data to the device.

Interrupt OUT pipes are not required if the host uses the control pipe to send reports using Set-Report requests.

The interrupt pipe is used for (Universal Serial Bus, 2001) :

- Receiving asynchronous ( unrequested) data from the device.
- Transmitting low latency data to the device.

### 4- Firmware

For a HID device to communicate with the host, the device Firmware must meet some requirements like (Universal Serial Bus, 2001) :

- The device must be identified as a HID in the device descriptor or in the interface descriptor.

- In addition, to the default control pipe, the Firmware of the device should include an Interrupt IN Endpoint.

- A report descriptor that represents the format of the transmitted and received device data should be included in the Firmware.

- The Firmware should support Get-Report Control transfers or Interrupt IN transfers in order to send data to the host.

- The Firmware should support Set-Report Control transfers in order to receive data, it may also support Interrupt OUT transfers.

### 2.13.3 Reports

A data report in a data packet may be preceded by a prefix called the report ID, if a device supports multiple reports of the same type, each report may contain different data and have its unique ID, in many cases it is better to have a single report for simplicity.

The report ID is an item in the report descriptor, if the report descriptor contains no report ID, the default value of zero is assumed for the ID, but in any way a report ID of zero should not be declared in a descriptor.

In transfers that uses a Set-Report or Get-Report request, the report ID should be specified by the host in the Setup transaction in the low byte of the value field, in an Interrupt transfer the Report ID should be the first byte sent with a report if the interface supports multiple reports with different IDs, if it supports the default ID of zero, the Report ID should not be sent with the report, applications running under Windows, should always precede any report to be sent with a report ID. If the Report ID is zero the Device Driver does not send it with the report data, reports read into applications running under Windows should begin with a report ID, this is done by the Device Driver which inserts an ID of zero if necessary (Axelson, 2005) .

## 2.14 Device driver

When a device is attached to the USB port, Windows detects the device and adds the suitable previously installed driver to manage communication between the device and the host, some

devices need to have a special driver, others don't, there are at least two ways to get device drivers ( Axelson, 2005):

- Supported classes such as disk drives, printers, keyboards…etc may use the device drivers included with Windows.

- Custom devices designed to be used with specific application, such as motor controllers and test instruments do not have a built-in drivers because Windows do not know about them, so the developer may need to write a special Device Driver to support his device or design the device to comply with the requirements of supported classes.

Under Windows code run in one of two modes : user or kernel mode, applications written by users must run in user mode, while USB drivers and almost all drivers run in kernel mode, in user mode the access to memory and other system resources is limited by Windows, while in kernel mode unrestricted access to memory and other resources is allowed to the code.

Usually applications use Win32 API functions to communicate with the operating system, drivers communicate with each other using I/O request packets.

### 2.14.1 Communication Flow

- The device must be attached to the USB port.

- Windows detects the attached device and enumerates it and decides upon the suitable Device Driver for the device, this is done by comparing the retrieved descriptors with the information contained in the INF file.

- The application has to get a handle that identifies the device, this is done by calling the CreateFile API function with a symbolic link that identifies the device.

- The device is now ready to transfer data upon the request of the host, for example when the user clicks a button to read data in an application.

- The data to be read should be stored in a buffer specified by the call, Windows has three functions used to exchange data with the device, these functions are ReadFile, WritFile and DeviceIoControl, a call to ReadFile causes the driver to retrieve data from the device or data stored in a buffer and did not be sent according to a host's request, DeviceIoControl can be used to transfer data in both directions instead of using ReadFile and WritFile.

- Writing to a device : similar to reading except for the direction and the API function used which is WritFile instead of ReadFile.

### 2.14.2 Writing Device Drivers

Choosing a Device Driver for a certain device depends on a combination of the performance needed, the cost and transfer rate (speed), the easiest approach to access a USB device is to use the Device Drivers included within Windows, some times the chip's vendor supplied a Device Driver to be used with the chip, this is a general purpose driver ready to be used by the developer, this driver should be accompanied by the needed documentation and source code, the last option available is to write a custom Device Driver, writing a driver is not a trivial task because it requires an investment in tools, experience in C programming and a good knowledge about how Windows communicate with hardware and applications, to write a driver you need Microsoft Visual C++ which is capable of compiling WDM (Win32 Driver Model) drivers, the compiler should include a programming environment and a debugger, other tools that may help include Windows' Device Developers kit (DDK), driver tool kits and advanced debugger (Axelson, 2005).

Following are the steps needed to develop a device driver (SCO Group, 2005).

***Preparation***

- Learn about the hardware. Most of the information you need can be found in the documentation for the device.

- Test the hardware to make sure it is functioning.

- Design the software. Even though the overall structure of a driver is not the same as an application program, good structured design remains important.

- Select a software maintenance and tracking utility.

***Implementation***

- Select one of the sample drivers to use as a starting point for your driver.

- Globally change the prefix used in the sample driver to your driver prefix.

- Modify the initialization entry point routines first and test loading and accessing the driver.

- Write base-level routines before interrupt-level routines.

- If applicable to the device, write and test any associated firmware.

- Develop utilities such as disk formatting, network administration, and diagnostic programs at the same time as the driver.

***Follow-up***

- As much as possible, use the testing phase to create error conditions that exercise the driver's ability to handle them.

- Evaluate the driver's performance both in isolation and in a production environment where other drivers are installed.

- Make sure documents affected by the creation of the driver are updated.

## 2.15 Device Testing

The USB interface is complex and improper functioning Firmware or PC software can make a peripheral impossible to use, so testing a USB product is essential, there are many tools

that help in testing a USB peripheral, these tests include free software tools such as the HidTest program, Protocol analyzers and other test equipment.

The HidTest program exercises the HID API of Windows for a specific device, it calls the relevant HID functions and shows results on the screen, this test is available from the site (http://www.lvr.com) .

Other testing software tools are available from the USB implementers Forum and Microsoft, developers can use these tools to test their devices and their host software, passing the tests can earn a product the right to display the USB Logo or the Microsoft Windows Logo, but to access these resources you have to join the Forum.

The Ultimate tool for USB development is a protocol analyzer, analyzers are used to monitor USB bus traffic, the analyzer is a combination of hardware and software that enables the developer to view every detail of the traffic on the bus, the analyzer collects data the developer requests, then it decodes and displays it in a variety of formats, the developer can watch what happens during enumeration , detect and examine the protocol and signaling errors, view the data being transferred during control, interrupt, bulk and isochronous transfers.

# *Chapter 3*

# USB Interface Board Hardware and Software

## 3.1 Chip Choices

When it comes to choose the chip for a project involving embedded controllers, the decision depends on many features such as:

- The function performed by the chip.
- The cost.
- The availability of the chip.
- The ease of development (the availability of tools, device driver, sample Firmware) and the developer experience with the device hardware and a suitable programming language.

## 3.2 Main Features of the Required Chip

Following are the required chip features based on the Board's Features :

- USB2.0 Full-speed compliant device.
- At least 8 KB RAM for application code space.
- 32 General Purpose I/O.
- Firmware loaded from PC or from I2C.
- Supports Control, Bulk, and Interrupt transfers.
- Can Support at least a total of 4-Input and 4-Output endpoints.

## 3.2 The TUSB3210

The TUSB3210 microcontroller chip was chosen to be the heart of the board, this chip is the one witch accomplishes the required Features of the board, The chip's CPU is clocked at 12 Megahertz, the chip is fully compliant with the USB version 2.0 full-speed-specification, it supports a USB transfer rate of 12 Megabits per second, Figure (3.1) shows the block diagram of the chip.

Figure (3.1) shows the block diagram of the chip.

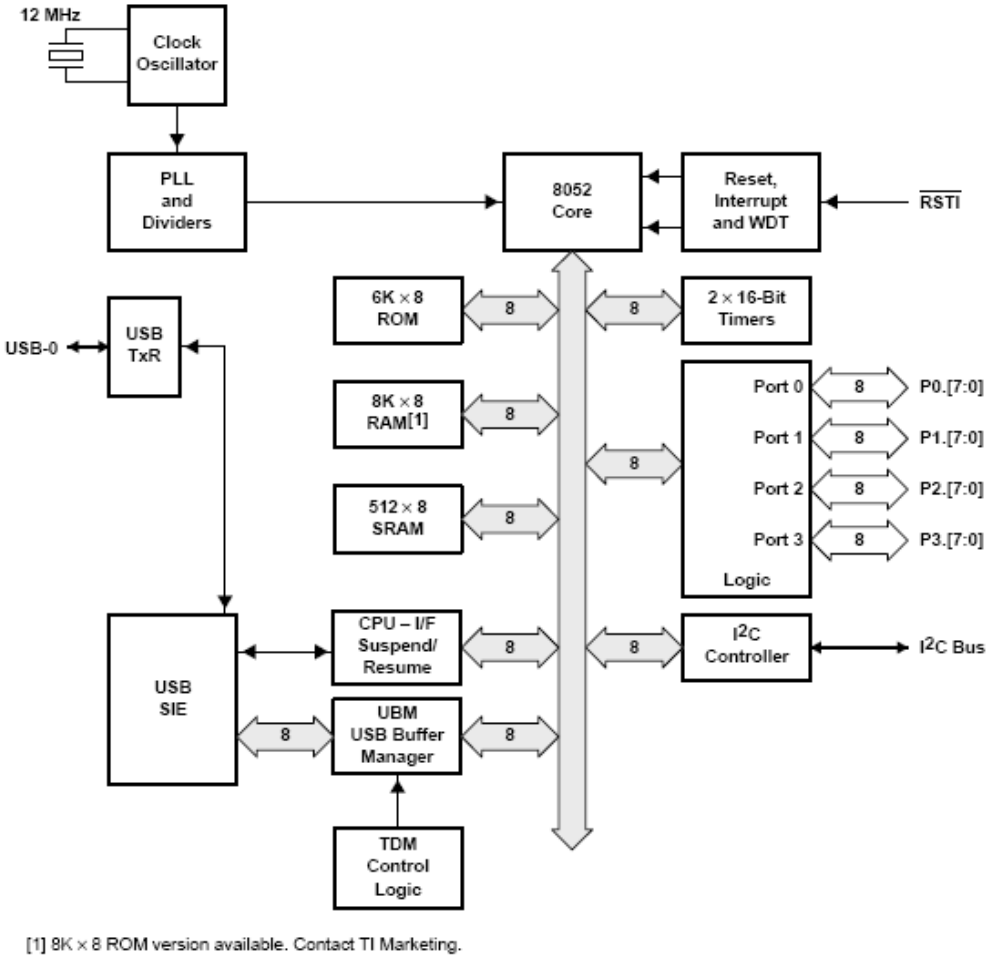



Figure 3.1 : Block diagram of the TUSB 3210 (Texas Instruments, 2003-c).

### 3.2.1 Why Choose the TUSB3210

This choice fulfills the minimum main requirements of the required chip, in addition the TUSB3210 is notable for the following reasons:-

- Since it is 8052 compatible, developers are familiar with the architecture and programming of this family, this makes the development process easier.

- The chip is well supported, it has Evaluation Modules and Product Developer's Kits, detailed Technical Documentation and other Utilities and Tools such as Apploader Driver, Bootcode Source Listing and Keyboard Sample Source Code.

- It's cheap, the chip costs about 2.5 $, additionally it's popular because many Forums are available on the Internet for developers interested in this chip .

- The availability of Host application and Firmware sample codes in the market, these codes may be used as a starting point in developing your own codes .

### 3.2.2 The Board Schematic Diagram

The USB Board has been built according to a schematic diagram provided by Texas Instruments, the schematic diagram can be found in Appendix F, the Hardware design is assumed to be generic, it is designed for use with personal computers running a USB enabled operating system, the operating system of the PC must be USB 1.1 specification compliant, the board consists of several circuits, following is a brief description for each circuit :

- **Power Supply Circuit** : As shown in here in the schematic, the board is supplied by a positive 5-volt power supply, there are two options for supplying the power to the circuit, bus-powered mode where 5 volts are supplied by the USB bus, this is achieved by setting JP(3) to position 1-2, the other option is the self powered mode where JP(3) is set to position 2-3, in addition pin 21 in the chip must be grounded for bus powered mode, in any case the input 5 volts are fed into a voltage regulator (278R33) which converts the 5 volts to 3.3 volts, the 3.3 volts are converted to positive 1.8 volts using R3 and R18 as voltage divider, the 1.8, 3.3 and 5 volts are used to supply the chip with the required voltage levels.

- **Reset Circuit** : when the switch push button is pushed down, pin 13 (reset pin) in the chip is connected to ground via resistor R15, this pin is the controller master reset signal pin which resets the controller.

- **Run/Suspend modes Circuit**: In this circuit D4 determines the state of the board, when D4 is ON, the board is powered and not suspended, when the SUSP pin (pin 16 in the TUSB3210) is in the low state, transistor Q2 is switched ON and a positive voltage drop is developed across the D4 diode, which as a result illuminates, when D4 is OFF, the board is not powered or suspended, once the SUSP terminal in the chip goes high, Q2 is turned OFF, and D4 is reversed biased and thus turns OFF.

- **The Main Circuit:** In this circuit, when the TUSB3210 is connected to the USB bus through Type B USB-shield as shown in the schematic, 5 volts are applied to the Base of the Q1 Transistor through R5, when pin 17 (PUR) of the TUSB3210 goes high (pull-up is enabled), Q1 is switched ON, as a result terminal DP0 (DP) is pulled high with respect to DM0 (DM) through resistor R4, this means that a Full speed device is attached to the bus, on the other hand if PUR terminal (pin 17) is pulled down to zero (pull-up disabled), Q1 switches OFF and no voltage difference is developed between DP0 and DM0, this means that: even if a device is attached it will not be enumerated by the host as long as the pull-up is disabled, the 12 MHz crystal oscillator is connected between pin 61 and 60 of the TUSB3210.

## 3.3 Writing Firmware

Any project that includes a controller chip is useless unless the developer writes the code that enables the device to communicate with the host and vice versa.

### 3.3.1 Hardware responsibilities.

The USB controller in the device should manage many of the communication's responsibilities automatically, the remaining tasks should be done by the Firmware which should supplement the hardware's capabilities, these tasks include sending data to the host, receiving data from the host, handling interrupt transfers and responding to interrupts generated by the hardware at the end of each transaction .

### 3.3.2 TUSB3210 Firmware[5] (Texas Instruments, 2000, 2001-a, 2003-d, 2004)

The best software (according to Texas Instruments and other USB Sites) for writing the Firmware code is the Keil uvision2 'C' compiler (Keil Electronik, 2003), other cheap alternatives exists such as the SDCC 8051 developing tool, in this project the Keil 'C' compiler was used to develop the Firmware for the following reasons :

- It is one of three packages the vendor advices to use for developing the Firmware.
- I evaluated the Demo version of this software, It is easy to use, and well documented.
- It combines project management, source code editing, and program debugging in one powerful environment.
- I am familiar with C++ programming .

The developed Firmware consists of eight header files (*. h) and six (*. c) files, some of the header files and functions used in these files are taken from the Texas Instruments TUSB2136 Generic Keyboard Demo Program[6] (Texas Instruments, 2000), the header files are:-

- Usbinit. h.
- Usb. h
- Delay. h
- Descriptor. h
- Reg52. h
- Tusb3210. h

[5] For Firmware source code, see Appendix D

[6] For more information refer to Appendix D

- Prog. h.
- Application. h.

The (*. c) files are named :

- Usbinit. c.
- Usb. c
- Delay. c
- Prog. c
- Application. c
- Main. c

Following is a general description of each of the above mentioned files.

#### 3.3.2.1 Usbinit. h

This file is a header file for initializing the USB board.

#### 3.3.2.2 Usb.h

This file is a header file for the USB protocol functions.

#### 3.3.2.3 Delay. h

- This file is a header file for the Delay functions.

#### 3.3.2.4 Descriptor .h

This file contains the definitions of the following descriptors :-

- Device Descriptor.

- Report Descriptor.
- Configuration Descriptor.
- Interface Descriptor.
- HID Descriptor.
- Input Endpoint 1 Descriptor.
- String Descriptors.

#### 3.3.2.5 Reg52. h[7]

This file contains the definitions of the 8052 controller's registers addresses and interrupt vectors.

#### 3.3.2.6 Tusb3210. h

- This file contains the definition of the Tusb3210 registers (Texas Instruments, 2000).

#### 3.3.2.7 Prog. h

This file is a header file for initializing the device and decoding the device data functions.

#### 3.3.2.8 Application. h

This file is a header file for the host application data exchange functions.

#### 3.3.2.9 Usbinit. C

This file consists of two functions (Texas Instruments, 2000) :-

- *InitializeUsbFunction (Void).*

[7] For more details refer to the TUSB3210 datasheets and the 8052 tutorial and reference.

- *UsbReset (Void).*

The first function is used to initialize the USB device and all its registers, it enables the external zero (EX0) and, global (EA) interrupts, it also enables the pull up to enumerate the device on the USB bus, this function calls the UsbReset ( ) function in order to Reset the USB device, when this function (InitializeUsbFunction) is called the device is disconnected from the bus.

The second function (UsbReset) resets the USB device, enables endpoint zero and endpoint one interrupts, it also enables the USB specific interrupts (SETUP, RESET, and STPW).

#### 3.3.2.10 Usb. C

Then file includes the following functions :

- *UsbGetConfiguration (Void)*: this function sends the current Configuration value to the host upon request (Texas Instruments, 2000) .

- *UsbSetConfiguration (Void)*: this function sets the device Configuration (at the end of a Set Configuration request, the device enters the configuration state) and Stalls output endpoint zero (Texas Instruments, 2000) .

The following functions are called on power up when the host starts enumerating the device and are used to obtain the device, configuration and string descriptors.

- *UsbGetDeviceDescriptor (Void).*

- *UsbGetConfigurationDescriptor (Void).*

- *UsbGetStringDescriptor (Void)* (Texas Instruments, 2000) .

The following functions are used for HID (Human Interface Devices) devices.

- *UsbGetHIDDescriptor (Void)*: sends HID descriptor to the host (Texas Instruments, 2000) .

- *UsbGetReportDescriptor (Void)*: sends report descriptor to the host.

- *UsbSetReport (Void)*: the Set Report request is sent by the host to the HID device, when the Set-Report Setup Packet is received, this initiates a "Receive Data Packet" sequence since the actual data value will be in the next packet on OEP0.

- *UsbGetInterface (Void)*: this function sends the current interface number to the host (Texas Instruments, 2000).

- *UsbSetInterface (Void)*: this function sets the interface number sent by the host (Texas Instruments, 2000).

- *UsbGetDeviceStatus (Void)*: this function sends device statues to the host (Texas Instruments, 2000).

- *UsbSetRemoteWakeup (Void)*: this function sets remote wake up (Texas Instruments, 2000) .

- *UsbClearRemoteWakeup (void)*: this function clears the remote wakeup option (Texas Instruments, 2000).

- *UsbGetInterfaceStatus (Void)*: this function sends the interface status to the host(Texas Instruments, 2000) .

- *UsbSetAddress (Void)* : this request is used by the host to assign an address to the nearly attached device, usually the device is given a zero address until the host assigns another address after the enumeration process has been successfully completed.

- *UsbSetEndpointHalt (Void)*: this function when called stops the input Endpoint 1 from sending data to the host.

- *UsbClearEndpointHalt (Void)*: this function enables the input Endpoint 1 to send data to the host.

- *UsbGetEndpointStatus (Void)*: this function when called allows the device to send the status of IEP1 (Input Endpoint1) to the host.

- *UsbNonStandardRequest (Void)*: this function allows any non standard or unrecognized request to arrive to it by default, the Endpoint 0 is automatically stalled to indicate that the request is invalid or unrecognized (Texas Instruments, 2000).

- Code *DEVIC-REQUEST-COMPAIRE tUsbRequestList [ ]*: this code defines a lookup table, using the structure defined in the header file. The values of constants used in this structure are defined in the usb. h file, this table includes the standard Device Requests, the standard Interface Requests, the Class specific Interface Requests and the standard Endpoint Requests (Texas Instruments, 2000) .

**Structure of the table:**

- ✓ bmRequestType: indicates the type of request, this is a bit mapped variable defined in the USB specification.

- ✓ bRequest: indicates the Request ID (Get Descriptor, Get Status, etc.), these are defined in the USB and HID specification and declared in "Usb. h".

- ✓ bValueL/H: additional values, purpose varies with requests.

- ✓ bIndexL/H: additional values, purpose varies with requests.

- ✓ bLengthL/H: number of bytes to transfer to or from the host.

- ✓ bCompareMask: indicates which of the above bytes should be compared to determine the function to call. For example, the mask 0x80 means only bmRequestType must match, 0xC0 means both bmRequestType and bRequest must match, If this variable is 0x00, as is the case in the last entry in the table, then no bytes are compared and, thus, ANY packet will pass the comparison stage.

- *usbDecodeAndProcesUsbRequest (Void)*: this function is called when a USB request has been received. It searches the tUSBRequestList [ ] structure defined in the previous section for a request that matches a given entry in the table and, when matched excites the corresponding function (Texas Instruments, 2000) .

- *usbStallEndpoint0 (Void)*: this function sets the STALL flag on both IEP0 (Input Endpoint zero) and OEP0 (Output Endpoint zero).

- *usbReceiveDataPacketOnEP0 (unsigned char * pBuffer)*: this function enable the device to receive a dada packet on EP0 (Endpoint zero), this function is called when a SET_REPORT token is received (Texas Instruments, 2000) .

- *usbReceiveNextPacketOnOEP0 (Void)*: this function allows the device to receive a data packet on EP0, if all data has not been sent, send the rest of data now.

- *usbSendZeroLengthPacketOnIEP0 (Void)*: this function allows the device to send a zero- length packet back to the host on IEP0, often called to acknowledge a packet received from the host that requires no data in the reply, just an acknowledgment of receipt (Texas Instruments, 2000).

- *usbSendDataPacketOnEP0 (unsigned char * pBuffer)*: this function allows the device to send data packet to EP0, this is used to send descriptors and other requests to the host, the length of data is defined in the Setup Packet (Texas Instruments, 2000).

- *usbSendNextPacketOnIEP0 (Void)*: this function allows the device to send the following packets to IEP0 if not all data has been transferred with the last IN token (Texas Instruments, 2000).

- *usbSendDataToHostOnEP1 (Void)*: this function allows the device to send the data in the device transmit buffer to the host on EP1, this data is from reading a port ,status ...etc, there is no defined length for the data, the length is defined in the Report Descriptor (64 bytes).

- *SetUpPacketInterruptHandler (Void)*: this function is called by the USB interrupt function when a Setup Packet is received. This function immediately sets both OEP0 and IEP0 to a NAK state, and sets the USBCTL to send / receive based on the direction of the request, then proceeds to call the usbDecodeAndProcessUsbRequest( ) function, which determines which function should be called to handle the given USB request (Texas Instruments, 2000).

- *OEP0InterruptHandler (Void)*: this function is called by the USB interrupt function when a USB interrupt is called by OEP0 (Texas Instruments, 2000).

- *IEP0InterruptHandler (Void)*: this function is called by the UsbInterrupt function when a USB interrupt is called by IEP0, this will happen once the data sent by calling usbSendNextPacketOnIEP0( ) and means that , the previous data packet has been sent, at that point there are two condition : either there is more data to send or there is not, it there is, we call usbSendNextPacketOnIEP0 ( ) to send next packet of data, if there is not any more data, we STALL, however, if the bStatusAction condition indicates that we were changing the devices, we do so at this point (Texas Instruments, 2000).

- *IEP1InterruptHandler (Void)*: this function is called by the USB Interrupt function when a USB interrupt is caused by IEP1, this will happen once the data sent by calling usbSendDataPacketOnIEP1 ( ) and means the previous data packet has been sent.

- *EX0_int (Void)*: this is an interrupt service routine for the external Interrupt 0 (Texas Instruments, 2000).

#### 3.3.2.11 Delay. C

This file consists of two functions, the first provides time delay for a specified time in milliseconds, the other function provides 5 microseconds time delay.

#### 3.3.2.12 Application .C

This file is responsible for data exchange between the peripheral and the host, it includes two functions :

- *Write_Data (Void)*: this function, when called, writes the data stored in the host buffer to Port one of the USB device. The device receives always a "59" bytes of data, knowing that "64" bytes except the report ID are sent, the first 5 bytes sent by the host are as follows:-

    1. Byte 1 = DataFromHost[0] : used for device information, in this case it is the hypothetical device which consists of LEDs.

    2. Byte 2 = DataFromHost[1] : contains the mode (write or read) "1" for read and "2" for write operations.

    3. Byte 3 = DataFromHost[2] : contains the size of data in bytes. (actual data Byte count)

    4. Byte 4 = DataFromHost[3] : indicates whether this packet is the last one (byte 4=0), or there is more data to follow (byte 4=1).

    5. Byte 5 = DataFromHost[4] : general purpose data byte.

This function writes all the data sent by the host and after sending the data it resets the device by calling the function *ResetDevice ( )*, it also sends a byte to the host indicating that it received the last data packet.

- *Read_Data (Void)*: this function is used to read the data from Port "1" of the device, the device receives 64 bytes from the host, in this stage no data is sent, bytes from one to five contains information as follows :-

Byte 1 = DataFromHost[0] : the hypothetical device connected to the USB ports (the LEDs).

Byte 2 = DataFromHost[1] : the mode which is one for reading from Port.

Byte 3 = DataFromHost[2] : the maximum read bytes which are in this case "63" bytes.

Byte 4 = DataFromHost[3] : not used in this mode.

Byte 5 = DataFromHost[4] : contains the actual read bytes defined by the user.

This function sends the data to the host when the device transmit buffer is full (contains a maximum of 63 bytes of data) the first byte Buffer[0] (from the 64 bytes sent) contains the number of bytes to be send to the host (size of data).

#### 3.3.2.13 Prog. C

This file includes two functions :

- *ResetDevice (Void)*: this function is responsible for initializing the device ports by setting them to zero except port 2 which is used for Interrupts , also the pull-ups of the chip's ports are enabled in order to enumerate the device.

- *DecodeDeviceData(unsigned char Data)* : This function is used to Decode the data from host to read port or write to port ,first Byte indicates the device(LEDs), second Byte indicates the mode (read, write)

#### 3.3.2.14 main. C

This is the main function, it initializes the TUSB3210 chip, it starts by setting the SDW (set by the boot program) in the configuration register of the chip, then it disables the watchdog timer and delays port 2 interrupt by 2 ms, then it resets the device using the function *Reset_Device ( )* and finally the USB registers are initialized using the function *InitializeUsbFunction ( ),* all these actions are forced to run in an infinite loop using a while(1) command.

