USB: The Universal Serial Bus

Benjamin David Lunt

Operating System Design
USB: The Universal Serial Bus

Benjamin David Lunt

Copyright 1984-2014 by Benjamin David Lunt
All rights reserved

No part of the content of this book may be copied, reproduced, or transmitted in any form by any means without written permission from the author.

About the Author

Benjamin David Lunt has been programming computers since 1984 and has always enjoyed the hardware side of the spectrum. He has programmed many different types of programs, mostly for DOS and other minimal operating systems, as well as a few programs for the Windows family.

About 13 years ago, he started to read up on the Universal Host Controller Interface (UHCI) and got hooked on the USB concept. He has been programming for USB ever since. One of the more known projects has been the USB support for the Bochs Emulator (http://bochs.sourceforge.net/).

Programming is not Ben's only line of work. He has been in the building trade since 1991, and enjoys remodeling homes.

He also enjoys hunting, fishing, and is an active member of his local Boy Scout troop.

Credits

Publisher
Forever Young Software

Editor
Benjamin David Lunt

Book Designer
Benjamin David Lunt

Cover Designer
Benjamin David Lunt

Production Team
Forever Young Software

Technical Review
Benjamin David Lunt

Special Thanks goes to
Loren “Boreal” Blaney

This book is dedicated to my daughter and son,

Larissa and John David

Published: February 2012
1st Edition: 20120121
2nd Edition: 20141010

Contents

Introduction
- The series as a whole
- Who is this series for?
- How to use this book?
- Things to know before reading this book
- Abbreviations
- Definitions
- Tables
- What you will need to use this book?
- Installing the Source Code on your computer
- What is on the disc?

- Prerequisite to using this book
- What is not discussed in this book
- Final word before we get started

Part 1

- Chapter 1 – Overview of the USB
- The process from connection to disconnection
- How the PC communicates with the USB
- How the USB communicates with the device
- The difference in USB speeds
- Other things to know and consider
- The design goal of this book
- Tested devices

Chapter 2 – The PCI Hardware

- Now that we have this information, what to do with it?
- Port I/O verses Memory Mapped I/O
- Determine the address space required for a function
- Other fields in the PCI address space
- Offset 0x60 for UHCI, EHCI, and xHCI Controllers
- EHCI and Companion Controller Function Numbers
- The PCI and Power Management

Chapter 3 – The UHCI Hardware

- UHCI Host Controller Detection
- Set up the UHCI controller
- The UHCI Port Status/Control Register
- Resetting a port
- Enabling the port
- Detecting multiple ports on a root hub
- Miscellaneous UHCI Notes

Chapter 4 – The UHCI Stack

- The UHCI Stack
- Transfer Descriptors
- Queues
- The FYSOS stack frame
- How the UHCI handles the Stack
- Summary

Chapter 5 – The OHCI Hardware

- The OHCI Controller’s Root Hub
- The OHCI Controller’s Port Register
- OHCI Host Controller Detection
- Set up the OHCI controller

- Setting the OHCI Controllers Operational Registers
- The OHCI Port Status/Control Register
- Powering the port
- Resetting the port
- Enabling the port
- Summary

Chapter 6 – The OHCI Stack
- The OHCI Stack
- Endpoint Descriptors
- Transfer Descriptors
- Summary

Chapter 7 – The EHCI Hardware
- Explanation of companion controllers
- Host Controller Capability Registers
- Host Controller Operational Registers
- Explanation of the PORTSC register
- EHCI Host Controller Detection
- Set up the EHCI controller
- The EHCI Port Status/Control Register
- Resetting and Enabling the Port
- Detecting multiple ports on a root hub

Chapter 8 – The EHCI Stack
- The EHCI Stack
- Transfer Descriptors
- Queue Heads
- Controller Schedule
- The FYSOS Periodic Stack Frame
- The FYSOS Asynchronous Stack Frame
- How the EHCI handles the Stack