## 3.4 The Host Application Software[8]

The host computer has to maintain the state of the USB bus and monitor all the devices attached to the bus, in a USB network, only one host exists .

The host controller is responsible for managing the physical part of the Universal Serial Bus, the host application software is responsible for communication with the host controller which communicates with the devices connected to the bus, the host system has three sections :

- The USB hardware interface.
- The system's Device Driver (software).
- The USB client software (application program and the peripheral Device Driver).

Windows 98 and later include every thing needed for an application to communicate with a HID class device, if a decision is made to use the HID Driver included with Windows, there is no need to write special drivers, before an application can communicate with a HID device, it has to identify the device and learn about the report's format, this is done by calling a series of API (Application Programmer's Interface) functions, first the application finds the attached HID device, it requests information about the device in order to find the suitable Device Driver, then it can exchange information with the device by sending and receiving reports.

Table (3.2) lists API (Application Programmer's Interface) functions used to establish communication and exchange data with HID devices (Axelson, 2005).

[8] For Host Application software's source code, see Appendix E

Table 3.1 : API functions used to establish communications and exchange data with a HID.

| API Function | DLL | Purpose |
|---|---|---|
| HidD-GetHidGuid | hid.dll | Obtain the GUID for the HID class |
| SetupDiGetClassDevs | setupapi.dll | Return a device information set containing all of the devices in a specified class. |
| SetupDiEnumDevuceInterfaces | setupapi.dll | Return information about a device in the device information set. |
| SetupDiGetDeviceInterfaceDetail | setupapi.dll | Return a device pathname. |
| SetupDiDestroyDeviceInfoList | setupapi.dll | Free resources used by SetupDiGetClass-Devs. |
| CreateFile | Kernel32.dll | Open communications with a device. |
| HidD-GetAttributes | Hid.dll | Return a Vendor ID, Product ID, and Version Number. |
| HidD-GetPreparsedData | Hid.dll | Return a handle to a buffer with information about the device's capabilities |
| HidP-GetCaps | Hid.dll | Return a structure describing the device's capabilities |
| HidD-FreePreparsedData | Hid.dll | Free resources used by HidD-GetPreparsedData. |
| Write-File | Kernel32.dll | Send an Output report to the device |
| ReadFile | Kernel32.dll | Read an Input report from the device |
| HidD-SetFeature | Hid.dll | Send a Feature report to the device |
| HidD-GetFeature | Hid.dll | Read a Feature report from the device |
| CloseHandle | Kernel32.dll | Free resources used by CreateFile |

### 3.4.1 Code Functions and Procedures

Code needed for communication with a HID can be written in C++, Delphi or Visual Basic, in this application Delphi (Borland corporation, 2002) was used to write the PC software. To use Delphi for writing the code you need the Delphi plus the HID Component (Marquerdt, 2004) which is needed to give complete access to all HID devices connected to computers using an Operating System like Windows 98 and later

versions, this component is a controller component which handles all the HID device plugs and unplugs.

Borland Delphi7 was chosen to develop the PC software for the following reasons :

- The full version is available in the local market with a low cost.
- I am familiar with Object Oriented Pascal programming which is the basis for Delphi programming.
- The HID component is available for free from the internet, many related sample codes are included which helps the developer to understand how PCs communicate with HIDs using API functions.

Two versions of the host software were produced, one to be used with practical applications and the other for testing the board, the latest will be described in the following sections.

The host application software consists of the following procedures and functions:-

#### 3.4.1.1 *HidCtlDeviceChange*

This procedure detects any new HID device when it is attached to the bus and enumerates it.

#### 3.4.1.2 *HidCtlEnumerate*

This function checks the attached device by index which is provided by the OnEnumerate event, if the index is within bounds and the device was not checked out already by CheckOutByIndex , the CheckOutByIndex function returns True, otherwise it returns False and HidDevice is set to nil. The HidCtlEnumerate function detects the attached device by checking the VID (vender ID) and PID (product ID), if these values match those for the TUSB3210, then the enumeration process continues and the main menu buttons are enabled, if the function returns False the Enumeration stops.

#### 3.4.1.3 *WriteBufferToDevice*

This procedure is activated if a device is connected and enumerated correctly, if so the report ID is set to zero ( Buf [0]= 0 , the first entry in the 65 bytes buffer), this value is always set zero. Then Buf [1] is set to 1 which is the connected device information, in this case it is the LEDs connected to Port 1 of the TUSB3210, then the "OutputReportByteLength" is determined and assigned to the variable "ToWrite" which is then sent as an argument to the "WriteFile" function which is an API function used to send the contents of the Buffer (report) from the host to the device.

#### 3.4.1.4 *SendDataPacketToDevice*

This procedure is responsible for preparing the data to be send to the device as 64 bytes packet, particularly :

- It fills the contents of data buffer from entry (Buf [6] to entry Buf [64]) with data bytes entered by the user .
- It processes the data to be displayed on the PC screen to the hexadecimal format.
- It determines according to the actual data size (Buf [3]) whether this data packet is the last packet (Buf [4]=0 ), or there is more data to be sent ( Buf [4]=1).
- It calls the WriteBufferToDevice procedure to transfer the data packets.

#### 3.4.1.5 *HidData*

This event automatically starts a thread to read the device, each time the device sends a report the event fires and presents the report (Marquerdt, 2004), this means that this procedure is responsible for receiving the reports sent by the device and decode it according to Firmware in order to take decisions.

In particular the following actions were monitored:-

- If the device is not in a read-status the procedure checks the first data byte received from the device if this byte is a "*" this means that the device has received the last data packet and is ready to receive a new packet.

- If the user clicks the read button then the device should read data from port 1 according to Firmware.

- If the user clicks the write button, the device should be ready to receive data as in the Firmware.

- Else the status of the device is set to NOTHING.

- If the user clicks the write button and the device sends a "w" to the host, this means that the last data packet was sent to the device and the host should stop sending data, if the write button is clicked again, the status of the device is set to NOTHING.

- If the status of the device is set to NOTHING then enable the interface buttons in order to start communication again with the device.

#### 3.4.1.6 *WriteDevice*

This procedure is used to send the report's data to the device, in details the following actions take place:-

- The status of the device is set to WRITE.

- Buf [2] is set to "2" to select the write mode.

- Buf [5] is set to zero indicating that it is not used in this mode.

- The procedure *SendDataPacketToDevice* is called to complete the writing process.

- The contents of the data packet are displayed on the screen.

#### 3.4.1.7 *ReadDevice*

This procedure is used to read the data from Port 1 of the device in particular:-

- The status of the device is set to READ.
- Buf [2] is set to "1" to select reading mode.
- Buf [3] is set to have the value of the global variable " MaxReadBytes= 63", which is the maximum bytes that could be read from Port 1 at a time.
- Buf [5] is set to a value equal to the "ActualReadBytes" which is set by the user in this application, in other applications this value might not be determined, but can be calculated depending on the application.
- The read buffer is filled with data bytes the device reads from Port 1, this is done by calling the *WriteBufferToDevice* procedure.

#### 3.4.1.8 *ShowBufferContents*

This procedure is used to display the contents of the reports sent by the host at any time on the screen.

#### 3.4.1.9 *HidCtlDeviceDataError*

This procedure is an event on the "TjvHidDevice", this event occurs if "On Data" encounters a read Error on calling "ReadFileEx", the parameters of the event are the device which encountered the read error and the error value gathered through "GetLastError " (Marquerdt, 2004)[9].

#### 3.4.1.10 *breadClick*

This procedure is used to activate the "*ReadDevice*" procedure by setting "*ButtonClicked*" to B-Read.

[9] For more details review the HID component documentation.

#### 3.4.1.11 *bwriteClick*

This procedure is used to activate the "*WriteDevice*" procedure by setting the "*ButtonClicked*" to B-Write.

#### 3.4.1.12 *bterminateClick*

This procedure is used to terminate the application.

#### 3.4.1.13 *SaveBtnClick*

This procedure is used to save the contents of the history list box (PC display) to a text file named by the user.

#### 3.4.1.14 *DisableButtons*

This procedure is used to disable the following buttons (options) on the main application interface menu :

- Writing .
- Reading .

#### 3.4.1.15 *EnableButtons*

This procedure is used to enable the following buttons (options) on the main application interface menu :

- Writing .
- Reading .

#### 3.4.1.16 *FormActivate*

This procedure is used to create and activate the Edit elements used to enter data by the user in the application interface menu .

#### 3.4.1.17 *ClearBtnClick*

This procedure is used to clear the contents of the history list box (display) on the application interface menu .

#### 3.4.1.18 *BuffBtnClick*

This procedure is used to call the "*ShowBufferContents*" procedure when the suitable button on the main interface form is clicked.

### 3.5 Board's Competitive Factors

This board has been built according to a schematic provided by Texas Instruments, the Texas Instruments has a similar commercially available board named TUSB3210GENPDK Generic Product Development Kit, this Kit has the same hardware design as the built board, this Kit is shipped with a Keyboard Firmware source code , this product costs 199$, concerning the built board and taking into consideration the bill of material for the board (can be found in Appendix G), the hardware components and other materials needed to build the board costs about 140 N.I.S., which is about 30$.

Compared to the TUSB3210GENPDK Kit, the built board has the same performance since it has the same hardware and a similar HID Firmware code, so the two boards are expected to have the same data transfer rate, this means that the cost of the built board is a competitive factor in favor of the built board.

### 3.6 Difficulties Faced

- The electronic components needed to build the board's circuits and the printed board needed to fix the tiny surface mounted TUSB3210 were not found in the local market (West Bank), some were purchased from U.S.A., the remaining were purchased from Haifa.

- The availability of the software system needed to develop the Firmware (Keil C51) was the major difficulty, the system is very expensive, it costs about 3000$ and can be purchased from the Keil dealer at Haifa, in the beginning of the

project and to overcome this difficulty I used the SDCC 8051 system to develop the Firmware, this system is available for free downloading from the Internet, the system is difficult to use and does not contain a simulator or a debugger like the keil, I managed to obtain the Firmware hex file, this file was converted to a binary file using a suitable utility, the Firmware was downloaded to the chip's RAM, but Windows can't recognize the board, this mains that the Firmware is not functioning, with this result we insisted on purchasing the Keil .

- I tried to benefit from the Keil Demo version, but the demo was restricted to 2Kbytes of object code, this restriction did not allow me to develop the Firmware which has more than 2Kbyts object code.

- After one year we managed to purchase the full version of the Keil C51 system at a low price, this makes the development process easier and more efficient.

- Many difficulties were faced during the development of the Firmware and host application software because this is my first experiment with microcontrollers, the availability of sample codes helped a lot in solving many problems.

*Chapter 4*

# USER MANUAL

This guide describes the setup and operation of the peripheral interface board, the user is assumed to be familiar with the universal serial bus (USB) protocol, also he should download and review other software needed for setup, this software is available for downloading from the Texas Instruments Site (www.ti.com).

## 4.1 Minimum Requirements to Operate the Board

The board is designed for use with a personal computer running a USB-enabled operating system, the PC should be 1.1 specification compliant (has a hardware support consisting of a USB host controller and a root hub with at least one USB port ), the main component of the board is the TUSB3210 from Texas Instruments .

The Firmware of the TUSB3210 can be downloaded from the host computer via USB bus every time power is supplied to the device (this enables debugging the Firmware easily)or it can be programmed via an inter-IC ($I^2C$) serial interface at power on from an EEPROM, when downloaded from the host the Firmware is loaded into an 8 Kbytes RAM memory using a built in boot loader. In this board the Firmware is downloaded via the USB port, this requires a driver on the PC to send the Firmware to the chip, once the Firmware is loaded into the RAM the boot loader software disconnects from the USB and the Firmware is executed.

### 4.1.1 USB Board Features

- Fully compliant with the USB release 1.11 HID specification.
- Data transfer rate : 64 Kbytes per second.
- High performance 12 MHz integrated 8052 controller.
- 256 X 8 RAM for internal Data.

- 8K X 8 RAM Code space.
- 512 X 8 shared RAM used for Data Buffers and Endpoint Descriptor Block.
- Four GPIO Ports (ports 0,1,2 and 3).
- Watchdog timer.
- Could be Bus or Self Powered.
- On-chip PLL generates 48 MHz.
- Power- down mode .
- Three 16- bit timer/counters.
- Supports SUSP and PUR pins.

Compared with the existing commercial boards, it differs in two aspects :

- ✓ The transfer rate : which is limited to 64 Kbytes per second, this relatively low rate results from using the HID Device Driver included with Windows which supports Control and Interrupt transfers only (Axelson, 2005).
- ✓ The board's software codes do not include DLLs, which make it easier to access any of the controller ports from the PC side.

### 4.1.2 Hardware Overview.

The board provides a platform that is practical and easy to use, it is designed to operate from an external 12-MHZ crystal, the board is set up for bus powered operation using 5-V to 3.3V voltage regulator, the UART is disabled, several test points have been added to the board for testing purposes (Texas Instruments, 2001-b).

Port 1 is used for data exchange, other ports' pins could be used as needed by the application, eight LEDs are connected to Port 1, these LEDs are connected in a common anode configuration, LED (D4) provides power and suspended status of the device (board).

### 4.1.3 Schematic Diagram

The complete schematic (Texas Instruments, 2001-b) of the board can be found in Appendix F.

### 4.1.4 Host Application Software.

The host software is written using Delphi 7.

The application interface form has the most needed programming functions such as:

- Save: saving the contents of the History List Box (screen) to a file.
- Read-from-Device: reads the bytes on port 1, the number of read bytes is set by the user in a special Edit location, the read bytes are displayed on the History List Box on the PC screen.
- Write-To-Device: sends the given number of bytes (maximum 59 bytes) to port 1, the actual number of bytes to send and the values of these bytes are filled by the user in the available Edit cells on the main form, the sent bytes are displayed on the History List Box.
- Show Buffer Contents: this option displays the contents of the buffer (at any time) on the History List Box (screen).
- Clear list: this option clears the contents of the History List Box (screen).
- Terminate Application: this option terminates the application.

The status of the device and the current action is displayed on the status bar at the top of the form.

Figure (4.1) shows the application interface form (screen).

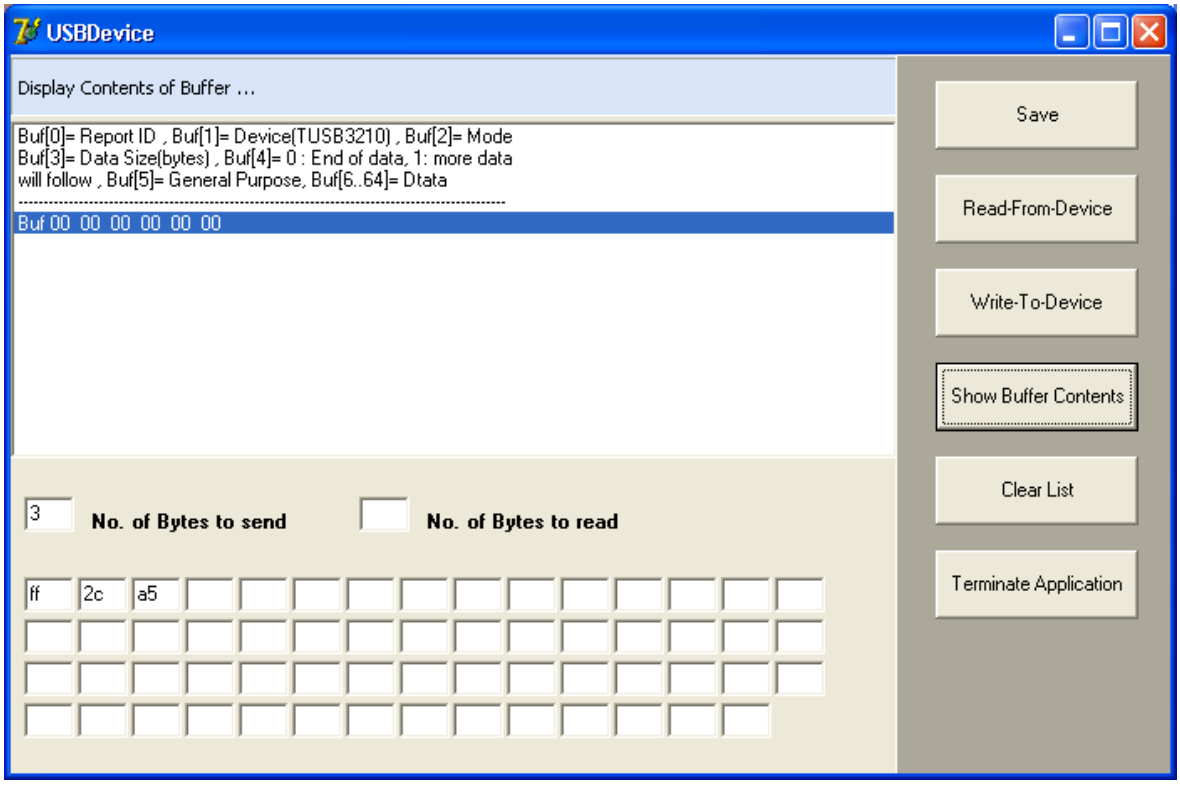



Figure 4.1 : The application interface form.

### 4.1.5 Communication Protocol

The protocol is needed to insure that communication between the host and the device is achieved correctly and as required, the host sends 64 bytes to the device, the format of the sent report is shown in figure (4.2), the contents of these bytes are as follows:-

- The first byte Buf [1]: contains information about the device connected to the USB port (port 1 ), in this case Buf [1]=1 means that, the selected device is the LEDs connected to port 1 for testing , this means that other value may be given to other connected device.

- The second byte Buf [2]: gives information about the mode ( action ), Buf [2]=1, means that the selected action is *Reading* from port 1, Buf [2]=2, means *Writing* to port 1.

- The third byte Buf [3]: contains the actual byte count.

- The fourth byte Buf [4]: the value of this entry indicates whether the data packet sent is the last one (Buf [4] =0) or whether more data is available to be send (Buf [4]=1).

- The fifth byte Buf [5]: this is a general purpose byte, it is used in the reading mode to indicate the actual number of bytes to read from port 1.

- The bytes from Buf [6] to Buf [64] are used for data bytes to be send to the device.

After each action the device should send a number of bytes to the host (Handshaking signals), informing it to send more data or to stop sending data.

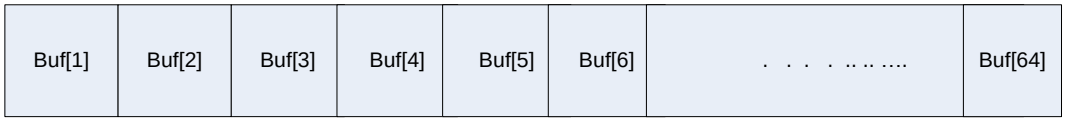



Figure 4.2 : Output report format

### 4.1.6 The Device Firmware

The Firmware of the TUSB3210 was written using the keil uvision2 "C51" Compiler, the main program (main .c) disables the watchdog timer and then resets the device using the function *ResetDevice( )*, then the USB registers are initialized using the "*InitializeUsbFunction ( )*" function, then the pull up is enabled to 3.3V to enumerate the device.

After the Boot Loader completes downloading the Firmware to the controller's RAM, it disconnects from the bus and the Firmware starts execution by setting the SDW bit in the configuration register to 1 to switch the memory map to normal mode (the 8k-RAM is mapped to code space), the device is now ready to be enumerated by the host, this is done by the host when it sends a number of SETUP tokens containing Requests to identify the device Interface and configure it's endpoints, the enumeration process uses the default pipe (endpoint 0) for the configuration of the device, during enumeration the host retrieves

several Descriptors, these Descriptors include the Interface Descriptor which identify the device as a HID, then the host assigns a unique address for the device and a suitable device driver according to the retrieved information.

The device is now ready to operate, if the host sends an IN token, then either status information or data packet is supposed to be transmitted by the device to the host, if the host wishes to send data, it issues an OUT token followed by the data payload, after receiving a SETUP, IN, or OUT transaction, the hardware triggers an interrupt which forces the Firmware to jump to a suitable Interrupt Handler to prepare endpoints and copy the data to its suitable buffer.

### 4.1.7 Communication Process.

Data sent by the host is decoded using the "*DecodeDeviceData ( )* " file, the first byte (Data [0]) contains information about functions connected to port 1 of the microcontroller, in this case (Data [0]=1) means that the function is the LEDs connected to port 1, the second byte (Data [1]) selects the desired mode, if (Data [1]=0x01),this indicates reading mode, if (Data [1]= 0x02), this indicates *Writing* mode, after executing the desired action the device is Reset using the "*ResetDevice ( )*" function, after the device finished processing the first Report (64 bytes) and upon the type of message sent by the device, the host should decide whether or not to send another report. When there is no data to be send the host should inform the device that this packet is the last one by sending a zero byte (Buf [4]= 0).

When reading data from port 1, the read report consists of 64 bytes, one byte for the actual number of read bytes (first byte) and the remaining bytes are the data read from port 1.

## 4.2 Building the Board.

The most important part of building the board is soldering the tiny TUSB3210 chip with many fine leads on the special printed circuit board, soldering this kind of chips requires confidence and experience.

### 4.2.1 Interfaces and USB Port

The USB board uses a standard cable connected from one end to the USB port of the PC, and from the other end to a standard type-B USB connector (this is the upstream port), in this design the circuit of the I²C serial interface was not built, but this option could be added to enable the user to access an I²C EEPROM, the UART port is embedded in the IC, in the current design it is not used but the circuit can be built (see the schematic diagram) and connected to the RS-232 port if needed.

### 4.2.2 Supplying Power to the Board

The TUSB3210 requires a power supply with the following rating:-

- Positive regulated 5 volts dc.
- The supplied current should be at least 0.5 amperes.

Power can be supplied to the chip using two modes :

- Self-powered mode: in this mode an external switching 5 volts dc power supply should be plugged into the input power socket (J1) on the board.
- Bus-powered mode: in this mode the board is supplied with a 5 dc volts via the USB cable.

According to the data sheet two voltage rates should be generated from the supplied 5 volts they are:-

- 3.3 volts which is generated from the 5 volts using a voltage regulator (278R33).
- 1.8 volts which is generated from the 3.3 volts using a voltage divider as in the schematic diagram.

When the board is powered correctly (D4) should turn on.

### 4.2.3 Light Emitting Diodes (LEDs) Used.

The board contains “9” LEDs:

- D4: when this LED is on, this indicates that the board is powered and not suspended, when it is off, the board is not powered or suspended.

- D5...D12: these LEDs are connected to port 1 of the microcontroller, they work as a hypothetical device connected to the port to show the complement of the ports' output and input bytes (the LEDs are connected in common anode configuration).

### 4.2.4 Jumpers

The following description may help the user to configure the board jumpers to the required mode of operation, as explained before, the Firmware can be downloaded from the host to the chips' RAM using a loading program supplied by the vendor or the Firmware can be stored on an I²C EEPROM, the board can be powered from an external 5 Volts supply or from the USB cable, following is a description of the jumpers used in the board :

- U2: this jumper is related to the board power mode, if it is set to position 2-3, the board is self-powered from an external source, if it is set to position 1-2, the board is bus-powered via the USB cable.

The following jumpers are not used in the current design, but if the user wishes to build the circuit containing the jumpers, he should configure them properly :

- JP2: this jumper is related to the RS-232 circuit, when this jumper is set to connect points 1 and 2, it connects P3.0 to R1OUT, when position 1-2 is off, it disconnects P3.0 from R1OUT.

- JP3: this jumper is related to the RS-232 circuit, when this jumper is set to connect points 1 and 2, it connects P3.1 to T1IN, when it is set to the off position, it disconnects P3.1 from T1IN.

- JP4: this jumper is related to the MCU's UART, when it is set to connect point 1 and 2, it enables the RS-232 port, when it is set to the off position it disables the RS-232 port.

## 4.3 Board Installation

After completing the construction and testing of the board we have two alternatives:

- If we have the Firmware downloaded to an EEPROM (not used in the current design), we do not need any Windows Driver, and when connecting the board to the USB port, it should be seen on the device manger as a HID device, if it appears as a HID, it is ready to be used.

- If the Firmware is to be loaded from the host computer (Texas Instruments, 2003-d) -as in the current design- , once the device is connected, Windows recognizes that a new hardware is connected and will ask the user for the Device Driver, this file is called TI Apploader Driver, it is available from Texas Instruments for downloading from their site.

### 4.3.1 The INF File

It is a text file containing information that Windows requires to identify a USB device, this file contains information that tells Windows what Driver to use and what information to store in the registry (Axelson, 2005), this file is created by the TI Apploader Driver during its installation, the INF file can be found in the `\Windows\temp` Directory, the developer needs to change the name of the Firmware binary file, the name of the Firmware file can be changed simply by replacing all the instances of that name in the INF file with the chosen filename, then the Apploader Driver should be reinstalled again.

This file contains much information, the most important is the name of the Firmware binary file that will be downloaded to the chips' RAM, this piece of information is needed by the Driver, the INF file should be copied the `\WINDOWS\INF` directory, the INF file has different names depending on the Windows version, for example:

- On Windows 98SE it is named "TIUPDatr.inf" .

- On Windows 2000, Windows ME and Windows XP it is named "OEMxxx.INF " where the "xxx " is a system-generated incrementing number (Texas Instruments, 2003-b).