Chapter 9 – The xHCI Hardware
- Host Controller Capability Registers
- Host Controller Extended Capabilities List
- xHCI USB Legacy Support
- xHCI USB Supported Protocol Capability
- Port Routing and Control
- Host Controller Operational Registers
- Explanation of the Save/Restore Operation
- Example of the Memory Mapped I/O Space
- The Root Hub’s Port Status and Control Registers
- Explanation of the PORTSC Register

- Explanation of the PORTPMSC Register
- Explanation of the PORTLI Register
- Explanation of the PORTHLPMC Register
- Explanation of the Host Runtime Register Set
- Explanation of the Doorbell Register Set
- Detecting and Resetting the Host Controller
- Set up the xHCI controller
- The xHCI Port Status/Control Register Set
- Pairing up the USB2 and USB3 port registers
- Resetting and Enabling the Port
- Detecting multiple ports on a root hub

Chapter 10 – The xHCI Schedule

- Memory Usage and Alignment
- The xHCI Device Context Area
- The xHCI Device Context Data Structure
- The xHCI Endpoint Context Data Structure
- The xHCI Transfer Descriptor
- The xHCI Transfer Request Block
- The xHCI Control Transfer Request Block
- The Setup Stage Transfer Request Block
- The Data Stage Transfer Request Block
- The Status Stage Transfer Request Block
- The Command Completion Transfer Request Block
- The xHCI Transfer Rings
- The Command Ring and TD Transfer Ring
- The Command's Event Ring
- Segment Tables
- Starting the Communication Process

Part 2

Chapter 11 – Device Enumeration with the UHCI

- Creating a Queue of Transfer Descriptors
- Getting the Device Descriptor of the Device
- Inserting your queue into the stack
- Setting the Address of the device
- Getting the String Descriptor(s) of the Device
- Getting/Setting the Configuration of the Device
- Short Packet Detect with the UHCI Stack

Chapter 12 – Device Enumeration with the OHCI

- Creating Endpoint Descriptors and Transfer Descriptors
- Getting the Device Descriptor of the Device

- Inserting your transfer descriptors into the stack
- When the Controller Has Processed the Endpoint Descriptor
- Setting the Address of the device
- Getting the String Descriptor(s) of the Device
- Getting/Setting the Configuration of the Device

Chapter 13 – Device Enumeration with the EHCI
- Driver Initialization and the BIOS
- Creating Queue Heads and Transfer Descriptors
- Inserting a Queue into the Stack
- Removing a queue from the Stack
- Controller Has Processed the Transfer Descriptors
- Setting the Address of the device
- Getting the String Descriptor(s) of the Device
- Getting/Setting the Configuration of the Device

Chapter 14 – Device Enumeration with the xHCI
- The New Cables and Backward Compatibility
- Is it a USB 2.0 or USB 3.0 Port?
- Driver Initialization
- Sending Commands to the Controller
- Detecting a Connection on the Root Hub
- Enabling a Slot for the Found Device
- Setting the Address of a Device
- Getting the Status of a Transfer
- Getting the Device Descriptor of the Device
- Getting the String Descriptor(s) of the Device
- Configuring an Attached Device
- Short Packet Detection

Part 3

Chapter 15 – Using the USB HID Mouse
- The HID Mouse Device Descriptors
- Configure the HID Mouse
- The HID Report
- Retrieving mouse movement

Chapter 16 – Using the USB HID Keyboard
- The HID Keyboard Device Descriptors
- Configure the HID Keyboard
- The HID Report
- Retrieving Keyboard Key Press Data
- Setting the Keyboards LED's
- Other Comments