As an example the INF file for the current board "TUSB3210.INF" is listed below.

```
; TI Application Firmware Loader Driver INF
;
;
; The .bin file listed under [SourceDisksFiles] and [DriverCopyFiles]
; AND referenced under the [DriverHwAddReg] section (These values
; MUST match) is the Bin file the driver will read and download to
; the device.
;
; To change the name of the file that will be downloaded to the
; device, not only change the value of the string 'FIRMWARE_FILENAME'
; at the bottom of this file, but also in [DriverCopyFiles] and
; [DriverHwAddReg] sections.
;
 [Version]
Signature=$CHICAGO$
Class=USB
LayoutFile=layout.inf
Provider=%MFGNAME%
[Manufacturer]
%MFGNAME%=DeviceList

;----------------------------------------------------------------------------
; Device directory
;----------------------------------------------------------------------------
[DestinationDirs]
DefaultDestDir=10,System32\Drivers
DriverCopyFiles = 10,SYSTEM32\DRIVERS   ; WINDOWS\SYSTEM32\DRIVERS
 [SourceDisksFiles]
ApLoader.sys=1
%FIRMWARE_FILENAME%=1          ; This is the bin file that the driver
                               ; will read and download to the device
[PreCopySection]
HKR,,NoSetupUI,,1
[SourceDisksNames]
1=%INSTDISK%,,,
[DeviceList]
%DESCRIPTION%=DriverInstall,USB\VID_0451&PID_2136

;----------------------------------------------------------------------------
;  Windows 2000 Sections
;----------------------------------------------------------------------------
```

```
[DriverInstall.NT]
CopyFiles=DriverCopyFiles
[DriverCopyFiles]
ApLoader.SYS
TUSB3210.BIN
 [DriverInstall.NT.Services]
AddService=APLOADER,2,DriverService
 [DriverService]
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%10%\system32\drivers\ApLoader.sys
 [DriverInstall.nt.hw]
AddReg=DriverHwAddReg
[DriverHwAddReg]
HKR,,FWFileName,,"TUSB3210.BIN"

;-----------------------------------------------------------------------------
;  Windows 98 Sections
;-----------------------------------------------------------------------------
[DriverInstall]
AddReg=DriverAddReg
CopyFiles=DriverCopyFiles
 [DriverAddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,ApLoader.sys

[DriverInstall.hw]
AddReg=DriverHwAddReg
;-----------------------------------------------------------------------------
;  String Definitions
;-----------------------------------------------------------------------------
[Strings]
MFGNAME="Texas Instruments"
INSTDISK="TI TUSB3210 Application Firware Loader Install Disk"
DESCRIPTION="TI TUSB3210 Application Firmware Loader"
FIRMWARE_FILENAME="TUSB3210"
```

**Notes:-**

1- The name of the Firmware file that should be downloaded is listed under [ SourceDisksFiles ] and [ DriverInstall.NT ] and referenced under [ DriverHwAddReg ] section, the name of the Firmware file under these sections must be the same (TUSB3210.bin) in the current design (Texas Instruments, 2003-b).

2- The INF file is configured to look for a particular “ *.bin ” file in the \WINDOWS\SYSTEM32\DRIVERS directory, this file should be an absolute binary file, if -as in the current design- we use the Keil Compiler to write the Firmware the “ *.bin ” file should be produced by the generated “ *.hex ” file using a special utility called “ hex2bin.exe ”, this file after being produced should be placed in the above mentioned directory (Texas Instruments, 2003-a).

3- The name of the firmware file can be changed simply by replacing all the instances of that name in the INF file with the chosen filename, then reinstall the Apploader Driver again (Texas Instruments, 2003-b).

### 4.3.2 The TI Apploader Driver

This is a Windows USB Device Driver, this driver enables the Firmware code written for the TUSB3210 chip to be downloaded from the host computer to the chips' RAM on power up, when the USB board built around the TUSB3210 is connected to the USB bus, Windows associates it with this driver which downloads the Firmware code to the chip's RAM (Texas Instruments, 2003-b).

To install the Apploader driver, and Setup the device the user should :

1- Download the Apploader Driver from the Texas Instruments web site (www.ti.com) .

2- The installation process should be done before the device (board) is connected to the USB port, if the device is already attached to the USB bus and recognized by Windows as “unknown hardware” the installation process should continue while keeping the device attached to the bus.

3- To install the Apploader Driver just double click on the executable file and follow the instructions of the Installation Wizard.

4- The file named 'TUSB3210.INF' should be copied to the Directory `\Windows\inf` before attaching the device to the USB port on the PC.

5- The files named 'TUSB3210.bin' and 'Aploader.sys[10]' should be copied to the Directory `\Windows\system32\Drivers`.

6- Attach the device to the USB port.

7- Follow the Wizard instructions to Setup the device.

**Comments**:

1- Before installing the Apploader, make sure that the EEPROM -if present- is not connected to the chip pins.

2- To uninstall the driver go to the "add/ remove" programs in the Windows control panel or run the install again (Texas Instruments, 2003-b).

[10] This file is created by the Apploader Driver when installed and can be found in the Directory Windows\temp

*Chapter 5*

# TESTING, DISCUSSION, CONTRIBUTIONS AND FUTURE WORK

In this thesis a peripheral USB Interface Board was built, the Firmware and Host Application software needed to operate the board were developed for a HID class device. Furthermore the USB concepts were briefly introduced and a user manual was produced to help others to build, install and use the board.

The design trajectory of this project was introduced which involves : choosing the suitable controller chip, building the Interface Board, deciding what type of Device Driver should be used to accomplish the communication process between the device connected to the chip's port and the PC, developing the Firmware and the PC software codes, writing the user manual for the board and finally verification and testing of the board and software.

This chapter presents conclusions and future work and highlights the main contributions. This chapter is organized as follows : section 5.1 summarizes the main conclusions of this thesis, section 5.2 introduces Testing the board and codes, section 5.3 introduces the Discussion, section 5.4 presents the main contributions and section 5.5 introduces the proposed future work.

## 5.1 Summary

Chapter 2 introduced an overview of USB concepts, this includes : comparing USB interface to other popular interfaces, terms definition, Host (PC) and peripheral duties, developing USB projects including the development tools and steps, how USB transfers data, USB transfer types, USB descriptors, USB device requests and HID (Human Interface Device) class devices. These concepts constitute the basis to understanding the USB communication protocol which is essential for developing the Firmware and Host application software.

Chapter 3 introduced the USB Interface Board hardware and software, this includes : how to choose the suitable chip for a USB project, a description of the TUSB3210 microcontroller chip which was chosen to be the heart of the Board, why choosing this chip and a detailed

description of the files of both the Firmware and the Host application software developed to operate the board.

Chapter 4 presented the Board's User Manual which includes : the hardware and software required for building and operating the USB interface, building steps of the board and its installation process.

## 5.2 Testing the Board and Codes

To test the board several alternatives exists, using free software tools such as the HidTest, using a protocol analyzer and other tools. the built board and its codes were tested using the HidTest program[11], and passes the tests, the best test for such a board is to use a protocol analyzer, but unfortunately the analyzer is not available, instead LEDs are used to watch the data transfer between the device and the host, this test if succeeded indicates the success of the enumeration processes and the data transfer protocol.

The Board was tested on a simple application which consists of "8" LEDs connected to port one of the microcontroller chip, when the board is connected to the PC USB port, the PC enumerates the board successfully, several bytes were sent from the PC to the board and received successfully on port 1, one byte was sent from port 1 to the host, the byte was received successfully, the board is capable of receiving and transmitting 64 bytes reports, this limitation on the number of data bytes sent and received is just for testing purposes.

The fact that we managed to send / receive data from/to the host implies that the data exchange process between the host and the board is successful; this indicates that the Hardware, Firmware and Host software are functioning as required.

## 5.3 Discussion

The board was built according to a ready made schematic provided by Texas Instruments, the board was built with some modifications on the diagram, these modifications include :

[11] For more details see Chapter 2, Device Testing section

- Changing the tiny surface mounted voltage regulator (TPS76333DBV) with the DIP mounted voltage regulator (278R33), this regulator converts 5 volts dc to 3.3 volts.

- Changing the tiny surface mounted transistors (MMBT4401,MMBT4403) with normal package transistors (2N2222A,2N4403).

- Additional circuit was added for testing the board, this circuit consists of 8 LEDs connected through current limiting resistors to port1's pins, this circuit is used to show the data exchange between the host and the board.

These changes simplifies the soldering and testing of the components on the board, the board's hardware is supposed to be generic in the sense that it allows access to all the GPIO pins of the microcontroller chip, the board's hardware functions exactly as planned.

The Board's schematic diagram is simple and includes a limited number of components, as a result building the board is not tedious, the board's Hardware costs about 30$, this is a low cost compared to the ready made board's cost of 199$.

The board's hardware could not be evaluated without the Firmware and the PC software codes, these codes were developed simultaneously, in order to develop the codes, the developer should have expertise in programming and software tools to enable him to write, compile, run and debug the codes, also codes samples could be of great help if available, the Firmware was developed using the Keil C51 package and Delphi 7 package was used to develop the PC software.

The developed Firmware code is generic and could be used by other developers to develop their own applications, this adds to the generic property of the hardware, on the other hand, the PC software is supposed to be generic also, it is somehow generic because it allows data to be exchanged between the PC and the board, but the user could not write to or read from the port he chooses, instead he can access only port "1"of the microcontroller, to enable the user to access any port ,Dynamic Link Libraries (DLLs) should be included in the software, unfortunately DLLs were not included due to lack of time, despite this fact, the board –as it is- can be used to develop several applications such as Barcode Reader, Flash Memory

Reader and General Purpose Controller, to implement these applications the developer should change the contents of some files as explained in section 5.5 .

Because the human Interface Device (HID) Driver included with Windows was used as the Device Driver of the PC software, data transfer rate is limited to a maximum of 64 Kbytes per second , this rate is relatively low compared to the Full speed transfer rate of 12 Mbits per second, this limitation is justified because we avoid the processes of writing a special Device Driver which is time consuming and requires big investment in tools, despite this limitation, the board is suitable to communicate with several HID applications.

The board's developed Firmware allow any USB HID enabled operating system to directly access the board, this includes Windows 98 and later Windows versions, MacOS, Unix/Linux, so I expect that when the board is attached to the USB port of any of these systems, it will be enumerated, when it comes to the way the user can access the board from the PC side, every operation system has its own API (Application Programmer's Interface) functions, in the current application, the host application software uses Win32 API functions, these functions can access devices running under Windows, so I expect that the board with the current application can't run under operating systems other than Windows, instead each operating system should have its own PC host application in order to access the board, the board was tested under Windows only.

## 5.4 Main Contributions

The ultimate objectives of this thesis are to build a Full speed USB Input/Output Interface Board, to develop the software codes needed to operate the board and to provide a user manual for the board, the board is assumed to be generic in away that enables users to access any of the board's ports.

The main contributions of this thesis can be summarized as follows :

- The design trajectory was presented and was followed in the next stages.

- Several decisions were taken concerning the suitable microcontroller chip, the Device Driver and the development tools needed to build and operate the Board, these decisions were based on obtaining the simplest, cheapest and efficient design.

- The Interface Board was built according to a schematic diagram provided by the chip's vendor.

- The Firmware code and the Host application software needed to operate the Board were developed for a HID class device using the suitable tools.

- A user manual describing how, install and operate the Board was produced.

- The Board and the codes were verified and tested in accordance with HID specifications.

Unfortunately the USB Full speed (12 Mbps) did not be achieved because the HID device driver was used which limits the transfer rate to a maximum of 64 KBytes per second, also the user can access only port one of the board because DLLs were not included in the developed codes due to lack of time, despite these limitations a HID board has been built, the Firmware and PC software were developed and a user manual was provided.

## 5.5 Future Work

The built Board and the developed codes are supposed to be generic, this means that other developers may use the board and the codes to develop several applications, the codes may or may not need some modifications depending on the specific application design, these modifications should include :

- The PC software : the data exchange procedures should be changed, these functions include : *WriteBufferToDevice, SendDataPacketToDevice* and *HidData* .

- The Firmware : All the protocol functions should not be changed, only the data exchange functions should be changed, these functions are included in the following files : *Prog.c, Application.c* and the corresponding header files, also the *report*

*descriptor* in the *descriptor.h* file has to be changed to cope with the new application data format.

I hope some body will develop the codes to include DLLs, this will enable users to access all the board's ports and narrow the gap between this product and the available similar commercial products.

Writing a custom Device Driver which enables using any of the four transfer types (instead of control and interrupt transfers in the HID device which limits the transfer rate) will enhance the board capabilities by obtaining almost the full speed transfer rate (12 Megabits/sec), this requires modifying the codes in order to cope with the new driver requirements.

The idea of building the board and developing the Firmware and host application codes can be followed to design a board and codes for the high-speed microcontroller chip TUSB6250 from Texas Instruments, this process requires changing some header files like *tusb3210.h*, *usb.h* and *descriptor.h* and some other minor modifications according to data sheets, using the HID Device Driver with the TUSB6250 gives a transfer rate of 24.576 Megabytes/sec, this rate is suitable for applications like the e-learning.

# REFERENCES


Axelson, J., (2005) : USB Complete, Third edition, Lakeview Research LLC, USA.

Borland Software Corporation (2002): Borland Delphi7 Enterprise.
(http:// www.borland.com).

Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, NEC Corporation (1998): Universal Serial Bus Specification, Release 1.1.
(http:// www.USB.org).

Keil Electronik GmbH/Keil Software, Inc. (2003): uvision2, Version 2.4.
(http:// www.Keil.com).

Marquerdt R. (2004): HID Component, Version 1.1, project JEDI.
(http:// www.delphi-jedi.org).

Peacock, C. (2002) : USB in a nutshell, third release, beyondlogic organization.
(http:// www.beyondlogic.org).

Texas Instruments (2001-a): TUSB2130 Boot Code Document for USB to General Purpose Device Controller (SLLU025A).
(http:// www.TI.com).

Texas Instruments (2001-b): TUSB3210 Generic Evaluation Board (SLLU031).
(http:// www.TI.com).

Texas Instruments (2003-a): TI Application Firmware Loader Driver INF.
(http:// www.TI.com).

Texas Instruments (2003-b): TI Apploader Driver (SLLC 160).
(http:// www.TI.com).

Texas Instruments (2003-c): TUSB3210 Universal Serial Bus General Purpose Device Controller (SLLS 466B).
(http:// www.TI.com).

Texas Instruments (2003-d): VIDs, PIDs, and Firmware (Design Decisions when Using TI USB Device Controllers) – SLLA 154 .
(http:// www.TI.com).

Texas Instruments (2004): TUSB2136/TUSB3210/TUSB5052 USB Firmware Programming Flow 8052 Embedded (SLLU020A), Revision A .
(http:// www.TI.com).

Universal Serial Bus (2001): Device Class Specification for Human Interface Devices (HID), Firmware specification, Version (1.11).
(http:// www.USB.org).

Vault Information Services LLC (2004) : The 8052 Tutorial & Reference, Version (1.0).
(http:// www.8052.com).

Texas Instruments (2000) : TUSB2136 Generic Keyboard Demo Program, USA.
(http://www.TI.com).

SCO Group Inc. (2005) : Developing Device Driver, USA.
(http:// www.sco.com).

# APPENDIX A

## USB Transfer Types

### A.1 Control Transfers.

Control transfers are typically used to:-

a) Carry the host requests needed for configuration of the device to the device so that the host enumerates the device.

b) Transfer small amounts of data.

All USB devices must support control transfers, this is done over the default pipe at Endpoint 0, control transfers are initiated by the host and can have up to three stages:

- The setup stage : It consists of three packets:-

1. The setup token packet which contains the address and the endpoint number.

2. The data packet which always has a PID type of DATA 0, and it includes a setup packet which tells the request type.

3. The handshake packet is the last packet to send, this packet is used to report the successful or failure of the transaction, if the device receives successfully the setup data it returns ACK otherwise the data is ignored and no handshake packet is send as shown in figure (A.1) (Peacock, 2002) .

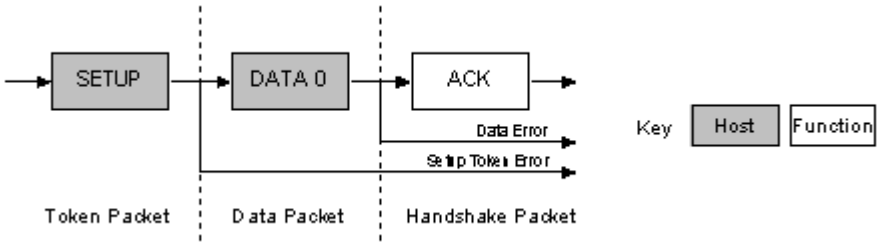



Figure A.1: The setup stage (Peacock, 2002).

- The Data Stage : This stage is optional, it consists of one or more IN or OUT transactions, the setup request contains the size of data to be transmitted, if the size of data exceeds the maximum packet size, another packet is transmitted which has the maximum packet size, until the end of data, the last packet may or may not has the maximum packet size, depending on the direction of data transfer the data stage has two alternatives :

A- **IN** : the host issues an IN token, when the device receives the IN token:-

1. It ignores the packet if there is an error in the IN token.

2. If the IN token was received correctly , it replay with a DATA packet containing the control data to be sent or a STALL packet indicating that the endpoint is not functioning or a NAK packet indicating that the endpoint has no data to send at this time. Figure (A.2) shows details (Peacock, 2002).

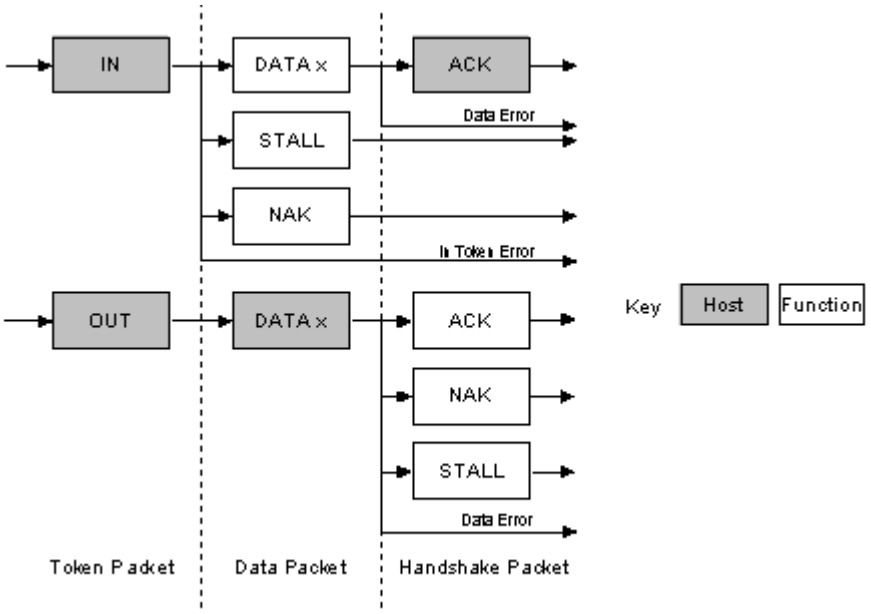



Figure A.2: IN and OUT token packets in a data stage (Peacock, 2002).

B- **OUT**: when the host sends a control data packet to the device, it sends an out token , then a data packet containing the control data as in figure (A.2) above.

1. If the OUT token or the data packet is not received correctly, the device ignores the packet.

2. If the device endpoint has received the data it issues an ACK informing the host that the transfer process was successful.

3. If the device endpoint is not ready to receive the data, the device issues a NAK.

4. If the endpoint is not functioning, the device returns a STALL.

- The status stage: In this stage the device issues a signal indicating the status of the overall request, this depends on the direction of the transfer :

A- **IN**: If the host has sent an IN token during the data stage, it should acknowledge the successful reception of data, this is done by sending an OUT token and then a zero length data packet, if the device issues an ACK, this means that the device has completed the last job and is now ready to do the next job, if an error occurs, the device issues a STALL, however if the device is still processing data, it issues a NAK informing the host to repeat the status stage later , this is shown in figure (A.3) (Peacock, 2002).

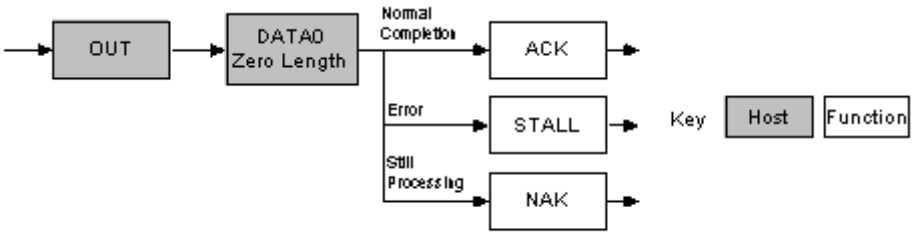



Figure A.3: The status stage, OUT token (Peacock, 2002).

B- **OUT**: if the host has sent an OUT token during the data stage.

1. If the data was received correctly, the device sends a zero length packet in response to an IN token.

2. If an error occurs, the device sends a STALL or a NAK if it is processing data, informing the host to repeat the status stage later. Figure (A.4) shows this case.

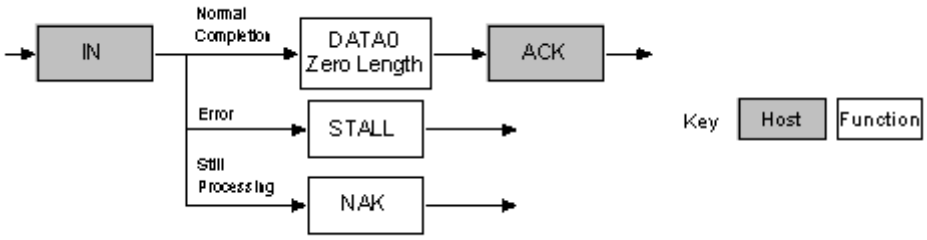



Figure A.4: The status stage, IN token (Peacock, 2002).

**Data size:**

The maximum data packet size for a low speed device is 8 bytes, for a full speed device the maximum may be 8, 16, 32, or 64 bytes, all data packets except the last one must have the maximum packet size, the host reads the maximum packet size from the device descriptor during enumeration.

**Speed:**

It is the amount of data that each transfer type can move, it depends on the speed of the device. Table (A.1) shows a comparison between the three speeds.

Table A.1: The maximum data transfer rate as related to transfer type and bus speed (Axelson, 2005).

| **Transfer Type** | **Maximum data-transfer rate per endpoint (kilobytes/second with data payload/transfer = maximum packet size for the speed)** | | |
|---|---|---|---|
| | **Low Speed** | **Full Speed** | **High Speed** |
| Control | 24 | 832 | 15,872 |
| Interrupt | 0.8 | 64 | 24,576 |
| Bulk | Not allowed | 1216 | 53,248 |
| isochronous | | 1023 | 24,576 |

## A.2 Interrupt Transfers

Interrupt transfers are useful when data has to be transferred within a specific amount of time; typical applications are keyboards, joysticks and mice. Interrupt transfers are popular because Windows includes drivers that enable applications to use interrupt transfers with devices that conform to the HID specification, usually interrupts are device generated, under USB if a device requires a service from the host, it must wait until the host polls it asking for data before it can report that it needs urgent attention, this type of transfer has two stages:

- Interrupt IN stage:

  In this stage the host polls periodically the interrupt endpoint, the endpoint descriptor contains the polling rate, each poll requires the host to send an IN token, if the IN token is not received correctly, the device ignores the packet and wait for new tokens.

Figure (A.5 upper part) shows the format of an IN transaction.

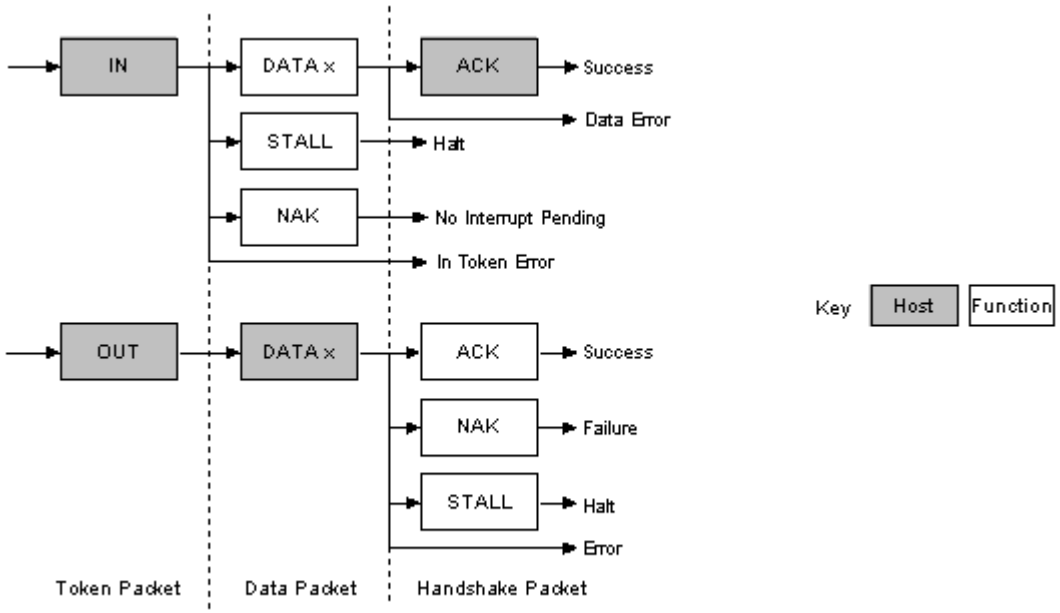



Figure A.5: Interrupt transfer, IN and OUT transactions format (Peacock, 2002).

- Interrupt OUT stage:

  In this stage the host issues an OUT token to send the interrupt data to the device, following the OUT token it sends a data packet containing the interrupt data, the device ignores the data if the OUT token or the data packet is corrupted.

  1. The device issues ACK if the endpoint received the data successfully.

  2. The device returns an NAK if the endpoint is busy in processing a previous packet.

  3. The device return a STALL if the endpoint is not functioning.

Fig (A.5 lower part) above shows the format of an OUT transaction

**Data size:**

For low-speed devices the maximum packet size ranges from 1 to 8 bytes, for full speed devices the maximum size ranges from 1 to 64 bytes, while for high speed device, it ranges from 1 to 1024 bytes.

**Speed:**

For low speed transfers, the transfer rate is 800 bytes per second, for full-speed transfers , the rate is 64-kilo bytes per second, while for high-speed transfers; it is 24.576 Megabytes per second.

## A.3 Isochronous Transfers

In this type of transfer, transfers occur continuously and periodically, data contained within the packet is typically time sensitive information, such as an audio or video stream, in isochronous transfers, there is no provision for retransmitting data received with errors.

When using isochronous transfers the following are provided (Peacock, 2002):

1. Guaranteed access to USB bandwidth.
2. Bounded latency.
3. Unidirectional stream pipe.
4. Error detection via CRC, but no retry or guarantee of delivery.
5. Used with full and high speeds only.
6. No data toggling.

Figure (A.6) below (Peacock, 2002), shows the format of an isochronous IN and OUT transactions. Since isochronous transfers can't retransmit data received with errors, there is no need for a handshake stage or error reporting.