Chapter 17 – Using the USB Mass Storage Device
The MSD Drive's Device Descriptors
Configure the MSD Drive
USB Mass Storage Device LUN's
Introduction to the Bulk-Only Protocol
Sending Commands Using the CBW/CSW Interface
Command Definition
Inquiry
Read Format Capacities
Request Sense
Read Capacity(10)/ Read Capacity(12)
Read(10)/Read(12)/Read(16)
Write(10)/Write(12) /Write(16)
Reset
Other Commands and Comments
Chapter 18 – Using the USB Floppy Storage Device
The Floppy Drive's Device Descriptors
Configure the Floppy Drive
Introduction to the Floppy Protocol
Sending Commands Using the CBI interface
UFI Command Definition
Inquiry
Read Format Capacities
Request Sense
Read(10)/Read(12)
Write(10)/Write(12)
Send Diagnostic/Reset
Other Commands and Comments
Chapter 19 – Using the USB External Hub
An Introduction to External Hubs
The Hub's Device Descriptors
Configure the External Hub
Detecting Number of Downstream Ports
Resetting and Enabling the Hub
Powering, Resetting, and Enabling a Port on the Hub
USB 2.0 Hubs and Transaction Translators
Other Comments
Chapter 20 – Using the USB Printer Device
The Printer's Descriptors
The Three Available Protocols

- Configure the Printer
- Get IEEE 1284 Device ID String
- Printing Text Using the PCL Language
- Other Comments

Chapter 21 – Using a USB HUB on the xHCI Root Hub
- The Hub's Device Descriptors
- The BOS Descriptor
- The Configuration Descriptor
- The Super Speed Endpoint Companion
- Configuring the External Hub and a Catch-22
- Set Depth Request
- Detecting Number of Downstream Ports
- Configuring the External Hub Again
- Resetting and Enabling the Hub
- Other Comments

Chapter 22 – Using a USB 3.0 Mass Storage device
- The MSD Drive's Device Descriptors
- The String Descriptors
- The BOS Descriptor
- The Configuration Descriptor
- Configure the MSD Drive
- Other Comments

Part 4

Chapter 23 – Using a USB Protocol Analyzer
- The Beagle from TotalPhase
- Setting up the hardware
- Receiving bus traffic
- Reading and Understanding the Report

Chapter 24 – The Human Interface Device
- A Brief Introduction to Human Interface Devices
- The HID Descriptor
- The Included HID Parser Code
- Other Comments

Part 5

Chapter 25 – Using a USB Video Camera
- The Camera's Device Descriptors
- Interface Association and Function Descriptors
- The Video Control Interface
- The Video Streaming Interface
- The Color Matching Descriptor

- Alternate Interface Descriptors
- A Known Video Device
- A Little More Information
- Configure the Camera for Video
- Other Commands and Comments

Chapter 26 – Setting Up the USB Video Camera

- Choosing/Setting the Camera's Resolution
- Resolution Negotiation
- Choosing the Alternate Interface
- Receiving Payload Data
- Stream Buffer
- Comments

Chapter 27 – The EHCI and ISO Transfers

- The EHCI Stack
- ISO Transfer Descriptors
- Creating an ISO Transfer
- Wrap Up

Appendix A - Disk Contents

- Installing the Source Code on your computer
- Finding the Specification Files

Appendix B - Included Utilities/Source Code

- Obtaining a C/C++ Compiler
- Obtaining an Assembler
- PKSFX(R) FAST! Self Extract Utility - v2.50
- DetCntlr -- Detect Controllers - v1.00.00
- GDevDesc -- Get Device Descriptor
- HIDParser -- Parse a HID Report Descriptor - v1.00.00
- MPutImg -- Write Disk Image to Floppy Drive - v0.10.01

Appendix C - List of Tables and Figures

- Figures
- Listings

Appendix D - Notes for all Controller Types

- Port Connection Status Change
- ISO Transfers
- Controller Quirks
- The BOS Descriptor
- Device Address Number
- String Descriptors with an Index of 0xEE
- The Configuration Descriptor's Bus Powered Bit