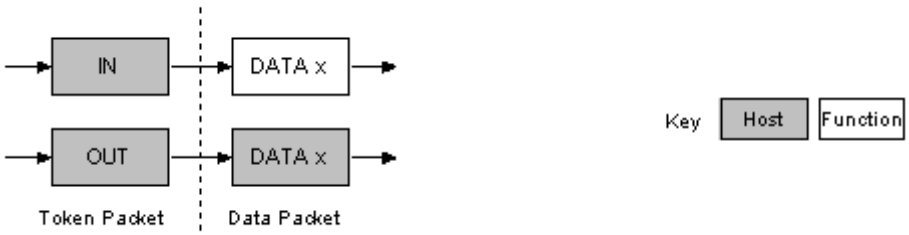



Figure A.6: Isochronous transfers, IN and OUT formats (Peacock, 2002)

**Data Size:**

For full speed endpoints, the maximum packet size ranges from 0 to 1023 bytes, for high speed endpoints, the maximum packet size could be 1024 bytes.

**Speed:**

A full speed transaction can transfer up to 1.023 Megabytes per second, while a high speed transaction has a transfer rate of 24.576 Megabytes per second.

## A.4 Bulk Transfers

Bulk transfers can send large amounts of data when time is not critical, typical uses include, sending data from host to printer and reading and writing to a disk. Bulk transfers provide error correction in the form of a CRC16 field and error detection and re-transition ensuring data is transmitted and received correctly and without errors, if the USB bus is busy while trying to send bulk data, the data may slowly and gradually go over the bus, this means that bulk transfers should only be used for time insensitive communication, because there is no guarantee of latency.

In particular bulk transfers have the following properties (Peacock, 2002):-

1. Can be used to transfer large amounts of data.
2. Provide error correction via CRC with guarantee of delivery.
3. No guarantee of bandwidth or minimum latency.
4. Used with full and high speed transfers only.

Figure (A.7) below shows the format of a bulk IN and OUT transaction.

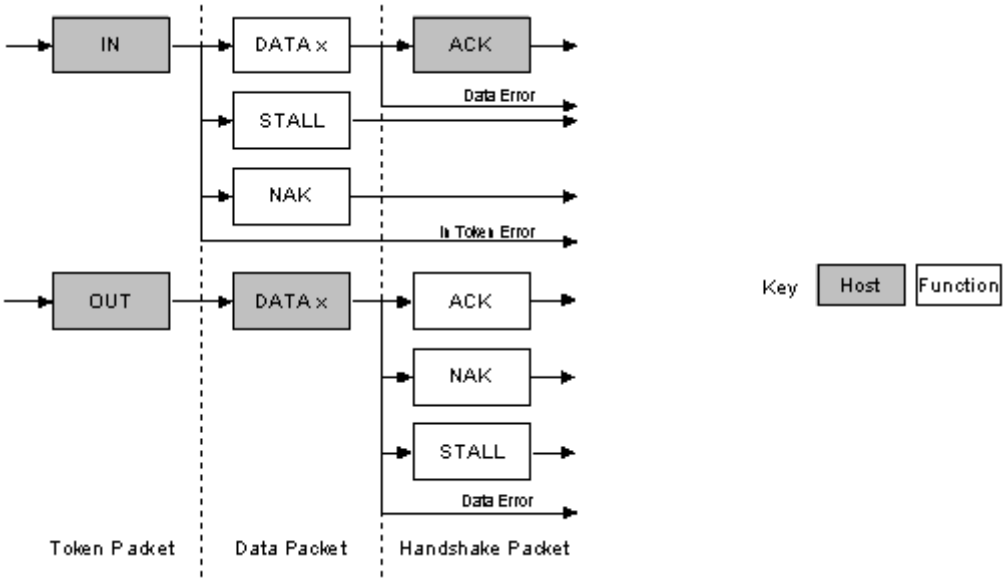



Figure A.7: Bulk transaction, IN and OUT formats (Peacock, 2002).

A- **IN**: The host issues an IN token indicating that it is ready to receive bulk data.

1. If the device receives the Token data with an error, it ignores the packet.

2. If the data was received correctly, the device sends an ACK, or a STALL packet indicating an error in the endpoint or a NAK packet indicating that the endpoint is busy in processing a previous packet.

B- **OUT**: When the host wants to send a bulk data packet to the device, it issues an OUT token and a data packet containing the bulk data immediately after the token:

1. If the OUT token or the data packet is not received correctly the device ignores the packet.

2. If the token was received correctly, the device sends a DATA packet containing the bulk data, or a STALL packet indicating that the endpoint is not functioning, or a NAK packet informing the host that the endpoint is functioning, but has no data to send at this time.

**Data size:**

A full speed bulk transfer has a maximum packet size of 8, 16, and 32 or 64 bytes, while for high speed, the maximum packet size is 512 bytes, the host reads the maximum packet size from the device's descriptor during the enumeration process.

**Speed:**

For full-speed bulk transfers the transfer rate is 1.216 Megabytes per second, while high-speed transfers have a transfer rate of 53.248 Megabytes per second.

# APPENDIX B

## B.1 USB Descriptors

### B.1.1 Device Descriptor

The device descriptor is the first descriptor the host reads immediately after the attachment of the device, a USB device can have one device descriptor, this descriptor presents the entire device, it includes the information needed by the host such as the supported USB version, maximum packet size, Vender and product IDs and the number of possible configurations the device can have .Table (B.1) shows the format of the device descriptor (Compaq, Intel, Microsoft, NEC, 1998).

Table B.1: Format of the device descriptor (Compaq, Intel, Microsoft, NEC, 1998).

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of the Descriptor in Bytes (18 bytes) |
| 1 | bDescriptorType | 1 | Constant | Device Descriptor (0x01) |
| 2 | bcdUSB | 2 | BCD | USB Specification Number which device complies to. |
| 4 | bDeviceClass | 1 | Class | Class Code (Assigned by USB Org) If equal to Zero, each interface specifies its own class code, If equal to 0xFF, the class code is vendor specified. Otherwise field is valid Class Code. |
| 5 | bDeviceSubClass | 1 | Subclass | Subclass Code (Assigned by USB Org) |
| 6 | bDeviceProtocol | 1 | Protocol | Protocol Code (Assigned by USB Org) |
| 7 | bMaxPacketSize | 1 | Number | Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64 |
| 8 | idVendor | 2 | ID | Vendor ID (Assigned by USB Org) |
| 10 | idProduct | 2 | ID | Product ID (Assigned by Manufacturer) |
| 12 | bcdDevice | 2 | BCD | Device Release Number |
| 14 | iManufacturer | 1 | Index | Index of Manufacturer String Descriptor |
| 15 | iProduct | 1 | Index | Index of Product String Descriptor |
| 16 | iSerialNumber | 1 | Index | Index of Serial Number String Descriptor |
| 17 | bNumConfigurations | 1 | Integer | Number of Possible Configurations |

- bcdUSB, the highest version of USB the device supports, the value is in binary – coded – decimal (BCD) format, the format of this field is 0xJJMN where JJ is the major version number, M is the minor and N is the sub minor, example: USB 1.1 is represented as 0x0110.

- bDeviceClass, bDeviceSubClass and bDeviceProtocol: used by the operating system to find a suitable device driver for the attached device, if we set bDeviceClass to be (0x00), this means that one device supports multiple classes.

- bMaxPacketSize : this field gives the maximum packet size for endpoint zero which all devices should support.

- idVender : the device descriptor for every commercial USB product must have a vender ID, usually this value should be written in the host INF file and if so, Windows uses this value to find a suitable devise driver for the device.

- idProduct : same as idVender above.

- bcdDevice : used to provide the device version number assigned by the developer, this field has the same format as the bcdUSB field.

- iManufacturer : this is an index which points to a string describing the manufacture, if unused; it should be set to zero.

- iProduct : an index pointing, to a string describing the product, if unused, it should be set to zero.

- iSerialNumber : an index pointing to string describing the serial number of the product, if unused, should be set to zero.

- bNumConfigurations : the number of configurations the device supports.

### B.1.2 Configuration Descriptor

Each USB device can have at least one configuration descriptor, the descriptor describes the device's features and abilities, usually one configuration is enough, but some devices support multiple configurations for multiple uses, the configuration descriptor contains information on the following :

- How the device is powered (bus or mains powered).
- The maximum power consumption of the device.
- The number of interfaces.

After receiving the device descriptor, the host issues requests to receive the device's configuration, the interface and endpoint descriptors, then the host issues a SetConfiguration command with a value that matches the bConfiguration value of one configuration, by doing so the host selects the desired configuration. Table (B.2) below shows the fields of the configuration descriptor (Compaq, Intel, Microsoft, NEC, 1998).

Table B.2: Configuration descriptor fields (Compaq, Intel, Microsoft, NEC, 1998).

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of Descriptor in Bytes |
| 1 | bDescriptorType | 1 | Constant | Configuration Descriptor (0x02) |
| 2 | wTotalLength | 2 | Number | Total length in bytes of data returned |
| 4 | bNumInterfaces | 1 | Number | Number of Interfaces |
| 5 | bConfigurationValue | 1 | Number | Value to use as an argument to select this configuration |
| 6 | iConfiguration | 1 | Index | Index of String Descriptor describing this configuration |
| 7 | bmAttributes | 1 | Bitmap | D7 Reserved, set to 1. (USB 1.0 Bus Powered)<br><br>D6 Self Powered<br><br>D5 Remote Wakeup<br><br>D4...0 Reserved, set to 0. |
| 8 | bMaxPower | 1 | mA | Maximum Power Consumption in 2mA units |

- bLength : the length of the descriptor in bytes.
- bDescriptorType : the constant Configuration (0x02).
- wTotalLength : the total length of data (in bytes) that the device returns.
- bNumberInterfaces : gives the number of interfaces the configuration supports, the minimum number is “1”.
- bConfigurationValue : used by the SetConfiguration request to identify the configuration.
- iConfiguration : this is an optional field, it is an index to a string descriptor that describes the configuration.
- bmAttributes : this field contains information about the way the device is powered. If bit 6=1 then the device is self-powered, if bit 5=1, the device supports remote wakeup feature, bit 6=0 the device is bus- powered in USB 1.1 and higher, bits (0-4) = 0, bit 7=1.
- bMaxPower : this field specifies how much current a device requires, the value of max power is equal to half the number of milli-amperes the device requires , for example : if the device needs 200 milli-amperes then bMaxPower = 100, the maximum power the device can drain from the bus should not exceed 500 mA according to specifications.

### B.1.3 Interface Descriptor

Interface is a set of Endpoints used by a device feature or function, the configuration interface descriptor contains information about the endpoints supported by the interface, each interface has an interface descriptor and a secondary endpoint descriptor for each endpoint supported by the interface. The interface descriptor conforms to the format shown in table (B.3) below (Peacock, 2002).

Table B.3: Format of the interface descriptor (Peacock, 2002).

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of Descriptor in Bytes (9 Bytes) |
| 1 | bDescriptorType | 1 | Constant | Interface Descriptor (0x04) |
| 2 | bInterfaceNumber | 1 | Number | Number of Interface |
| 3 | bAlternateSetting | 1 | Number | Value used to select alternative setting |
| 4 | bNumEndpoints | 1 | Number | Number of Endpoints used for this interface |
| 5 | bInterfaceClass | 1 | Class | Class Code (Assigned by USB Org) |
| 6 | bInterfaceSubClass | 1 | SubClass | Subclass Code (Assigned by USB Org) |
| 7 | bInterfaceProtocol | 1 | Protocol | Protocol Code (Assigned by USB Org) |
| 8 | iInterface | 1 | Index | Index of String Descriptor Describing this interface |

- bLength : the length of the descriptor (in bytes).
- bDescriptorType : the constant Interface (0x04).
- bInterfaceNumber : this field is an index which identifies the interface, each interface must have a descriptor with unique bInterfaceNumber, the default value for this field is zero.
- bAlternateSetting : the field could be used to specify alternative interfaces, the default value of this field is zero.
- bNumberEndpoints : the number of endpoints supported by the interface excluding Endpoint zero. For devices that support endpoint zero only, the value of this field is zero.
- bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol : these fields are used to specify supported classes such as HID and mass storage, this facility eliminates the need to write specific device drivers for the device and instead use class drivers.

- iInterface : this field contains an index to a string that describes the interface.

### B.1.4 Endpoint Descriptors

Every endpoint mentioned in the interface descriptor has its endpoint descriptor except endpoint zero, which has no descriptor and every device must support it since it is a control endpoint and is configured before requesting any descriptors, the information contained in the endpoint descriptors are used by the host to determine the bandwidth requirements of the device (Peacock, 2002) .

The endpoint descriptor conforms to the format shown in table (B.4) below.

Table B.4: Format of the endpoint descriptor (Peacock, 2002).

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of Descriptor in Bytes (7 bytes) |
| 1 | bDescriptorType | 1 | Constant | Endpoint Descriptor (0x05) |
| 2 | bEndpointAddress | 1 | Endpoint | Endpoint Address<br>Bits 0...3b Endpoint Number.<br>Bits 4...6b Reserved. Set to Zero<br>Bit 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoint) |
| 3 | bmAttributes | 1 | Bitmap | Bits 0..1 Transfer Type<br><br>00 = Control<br>01 = Isochronous<br>10 = Bulk<br>11 = Interrupt<br><br>Bits 2..7 are reserved. If Isochronous endpoint,<br>Bits 3..2 = Synchronisation Type (Iso Mode)<br><br>00 = No Synchonisation<br>01 = Asynchronous<br>10 = Adaptive<br>11 = Synchronous |

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| | | | | Bits 5..4 = Usage Type (Iso Mode)<br><br>00 = Data Endpoint<br>01 = Feedback Endpoint<br>10 = Explicit Feedback Data Endpoint<br>11 = Reserved |
| 4 | wMaxPacketSize | 2 | Number | Maximum Packet Size this endpoint is capable of sending or receiving |
| 6 | bInterval | 1 | Number | Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Isochronous must equal 1 and field may range from 1 to 255 for interrupt endpoints. |

- bEndpointAddress : this field indicates the endpoint number and direction.
- bmAttribute : this field specifies the type of transfer the endpoint supports.
- wMaxPacketSize : this field indicates the maximum number of data bytes the endpoint can transfer in a transaction.
- bInterval : this field is used to specify the polling interval of certain transfers.

### B.1.5 String Descriptors

String descriptors are optional, they contain descriptive text such as : the device manufacture, product and serial number, if the developer did not use these descriptors, he should set to zero any string index fields of the descriptors, this means that string descriptors are not used.

Table (B.5) (Peacock, 2002) shows the format of String Descriptor Zero, this descriptor is read by the host to determine what languages are available, if a language is supported, it is referenced by sending the language ID in the wIndex field of a Get-Descriptor (string) request (Peacock, 2002).

Table B.5: The format of string descriptor zero (Peacock, 2002) .

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of Descriptor in Bytes |
| 1 | bDescriptorType | 1 | Constant | String Descriptor (0x03) |
| 2 | wLANGID[0] | 2 | number | Supported Language Code Zero<br>(e.g. 0x0409 English - United States) |
| 4 | wLANGID[1] | 2 | number | Supported Language Code One<br>(e.g. 0x0c09 English - Australian) |
| n | wLANGID[x] | 2 | number | Supported Language Code x<br>(e.g. 0x0407 German - Standard) |

All subsequent strings conform to the format shown in table (B.6) below.

Table B.6: Format of all subsequent strings (Peacock, 2002).

| Offset | Field | Size | Value | Description |
|---|---|---|---|---|
| 0 | bLength | 1 | Number | Size of Descriptor in Bytes |
| 1 | bDescriptorType | 1 | Constant | String Descriptor (0x03) |
| 2 | bString | n | Unicode | Unicode Encoded String |

## B.2 USB standard requests

There are three requests :

### 1. Device Requests

All devices must respond to standard requests, table (B.7) below summarizes USB 1.1 standard device requests (Peacock, 2002,Compaq, Intel, Microsoft and NEC, 1998).

Table B.7: USB 1.1 standard device requests (Peacock, 2002,Compaq, Intel, Microsoft and NEC, 1998).

| bmRequestType | bRequest | wValue | wIndex | wLength (Byte) | Data |
|---|---|---|---|---|---|
| 1000 0000b | GET_STATUS (0x00) | Zero | Zero | Two | Device Status |
| 0000 0000b | CLEAR_FEATURE (0x01) | Feature Selector | Zero | Zero | None |
| 0000 0000b | SET_FEATURE (0x03) | Feature Selector | Zero | Zero | None |
| 0000 0000b | SET_ADDRESS (0x05) | Device Address | Zero | Zero | None |
| 1000 0000b | GET_DESCRIPTOR (0x06) | Descriptor Type & Index | Zero or Language ID | Descriptor Length | Descriptor |
| 0000 0000b | SET_DESCRIPTOR (0x07) | Descriptor Type & Index | Zero or Language ID | Descriptor Length | Descriptor |
| 1000 0000b | GET_CONFIGURATION (0x08) | Zero | Zero | 1 | Configuration Value |
| 0000 0000b | SET_CONFIGURATION (0x09) | Configuration Value | Zero | Zero | None |

- Get Status : this request is directed to the device, the host requests information about the device, the device responds by sending a data packet, the information needed are contained in D0 and D1 of the 16 bits packet, the packet has the format shown below in Figure (B.1)(Peacock, 2002,Compaq, Intel, Microsoft and NEC, 1998).

| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | | Remote Wakeup | Self Powered |

Figure B.1: Data packet format sent by the device in response to a Get Status request (Peacock, 2002,Compaq, Intel, Microsoft and NEC, 1998).

- Bit ”0” (Do) is the Self-Powered field : 0 = bus-powered, 1 = self powered.

- Bit “1” (D1) is the Remote Wakeup field : default on reset is “0” (disabled) .

- All other bits are reserved.

For endpoint requests, only bit “0” is defined:-

- Bit 0 = 1 indicates a HALT condition.

- Set Feature : using this field the host requests to enable a feature on a device, interface or endpoint, the USB specification defines two features:

  - DEVICE-REOMOTE-WAKEUP : when set by the host, a suspended device signals the host to resume communications.

  - ENDPOINT – HALT : with a value of zero.

- Clear Feature : using this field the host requests to disable a feature on a device, interface or endpoint, the USB specification defines two features.

  - DEVICE-REMOTE-WAKEUP : with a value of “1”, applies to device.

  - ENDPOINT-HALT : with a value of zero applies to endpoint.

- Set Address : this field is used during the enumeration process, it is used to give a unique address to the device, the address is specified in the wValue field. 'this request is unlike most other requests because the device doesn’t carry out the request until it has completed the status stage of the request by sending a 0-length data packet' (Axelson, 2005), before this the device is assigned address 0, after the completion of Set Address, it will be assigned an address other than “0”.

- Set Descriptor : this field is used by the host to add a descriptor or update an existing descriptor, this request enables the host to add descriptors different from those in the Firmware, or to change existing descriptors.

- Get Descriptor : this field is used by the host to request a specific descriptor, when the host sends a request for a configuration descriptor -for example- , the device

responds by returning the configuration descriptor and all interface descriptors for that configuration and all endpoint descriptors for the interface.

- Set Configuration : this field is used to set the device configuration, at the end of a Set Configuration request, the device enters the configuration state.

- Get Configuration : this field enables the host to request the value of the current device configuration.

**2. Interface Requests**

According to USB specification, there are five standard interface requests, table (B.8) below shows details.

Table B.8: Standard interface requests (Peacock, 2002) .

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 1000 0001b | GET_STATUS (0x00) | Zero | Interface | Two | Interface Status |
| 0000 0001b | CLEAR_FEATURE (0x01) | Feature Selector | Interface | Zero | None |
| 0000 0001b | SET_FEATURE (0x03) | Feature Selector | Interface | Zero | None |
| 1000 0001b | GET_INTERFACE (0x0A) | Zero | Interface | One | Alternate Interface |
| 0000 0001b | SET_INTERFACE (0x0B) | Alternative Setting | Interface | Zero | None |

- wIndex : is used to specify the referring interface for requests directed to the interface, the format of this field is shown below in figure (B.2) (Peacock, 2002), this field is used by the host.

| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | Interface Number | | | | | | | |

Figure B. 2: The format of the wIndex Field used by the host (Peacock, 2002).

- Get Status : is used to return the status of the interface.

- Clear Feature : used to request to disable an interface feature, according to specification, we have no interface features.

- Set Feature : this field is used by the host to enable an interface feature .

- Get Interface : devices with configurations that support multiple settings for an interface are requested by the host to send the current setting .

- Set Interface : devices with configurations that support multiple settings for an interface are requested by the host to use a specific setting.

**3. Endpoint requests**

The details of these requests (Peacock, 2002) are shown in table (B.9).

Table B.9: Details of standard endpoint requests (Peacock, 2002).

| bmRequestType | bRequest | wValue | wIndex | wLength | Data |
|---|---|---|---|---|---|
| 1000 0010b | GET_STATUS (0x00) | Zero | Endpoint | Two | Endpoint Status |
| 0000 0010b | CLEAR_FEATURE (0x01) | Feature Selector | Endpoint | Zero | None |
| 0000 0010b | SET_FEATURE (0x03) | Feature Selector | Endpoint | Zero | None |
| 1000 0010b | SYNCH_FRAME (0x12) | Zero | Endpoint | Two | FrameNumber |

- wIndex : is used to specify the referring endpoint and direction for requests directed to an endpoint, the format of this field is as shown below in figure (B.3) (Peacock, 2002).

| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | Dir | Reserved | | | Endpoint Number | | | |

Figure B. 3: The format of the wIndex field (Peacock, 2002).

- Get Status : the host requests the status of an endpoint , the contents of this field are two bytes indicating the endpoint status (HALT/STALL) , the format of these two bytes are shown below in figure (B.4) (Peacock, 2002).

| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Reserved | | | | | | | | | | | | | | | Halt |

Figure B. 4: The format of the get status field (Peacock, 2002).

- Clear Feature : the host requests to disable an endpoint feature, the USB specification defines one feature applies to endpoints, which is ENDPOINT-HALT with a value of zero.

- Set Feature : the endpoint is requested to set or enable a feature by the host, USB specification defines one feature applies to endpoints, which is ENDPOINT-HALT with a value of zero.

- Synch Frame : the device reports an endpoint synchronization frame .

# APPENDIX C

## C.1 HIDs Descriptors

A HID class device uses the following standard USB descriptors.

- Device.
- Configuration.
- Interface.
- Endpoint.

The interface descriptor is of special importance in writing Firmware for a HID device since it is the descriptor where the device is defined as a HID, this is done by setting the class – code byte in the interface descriptor to "3" to define the device as a HID.

The HID class devices use the following Class-Specific descriptors in addition to the standard descriptors:

- HID.
- Report.
- Physical.

### C.1.1 HID Class Descriptor.

The main purpose of this descriptor is to identify the length of the descriptor and additional descriptors to be used in HID communications, this descriptor has at least seven or more fields, table (C.1) shows details (Universal Serial Bus, 2001).

Table C.1: Fields of a HID class descriptor (Universal Serial Bus, 2001).

| **Offset (decimal)** | **Field** | **Size (bytes)** | **description** |
|---|---|---|---|
| 0 | bLenght | 1 | Descriptor size in bytes |
| 1 | bDescriptorType | 1 | 21h indicates the HID class |
| 2 | bcdHID | 2 | HID specification release number (BCD) |
| 4 | bCountryCode | 1 | Numeric expression identifying the country for localized hardware (BCD) |
| 5 | bNumDescriptors | 1 | Number of subordinate class descriptors supported |
| 6 | bDescriptorType | 1 | The type of class descriptor |
| 7 | wDescriptorLength | 2 | Total length of report descriptor |
| 9 | bDescriptorType | 1 | Constant identifying the type of descriptor. Optional, for devices with more than one descriptor. |
| 10 | wDescriptorLength | 2 | Total length of descriptor. Optional, for devices with more than one descriptor. May be followed by additional bDescriptorType and wDescriptorLength fields. |

- bLength : descriptor length in bytes.
- bDescriptorType : constant (0x21) specifying the HID class.
- bcdHID : a binary coded decimal numeric expression identifying the HID version, example : version 1.1 is (0110 h).
- bCountryCode : a code identifying the country if the device is localized, if not the value in this field is (00 h).
- bNumDescriptors : a number identifying the number of class descriptors, this value should be at least 1 (report descriptor).
- bDescriptorType : report or physical descriptor, a HID must support at least one report descriptor.
- wDescriptorLength : length of descriptor specified in the previous field.

- bDescriptorType, wDescriptorLength : these fields identify the type and the length of any additional descriptors.

### C.1.2 Report descriptors

A report descriptor is made up of items that provide information about the device uses, it defines the format of the data that performs the device tasks, for example, if the device is a relay controller, the data consist of codes that determine which relay or relays to open or close, the report descriptor must determine in advance the size and contents of a HID report, the report descriptor length vary from device to device, the host retrieves the descriptor by a Get-Descriptor request. Listing (C.1) shows on example of a report descriptor, in this descriptor the input and output reports are described (Axelson, 2005).

Listing C.1 : An example of a report descriptor.

```
hid_report_desc_table :
    db 06h, A0h, FFh        ;       Usage Page (vendor defined)
    db 09h, A5h             ;       Usage (vendor defined)

    db A1h, 01h             ;       Collection (Application)
    db 09h, A6h             ;       Usage (vendor defined)

; the input report
    db 09h, A7h             ;       Usage (vendor defined)
    db 15h, 80h             ;       Logical Minimum (-128)
    db 25h, 7Fh             ;       Logical Maximum (127)
    db 75h, 08h             ;       Report Size (8) (bits)
    db 95h, 02h             ;       Report Count (2) (fields)
    db 81h, 02h             ;       Input (Data, Variable, Absolute)

; the output report
    db 09h, A9h             ;       Usage (vendor defined)
    db 15h, 80h             ;       Logical Minimum (-128)
    db 25h, 7Fh             ;       Logical Maximum (127)
    db 75h, 08h             ;       Report Size (8) (bits)
    db 95h, 02h             ;       Report Count (2) (fields)
    db 91h, 02h             ;       Output (Data, Variable, Absolute)

    db C0h                  ;       End collection
End_hid_report_desc_table:
```

As shown in the descriptor : the input report has a size of “2” bytes to be send to the host, the output report has also “2” bytes to be send to the device, all the items listed in the above report descriptor are needed in all reports, each item in this descriptor consists of a byte that identifies the item, the item’s data may be one or more bytes, in particular each item in the above descriptor means (Axelson, 2005):

- Usage Page: this item specifies the general function of the device (generic desktop controls, game control …etc), this item has a value of (06 h), according to HID specification this defines a single operating mode for a control, in the above example the usage page value (FFA0 h) is a vender defined value.