Appendix E - Request Sense Return Data

Appendix F - Brief History of the USB
Specification/Hardware Release History
Brad Hosler - Founder of USB
xHCI TRB Types and Completion Codes
Setup Packet
bmRequestType
bRequest
Descriptor Types
Appendix H - USB OTG (On The Go)
Appendix I - USB Class Codes
Appendix J - Device Firmware Upgrade
Upgrade Process
Wrap up and Comments
Appendix K - New Type C USB Connector
Appendix Q - Building a Simple USB Device
Our ATTiny2313 Breakout Board
Soldering the Components
The Programmer
The Cross Compiler
Fusing and Flashing
Build Blink Device
Troubleshooting
Building an Actual USB Device
Build *OUTPUT* Device
Build *INPUT* Device
Conclusion
Other Comments
Appendix X - For More Information
Bibliography

Introduction

The series as a whole

This book is part of a whole, a series of books describing how to design and write an operating system. The operating system in question is called *FYSOS, aka Konan*. Throughout this book you may see references to this as a whole. I will try to make this book as independent of the whole series as possible, but there may be times when I refer to another book in the series.

Currently, this series, *Operating System Design*, includes:

- The System Core
- The Virtual File System
- The ATA Hard drives and FDC Floppy drives
- The SCSI interface for optical and other media
- Input Devices
- The Serial and Parallel ports
- The PCI/PCIe Bus
- **The Universal Serial Bus**
- The Graphical User Interface
- The Network Interface

At the writing of this edition, three books have been written, shown above in bold. Other books in the series will be written when time and interest allows.

Who is this series for?

Have you ever wanted to write your own Operating System? Have you ever wondered how and where to start? How does the computer know where my code is? How do I make the computer do what I want? Have you ever wondered about these things?

Within the pages of each book in this series, and with the code and data on their accompanied CD-ROM's, I will take you through the process of creating a minimal, though fully functional operating system. I will discuss the boot process, where and how to get your operating system loaded and executed, how to detect and interact with the hardware, as well as creating a user interface.

I will also discuss the pros and cons of the process of writing and testing your code. What kind of hardware to use for testing, and what software you will need to create your own operating system.

Each book in the series should be independent of another. It is not required that you have another book in the series to use this or any other book within the series.

The included CD-ROM contains code and data from each chapter.
This book, USB: The Universal Serial Bus, has been the most asked about part of this series. Therefore, I am going to include it first. Other books in this series will be released when they become available.

The subject in this book is for those who want to learn how to study, program, and learn how to incorporate USB in their own operating system design, imbedded projects, and/or applications that have full control of the hardware with no underlining operating system to get in the way.

To show this, I show how to and include code to detect and setup the USB hardware, detect attached devices, enumerate, and communicate with a USB mouse, USB keypad, USB jump drive, and a 4-port USB external hub, as well as other devices.

How to use this book?

There are small icons displayed in this book from time to time. These icons indicate that this paragraph or text box is of difference than the main text. Here are these icons and their descriptions:

Indicates that the listing is included on the disc.

Indicates that this is a note box or “please note this” item.

Indicates that this is an “off the subject” note and is of interest only.

There are also some places where I may ask you to enter text into the DOS prompt or text editor. The actual text to be entered will have a shaded background while a specified key will be italic and enclosed in brackets. Here is an example on loading the file *demo.com* with DEBUG.

At the C:\ prompt enter the following:

C:\>DEBUG demo.com<*enter*>

There are numeric values throughout the text. When a value other than decimal is given, sometimes it will have the decimal value in parentheses just after it. For example, if I have the value 0FFh, if the subject calls for it, I will proceed it with the following: (255d) – meaning 255 in decimal format.

Things to know before reading this book

When I use a physical memory address, I always use a number of leading zeros depending on the memory access size I am currently using. If I am discussing something in 16-bit, I use five digits to indicate the address. Since in 16-bit real mode, the CPU has a 20-bit address bus, this 20-bit address can be shown in five hex digits. For example, if I were pointing to offset 0x123 in memory, I would use

```
0x00123
```

to describe this address. Four bits per hex digit, times five digits, is twenty bits. All addresses in 16-bit real mode will have this notation.

When I am discussing memory addresses in 32-bit protected mode, I will use the same notation except that it will have eight digits on the right side of the 'x'.

The USB hardware uses precise timing when accomplishing some tasks. Therefore, this book uses the symbol uS for microseconds and mS for milliseconds. A microsecond is defined as one millionth of a second, while a millisecond is defined as one thousandth of a second.

Since I use C and Assembly to build all of the projects in this series, I will assume that you know enough about these two languages and I will use these two languages throughout this book. If you are not familiar with C, please study up on C before dwelling deep into this book. If you are not familiar with Assembly, you may be able to use this book without any knowledge of Assembly. However, any operating system writing and understanding will require some Assembly language knowledge.

Throughout this book I will use the terms "system software" and "your driver" often. These two terms relate to the software that you are writing whether it be BIOS code, an Operating System, or simply a driver for an already created Operating System. When I use these two terms, I am talking about the code you are writing.

Abbreviations

The list starting below shows the abbreviations that might be used throughout this book.

USB	Universal Serial Bus
USB-IF	Universal Serial Bus-Implementers Forum (www.usb.org)
HC	Host Controller
UHCI	Universal Host Controller Interface
OHCI	Open Host Controller Interface
EHCI	Enhanced Host Controller Interface
xHCI	Extensible Host Controller Interface
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
ISA	Industry Standard Architecture
PCB	Printed Circuit Board
CPU	Central Processing Unit
IRQ	Interrupt Request
mS	Millisecond
uS	Microsecond
nS	Nanosecond
mA	Milliamps
HID	Human Interface Device
CRLF	Carriage Return Line Feed pair. ASCII pair 13/10.
QWERTY	The first six letters on a standard keyboard from top left to right. Is used to indicate a full sized keyboard.
LS	Low Speed
FS	Full Speed
HS	High Speed
SS	Super Speed
UFI	USB Floppy Interface
CBI	Control/Bulk/Interrupt (used with UFI)
CB	Control/Bulk
CDB	Command Data Block
SCSI	Small Computer System Interface
DOS	Disk Operating System (Usually referred to MS-DOS □)
FreeDOS	A freeware clone of MS-DOS found at www.freedos.org
i.e.	Latin for *id est*, or *in other words*.
e.g.	Latin for *exempli gratia*, *for the sake of example* or *for example*.
FAT	File Allocation Table. Microsoft's FAT file system
LBA	Logical Block Address
CHS	Cylinder / Head / Sector

When declaring an amount of memory or data, or when describing bits, I will use the following notation:

MSb - most significant bit. The left most or highest bit.
LSb - least significant bit. The right most or lowest bit.
MSB - most significant byte.
LSB - least significant byte.
Bit - a single bit of either 0 or 1.
Bits 7:5 - declares a series of three bits, bits 7, 6, and 5.
Nibble - a 4-bit field, the low or high 4 bits of a byte.
Byte - a single byte of 8 bits.
Word - two consecutive bytes totaling 16 bits.
Dword - four consecutive bytes totaling 32 bits.
Qword - eight consecutive bytes totaling 64 bits.
Page - Four Kilobytes of aligned memory.
K - Kilobyte. 1024 bytes. 100k = 100 * 1024
M - Megabyte. 1024 kilobytes. 100M = 100 * 1024 * 1024
G - Gigabyte. 1024 megabytes. 1G = 1024 * 1024 * 1024

When declaring variables and/or labels with a size in the C/C++ code throughout this series, I use type defines such as

```
bit8u  variable0;
bit16s variable1;
bit32u variable2;
bit64s variable3;
```

bit8u simply is an 8-bit byte and is unsigned, while bit16s is a 16-bit word and is signed. For more information on these types, please see ctype.h in the \utils\include\ directory on the disc.