- usage: this item is a subset of the usage page, its value is (09h) and it describes the function of the individual report, example of usages available for generic desktop controls are keyboards, mice, …etc, if the usage page is vender -defined, the usage should be a vender -defined, in this example it was given the value of (A5 h) .

- Collection (Application): each report descriptor, should have an application collection, this is necessary for Windows to enumerate the device, this item and all the items that follow it performs together a single function such as a keyboard or a mouse, the usage that follows this item has a value of (A6 h) and is vender- defined value.

- Logical minimum and maximum: this item determines the range of values the report can contain. In the example the values are (80 h) and (7F h) which is within the range from (-128 to +127), negative values are expressed in 2's complement format.

- Report size: this item defines the number of bits in any data item, in this particular example this item has a value of (75 h) and each data item consists of eight bits.

- Report count: this item identifies the number of data items in the report, in this example it has a value of (95 h) and the report contains two bytes (data items).

- Input (Data, Variable, Absolute): this item specifies the direction of data, whether it is from the host to the device (91 h) or from the device to the host (81 h), it also gives other information about the data.

- End collocation: this item indicates the end of the application collection.

### C.1.3 Physical descriptor:

A physical descriptor is a data structure that provides information about the specific part or parts of the human body that are activating a control or controls (Universal Serial Bus, 2001), for example " the right hand thump is used to activate buttons".

Physical descriptors are optional, the host retrieves physical descriptors by sending a Get-Descriptor request to the device.

## C.2 HID Specific Requests

According to the HID specification, there are six HID – specific requests, table (C.2) below shows details.

Table C.2: HID specific requests (Axelson, 2005).

| Request # | request | Data source | value | index | Data Length (bytes) | Data Stage contents | Required? |
|---|---|---|---|---|---|---|---|
| 01 h | Get-Report | device | Report Type, Report ID | interface | report length | report | Yes |
| 02 h | Get-Idle | device | Report ID | interface | 1 | idle duration | no |
| 03 h | Get-Protocol | device | 0 | interface | 1 | protocol | required for boot devices |
| 09 h | Set-Report | host | Report Type, Report ID | interface | report length | report | no |
| 0A h | Set-Idle | host | idle Duration, Report ID | interface | 0 | none | no |
| 0B h | Set-Protocol | host | protocol | interface | 0 | none | required for boot devices |

As explained in the table above the Get-Report request is required for all HIDs while Get-Protocol and Set-Protocol are required for boot devices, the remaining requests are optional, table (C.3) below shows the format of any request (Universal Serial Bus, 2001).

Table C.3: Format of any HID request (Universal Serial Bus, 2001) .

| **Part** | **Offset/Size (Bytes)** | **Description** |
|---|---|---|
| bmRequestType | 0/1 | Bits specifying characteristics of request. Valid values are 10100001 or 00100001 only based on the following description:<br>7 Data transfer direction<br>0 = Host to device<br>1 = Device to host<br>6..5 Type<br>1 = Class<br>4..0 Recipient<br>1 = Interface |
| bRequest | 1/1 | A specific request. |
| wValue | 2/2 | Numeric expression specifying word-size field (varies according to request.) |
| wIndex | 4/2 | Index or offset specifying word-size field (varies according to request.) |
| wLength | 6/2 | Numeric expression specifying number of bytes to transfer in the data phase. |

A detailed description of each request is given below (Universal Serial Bus, 2001).

- Get -Report: this request allows the host to receive data from the device using control transfers.

| Part | Description |
|---|---|
| bmRequestType | 1010 0001 |
| bRequest | Get-Report |
| wValue | Report type and report ID<br><br>High byte = Report Type (1=input, 2= output, 3= feature), low byte = report ID, default report ID = 0. |
| wIndex | Number of supported interfaces. |
| wLength | Report length. |
| Data | Report. |

According to HID specification the host should use Interrupt IN pipe to obtain periodic data, using Interrupt OUT pipe for sending output reports is optional, all HIDs must support this request.

- Set -Report: this request allows the host to send data to the device using Control transfers.

| Part | Description |
|---|---|
| bmRequestType | 0010 0001 |
| bRequest | Set-Report |
| wValue | Report Type (high byte, 1= input, 2=output, 3=feature) and Report ID (low byte, default value=0). |
| wIndex | Number of interfaces supported. |
| wLength | Report length. |
| Data | Report |

For devices that do not have an Interrupt OUT Endpoint, using this request is the only way that enables the host to send data to the device, this request is not required for HID devices.

- Get -Idle : this request is used by the host to read the current idle rate from a device.

| Part | Description |
|---|---|
| bmRequestType | 1010 0001 |
| bRequest | Get-Idle |
| wValue | High byte = 0. low byte=report ID |
| wIndex | Number of interfaces that support this request |
| wLength | 1 (one) |
| Data | Idle rate expressed in units of milliseconds. |

HID devices are not required to support this request.

- Set-Idle: when the data received hasn't changed since the last report, the Set-Idle request limits the frequency of the Interrupt IN endpoint and thus saves bandwidth.

| Part | Description |
|---|---|
| bmRequestType | 0010 0001 |
| bRequest | Set-Idle |
| wValue | Sets the duration (high byte) or the maximum a mount of time between successive reports, if the value is zero the device will send reports only when the report data has changed, otherwise it sends a NAK, the low byte indicates report ID. |
| wIndex | Number of interfaces that support this request |
| wLength | Zero |
| Data | Not applicable |

This request is not required for HID devices, during a HID enumeration the Windows Device Driver sets the idle rate to zero, if this request is supported by a HID, the device sends a report when the report data has changed, if the device returns a STALL when receiving this

request, this means that the request is not supported and reports can be send regardless of the change of data.

- Get-Protocol: this request is used by the host to read which protocol is currently active on the device.

| **Part** | **Description** |
|---|---|
| bmRequestType | 101 0001 |
| bRequest | Get-Protocol |
| wValue | 0 (zero) |
| wIndex | Number of interface that support this request. |
| wLength | 1 (one) |
| Data | 0=boot protocol, 1= report protocol |

This request is supported by boot devices.

- Set-Protocol: this request is used by the host to switch between boot and report protocols, it is supported by boot devices.

| **Part** | **Description** |
|---|---|
| bmRequestType | 0010 0001 |
| bRequest | Set-Protocol |
| wValue | 0 = boot protocol , 1 = report protocol. |
| wIndex | Number of interfaces supporting this request. |
| wLength | 0 (zero) |
| Data | Not applicable. |

# APPENDIX D

## FIRMWARE SOURCE CODE

### D.1 Header Files

#### D.1.1 Usbinit.h

```
#ifndef USBINIT_H
#define USBINIT_H
//*****************************************************************************
// Prototypes
//*****************************************************************************
void InitializeUsbFunction(void);
void UsbReset(void);
#endif
```

### D.1.2 Usb.h

**-- Description: header file for USB functions**

```
#ifndef USB_H
#define USB_H

//***************************************************************************
// Enumeration Definitions
//***************************************************************************
typedef enum
{
   STATUS_ACTION_NOTHING,
   STATUS_ACTION_DATA_IN,
   STATUS_ACTION_DATA_OUT
} tSTATUS_ACTION_LIST;

//***************************************************************************
// Stucture Definitions
//***************************************************************************
// DEVICE_REQUEST Structure
typedef struct _tDEVICE_REQUEST
{
   unsigned char bmRequestType;       // See bit definitions below
   unsigned char bRequest;            // See value definitions below
   unsigned char bValueL;             // Meaning varies with request type
   unsigned char bValueH;             // Meaning varies with request type
   unsigned char bIndexL;             // Meaning varies with request type
   unsigned char bIndexH;             // Meaning varies with request type
   unsigned char bLengthL;            // Number of bytes of data to transfer (LSByte)
   unsigned char bLengthH;            // Number of bytes of data to transfer (MSByte)
} tDEVICE_REQUEST;

typedef struct _tDEVICE_REQUEST_COMPARE
{
   unsigned char bmRequestType;       // See bit definitions below
   unsigned char bRequest;            // See value definitions below
   unsigned char bValueL;             // Meaning varies with request type
   unsigned char bValueH;             // Meaning varies with request type
   unsigned char bIndexL;             // Meaning varies with request type
   unsigned char bIndexH;             // Meaning varies with request type
   unsigned char bLengthL;            // Number of bytes of data to transfer (LSByte)
   unsigned char bLengthH;            // Number of bytes of data to transfer (MSByte)
   unsigned char bCompareMask;        // MSB is bRequest, if set 1, bRequest should be matched,
LSB is bLengthH
   void (*pUsbFunction)(void);
// function pointer
} tDEVICE_REQUEST_COMPARE, *ptDEVICE_REQUEST_COMPARE;

//***************************************************************************
// Constant Definitions
//***************************************************************************
// USB Device VID and PID Definition
#define VID_L  0x51                    // TI = 0x0451
```

```
#define VID_H  0x04
#define PID_L  0x10                    // TUSB3210 = 0x3210
#define PID_H  0x32
#define VER_L  0x00                    // Version 1.0
#define VER_H  0x01
#define NO_MORE_DATA    0xFFFF      // 0 means, send a null packet, 0xFF => no more data

//*****************************************************************************
// DEVICE REQUEST
//*****************************************************************************
#define SIZEOF_DEVICE_REQUEST   0x08

//  Bit definitions for DEVICE_REQUEST.bmRequestType
//  Bit 7:  Data direction
#define USB_REQ_TYPE_OUTPUT       0x00   // 0 = Host sending data to device
#define USB_REQ_TYPE_INPUT        0x80   // 1 = Device sending data to host

//  Bit 6-5: Type
#define USB_REQ_TYPE_MASK         0x60   // Mask value for bits 6-5
#define USB_REQ_TYPE_STANDARD     0x00   // 00 = Standard USB request
#define USB_REQ_TYPE_CLASS        0x20   // 01 = Class specific
#define USB_REQ_TYPE_VENDOR       0x40   // 10 = Vendor specific

//  Bit 4-0: Recipient
#define USB_REQ_TYPE_RECIP_MASK     0x1F   // Mask value for bits 4-0
#define USB_REQ_TYPE_DEVICE         0x00   // 00000 = Device
#define USB_REQ_TYPE_INTERFACE      0x01   // 00001 = Interface
#define USB_REQ_TYPE_ENDPOINT       0x02   // 00010 = Endpoint
#define USB_REQ_TYPE_OTHER          0x03   // 00011 = Other

// Values for DEVICE_REQUEST.bRequest
// Standard Device Requests
#define USB_REQ_GET_STATUS              0
#define USB_REQ_CLEAR_FEATURE           1
#define USB_REQ_SET_FEATURE             3
#define USB_REQ_SET_ADDRESS             5
#define USB_REQ_GET_DESCRIPTOR          6
#define USB_REQ_SET_DESCRIPTOR          7
#define USB_REQ_GET_CONFIGURATION       8
#define USB_REQ_SET_CONFIGURATION       9
#define USB_REQ_GET_INTERFACE           10
#define USB_REQ_SET_INTERFACE           11
#define USB_REQ_SYNCH_FRAME             12

//*****************************************************************************
// HID CLASS Requests
//*****************************************************************************
#define USB_REQ_GET_REPORT      0x01
#define USB_REQ_GET_IDLE        0x02
#define USB_REQ_GET_PROTOCOL    0x03
#define USB_REQ_SET_REPORT      0x09
#define USB_REQ_SET_IDLE        0x0A
#define USB_REQ_SET_PROTOCOL    0x0B
```

```
//*****************************************************************************
// DESCRIPTOR TYPES
//*****************************************************************************
//  Descriptor Type Values
#define DESC_TYPE_DEVICE            1     // Device Descriptor (Type 1)
#define DESC_TYPE_CONFIG            2     // Configuration Descriptor (Type 2)
#define DESC_TYPE_STRING            3     // String Descriptor (Type 3)
#define DESC_TYPE_INTERFACE         4     // Interface Descriptor (Type 4)
#define DESC_TYPE_ENDPOINT          5     // Endpoint Descriptor (Type 5)
#define DESC_TYPE_HID               0x21  // HID Descriptor (Type 0x21)
#define DESC_TYPE_REPORT            0x22  // Report Descriptor (Type 0x22)
#define DESC_TYPE_PHYSICAL          0x23  // Physical Descriptor (Type 0x23)

//*****************************************************************************
// FEATURES
//*****************************************************************************
//  Feature Selector Values
#define FEATURE_REMOTE_WAKEUP       1     // Remote wakeup (Type 1)
#define FEATURE_ENDPOINT_STALL      0     // Endpoint stall (Type 0)

//*****************************************************************************
// GET STATUS VALUES
//*****************************************************************************
// Device Status Values
#define DEVICE_STATUS_REMOTE_WAKEUP   0x02
#define DEVICE_STATUS_SELF_POWER      0x01


//*****************************************************************************
// DECSRIPTOR SIZES
//*****************************************************************************
#define SIZEOF_DEVICE_DESCRIPTOR            0x12
#define SIZEOF_CONFIG_DESCRIPTOR            0x09
#define SIZEOF_INTERFACE_DESCRIPTOR         0x09
#define SIZEOF_ENDPOINT_DESCRIPTOR          0x07
#define SIZEOF_HID_DESCRIPTOR               0x09
#define SIZEOF_CONFIG_DESC_GROUP SIZEOF_CONFIG_DESCRIPTOR +
SIZEOF_INTERFACE_DESCRIPTOR + SIZEOF_HID_DESCRIPTOR +
SIZEOF_ENDPOINT_DESCRIPTOR

//  Bit definitions for CONFIG_DESCRIPTOR.bmAttributes
#define CFG_DESC_ATTR_SELF_POWERED      0x40   // Bit 6: If set, device is self powered
#define CFG_DESC_ATTR_BUS_POWERED       0x80   // Bit 7: If set, device is bus powered
#define CFG_DESC_ATTR_REMOTE_WAKE     0x20     // Bit 5: If set, device supports remote
wakeup

//  Bit definitions for EndpointDescriptor.EndpointAddr
#define EP_DESC_ADDR_EP_NUM    0x0F   // Bit 3-0: Endpoint number
#define EP_DESC_ADDR_DIR_IN    0x80   // Bit 7: Direction of endpoint, 1/0 = In/Out

//  Bit definitions for EndpointDescriptor.EndpointFlags
#define EP_DESC_ATTR_TYPE_MASK   0x03   // Mask value for bits 1-0
#define EP_DESC_ATTR_TYPE_CONT   0x00   // Bit 1-0: 00 = Endpoint does control transfers
#define EP_DESC_ATTR_TYPE_ISOC    0x01   // Bit 1-0: 01 = Endpoint does isochronous
transfers
```

```
#define EP_DESC_ATTR_TYPE_BULK   0x02   // Bit 1-0: 10 = Endpoint does bulk transfers
#define EP_DESC_ATTR_TYPE_INT    0x03   // Bit 1-0: 11 = Endpoint does interrupt
transfers

//*****************************************************************************
// Prototypes
//*****************************************************************************
void usbGetConfiguration(void);
void usbSetConfiguration(void);
void usbGetDeviceDescriptor(void);
void usbGetConfigurationDescriptor(void);
void usbGetStringDescriptor(void);
void usbGetHIDDescriptor(void);
void usbGetReportDescriptor(void);
void usbSetReport(void);
void usbGetInterface(void);
void usbSetInterface(void);
void usbGetDeviceStatus(void);
void usbSetRemoteWakeup(void);
void usbClearRemoteWakeup(void);
void usbGetInterfaceStatus(void);
void usbSetAddress(void);
void usbSetEndpointHalt(void);
void usbClearEndpointHalt(void);
void usbGetEndpointStatus(void);
void usbNonStandardRequest(void);
void usbDecodeAndProcessUsbRequest(void);
void usbStallEndpoint0(void);
void usbReceiveDataPacketOnEP0(unsigned char * pBuffer);
void usbReceiveNextPacketOnOEP0(void);
void usbSendZeroLengthPacketOnIEP0(void);
void usbSendDataPacketOnEP0(unsigned char * pBuffer);
void usbSendNextPacketOnIEP0(void);
void usbSendDataToHostOnEP1(void);
void SetupPacketInterruptHandler(void);
void OEP0InterruptHandler(void);
void IEP0InterruptHandler(void);
void IEP1InterruptHandler(void);

#endif
```

### D.1.3 Delay.h

```
-- Description: header file for delay in ms
--------------------------------------------------------------------------------

#ifndef DELAY_H
#define DELAY_H

//*****************************************************************************
// Prototypes
//*****************************************************************************
void Delay_ms(unsigned int time);
void Delay_5us(void);

#endif
```

### D.1.4 Descriptor.h

-- Description: header file with descriptor definitions
---

```
#ifndef DESCRIPTOR_H
#define DESCRIPTOR_H

//*****************************************************************************
// Device descriptor
//*****************************************************************************
unsigned char code romDeviceDescriptor[] = {
   SIZEOF_DEVICE_DESCRIPTOR,      // Length of this descriptor (12h bytes)
   DESC_TYPE_DEVICE,              // Type code of this descriptor (01h)
   0x10,0x01,                     // Release of USB spec (Rev 1.1)
   0,                             // Device's base class code
   0,                             // Device's sub class code
   0,                             // Device's protocol type code
   EP0_MAX_PACKET_SIZE,           // End point 0's max packet size = 8
   VID_L,VID_H,                   // Vendor ID for device, TI=0x0451
   PID_L,PID_H,                   // Product ID for device, 0x2136
   VER_L,VER_H,                   // Revision level of device, Rev=1.0
   1,                             // Index of manufacturer name string desc
   2,                             // Index of product name string desc
   3,                             // Index of serial number string desc
   1                              // Number of configurations supported
   };

//*****************************************************************************
// Report descriptor
// generated with HID-Generator
//*****************************************************************************
unsigned char code romReportDescriptor[] =
{
0x06, 0xA0, 0xFF,                 // Usage Page (vendor defined)
0x09, 0x01,                       // Usage (I/O Device)
0xA1, 0x01,                       // Collection (Application)
 // input report
0x19, 0x00,                       // Usage_Minimum(Unicode Char 0)
0x29, 0x3F,                       // Usage_Maximum(Unicode Char 63)
0x15, 0x80,                       // Logical Minimum (-128)
0x25, 0x7F,                       // Logical Maximum (127)
0x75, 0x08,                       // Report Size (8 bit)
0x95, 0x40,                       // Report Count (64 Bytes)
0x81, 0x02,                       // Input (Data, Variable, Absolute)

// output report
0x19, 0x00,                       // Usage_Minimum(Unicode Char 0)
0x29, 0x3F,                       // Usage_Maximum(Unicode Char 63)
0x91, 0x02,                       // Output (Data, Variable, Absolute)
0xC0                              // End Collection
};
```

```
//*****************************************************************************
// Configuration descriptor group
//*****************************************************************************
unsigned char code romConfigurationDescriptorGroup[] =
{
   // Configuration  Descriptor, size=0x09
   SIZEOF_CONFIG_DESCRIPTOR,          // bLength
   DESC_TYPE_CONFIG,                  // bDescriptorType
   SIZEOF_CONFIG_DESC_GROUP, 0x00, // wTotalLength
   1,                                 // bNumInterfaces
   1,                                 // bConfigurationValue
   0,                                 // iConfiguration, string index
   CFG_DESC_ATTR_BUS_POWERED,     // bmAttributes, bus bootcode
   0x64,                              // Max. Power Consumption at 2mA unit (200mA)

   // Interface Descriptor, size = 0x09
   SIZEOF_INTERFACE_DESCRIPTOR,   // bLength
   DESC_TYPE_INTERFACE,               // bDescriptorType
   0,                                 // bInterfaceNumber
   0,                                 // bAlternateSetting
   1,                                 // bNumEndpoints
   3,                                 // bInterfaceClass: 3 = HID class
   0,                          // bInterfaceSubClass: 0 = no Subclass, 1 = boot device
   0,                          // bInterfaceProtocol: 0 = no protocol, 1 = keyboard
   0,                         // iInterface, string index

   // HID DESCRIPTOR (9 bytes)
   SIZEOF_HID_DESCRIPTOR,          // bLength of HID descriptor
   DESC_TYPE_HID,                  // HID Descriptor Type: assigned by USB
   0x10,0x01,                      // HID Revision number 1.1
   0,                              // Target country
   1,                              // Number of HID classes to follow
   DESC_TYPE_REPORT,               // Report descriptor type
   sizeof(romReportDescriptor), 0x00,      // Total length of report descriptor

    // Input Endpoint 1 Descriptor, size = 0x07
   SIZEOF_ENDPOINT_DESCRIPTOR,    // bLength
   DESC_TYPE_ENDPOINT,              // bDescriptorType
   0x81,                          // bEndpointAddress; bit7=1 for IN, bits 3-0=1 for EP1
   EP_DESC_ATTR_TYPE_INT,          // bmAttributes, interupt transfer for HID IN
   EP1_MAX_PACKET_SIZE, 0x00,      // wMaxPacketSize
  1                                // bInterval
};

//*****************************************************************************
// String descriptors
//*****************************************************************************
char code mfgDescription[] = "Peripheral USB Interface Board By Khaled Murad";
char code prodDescription[] = "USB-Peripheral Interface Board";
char code revDescription[] = "Version 1.0";

#endif
```

### D.1.5 Reg52.h

-- Description: header file for 8052 controllers
---

```
#ifndef REG52_H
#define REG52_H

/*  BYTE Registers  */
sfr P0    = 0x80;
sfr P1    = 0x90;
sfr P2    = 0xA0;
sfr P3    = 0xB0;
sfr PSW   = 0xD0;
sfr ACC   = 0xE0;
sfr B     = 0xF0;
sfr SP    = 0x81;
sfr DPL   = 0x82;
sfr DPH   = 0x83;
sfr PCON  = 0x87;
sfr TCON  = 0x88;
sfr TMOD  = 0x89;
sfr TL0   = 0x8A;
sfr TL1   = 0x8B;
sfr TH0   = 0x8C;
sfr TH1   = 0x8D;
sfr IE    = 0xA8;
sfr IP    = 0xB8;
sfr SCON  = 0x98;
sfr SBUF  = 0x99;

/*  8052 Extensions  */
sfr T2CON    = 0xC8;
sfr RCAP2L   = 0xCA;
sfr RCAP2H   = 0xCB;
sfr TL2      = 0xCC;
sfr TH2      = 0xCD;

/*  BIT Registers  */
/*  PSW  */
sbit CY   = PSW^7;
sbit AC   = PSW^6;
sbit F0   = PSW^5;
sbit RS1  = PSW^4;
sbit RS0  = PSW^3;
sbit OV   = PSW^2;
sbit P    = PSW^0; //8052 only

/*  TCON  */
sbit TF1  = TCON^7;
sbit TR1  = TCON^6;
sbit TF0  = TCON^5;
sbit TR0  = TCON^4;
```

```
sbit IE1   = TCON^3;
sbit IT1   = TCON^2;
sbit IE0   = TCON^1;
sbit IT0   = TCON^0;

/*  IE  */
sbit EA    = IE^7;
sbit ET2   = IE^5; //8052 only
sbit ES    = IE^4;
sbit ET1   = IE^3;
sbit EX1   = IE^2;
sbit ET0   = IE^1;
sbit EX0   = IE^0;

/*  IP  */
sbit PT2   = IP^5;
sbit PS    = IP^4;
sbit PT1   = IP^3;
sbit PX1   = IP^2;
sbit PT0   = IP^1;
sbit PX0   = IP^0;

/*  SCON  */
sbit SM0   = SCON^7;
sbit SM1   = SCON^6;
sbit SM2   = SCON^5;
sbit REN   = SCON^4;
sbit TB8   = SCON^3;
sbit RB8   = SCON^2;
sbit TI    = SCON^1;
sbit RI    = SCON^0;

/*  T2CON  */
sbit TF2      = T2CON^7;
sbit EXF2     = T2CON^6;
sbit RCLK     = T2CON^5;
sbit TCLK     = T2CON^4;
sbit EXEN2    = T2CON^3;
sbit TR2      = T2CON^2;
sbit C_T2     = T2CON^1;
sbit CP_RL2 = T2CON^0;

/*-------------------------------------------------
P0 Bit Registers
-------------------------------------------------*/
sbit P0_0 = P0^0;
sbit P0_1 = P0^1;
sbit P0_2 = P0^2;
sbit P0_3 = P0^3;
sbit P0_4 = P0^4;
sbit P0_5 = P0^5;
sbit P0_6 = P0^6;
sbit P0_7 = P0^7;
```

```
/*-----------------------------------------------
P1 Bit Registers
-----------------------------------------------*/
sbit P1_0 = P1^0;
sbit P1_1 = P1^1;
sbit P1_2 = P1^2;
sbit P1_3 = P1^3;
sbit P1_4 = P1^4;
sbit P1_5 = P1^5;
sbit P1_6 = P1^6;
sbit P1_7 = P1^7;

/*-----------------------------------------------
P2 Bit Registers
-----------------------------------------------*/
sbit P2_0 = P2^0;
sbit P2_1 = P2^1;
sbit P2_2 = P2^2;
sbit P2_3 = P2^3;
sbit P2_4 = P2^4;
sbit P2_5 = P2^5;
sbit P2_6 = P2^6;
sbit P2_7 = P2^7;


/*-----------------------------------------------
P3 Bit Registers
-----------------------------------------------*/
sbit P3_0 = P3^0;
sbit P3_1 = P3^1;
sbit P3_2 = P3^2;
sbit P3_3 = P3^3;
sbit P3_4 = P3^4;
sbit P3_5 = P3^5;
sbit P3_6 = P3^6;
sbit P3_7 = P3^7;

/*-----------------------------------------------
Interrupt Vectors:
Interrupt Address = (Number * 8) + 3
-----------------------------------------------*/
#define IE0_VECTOR    0  /* 0x03 External Interrupt 0 */
#define TF0_VECTOR    1  /* 0x0B Timer 0 Interrupt, used for all internal peripherals*/
#define IE1_VECTOR    2  /* 0x13 External Interrupt 1, used for P2[7:0] interrupt */
#define TF1_VECTOR    3  /* 0x1B Timer 1 Interrupt*/
#define SIO_VECTOR    4  /* 0x23 UART Interrupt */
#define TF2_VECTOR    5  /* 0x2B Timer 2 Interrupt */

#endif
```