When I write a numeric value, I will use the decimal form unless otherwise noted, or it is obvious that a different form is better for the current context. All decimal numbers will be in the form of 1234567890, while all hexadecimal numbers will use the form of 0x01234567, the standard C form with a preceding 0x, or will have a trailing ‘h’ as in 01234567h. All binary numbers will be in the form of 11001100b and have a trailing ‘b’. Hexadecimal or Binary numbers might have an underscore within the number sequence as 0101_0101b. This is to help the eye see the bit positions easier. For example, a 32-bit dword in binary form could be 01010101111001001100100100101001b. This is difficult to count where bit 23 might be. However, the same binary representation of 0101_0101_1110_0100_1100_1001_0010_1001b is much easier to find a numbered bit.

Definitions

The following words may be used within this book. I list their meaning here to clarify what is intended.

The words *should, shall, will,* or *must* are used to indicate mandatory requirements. This means that it is mandatory for a device or function when this term is used.

The words *may, might,* or *can* are used to indicate that it is recommended or is a possibility for a device when this term is used. The word *may* is usually referred to as, *is permitted to*. The word *can* is usually referred to, *is capable of*.

The word *Host* is referring to the code, operating system, driver, or whatever code is currently requesting access to the USB hardware. *Host Controller* is referring to the current hardware the Host is accessing.

Little-endian is used to indicate that a value larger than a byte is stored lowest byte first, then the next lowest byte, and so on. This is considered the Intel format.

Big-endian is just the opposite. The highest byte is stored first, then the next highest, and so on. This is considered the IBM format.

An example of the little-endian format would be if the value of 0x01234567 were to be written to memory, it would be written as

0x00000000 67 45 23 01

with the lowest byte written to the lower memory byte, the next lowest byte written to the next byte in memory, and so on.

Tables

Most tables within this book will have the first column as an offset column, and the second as a size column. The offset is the zero-based offset from the start of the object described, while the size is shown in bytes, unless otherwise noted.

What you will need to use this book?

I am going to use an Intel compatible processor with a PCI[1] bus, a single UHCI 2-port USB card, a single OHCI 4-port USB card, a single EHCI 4-port USB card, a single xHCI 2-port USB card, a USB mouse, a USB keypad, a 4-port external bus-powered hub, and a USB 256meg-thumb drive. The code will be protected-mode code using a DPMI and written in 16-bit or 32-bit C and/or assembly. Instructions on obtaining a C compiler and assembler are included in Appendix B.

Since the overall topic of the books in this series are writing your own operating system, none of the documentation or code examples in this book describe how to call the underlining operating system to do the work. That would defeat the purpose of this book. Therefore, all of the documentation and code in this book describe how to use the physical hardware to complete the task at hand. Therefore, you must be able to have full access to the hardware. Good old DOS allows us to have full access, so all code examples are written using a DOS compiler or assembler and should only be executed on a DOS machine. They will not work on an emulated DOS machine, such as a DOS box under a modern commercial operating system. The examples do use a DOS Protected Mode Interface (DPMI) driver for DOS. However, this is to allocate physical memory and so these examples can be executed in DOS, accessing high memory addresses.

If you do not have a DOS bootable machine, a floppy disk image is available on the included CDROM. Using the included installer, mputimg.exe, you can create a FreeDOS bootable 1.44meg floppy disk.

To be able to execute the included application source code, and see what the results are, you must have a similar set-up as described here. You may also need to have a fair knowledge of C and Intel x86 assembly to use and understand the source code.

I may refer to code within the FYSOS operating system throughout this book. If I do, there will be comments and descriptions in the code of this book to give similar or exact results, and documented accordingly.

Installing the Source Code on your computer

To install the source and applications from the CDROM to your computer, I suggest you create a directory of the following:

C:\FYSOS\main\usb\

If you have installed items from other books in this series, the C:\FYSOS\main\ directory may already be created.

It is not required to use this directory structure to use this book. However, if you obtain and use more than one of the books in this series, this may make it easier for you to follow along with the series.