### D.1.6 Tusb3210.h

**-- Description: header file defining the registers of the tusb3210**
**This file contains definitions from the TUSB2136 Generic Keyboard Demo Program**
**From Texas Instruments (Texas Instruments, 2000) .**
---

```
#ifndef TUSB3210_H
#define TUSB3210_H

/*-------------------------------------------------------------------------+
| Constant Definition                                                      |
+-------------------------------------------------------------------------*/

// USB related Constant
#define MAX_ENDPOINT_NUMBER     0x03
#define EP0_MAX_PACKET_SIZE     0x08
#define EP1_MAX_PACKET_SIZE     0x40    // 64 bytes FIFO

/*-------------------------------------------------------------------------+
| DATA BUFFER (368 Byte, XDATA range = FD80...FEEF
+-------------------------------------------------------------------------*/

// Buffer for descriptors (0xFD80...0xFDB3)
#define START_OF_USER_BUFFER_ADDRESS    0xFD80



// Buffer address for Endpoints 1 (0xFDB8...0xFEB7)
#define IEP1_X_BUFFER_ADDRESS   0xFDB8   // Input Endpoint 1 X Buffer Base-address (64
Bytes)
#define OEP1_X_BUFFER_ADDRESS   0xFDF8   // Output Endpoint 1 X Buffer Base-address
(64 Bytes)
#define DATAFROMHOST_ADDRESS   0xFE38// Data received from host on SET_REPORT (64
Bytes)
#define DATATOHOST_ADDRESS      0xFE78  // Data to be sent to the host via IEP1 (64 Bytes)

// Endpoint 0: buffer address
#define OEP0_BUFFER_ADDRESS   0xFEF0   // Output Endpoint 0 Buffer Base-address hard
wired
#define IEP0_BUFFER_ADDRESS   0xFEF8      // Input Endpoint 0 Buffer Base-address hard
wired
// Setup packet block
#define EP0_SETUP_ADDRESS   0xFF00      // Setup packet request type
// MCU Configuration Register
#define MCNFG_SDW      0x01   // 0: start from BootRom, 1: start from RAM
#define MCNFG_XINT     0x40   // INT1 source control bit
                              // 0:P3.3 1:P2[7:0]
// Watchdog Register
#define WDCSR_WDT      0x01    // Watchdog timer reset bit
                               // write a 1 to reset timer
#define WDCSR_WDR      0x40   // Watchdog reset indication bit
                               // 0:a power-up or USB reset
                               // 1:watchdog timeout reset occurred.
```

```
#define WDCSR_WDE       0x80    // Watchdog timer enable bit
                                // 0:disable(only cleared on power-up, USB or WDT reset)
                                // 1:enable
// EndPoint Desciptor Block
#define EPCNF_USBIE     0x04    // USB Interrupt on Transaction Completion. Set By MCU
                                // 0:No Interrupt, 1:Interrupt on completion
#define EPCNF_STALL     0x08    // USB Stall Condition Indication. Set by UBM
                                // 0: No Stall, 1:USB Install Condition
#define EPCNF_DBUF      0x10    // Double Buffer Enable. Set by MCU
                                // 0: Primary Buffer Only(X-buffer only), 1:Toggle Bit Selects Buffer
#define EPCNF_TOGLE     0x20    // USB Toggle bit. This bit reflects the toggle sequence bit of
DATA0 and DATA1.
#define EPCNF_ISO       0x40    // ISO=0, Non Isochronous transfer. This bit must be cleared
by MCU since only non isochronous transfer is supported.
#define EPCNF_UBME      0x80    // UBM Enable or Disable bit. Set or Clear by MCU.
                                // 0:UBM can't use this endpoint
                                // 1:UBM can use this endpoint

#define EPBCT_BYTECNT_MASK 0x7F // MASK for Buffer Byte Count
#define EPBCNT_NAK      0x80    // NAK bit
                                // 0:buffer contains valid data
                                // 1:buffer is empty
// USB Registers
#define USBSTA_STPOW    0x01    // Setup Overwrite Bit. Set by hardware when setup packet
is received
                                // while there is already a packet in the setup buffer.
                                // 0:Nothing, 1:Setup Overwrite
#define USBSTA_RWUP     0x02    // Remote wakeup overwrite bit
#define USBSTA_SETUP    0x04    // Setup Transaction Received Bit. As long as SETUP is '1',
            // IN and OUT on endpoint-0 will be NAKed regardless of their real NAK bits values.
#define USBSTA_PWON     0x08    // Power Request for port3
#define USBSTA_PWOFF    0x10    // Power Off Request for port3
#define USBSTA_RESR     0x20    // Function Resume Request Bit. 0:clear by MCU, 1:Function
Resume is detected.
#define USBSTA_SUSR     0x40    // Function Suspended Request Bit. 0:clear by MCU,
1:Function Suspend is detected.
#define USBSTA_RSTR     0x80    // Function Reset Request Bit. This bit is set in response to a
global or selective suspend condition.
                                // 0:clear by MCU, 1:Function reset is detected.

#define USBMSK_STPOW    0x01    // Setup Overwrite Interrupt Enable Bit
                                // 0: disable, 1:enable
#define USBMSK_RWUP
                                                                        0x02
#define USBMSK_SETUP    0x04    // Setup Interrupt Enable Bit
                                // 0: disable, 1:enable
#define USBMSK_RESR     0x20    // Function Resume Interrupt Enable Bit
                                // 0: disable, 1:enable
#define USBMSK_SUSR     0x40    // Function Suspend Interrupt Enable Bit
                                // 0: disable, 1:enable
#define USBMSK_RSTR     0x80    // Function Reset Interrupt Enable Bit
                                // 0: disable, 1:enable

#define USBCTL_DIR      0x01    // USB traffic direction 0: USB out packet, 1:in packet (from
TUSB3210 to Host)
```

```
#define USBCTL_SIR      0x02     // Setup interrupt status bit
                                 // 0: SETUP interrupt is not served.
                                 // 1: SETUP interrupt in progess
#define USBCTL_SELF     0x04     // Bus/self powered bit
                                 // 0: bus, 1:self
#define USBCTL_RWE      0x08     // remote wakeup enable bit
                                 // 0: disable, 1:enable
#define USBCTL_FRSTE    0x10     // Function Reset Condition Bit.
                                 // This bit connects or disconnects the USB Function Reset
from the MCU reset
                                 // 0:not connect, 1:connect
#define USBCTL_RWUP     0x20     // Device Remote Wakeup Request
                                 // 0:nothing, 1:remote wakeup request to USB Host
#define USBCTL_U12      0x40     // USB Hub version
                        // 0:1.x, 1:2.x
#define USBCTL_CONT     0x80     // Connect or Disconnect Bit
                                 // 0:Upstream port is disconnected. Pull-up disabled
                                 // 1:Upstream port is connected. Pull-up enabled


// Interrupt vector values for INT0
#define VECINT_NO_INTERRUPT              0x00
#define VECINT_OUTPUT_ENDPOINT1          0x12
#define VECINT_OUTPUT_ENDPOINT2          0x14
#define VECINT_OUTPUT_ENDPOINT3          0x16

#define VECINT_INPUT_ENDPOINT1           0x22
#define VECINT_INPUT_ENDPOINT2           0x24
#define VECINT_INPUT_ENDPOINT3           0x26

#define VECINT_STPOW_PACKET_RECEIVED   0x30        // USBSTA
#define VECINT_SETUP_PACKET_RECEIVED   0x32        // USBSTA
#define VECINT_POWER_ON                  0x34
#define VECINT_POWER_OFF                 0x36
#define VECINT_RESR_INTERRUPT            0x38        // USBSTA
#define VECINT_SUSR_INTERRUPT            0x3A        // USBSTA
#define VECINT_RSTR_INTERRUPT            0x3C        // USBSTA
#define VECINT_I2C_RXF_INTERRUPT         0x40        // I2CSTA
#define VECINT_I2C_TXE_INTERRUPT         0x42        // I2CSTA
#define VECINT_INPUT_ENDPOINT0           0x44
#define VECINT_OUTPUT_ENDPOINT0          0x46

//I2C Registers
#define I2CSTA_SWR       0x01      // Stop Write Enable
                                   // 0:disable, 1:enable
#define I2CSTA_SRD       0x02      // Stop Read Enable
                                   // 0:disable, 1:enable
#define I2CSTA_TIE       0x04      // I2C Transmitter Empty Interrupt Enable
                                   // 0:disable, 1:enable
#define I2CSTA_TXE       0x08      // I2C Transmitter Empty
                                   // 0:full, 1:empty
#define I2CSTA_400K      0x10      // I2C Speed Select
                                   // 0:100kHz, 1:400kHz
#define I2CSTA_ERR       0x20      // Bus Error Condition
                                   // 0:no bus error, 1:bus error
```

```
#define I2CSTA_RIE        0x40      // I2C Receiver Ready Interrupt Enable
                                    // 0:disable, 1:enable
#define I2CSTA_RXF        0x80      // I2C Receiver Full
                                    // 0:empty, 1:full
#define I2CADR_READ       0x01      // Read Write Command Bit
                                    // 0:write, 1:read

/*---------------------------------------------------------------------------+
| ENDPOINT 0 SETUP BLOCK (EDB, XDATA range = FF00...FF07
+---------------------------------------------------------------------------*/

#define pEP0_SETUP_ADDRESS   ( (char xdata *)0xFF00)

/*---------------------------------------------------------------------------+
| ENDPOINT 1..3 BLOCKS (EDB, XDATA range = FF08...FF7F
+---------------------------------------------------------------------------*/
// Output Endpoint 1: configuration
#define OEPCNF_1        (* (char xdata *)0xFF08)   // Output Endpoint 1 Configuration
#define OEPBBAX_1       (* (char xdata *)0xFF09)   // Output Endpoint 1 X-Buffer Base-address
#define OEPBCTX_1       (* (char xdata *)0xFF0A)   // Output Endpoint 1 X Byte Count
#define OEPBBAY_1       (* (char xdata *)0xFF0D)   // Output Endpoint 1 Y-Buffer Base-address
#define OEPBCTY_1       (* (char xdata *)0xFF0E)   // Output Endpoint 1 Y Byte Count
#define OEPSIZXY_1      (* (char xdata *)0xFF0F)   // Output Endpoint 1 XY-Buffer Size

// Output Endpoint 2: configuration
#define OEPCNF_2        (* (char xdata *)0xFF10)   // Output Endpoint 2 Configuration
#define OEPBBAX_2       (* (char xdata *)0xFF11)   // Output Endpoint 2 X-Buffer Base-address
#define OEPBCTX_2       (* (char xdata *)0xFF12)   // Output Endpoint 2 X Byte Count
#define OEPBBAY_2       (* (char xdata *)0xFF15)   // Output Endpoint 2 Y-Buffer Base-address
#define OEPBCTY_2       (* (char xdata *)0xFF16)   // Output Endpoint 2 Y Byte Count
#define OEPSIZXY_2      (* (char xdata *)0xFF17)   // Output Endpoint 2 XY-Buffer Size

// Output Endpoint 3: configuration
#define OEPCNF_3        (* (char xdata *)0xFF18)   // Output Endpoint 3 Configuration
#define OEPBBAX_3       (* (char xdata *)0xFF19)   // Output Endpoint 3 X-Buffer Base-address
#define OEPBCTX_3       (* (char xdata *)0xFF1A)   // Output Endpoint 3 X Byte Count
#define OEPBBAY_3       (* (char xdata *)0xFF1D)   // Output Endpoint 3 Y-Buffer Base-address
#define OEPBCTY_3       (* (char xdata *)0xFF1E)   // Output Endpoint 3 Y Byte Count
#define OEPSIZXY_3      (* (char xdata *)0xFF1F)   // Output Endpoint 3 XY-Buffer Size

// Input Endpoint 1: configuration
#define IEPCNF_1        (* (char xdata *)0xFF48)   // Input Endpoint 1 Configuration
#define IEPBBAX_1       (* (char xdata *)0xFF49)   // Input Endpoint 1 X-Buffer Base-address
#define IEPBCTX_1       (* (char xdata *)0xFF4A)   // Input Endpoint 1 X Byte Count
#define IEPBBAY_1       (* (char xdata *)0xFF4D)   // Input Endpoint 1 Y-Buffer Base-address
#define IEPBCTY_1       (* (char xdata *)0xFF4E)   // Input Endpoint 1 Y Byte Count
#define IEPSIZXY_1      (* (char xdata *)0xFF4F)   // Input Endpoint 1 XY-Buffer Size

// Input Endpoint 2: configuration
#define IEPCNF_2        (* (char xdata *)0xFF50)   // Input Endpoint 2 Configuration
#define IEPBBAX_2       (* (char xdata *)0xFF51)   // Input Endpoint 2 X-Buffer Base-address
#define IEPBCTX_2       (* (char xdata *)0xFF52)   // Input Endpoint 2 X Byte Count
#define IEPBBAY_2       (* (char xdata *)0xFF55)   // Input Endpoint 2 Y-Buffer Base-address
#define IEPBCTY_2       (* (char xdata *)0xFF56)   // Input Endpoint 2 Y Byte Count
#define IEPSIZXY_2      (* (char xdata *)0xFF57)   // Input Endpoint 2 XY-Buffer Size
```

```
// Input Endpoint 3: configuration
#define IEPCNF_3        (* (char xdata *)0xFF58)   // Input Endpoint 3 Configuration
#define IEPBBAX_3       (* (char xdata *)0xFF59)   // Input Endpoint 3 X-Buffer Base-address
#define IEPBCTX_3       (* (char xdata *)0xFF5A)   // Input Endpoint 3 X Byte Count
#define IEPBBAY_3       (* (char xdata *)0xFF5D)   // Input Endpoint 3 Y-Buffer Base-address
#define IEPBCTY_3       (* (char xdata *)0xFF5E)   // Input Endpoint 3 Y Byte Count
#define IEPSIZXY_3      (* (char xdata *)0xFF5F)   // Input Endpoint 3 XY-Buffer Size

/*-------------------------------------------------------------------------+
| INTERNAL MEMORY MAPPED REGISTERS (MMR, XDATA range = FF80...FFFF
+-------------------------------------------------------------------------*/

// Endpoint 0 Descriptor Registers
#define IEPCNFG_0       (* (char xdata *)0xFF80)   // Input Endpoint Configuration Register
#define IEPBCNT_0       (* (char xdata *)0xFF81)   // Input Endpoint 0 Byte Count
#define OEPCNFG_0       (* (char xdata *)0xFF82)   // Output Endpoint Configuration Register
#define OEPBCNT_0       (* (char xdata *)0xFF83)   // Output Endpoint 0 Byte Count

// Interrupt Registers
#define INTCFG          (* (char xdata *)0xFF84)   // Interrupt P2 delay

// Miscellaneous Registers
#define MCNFG           (* (char xdata *)0xFF90)   // MCU Configuration Register

#define VECINT          (* (char xdata *)0xFF92)   // Vector Interrupt Register
#define WDCSR           (* (char xdata *)0xFF93)   // Watchdog Timer, Control & Status Register

// Pull-up enable Registers
#define PUR0        (* (char xdata *)0xFF94)   // Pull-up control register 1=Enabled  0=Disabled
#define PUR1        (* (char xdata *)0xFF95)   // Pull-up control register 1=Enabled 0=Disabled
#define PUR2        (* (char xdata *)0xFF96)   // Pull-up control register 1=Enabled 0=Disabled
#define PUR3        (* (char xdata *)0xFF97)   // Pull-up control register 1=Enabled 0=Disabled

// I2C Registers
#define I2CSTA          (* (char xdata *)0xFFF0)   // I2C Status and Control Register
#define I2CDAO          (* (char xdata *)0xFFF1)   // I2C Data Out Register
#define I2CDAI          (* (char xdata *)0xFFF2)   // I2C Data In Register
#define I2CADR          (* (char xdata *)0xFFF3)   // I2C Address Register

// VID/PID selection Register
#define VIDSTA          (* (char xdata *)0xFFF6)   // VID/PID status register

// USB Registers
#define USBCTL          (* (char xdata *)0xFFFC)   // USB Control Register
#define USBMSK          (* (char xdata *)0xFFFD)   // USB Interrupt Mask Register
#define USBSTA          (* (char xdata *)0xFFFE)   // USB Status Register
#define FUNADR          (* (char xdata *)0xFFFF)   // This register contains the device
function address.

#endif
```

### D.1.7 Prog.h

Description: header file for device functions

```
-------------------------------------------------------------------------------
#ifndef PROG_H
#define PROG_H

//*****************************************************************************
// Enumeration Definition
//*****************************************************************************

typedef enum
{
   NOTHING,
   WRITING
} tPROGR_MODE;

//*****************************************************************************
// Prototypes
//*****************************************************************************
void ResetDevice(void);
void DecodeDeviceData(unsigned char * Data);

#endif
```

### D.1.8 Application.h

```
-- Description: header file for application functions
-------------------------------------------------------------------------------

#ifndef _Application_H
#define _Application_H

//*****************************************************************************
// Prototypes
//*****************************************************************************

void Read_Data(void);
void Write_Data(void);

#endif
```

## D.2 *.C Files

### D.2.1 Usbinit.c

-- Description: USB init functions
---

```
#include "reg52.h"
#include "Tusb3210.h"
#include "Usb.h"
#include "Usbinit.h"

//***************************************************************************
// extern variables
//***************************************************************************
extern unsigned int BytesRemainingOnIEP0;
extern unsigned int BytesRemainingOnOEP0;
extern unsigned char StatusAction;
extern unsigned char ConfigurationNumber;
extern unsigned char InterfaceNumber;
extern unsigned char deviceReady;

//***************************************************************************
// Initializes the USB function and all registers
// if this function is called, the function is disconnected from usb
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void InitializeUsbFunction(void)
{
IT0 = 0;    // Edge-triggered interrupt EX0
EX0 = 1;   // Enable external 0 interrupt (USB interrupt source)
EA = 1;     // Enable global interrupts
UsbReset(); // Reset the USB Function
// now enable the pull up to enumerate the device on usb
USBCTL |= USBCTL_CONT;
}

//***************************************************************************
// Description: This initializes or resets the USB function
//***************************************************************************
void UsbReset(void)
{
deviceReady = 0;                        // Device is not currently ready
   ConfigurationNumber = 0x00;   // device is unconfigured
   InterfaceNumber = 0x00;
   FUNADR = 0x00;                    // no device address
   BytesRemainingOnIEP0 = NO_MORE_DATA;
   BytesRemainingOnOEP0 = NO_MORE_DATA;
   StatusAction = STATUS_ACTION_NOTHING;

   // enable endpoint 0 interrupt
   IEPCNFG_0 = EPCNF_USBIE | EPCNF_UBME;
   OEPCNFG_0 = EPCNF_USBIE | EPCNF_UBME;
```

```
   // NAK both endpoints
   IEPBCNT_0  = EPBCNT_NAK;
   OEPBCNT_0  = EPBCNT_NAK;
   // enable input Endpoint 1 interrupt and set configurations
   // only use primary (X) buffer
   IEPCNF_1 = EPCNF_USBIE | EPCNF_UBME;
   IEPBBAX_1 = (unsigned char)(IEP1_X_BUFFER_ADDRESS >> 3 & 0x00FF);
   IEPBCTX_1 = EPBCNT_NAK;       // no data
   IEPSIZXY_1 = EP1_MAX_PACKET_SIZE;
    // Enable the USB-specific Interrupts; SETUP, RESET and STPOW
   USBMSK = USBMSK_STPOW | USBMSK_SETUP | USBMSK_RSTR | USBMSK_RESR |
USBMSK_SUSR;
}
```

### D.2.2 Usb.c

```
-- Description: USB functions
/*-------------------------------------------------------------------------------------------------
Some function used are from the Texas InstrumentTUSB2136 Generic Keyboard
Demo Program (Texas Instruments, 2000).
-------------------------------------------------------------------------------------------------*/

#include "reg52.h"
#include "Tusb3210.h"
#include "Usb.h"
#include "Usbinit.h"
#include "Prog.h"
#include "Descriptor.h"
#include <string.h>

//***************************************************************************
// external variables
//***************************************************************************

//***************************************************************************
// global variables
//***************************************************************************
unsigned char HostAskMoreDataThanAvailable;     // If host ask more data then TUSB3210 has
                                                // It will send one zero-length packet
                                                // if the asked lenght is a multiple of
                                                // max. size of endpoint 0
unsigned char * pIEP0Buffer;                    // A buffer pointer to input end point 0
                                                // Data sent back to host is copied from
                                                // this pointed memory location
unsigned int BytesRemainingOnIEP0;              // For endpoint zero transmitter only
                                                // Holds count of bytes remaining to be
                                                // transmitted by endpoint 0.  A value
                                                // of 0 means that a 0-length data packet
                                                // A value of 0xFF means that transfer
                                                // is complete
unsigned int BytesRemainingOnOEP0;              // For endpoint zero transmitter only
                                                // Holds count of bytes remaining to be
                                                // received by endpoint 0.  A value
                                        // of 0 means that a 0-length data packe
                                        // A value of 0xFFFF means that transfer is complete
unsigned char * pOEP0Buffer;        // A buffer pointer to output end point 0
                                    // Data sent from host is copied to
                                    // this pointed memory location
                                    // is complete.

unsigned char ConfigurationNumber = 0;  // Set to 1 when USB device has been
                                        // configured, set to 0 when unconfigured
unsigned char InterfaceNumber = 0;      // The interface number selected
unsigned int DeviceFeatures = 0;        // The device features
unsigned char Suspended = 0;            // Indicates whether the device is suspended or not
unsigned char deviceReady = 0;
unsigned char StatusAction;             // Indicates the current state of sending
                                        // receiving data packets.
```

```
//*****************************************************************************
// XDATA buffer locations
//*****************************************************************************
// locations of Descriptors in RAM (user definied)
unsigned char xdata Descriptor[SIZEOF_DEVICE_DESCRIPTOR] _at_
START_OF_USER_BUFFER_ADDRESS;

// location of Endpoint 1 buffers, use only X-buffer (user definied)
unsigned char xdata IEP1Buffer[EP1_MAX_PACKET_SIZE] _at_
IEP1_X_BUFFER_ADDRESS;
unsigned char xdata OEP1Buffer[EP1_MAX_PACKET_SIZE] _at_
OEP1_X_BUFFER_ADDRESS;
// location of Endpoint 0 buffers (fixed by datasheet)
unsigned char xdata OEP0Buffer[EP0_MAX_PACKET_SIZE] _at_ OEP0_BUFFER_ADDRESS;
unsigned char xdata IEP0Buffer[EP0_MAX_PACKET_SIZE] _at_ IEP0_BUFFER_ADDRESS;
// location of setup packet block (fixed by datasheet)
tDEVICE_REQUEST xdata tSetupPacket _at_ EP0_SETUP_ADDRESS;

// location of the data received from host on SET_REPORT
unsigned char xdata DataFromHost[EP1_MAX_PACKET_SIZE] _at_
DATAFROMHOST_ADDRESS;
unsigned char xdata * pDataFromHost = DATAFROMHOST_ADDRESS;

// location of data to be transfered to host by the device
unsigned char xdata DataToHost[EP1_MAX_PACKET_SIZE] _at_ DATATOHOST_ADDRESS;
unsigned char xdata * pDataToHost = DATATOHOST_ADDRESS;

//*****************************************************************************
// Send Configuration value to Host
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbGetConfiguration(void)
{
   BytesRemainingOnIEP0 = 1;
   usbSendDataPacketOnEP0(&ConfigurationNumber);
}


//*****************************************************************************
// Set Configuration from Host and stall output endpoint 0
 // This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbSetConfiguration(void)
{
   OEPCNFG_0 |= EPCNF_STALL;
   ConfigurationNumber = tSetupPacket.bValueL;
   usbSendZeroLengthPacketOnIEP0();
}
```

```
//*****************************************************************************
// The following functions are called at initial device enumeration, and are
// used to obtain the device, configuration, and string descriptors
//*****************************************************************************
void usbGetDeviceDescriptor(void)
{
   OEPBCNT_0 = 0x00;
   BytesRemainingOnIEP0 = SIZEOF_DEVICE_DESCRIPTOR;
   usbSendDataPacketOnEP0(&romDeviceDescriptor);
   // Once the Device Descriptor has been sent, the device can work
deviceReady = 1;
}

//*****************************************************************************
// Get Configuration descriptor
//*****************************************************************************
void usbGetConfigurationDescriptor(void)
{
   OEPBCNT_0 = 0x00;
   BytesRemainingOnIEP0 = SIZEOF_CONFIG_DESC_GROUP;
   usbSendDataPacketOnEP0(&romConfigurationDescriptorGroup);
}

//*****************************************************************************
// Get String descriptor
// sends the wanted descriptor string to the host
// 0: language info
// 1: manufacturer description
// 2: product description
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbGetStringDescriptor(void)
{
  unsigned char Temp;
  unsigned char stringOffset = 0;

   OEPBCNT_0 = 0x00;
switch(tSetupPacket.bValueL)
{
case 0: // LANGUAGE ID
Descriptor[0] = 4;                  // Length of language descriptor ID
Descriptor[1] = DESC_TYPE_STRING;   // LANGID tag
Descriptor[2] = 0x09;       // Low byte of 0x0409 (English)
Descriptor[3] = 0x04;       // High byte of 0x0409 (English)
break;
case 1: // MANUFACTURER DESCRIPTION
Descriptor[stringOffset++] = strlen(mfgDescription) * 2 + 2;   // Length of this string
Descriptor[stringOffset++] = DESC_TYPE_STRING;             // String descriptor type
for(Temp = 0; Temp < strlen(mfgDescription); Temp++)
{
Descriptor[stringOffset++] = mfgDescription[Temp];    // Insert the character from the string
Descriptor[stringOffset++] = 0x00;             // Insert a trailing 00h for Unicode representation
}
break;
```

```
case 2: // PRODUCT DESCRIPTION
Descriptor[stringOffset++] = strlen(prodDescription) * 2 + 2;  // Length of this string
Descriptor[stringOffset++] = DESC_TYPE_STRING;        // String descriptor type
for(Temp = 0; Temp < strlen(prodDescription); Temp++)
{
Descriptor[stringOffset++] = prodDescription[Temp];       // Insert the character from the string
Descriptor[stringOffset++] = 0x00; // Insert a trailing 00h for Unicode representation
}
break;
case 3: // VERSION DESCRIPTION
Descriptor[stringOffset++] = strlen(revDescription) * 2 + 2;      // Length of this string
Descriptor[stringOffset++] = DESC_TYPE_STRING;               // String descriptor type
for(Temp = 0; Temp < strlen(revDescription); Temp++)
{
Descriptor[stringOffset++] = revDescription[Temp];     // Insert the character from the string
Descriptor[stringOffset++] = 0x00;              // Insert a trailing 00h for Unicode representation
}
break;
default :
// default send a zero data packet
Descriptor[0] = 0;
break;
}
  BytesRemainingOnIEP0 = Descriptor[0];
   usbSendDataPacketOnEP0(&Descriptor);
}

//***************************************************************************
// the following 3 functions are used for HID devices
//***************************************************************************
// Get HID descriptor
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void usbGetHIDDescriptor(void)
{
unsigned char Temp;
// Copy the DEVICE DESCRIPTOR from program "ROM" to XRAM
   for(Temp = 0; Temp < SIZEOF_HID_DESCRIPTOR; Temp++)
    Descriptor[Temp] = romConfigurationDescriptorGroup[SIZEOF_CONFIG_DESCRIPTOR +
SIZEOF_INTERFACE_DESCRIPTOR + Temp];
   OEPBCNT_0 = 0x00;
   BytesRemainingOnIEP0 = SIZEOF_HID_DESCRIPTOR;
   usbSendDataPacketOnEP0(&Descriptor);
}

//***************************************************************************
// Send report descriptor to host
//***************************************************************************
void usbGetReportDescriptor(void)
{
   BytesRemainingOnIEP0 = sizeof(romReportDescriptor);
   usbSendDataPacketOnEP0(&romReportDescriptor);
}
```

```
//*****************************************************************************
// The Set_Report request is sent by the host to a typical HID device.
// when the Set_Report setup packet is received, we initiate a
// "Receive Data Packet" sequence since the actual data value will be
// in the following packet on OEP0.
//*****************************************************************************
void usbSetReport(void)
{
   usbReceiveDataPacketOnEP0((unsigned char *)&DataFromHost);
}



//*****************************************************************************
// Send Interface number to host
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbGetInterface(void)
{
   BytesRemainingOnIEP0 = 1;
   usbSendDataPacketOnEP0(&InterfaceNumber);
}



//*****************************************************************************
// Set Interface number which we get from host
// This function is taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbSetInterface(void)
{
   OEPCNFG_0 |= EPCNF_STALL;              // control write without data stage
   InterfaceNumber = tSetupPacket.bIndexL;
   usbSendZeroLengthPacketOnIEP0();
}



//*****************************************************************************
// Send device status to host
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbGetDeviceStatus(void)
{
   BytesRemainingOnIEP0 = 2;
  usbSendDataPacketOnEP0((unsigned char *)&DeviceFeatures);
}
```

```
//*****************************************************************************
// Set remote wake up
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbSetRemoteWakeup(void)
{
USBCTL |= USBCTL_RWE;
DeviceFeatures |= 0x0200;                          // bit 0: self power, bit 1: remote wake up
   OEPCNFG_0 |= EPCNF_STALL;                        // first low, then high byte => 0x0200
   usbSendZeroLengthPacketOnIEP0();
}

//*****************************************************************************
// clear remote wake up
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbClearRemoteWakeup(void)
{
USBCTL &= ~USBCTL_RWE;
DeviceFeatures &= ~0x0200;                    // bit 0: self power, bit 1: remote wake up
   OEPCNFG_0 |= EPCNF_STALL;
   usbSendZeroLengthPacketOnIEP0();
}

//*****************************************************************************
// Send interface status to host
// return 0x00 0x00 (reserved for future use)
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// From Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbGetInterfaceStatus(void)
{
unsigned int StatusBuffer = 0x00;
   OEPCNFG_0 |= EPCNF_STALL;
   BytesRemainingOnIEP0 = 2;
usbSendDataPacketOnEP0((unsigned char *)&StatusBuffer);
}

//*****************************************************************************
// The SetAddress request allows the host to assign an address to this device.
// The device starts with an address of 00h, as do all USB devices, until
// the host specifically assigns it another address
//*****************************************************************************
void usbSetAddress(void)
{
   if (tSetupPacket.bValueL < 128)
   {
      FUNADR = tSetupPacket.bValueL;
      usbSendZeroLengthPacketOnIEP0();
   }
   else usbStallEndpoint0();
}
```

```
//*****************************************************************************
// stop the input Endpoint 1 from sending data to the host
//*****************************************************************************
void usbSetEndpointHalt(void)
{
if (tSetupPacket.bIndexL == 0x81) IEPCNF_1 &= ~EPCNF_UBME;
   usbSendZeroLengthPacketOnIEP0();
}


//*****************************************************************************
// enable the input Endpoint 1 to send data to the host
//*****************************************************************************
void usbClearEndpointHalt(void)
{
if (tSetupPacket.bIndexL == 0x81) IEPCNF_1 |= EPCNF_UBME;
   usbSendZeroLengthPacketOnIEP0();
}


//*****************************************************************************
// Send the status of IEP1 to host
//*****************************************************************************
void usbGetEndpointStatus(void)
{
unsigned int EndpointStatus = 0x0100;
if(tSetupPacket.bIndexL == 0x81 && IEPCNF_1 & EPCNF_UBME) EndpointStatus = 0x0000;
BytesRemainingOnIEP0 = 2;
usbSendDataPacketOnEP0((unsigned char *)&EndpointStatus);
}


/*****************************************************************************
// Any non-standard or unrecognized request will arrive at the following
// function by default.  We automatically stall the endpoint to indicate
// it's an invalid or unrecognized request.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbNonStandardRequest(void)
{
usbStallEndpoint0();
}



//*****************************************************************************
// USB REQUEST TABLE:
// This section of code defines the lookup table, using the structure
// defined in the header file. The values of the constants used in this
// structure are also defined in "usb.h".
// This Table was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
```

```
code tDEVICE_REQUEST_COMPARE tUsbRequestList[] =
{
// STANDARD DEVICE REQUESTS

   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_GET_STATUS,
   0x00,0x00,         // wValue always 0
   0x00,0x00,         // wIndex always 0
   0x02,0x00,         // Length is 2
   0xff,&usbGetDeviceStatus,  // all values must match

// CLEAR DEVICE FEATURE
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_CLEAR_FEATURE,
   FEATURE_REMOTE_WAKEUP,0x00,    // Feature Selector
   0x00,0x00,            // wIndex is 0
   0x00,0x00,            // Length is 0
   0xff,&usbClearRemoteWakeup,   // all values must match

   // SET DEVICE FEATURE
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_SET_FEATURE,
   FEATURE_REMOTE_WAKEUP,0x00,    // Feature Selector
   0x00,0x00,         // wIndex is 0
   0x00,0x00,         // wLength is 0
   0xff,&usbSetRemoteWakeup,     // all values must match

   // SET ADDRESS
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_SET_ADDRESS,
   0xff,0x00,       // Device address
   0x00,0x00,       // wIndex is 0
   0x00,0x00,       // wLength is 0
   0xdf,&usbSetAddress,    // all values, except address must match

   // GET DEVICE DESCRIPTOR
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_GET_DESCRIPTOR,
   0xff,DESC_TYPE_DEVICE,     // Descriptor Type & Index
   0xff,0xff,                 // Zero or Language ID
   0xff,0xff,                 // Descriptor Length
   0xd0,&usbGetDeviceDescriptor,          // only bReqType, bReq and decriptor type must match

   // GET CONFIGURATION DESCRIPTOR
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_GET_DESCRIPTOR,
   0xff,DESC_TYPE_CONFIG,    // Descriptor Type & Index
   0xff,0xff,    // Zero or Language ID
   0xff,0xff,    // Descriptor Length
   0xd0,&usbGetConfigurationDescriptor,// only bReqType, bReq and decriptor type must match

   // GET STRING DESCRIPTOR
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
   USB_REQ_GET_DESCRIPTOR,
   0xff,DESC_TYPE_STRING,    // Descriptor Type & Index
```

```
    0xff,0xff,     // Zero or Language ID
    0xff,0xff,     // Descriptor Length
    0xd0,&usbGetStringDescriptor,          // only bReqType, bReq and decriptor type must match

// SET DESCRIPTOR not implemented

// GET CONFIGURATION
    USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
    USB_REQ_GET_CONFIGURATION,
    0x00,0x00,      // always 0
    0x00,0x00,     // always 0
    0x01,0x00,     // wLength is 1
    0xff,&usbGetConfiguration,   // all values must match

    // SET CONFIGURATION
    USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_DEVICE,
    USB_REQ_SET_CONFIGURATION,
    0xff,0x00,    // configuration value
    0x00,0x00,    // zero
    0x00,0x00,     // wLength is 0
    0xdf,&usbSetConfiguration,   // all values, except configuration value must match

//******************************
// STANDARD INTERFACE REQUESTS
//******************************
    // GET INTERFACE STATUS
    USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD |
 USB_REQ_TYPE_INTERFACE,
    USB_REQ_GET_STATUS,
    0x00,0x00,  // zero
    0xff,0x00,   // interface number
    0x02,0x00, // length is 2
    0xf7,&usbGetInterfaceStatus,   // all values, except interface number must match

// CLEAR/SET FEATURE not implemented

// GET INTERFACE
    USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD |
 USB_REQ_TYPE_INTERFACE,
    USB_REQ_GET_INTERFACE,
    0x00,0x00,   // zero
    0xff,0xff,      // interface number
    0x01,0x00,   // length is 1
    0xf3,&usbGetInterface,   // all values, except interface number must match

// SET INTERFACE FEATURE
    USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD |
 USB_REQ_TYPE_INTERFACE,
 USB_REQ_SET_INTERFACE,
  0xff,0x00,               // Alternative Setting
  0xff,0x00,              // Interface number
  0x00,0x00,             // length is 0
  0xd7,&usbSetInterface,  // all value, except alternate settings and interface number must match
```

```
  // GET HID DESCRIPTOR
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD |
USB_REQ_TYPE_INTERFACE,
   USB_REQ_GET_DESCRIPTOR,
   0xff,DESC_TYPE_HID,  // Descriptor Type & Index
   0xff,0xff,                    // Zero or Language ID
   0xff,0xff,                   // Descriptor Length
   0xd0,&usbGetHIDDescriptor,    // only bReqType, bReq and decriptor type must match

   // GET REPORT DESCRIPTOR
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD |
USB_REQ_TYPE_INTERFACE,
   USB_REQ_GET_DESCRIPTOR,
   0xff,DESC_TYPE_REPORT,   // Report Type(High) and Index(Low)
   0xff,0xff,                     // Interface number
   0xff,0xff,                    // descriptor length
   0xd0,&usbGetReportDescriptor,    // only bReqType, bReq and report type must match



//**********************************
// CLASS SPECIFIC INTERFACE REQUESTS
//**********************************
   // SET REPORT
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_CLASS | USB_REQ_TYPE_INTERFACE,
   USB_REQ_SET_REPORT,
   0xff,0xff,          // Report type and Report ID
   0xff,0xff,          // Interface number
   0xff,0xff,          // Report length
   0xc0,&usbSetReport,    // only bReqType and bReq must match

   // GET REPORT not supported
   // GET/SET IDLE not supported
   // GET/SET PROTOCOL not supported

//******************************
// STANDARD ENDPOINT REQUESTS
//******************************
   // GET ENDPOINT STATUS
   USB_REQ_TYPE_INPUT | USB_REQ_TYPE_STANDARD | USB_REQ_TYPE_ENDPOINT,
   USB_REQ_GET_STATUS,
   0x00,0x00,   // always 0
   0xff,0x00,    // Endpoint number
   0x02,0x00,   // wLength is 2
   0xf7,&usbGetEndpointStatus,    // all values, except endpoint number must match

   // CLEAR ENDPOINT FEATURE
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD |
USB_REQ_TYPE_ENDPOINT,
   USB_REQ_CLEAR_FEATURE,
   FEATURE_ENDPOINT_STALL,0x00,  // Feature selector
   0xff,0x00,                                  // Endpoint number
   0x00,0x00,                                  // wLength is 0
   0xf7,&usbClearEndpointHalt,            // all values, except endpoint number must match
```

```
// SET ENDPOINT FEATURE
   USB_REQ_TYPE_OUTPUT | USB_REQ_TYPE_STANDARD |
USB_REQ_TYPE_ENDPOINT,
   USB_REQ_SET_FEATURE,
   FEATURE_ENDPOINT_STALL,0x00,     // Feature selector
   0xff,0x00,                       // Endpoint number
   0x00,0x00,                       // wLength is 0
   0xf7,&usbSetEndpointHalt,        // all values, except endpoint number must match

// SYNCH FRAME is not implemented

// END OF LIST CATCH-ALL REQUEST:
// This will match any USB request since bCompareMask is 0x00.
   0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,
   0x00,&usbNonStandardRequest
};


//*****************************************************************************
//This function is called when a USB request has been received.  It searches
// the tUsbRequestList[] structure defined in the previous section for a
// request that matches a given entry in the table and, when matched, executes
// the corresponding function.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbDecodeAndProcessUsbRequest(void)
{
   unsigned char Mask, Result, Temp;
   unsigned char * pUsbRequestList;          // save code space

// We initialize the pUsbRequestList pointer to the beginning of the
// tUsbRequestList[] so that we can subsequently traverse the table
// by incrementing the pUsbRequestList pointer.
   pUsbRequestList = (unsigned char *)&tUsbRequestList[0];

// Cycle indefinitely until we've found an entry in the tUsbRequestList[]
// table. Since the last entry in the table has a 0x00 mask, we'll
// *always* find a match, so this cycle will always exit.
   while(1)
   {
      Result = 0x00;
      Mask   = 0x80;

      // We cycle through fields 0 through 7, which correspond to the 8 fields
      // in each entry of tUsbRequestList.  If the given byte in the packet
      // we just receive is equal to the corresponding byte in the table, we
      // set that bit in the result, indicating a byte which matched.  Otherwise,
      // we don't set the bit which means that byte didn't match.
      for(Temp = 0; Temp < 8; Temp++)
      {
         if(*(pEP0_SETUP_ADDRESS + Temp) == *(pUsbRequestList + Temp)) Result |= Mask;
          Mask >>= 1;
      }
```

```
      // At this point, bResult holds 8 bits which indicate whether each of the
      // bytes in the packet matched the corresponding bytes in the tUsbRequestList[]
      // table.  We then AND the mask value in the table with the result so that
      // we only are comparing the bits required in the mask.  If the resulting
      // value is equal to the mask, that means that all significant bytes match.
      // This is done since any bit that is clear in the mask is a "don't care", so
      // the AND makes sure we don't reject a "valid" comparison beause a don't
      // care bit actually matched.
      if((*(pUsbRequestList + Temp) & Result) == *(pUsbRequestList + Temp)) break;

      // If we haven't found a matching entry yet, we advenced the pointer to point
      // to the next entry in the table, and keep looking.
      pUsbRequestList += sizeof(tDEVICE_REQUEST_COMPARE);
   }

   // We check to see if any more setup packet(s) have been received and, if so, we
   // anbandon this one to handle the next one.
   if(USBSTA & (USBSTA_SETUP | USBSTA_STPOW) != 0x00) return;

// If we've reached this point of the function, we've found the function that should
// be called given the current request.  So we call it...
((ptDEVICE_REQUEST_COMPARE)pUsbRequestList)->pUsbFunction();
}



//***************************************************************************
// Sets the STALL flag on both IEP0 and OEP0.
//***************************************************************************
void usbStallEndpoint0(void)
{
   IEPCNFG_0 |= EPCNF_STALL;
   OEPCNFG_0 |= EPCNF_STALL;
}




//***************************************************************************
// Receive a data packet on EP0
// sets buffer address and bytes remaining and counter to 0 => receiving on
// next OUT token
// called when a SET_REPORT token is received.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void usbReceiveDataPacketOnEP0(unsigned char * pBuffer)
{
   pOEP0Buffer = pBuffer;
   BytesRemainingOnOEP0 = (unsigned int)(tSetupPacket.bLengthH << 8) | (unsigned
int)tSetupPacket.bLengthL;
   StatusAction = STATUS_ACTION_DATA_OUT;
   OEPBCNT_0 = 0x00;
}
```

```
//***************************************************************************
// Receive a data packet on EP0
// if not all data has been sent, send the rest now
//***************************************************************************

void usbReceiveNextPacketOnOEP0(void)
{
   unsigned char Index, Byte;
   Byte = OEPBCNT_0 & EPBCT_BYTECNT_MASK;
    // enter if received bytes (should be 8) arrived
   if(BytesRemainingOnOEP0 >= (unsigned int)Byte)
   {
      for(Index = 0; Index < Byte; Index++) *pOEP0Buffer++ = OEP0Buffer[Index];
      BytesRemainingOnOEP0 -= (unsigned int)Byte;
      // if we are waiting for more data, wait for the next OUT token
      if(BytesRemainingOnOEP0 > 0)
     {
         OEPBCNT_0 = 0x00;
         StatusAction = STATUS_ACTION_DATA_OUT;
     }
// if all bytes are received, stall endpoint
      else
     {
         OEPCNFG_0 |= EPCNF_STALL;
         StatusAction = STATUS_ACTION_NOTHING;
        // data has been received here
        // now decode the packet
        DecodeDeviceData(&DataFromHost);
      }
   }
// if to much data has been sent, stall endpoint
   else
   {
   OEPCNFG_0 |= EPCNF_STALL;
   StatusAction = STATUS_ACTION_NOTHING;
   }
}



//***************************************************************************
// Sends a 0-length packet back to the host on IEP0. Often called to
// acknowledge a packet received from the host that requires no data in the
// reply, just an acknowledgement of receipt.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void usbSendZeroLengthPacketOnIEP0(void)
{
   BytesRemainingOnIEP0 = NO_MORE_DATA;
   StatusAction = STATUS_ACTION_NOTHING;
   IEPBCNT_0 = 0x00;
}
```

```
//*****************************************************************************
// Send data packet to EP0
// used for sending descriptors and other requests to the host
// length of data is defined in the setup packet.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************

void usbSendDataPacketOnEP0(unsigned char * pBuffer)
{
   unsigned int Temp;
   pIEP0Buffer = pBuffer;
   Temp = (unsigned int)(tSetupPacket.bLengthH << 8) | (unsigned int)tSetupPacket.bLengthL;
   // Limit transfer size to Length if needed
   // this prevent USB device sending 'more than require' data back to host
   if(BytesRemainingOnIEP0 >= Temp)
   {
      BytesRemainingOnIEP0 = Temp;
      HostAskMoreDataThanAvailable = 0;
   }
       else HostAskMoreDataThanAvailable = 1;

       usbSendNextPacketOnIEP0();
}


//*****************************************************************************
// Send following packets to IEP0 if not all data has been transfered
// with the last IN token.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//*****************************************************************************
void usbSendNextPacketOnIEP0(void)
{
   unsigned char PacketSize, Index;

   // First check if there are bytes remaining to be transferred
   if(BytesRemainingOnIEP0 != NO_MORE_DATA)
   {
      if(BytesRemainingOnIEP0 > EP0_MAX_PACKET_SIZE)
                                                                                {
         // More bytes are remaining than will fit in one packet
         // there will be More IN Stage
         PacketSize = EP0_MAX_PACKET_SIZE;
         BytesRemainingOnIEP0 -= EP0_MAX_PACKET_SIZE;
         StatusAction = STATUS_ACTION_DATA_IN;
      }
      else if (BytesRemainingOnIEP0 < EP0_MAX_PACKET_SIZE)
      {
         // The remaining data will fit in one packet.
         // This case will properly handle wBytesRemainingOnIEP0 == 0
         PacketSize = (unsigned char)BytesRemainingOnIEP0;
         BytesRemainingOnIEP0 = NO_MORE_DATA;       // No more data need to be Txed
         StatusAction = STATUS_ACTION_NOTHING;
      }
```

```
      else
      {
         PacketSize = EP0_MAX_PACKET_SIZE;
         if(HostAskMoreDataThanAvailable == 1)
         {
            BytesRemainingOnIEP0 = 0;
            StatusAction = STATUS_ACTION_DATA_IN;
         }
         else
         {
            BytesRemainingOnIEP0 = NO_MORE_DATA;
            StatusAction = STATUS_ACTION_NOTHING;
         }
      }

      for(Index = 0; Index < PacketSize; Index++) IEP0Buffer[Index] = *pIEP0Buffer++;
      IEPBCNT_0 = PacketSize;
                                                                                    }
   else StatusAction = STATUS_ACTION_NOTHING;
}




//*****************************************************************************
// Send the buffer DataToHost on EP1
// this data is data from reading a device, status,....
// there is no defined length, because this function is not called
// because of the setup packet
// but length is defined inthe report descriptor (64 Bytes)
//*****************************************************************************
void usbSendDataToHostOnEP1(void)
{
unsigned char Index;

// wait for the NAK
Index = IEPBCTX_1;
while(Index != EPBCNT_NAK) Index = IEPBCTX_1;
for(Index = 0; Index < EP1_MAX_PACKET_SIZE; Index++) IEP1Buffer[Index] =
DataToHost[Index];
   IEPBCTX_1 = Index;
}
```

```
//***************************************************************************
// This function is called by the UsbInterrupt function when
// a setup packet is received.  This function immediately sets both
// OEP0 and IEP0 to a NAK state, sets the USBCTL to send/receive based
// on the direction of the request, then proceeds to call the
// usbDecodeAndProcessUsbRequest() function which determines which
// function should be called to handle the given USB request.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void SetupPacketInterruptHandler(void)
{
   // Hardware clears STALL in both data endpoints once valid setup packet is
   // received.  NAK both data endpoints.
   IEPBCNT_0  = EPBCNT_NAK;
   OEPBCNT_0  = EPBCNT_NAK;
   USBSTA = USBSTA_SETUP; // from now, hardware will refer NAK bit in I/OEPBCNT
   // Copy the MSB of bmRequestType to DIR bit of USBCTL to indicate the
   // direction of the transfer.
   if((tSetupPacket.bmRequestType & USB_REQ_TYPE_INPUT) == USB_REQ_TYPE_INPUT)
      USBCTL |= USBCTL_DIR;
   else
   USBCTL &= ~USBCTL_DIR;
// Clear the bStatusAction to indicate that, at this point, nothing is
// happening (it may be set to DATA_OUT by specific functions that
// expect a DATA packet following the setup packet).
   StatusAction = STATUS_ACTION_NOTHING;
// Call the function that determines which function should be called to
// handle the specific USB request.
   usbDecodeAndProcessUsbRequest();
}



//***************************************************************************
// This function is called by the UsbInterrupt function when
// a USB interrupt is called by OEP0.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void OEP0InterruptHandler(void)
{
// We clear the IEP0 byte count since we have nothing to send out.
   IEPBCNT_0 = 0x00;
// We now handle the interrupt based on the bStatusAction condition.
// If we are in a DATA_OUT condition, we call the usbReceiveNextPacketOnEP0
// function to copy the data payload to its correct buffer.  If we are
// not expecting any data on OEP0, we set the stall flag to stall the
// endpoint and abort any additional data that may otherwise be
// sent.
   if(StatusAction == STATUS_ACTION_DATA_OUT)
   usbReceiveNextPacketOnOEP0();    // Handle this data packet
   else
   OEPCNFG_0 |= EPCNF_STALL;   // We weren't expecting data
}
```

```
//***************************************************************************
// This function is called by the UsbInterrupt function when
// a USB interrupt is caused by IEP0. This will happen once the data
// sent by calling usbSendNextPacketOnIEP0 and means the previous data
// packet has been sent. At that point, there are two conditions:
// either there is more data to send or there isn't. If there is, we
// call usbSendNextPacketOnIEP0 to send the next packet of data.  If
// there isn't anymore data, we stall. However, if the bStatusAction
// condition indicates that we were changing the devices address, we
// do so at this point.
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void IEP0InterruptHandler(void)
{
// We clear the OEP0 byte count since we are not expecting any data.
   OEPBCNT_0 = 0x00;
// We now handle the interrupt based on the bStatusAction condition.
// If we are in a DATA_IN condition, we call the usbSendNextPacketOnIEP0
// function to send the next data payload packet.  If we are in a
// Set Address mode, we modify the address.  In any other case, we've
// sent all the data we had to send, so we stall the endpoint to indicate
// there is no more data to send.
   if(StatusAction == STATUS_ACTION_DATA_IN) usbSendNextPacketOnIEP0();
   else IEPCNFG_0 |= EPCNF_STALL;    // No more data to send
}
```

```
//***************************************************************************
// This function is called by the UsbInterrupt function when
// a USB interrupt is caused by IEP1. This will happen once the data
// sent by calling usbSendDataPacketOnEP1 and means the previous data
// packet has been sent.
//***************************************************************************
void IEP1InterruptHandler(void)
{
OEPBCNT_0 = 0x00;
}
```

```
//***************************************************************************
// Interrupt Service Routine for EX0
// This function was taken from the TUSB2136 Generic Keyboard Demo Program
// from Texas Instruments (Texas Instruments, 2000) .
//***************************************************************************
void EX0_int(void) interrupt IE0_VECTOR       // External Interrupt 0
{
   EA = 0;                    // Disable any further interrupts

   switch (VECINT)
```

```
{                   // Identify Interrupt ID
      case VECINT_OUTPUT_ENDPOINT0:
         VECINT  = 0x00;
         OEP0InterruptHandler();
         break;

      case VECINT_INPUT_ENDPOINT0:
         VECINT  = 0x00;
         IEP0InterruptHandler();
         break;

      case VECINT_INPUT_ENDPOINT1:
         VECINT  = 0x00;
         IEP1InterruptHandler();
         break;

      case VECINT_STPOW_PACKET_RECEIVED:
         VECINT  = 0x00;
         // clear setup packet flag
         USBSTA = USBSTA_STPOW;
         SetupPacketInterruptHandler();
         break;

      case VECINT_SETUP_PACKET_RECEIVED:
         VECINT  = 0x00;
         // clear setup packet flag
         USBSTA = USBSTA_SETUP;
         SetupPacketInterruptHandler();
         break;

      case VECINT_RSTR_INTERRUPT:
         VECINT  = 0x00;
         // clear reset flag
         USBSTA = USBSTA_RSTR;
         UsbReset();
         break;

     case VECINT_RESR_INTERRUPT:
        VECINT = 0x00;
        USBSTA = USBSTA_RESR;
        Suspended = 0;
        break;

     case VECINT_SUSR_INTERRUPT:
      VECINT = 0x00;
      USBSTA = USBSTA_SUSR;
      Suspended = 1;
      break;

      default:
     VECINT = 0x00;
      break;              // unknown interrupt ID
   }
   EA = 1;              // Enable the interrupts again
}
```

### D.2.3 Delay.c

```
-- Description: delay time in ms
-------------------------------------------------------------------------
#include "Delay.h"
#include "reg52.h"
#include <intrins.h>

#define SYS_CLOCK   48


//*************************************************************************
// Delay with time * 1ms
// with 48MHz ossillator
//*************************************************************************

void Delay_ms(unsigned int time)
{
TMOD = 0x01;    // Timer 0 16 bit Mode
TR0 = 0;
while(time--)
{
      TL0 = (65536L-1000L*SYS_CLOCK/12) % 256L;
      TH0 = (65536L-1000L*SYS_CLOCK/12) / 256L;
      TR0 = 1;              // start timer
      while(!TF0);
      TF0 = 0;
   }
   TR0 = 0;
}

//*************************************************************************
// Delay with 5us
// with 48MHz ossillator
//*************************************************************************
void Delay_5us(void)
{
  nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
_nop_(); _nop_();
}
```

### D.2.4 Application.c

```
-- Description: Application programming functions
------------------------------------------------------------------------*/
#include "reg52.h"
#include "Application.h"
#include "Prog.h"
#include "Delay.h"
#include "Usb.h"
#include "Tusb3210.h"
#include <intrins.h>
//***************************************************************************
// extern variables
//***************************************************************************
extern tPROGR_MODE ProgMode;
extern unsigned char xdata * pDataToHost;
extern unsigned char xdata * pDataFromHost;



//***************************************************************************
// Writing data to port P1
// DataFromHost[2]: Bytes to write from the actual packet
// DataFromHost[3]: 0: the last packet was sent, not 0: more data will come
//***************************************************************************
void Write_Data(void)
{
unsigned char Counter;

// do this only one time if writing to port
if (ProgMode == NOTHING)
{
Delay_ms(10);
ProgMode = WRITING;

}

// device receives always max 59 bytes of data
// 64 bytes are sent, but 1 for device info, 1 for mode,
// 1 byte for size info and 2 bytes for other info => max 59 bytes data

for (Counter = 0; Counter < *(pDataFromHost+2); Counter++)
 {
// apply data on Port1
P1 = *(pDataFromHost+Counter+5);
Delay_5us();
_nop_();
while(P2_2 == 0);          // wait for button to be pressed (to go High)
                           // indicating  ready to receive next byte
//while(P2_2 == 1);          // continue reading port without interrupt
  _nop_();
  _nop_();
Delay_ms(60);
 } // end for
```

```
// if the last packet was sent, exit writing
if (*(pDataFromHost+3) == 0) // Buf[4]
{
// now power off
ResetDevice();
ProgMode = NOTHING;
*pDataToHost = 'w';
usbSendDataToHostOnEP1();
                                                                                    }
}



//*****************************************************************************
// Read data from P1
// DataFromHost[2]: Max Bytes to read from port
//*****************************************************************************
void Read_Data(void)
{
unsigned int Counter, ActualBytesToRead, MaxBytesToRead;
unsigned char BytesToSend;

MaxBytesToRead = (*(pDataFromHost+2));    // Buf[3]
ActualBytesToRead = (*(pDataFromHost+4)); // Buf[5]
Delay_5us();
_nop_();

// P1 to high to enable reading
P1 = 0xFF;

BytesToSend = 0;
for (Counter = 0; Counter < MaxBytesToRead; Counter++)
{
// read a byte
*(pDataToHost+BytesToSend+1) = P1;
BytesToSend++;
// if the buffer is full, send a packet to host
// max data is 63 bytes + (1 byte for size)
if ((BytesToSend == 63) || (BytesToSend == ActualBytesToRead))
  {
   // now send the data to host on next IN token
   // buffer[0] is size info
   *pDataToHost = BytesToSend;
   usbSendDataToHostOnEP1();
   BytesToSend = 0;

  } // enf if
 _nop_();
 _nop_();
} // end for
 if ((BytesToSend == 63) || (BytesToSend == ActualBytesToRead))
break;
} // end for
```

```
if (BytesToSend != 0)
  {
    // finally send the rest of the readbuffer if size < 63 Bytes
   *pDataToHost = BytesToSend;
   usbSendDataToHostOnEP1();
   }
} // end function
```

### D.2.5 Prog.c

-- Description: Main programme functions
---

```
#include "reg52.h"
#include "Tusb3210.h"
#include "Prog.h"
#include "Usb.h"
#include "Delay.h"
#include "application.h"


//-----------------------------------------------------------------------------
// extern variables
//-----------------------------------------------------------------------------
extern unsigned char xdata * pDataToHost;
extern unsigned char xdata * pDataFromHost;

//-----------------------------------------------------------------------------
// global variables
//-----------------------------------------------------------------------------
tPROGR_MODE ProgMode;

//-----------------------------------------------------------------------------
// Initializes the Device ports and enable pull ups
//-----------------------------------------------------------------------------
void ResetDevice(void)
{
// initialize Ports
PUR0 = 0;     // enable pullups for Port0
PUR1 = 0;     // enable pullups for Port1
PUR2 = 0;     // enable pullups for Port2
PUR3 = 0;     // enable pullups for Port3

P0 = 0x00;     // initialize ports
P1 = 0x00;
P3 = 0x00;
}


//-----------------------------------------------------------------------------
// Decode the data from host to read port or write to port
// first Byte indicates read/write
// second Byte indicates mode (read,write)
//-----------------------------------------------------------------------------
void DecodeDeviceData(unsigned char * Data)
{
// select read/write
switch (Data[0])
{

// 1: TUSB3210 selected
case 0x01:
```

```
switch (Data[1])
{
// 1: read selected
case 0x01 : Read_Data(); break;
// 2: write selected

case 0x02: Write_Data(); break;
}
break;


default: break;
}

// tell host, that data has been Written
// and new data can be send
if (ProgMode == WRITING )
{
*pDataToHost = '*';
usbSendDataToHostOnEP1();
}
// else power off
else ResetDevice();
```

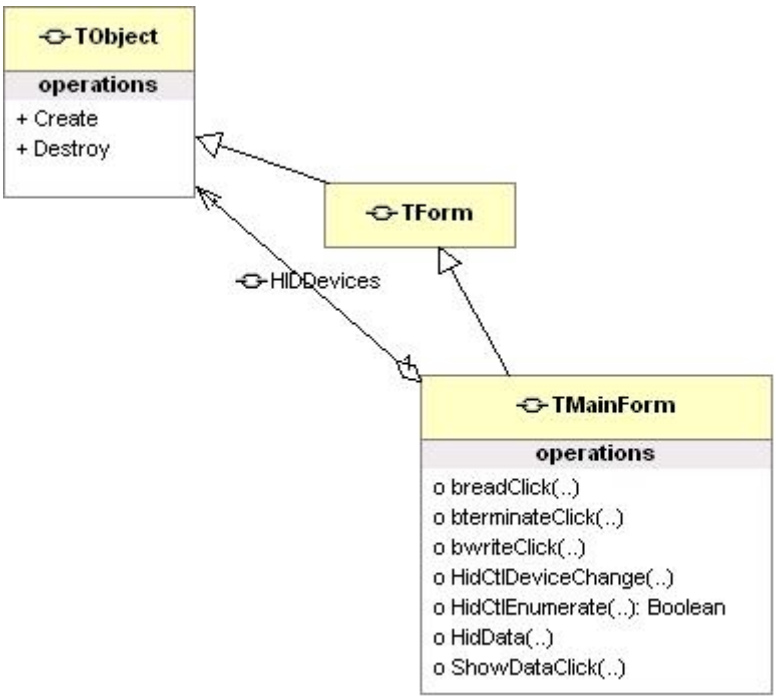
### D.2.6 Main.c

```
-- Description: main program
-----------------------------------------------------------------------

#include "reg52.h"
#include "Tusb3210.h"
#include "Usbinit.h"
#include "Usb.h"
#include "Prog.h"

//***************************************************************************
// extern variables
//***************************************************************************
extern unsigned char deviceReady;
extern tPROGR_MODE ProgMode;

//***************************************************************************
// Main routine to initialize the TUSB3210
//***************************************************************************
void main()
{
// misc. settings
   MCNFG = (MCNFG_XINT | MCNFG_SDW);              // SDW is already set by boot program
WDCSR = 0x00;    // disable watchdog
INTCFG = 0x02;   // P2 Interrupt delay in ms (here 2ms)
INTCFG = 0x01;   // P2 Interrupt delay in ms (here 1ms)
ProgMode = NOTHING;    // actual state: nothing

// initialize device pins
ResetDevice();

// initialize USB registers
InitializeUsbFunction();

while(1);
}
```

# APPENDIX E

## PC SAMPLE SOURCE CODE

### E.1 UML Diagram




### E.2 PC Software Source Code.

```
procedure TMainForm.HidCtlDeviceChange(Sender: TObject);
var
Dev: TJvHidDevice;
Counter: Byte;
begin
  CurrentDevice := nil;
    // if not found , disable buttons
    DisableButtons;
    StatusBar.SimpleText := 'USB HID Device not found';

if HIDDevCount > 0 then
begin
for Counter := 0 to HIDDevCount-1 do
    begin
      Dev := TJvHidDevice(HIDDevices[Counter]);
      Dev.Free;
    end;
 HIDDevCount := 0;
```

```
   end;
  HidCtl.Enumerate;
end;


function TMainForm.HidCtlEnumerate(const HidDev: TJvHidDevice;
  const Idx: Integer): Boolean;
var
  Dev: TJvHidDevice;
begin
   HidCtl.CheckOutByIndex(Dev, Idx);
   HIDDevices[HIDDevCount] := Dev;
   inc(HIDDevCount);

   // TUSB3210 Device has these values
   // Detect the Device and enable buttons
   // if the correct VID/PID is found
  if (HidDev.Attributes.VendorID = $0451) and
     (HidDev.Attributes.ProductID = $3210)  then
     begin
       EnableButtons;
       StatusBar.SimpleText := 'USB HID Device is connected';
       CurrentDevice := Dev;
     end;
   Result := true;

end;


//**************************************************************
// The Buffer sends to the Device 64 Bytes + one Byte Report ID
//**************************************************************
  Procedure TMainForm.WriteBufferToDevice;
  var
  Written: Cardinal;
  ToWrite: Cardinal;
begin
if Assigned(CurrentDevice) then
  begin
   // report ID is always zero
    Buf[0] := 0;

       //*******************************
       // Buf[1]: device information
       // 1: TUSB3210
       // 2,3,4, etc..: Other connected
       //             Devices If any
       //*******************************

Buf[1] := 1 ; // TUSB3210
    ToWrite := CurrentDevice.Caps.OutputReportByteLength;
    CurrentDevice.WriteFile(Buf, ToWrite, Written);
   end;
 end;
```

```
//***************************************************************************
// Preparing  the data to be send to the Device as a 64 Bytes packet .
//***************************************************************************
procedure TMainForm.SendDataPacketToDevice;
var
ByteCounter, MaxPacketSize: byte;

begin

  // Buf[5] is also a general information Byte
  // =>  data starts at Buf[6]

     MaxPacketSize := 59;
     datalength := StrToInt(BytesToSend.Text);

      // initialize the Buffer
      for ByteCounter := 1 to MaxPacketSize do
      Buf[ByteCounter+5] := 0;

    // Filling the data from the list (Filled by the user)
       for ByteCounter := 1 to MaxPacketSize do
       begin
        if Edits[ByteCounter].Text <> ' ' then
        begin
          Buf[ByteCounter+5] :=  StrToIntDef('$'+ Edits[ByteCounter-1].Text,0);
          Edits[ByteCounter-1].Text := Format('%.2x', [Buf[ByteCounter+5]]);;
         inc(SentBytes);
         if datalength = SentBytes then break;
       end;
    end;

   // Buf[3]: Actual byte count in packet
   // Buf[4]= (0): Indicates that the last packet was sent
   // Buf[4]= (1): Indicates that more data will follow

   if datalength = SentBytes then
     begin

      Buf[3] := SentBytes; // actual Byte count
      Buf[4] := 0;          // last packet was sent (end of data)
     end
       else
         begin
          Buf[3] := SentBytes;
          Buf[4] := 1; // more data will follow
         end;

    WriteBufferToDevice;
end;
```

```
//***************************************
// Receiving the data sent by the device
//***************************************
procedure TMainForm.HidData(const HidDev: TJvHidDevice; ReportID: Byte;
 const Data: Pointer; Size: Word);
var
Counter : Byte;
begin
   // writing  of the last data packet has finished =>
   // send next data packet if a '*' is received. That means,
   // that the device has received the last data packet
   // and is ready to receive a new data packet
    if not (Status = READ) then
      if Byte(PChar(Data)[0]) = ord('*') then SendDataPacketToDevice;
      if ((ButtonClicked = B_READ) or (Status = READ))then
        begin
            for Counter:= 1 to Byte(PChar(Data)[0]) do
            readbuffer := readbuffer + Format ('%.2x ',[Cardinal(PChar(Data)[Counter])]);
        //if ((length(readbuffer) = MaxReadBytes) or (length(readbuffer) = ActualReadBytes)) then
            HistoryListBox.ItemIndex := HistoryListBox.Items.Add(readbuffer);
            Status := NOTHING;
       end
        else if ButtonClicked = B_WRITE then WriteDevice
          else Status := NOTHING ;

       // if last packet was sent to device stop sending data
       if (Status = WRITE) and (Byte(PChar(Data)[0])= ord('w')) then
      if ButtonClicked = B_WRITE then Status := NOTHING;
      // if just detecting device, then exit
      if  Status = NOTHING then
        Status := NOTHING ;
        // enable buttons
        EnableButtons;
end;


//*************************************
// Write to the Device
//*************************************
Procedure TMainForm.WriteDevice;
var
 Str : string;
 I : integer;
begin
 StatusBar.SimpleText := ' Writing to port...';
 Status := WRITE;
   Buf[2] := 2;  // select mode (writing)
   Buf[5] := 0;  // not used in writing mode
   SentBytes := 0;
   SendDataPacketToDevice;
      Str := Format('W %.2x  ', [Buf[6]]); // first data byte
      for I := 7 to (datalength+5) do
```

```
      Str := Str + Format('%.2x  ', [Buf[I]]);
      HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str);
 end;
//************************************
// Read the Device
//************************************
 Procedure TMainForm.ReadDevice;
 begin
     StatusBar.SimpleText := ' Reading from port...';
     Status := READ;
   Buf[2] := 1; // select mode (reading)
   MaxReadBytes := 63; // one byte for size information
   ActualReadBytes := StrToInt(BytesToRead.Text); // actual Byte count
      Buf[3] := MaxReadBytes ; // data size
      Buf[5] := ActualReadBytes; // actual Byte count
      readbuffer := '';
      WriteBufferToDevice;
end;


 procedure TmainForm.ShowBufferContents;
 var
  Str,Str1,Str2,Str3,Str4 : string;
  I : integer;
 begin
   Str1 := 'Buf[0]= Report ID , Buf[1]= Device(TUSB3210) , Buf[2]= Mode';
   Str2 := 'Buf[3]= Data Size , Buf[4]= 0:No more data ,1:More data will';
   Str3 := 'follow , Buf[5]= General Purpose, Buf[6..64]= Dtata';
   Str4 := '------------------------------------------------------------'+
           '-----------------------------------';
   HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str1);
   HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str2);
   HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str3);
   HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str4);
   StatusBar.SimpleText := ' Display Contents of Buffer ...';
   Str := Format('Buf %.2x  ', [Buf[0]]); // first data byte
   for I := 1 to (datalength+5) do
   Str := Str + Format('%.2x  ', [Buf[I]]);
   HistoryListBox.ItemIndex := HistoryListBox.Items.Add(Str);
  end;

procedure TMainForm.HidCtlDeviceDataError(const HidDev: TJvHidDevice;
  Error: Cardinal);
 begin
  HistoryListBox.ItemIndex  :=  HistoryListBox.Items.Add(Format('READ  ERROR:  %s  (%x)',
[SysErrorMessage(Error), Error]));
 end;
procedure TMainForm.breadClick(Sender: TObject);
begin
   ButtonClicked := B_READ;
   if (BytesToRead.Text <> '' ) then
   ReadDevice
    else
     begin
       Status := NOTHING;
```

```
      StatusBar.SimpleText := 'To Read from the port , Enter the Number of Bytes to Read';
    end;
end;

procedure TMainForm.bwriteClick(Sender: TObject);
begin
 ButtonClicked := B_Write;
 if ((BytesToSend.Text <> '') and (StrToInt(BytesToSend.Text) <= 59)) then
   WriteDevice
   else
   begin
     Status := NOTHING;
     StatusBar.SimpleText := 'To write to the port ,Enter the Number of Bytes to Send ( Maximum
59 Bytes)';
  end;
end;


procedure TMainForm.bterminateClick(Sender: TObject);
begin
  Application.Terminate;
end;


procedure TMainForm.SaveBtnClick(Sender: TObject);
begin
    ForceCurrentDirectory := True;
    if SaveDialog.Execute then
    HistoryListBox.Items.SaveToFile(SaveDialog.FileName);
end;



// ******************************
// ****** disable buttons ********
// ******************************
Procedure TMainForm.DisableButtons;
 begin
    bwrite.Enabled := false;
    bread.Enabled := false;
 end;


// ******************************
// ****** enable buttons ********
// ******************************
 Procedure TMainForm.EnableButtons;
 begin
     bwrite.Enabled := true ;
     bread.Enabled := true;
 end;
procedure TMainForm.FormActivate(Sender: TObject);
var
  I, J: Integer;
begin
```

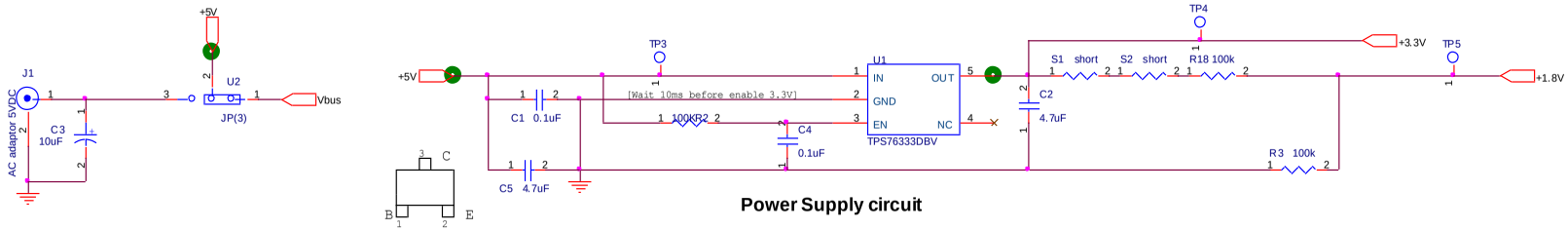
```
  Edits[0] := Edit1;
  for I := 1 to High(Edits) do
    Edits[I] := TEdit.Create(Self);
  for J := 0 to 3 do
    for I := 0 to 14 do
      with Edits[J*15 + I] do
      begin
        Visible  := True;
        Left     := Edit1.Left + I*(Edit1.Width+2);
        Top      := Edit1.Top  + J*(Edit1.Height+4);
        Width    := Edit1.Width;
        Anchors  := Edit1.Anchors;
        //Edits[58].Visible := False;
        Edits[59].Visible := False;
        if not Assigned(Parent) then
          Parent := Edit1.Parent;
        TabOrder := Edit1.TabOrder + J*15 + I;
      end;
  HidCtl.OnDeviceChange := HidCtlDeviceChange;
end;



procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
  Status := NOTHING;
  HistoryListBox.Items.Clear;
  StatusBar.SimpleText := ' Device is Ready...';
end;



procedure TMainForm.BuffBtnClick(Sender: TObject);
begin
  ShowBufferContents;
end;
```

# APPENDIX F

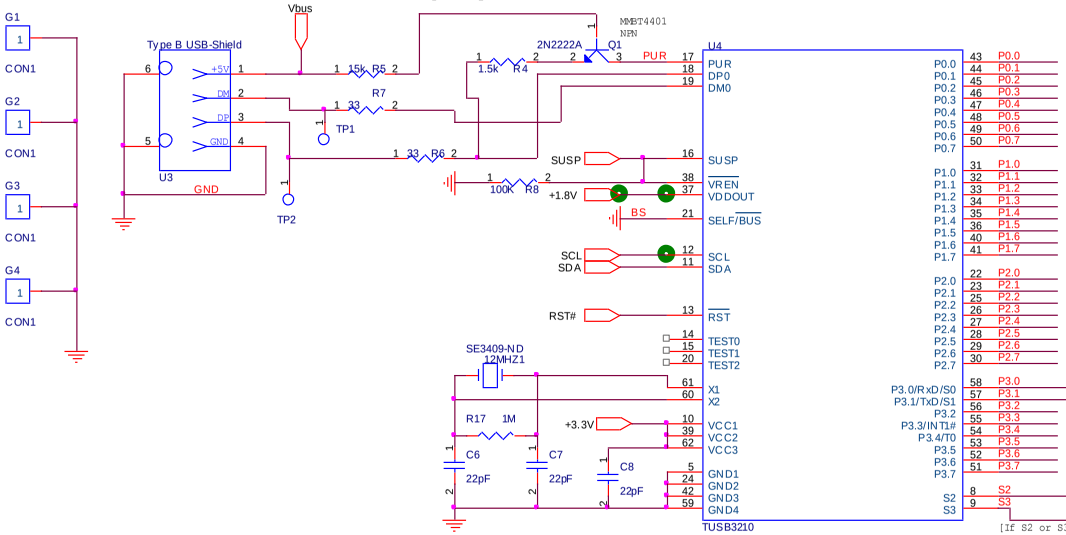
## BOARD SCHEMATIC DIAGRAM

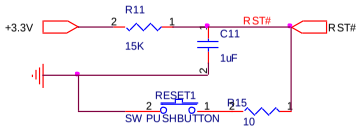
The schematic diagram of the board (Texas Instruments, 2001-b) is show in the next page, the diagram includes :

- The Generic Board built around the TUSB3210.
- The Power Supply circuit providing the board with a regulated 3.3 volts.
- The Reset circuit which resets the TUSB3210 by pushing a button.
- The EEPROM circuit (not used in the built board) .
- The Run and Suspend circuit.
- The RS232 Serial Port circuit (not used in the built board) .

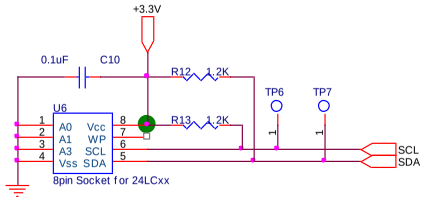



**Power Supply circuit**

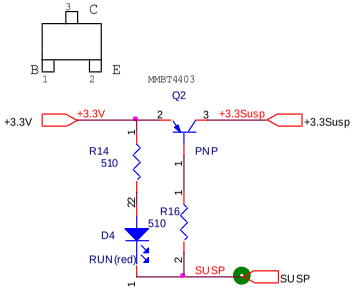


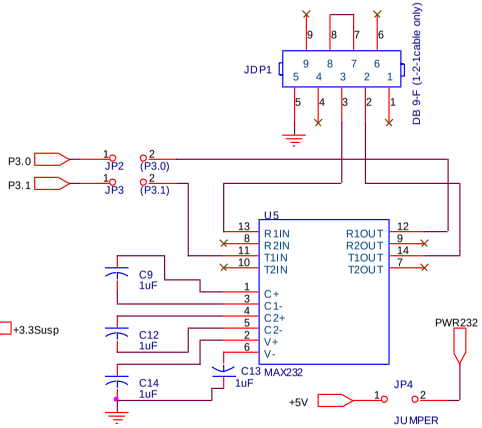

**Reset circuit**




**EEPROM circuit**




**Run Suspend modes circuit**




**RS232 Serial Port circuit**

# APPENDIX G

## BOARD'S BILL OF MATERIAL

The table below is a bill of material listing all the needed components to build the board (Texas Instruments, 2001-b).

| Item | Quantity | Reference | Part |
|---|---|---|---|
| 1 | 3 | C1, C4, C10 | Capacitor, 0.1 µF |
| 2 | 2 | C2, C5 | Capacitor, 4.7 µF |
| 3 | 1 | C3 | Capacitor, 10 µF |
| 4 | 3 | C6, C7, C8 | Capacitor, 22 pF |
| 5 | 5 | C9, C11, C12, C13, C14 | Capacitor, 1 µF |
| 6 | 9 | D4, D5...D12 | LEDs |
| 7 | 4 | G1, G2, G3, G4 | CON1 |
| 8 | 1 | JDP1 | DB 9–F (1–2–1 cable only) |
| 9 | 1 | JP1 | Connector plug 25×2 |
| 10 | 1 | JP2 | Jumper (P3.0) |
| 11 | 1 | JP3 | Jumper (P3.1) |
| 12 | 1 | JP4 | Jumper |
| 13 | 1 | J1 | AC adaptor 5 Vdc |
| 14 | 1 | Q1 | NPN transistor, 2N2222A |
| 15 | 1 | Q2 | PNP transistor, 2N4403 |
| 16 | 1 | RESET1 | Pushbutton switch |
| 17 | 6 | R2, R3, R8, R9, R10, R18 | Resistor, 100 kΩ |
| 18 | 1 | R4 | Resistor, 1.5 kΩ |
| 19 | 2 | R5, R11 | Resistor, 15 kΩ |
| 20 | 2 | R7, R6 | Resistor, 33 Ω |
| 21 | 2 | R13, R12 | Resistor, 1.2 kΩ |
| 22 | 2 | R14, R16 | Resistor, 510 Ω |
| 23 | 1 | R15 | Resistor, 10 Ω |
| 24 | 1 | R17 | Resistor, 1 MΩ |
| 25 | 2 | S2, S1 | Short |
| 26 | 7 | TP1, TP2, TP3, TP4, TP5, TP6, TP7 | Test point |
| 27 | 1 | U1 | TPS76333DBV or 278R33 |
| 28 | 1 | U2 | Jumper. JP(3) |
| 29 | 1 | U3 | Type-B USB shield |

| Item | Quantity | Reference | Part |
|---|---|---|---|
| 30 | 1 | U4 | USB controller, TUSB3210 |
| 31 | 1 | U5 | MAX232 |
| 32 | 1 | U6 | Eight-pin socket for 24LCxx |
| 33 | 1 | 12 MHz | SE3409-ND |