I use the DOS only version of MS □ Win98SE □ as my test host. However, I use MS □ WinXP □ as my development host and will assume a similar host from here on.

To install the files from the CDROM to your hard drive, copy the whole contents of the \main\ directory to your hard drive. This will install all the source code and application data you will need for this book. See Appendix B on how to obtain and installation of the C compiler and assembler.

If you are using DOS, use the following:

C:\xcopy D:\FYSOS\MAIN\USB C:\FYSOS\MAIN\USB /E /I /-Y /D<*enter*>

Where the C: drive is your host drive, and the D: drive is the CDROM drive.

What is on the disc?

As mention in the last section, there is a disc included with this book. It contains all of the code contained in this book except the small snippets, which are just a few lines of code rather than a complete program. Please see Appendix X on how to obtain the CDROM.

A code listing in this book will have a small image of a disk at the beginning of the listing declaring that the disc contains this listing.

Also included on the disc are all utilities mentioned in the text. You can find these utilities in the \main\usb\utils directory. The PKUNZIP.EXE file is used to unzip a utility that has the .ZIP extension. The PKUNZIP.EXE file is shareware. To see more about it, you can go to the PKWARE site at http://www.pkware.com.

There are multiple other programs that can access .ZIP files. For example, the Windows Explore program will see them as regular folders. WinRAR at http://www.winrar.com/, WinZip at http://www.winzip.com/win/en/index.htm, along with other free software packages will also extract the .ZIP files.

See Appendix A for a complete listing of the disc's contents, and Appendix X for information on how to receive the disc.

Prerequisite to using this book

It is not required that you know much or anything about the USB to use and understand this book. However, it will be to your benefit if you have briefly read through each of the specification files described in Appendix A. It is not required that you understand anything that you have read of these specifications, though it will make it easier to find a reference in those specifications when you come across it when reading this book.

It will also be beneficial to you, but not required, if you have knowledge of operating system development programming, how memory is used, how to communicate with the computer's hardware, and such subjects of this matter.

What is not discussed in this book

I will not go in to much detail on the electrical mechanics of the USB. In other words, I will not describe or have illustrations showing the layout of the different controller chips, the current pathways, or pin-outs of the PCI cards. This book is not used for physical manufacturing of USB devices and/or controllers. This book is used to describe the way your operating system, BIOS, or stand-alone driver will communicate with each controller and attached devices. However, for those interested, see Appendix Q which describes how to create USB devices using PCB's and a soldering gun.

Final word before we get started

All of the information in this book and the code on the disc is accurate to the best of my knowledge. If there is ever a discrepancy, the associated specification takes precedence. If there is an error in the accuracy of this book's information, you use it at your own risk. I give no warranty, either expressed or implied, to the correctness of this book. However, if you do find an error, please let me know, so that I may correct it in a later edition. I may list your name as a contributor.

Enough with the pleasantries, shall we get started?

Part 1

Chapter 1 – Overview of the USB

What makes the USB system better than another system? Why should we use it instead of the standard UART serial or parallel? Well, one thing for sure is that you can have up to 127 devices plugged into the same root hub with USB, where as you would have to have 127 UART cards and 127 corresponding ISA or PCI slots. Second, USB devices can be hot pluggable, meaning they can be plugged in and unplugged without having to turn off and/or reboot the machine.

Another advantage of a USB device is that when it is plugged in, the host software requests a device identify request. The device then sends a packet of data stating what kind of device it is. From this packet, the host can then request other identification packets, including strings and configurations, then load the appropriate drivers. With a UART serial device, the user must install the driver, which assumes the user has enough knowledge to do so.

A combination of USB devices can draw up to 500mA of power from the bus[2]. If this is not enough, or the device needs a very precise amount of current, the device can be externally powered. This option is also stated in one of the request packets the host may request.

If there are at most 127 devices on a single controller, how is it that a device knows what packet to receive and what packets to ignore? When a device is plugged in and properly reset, it is then in the default state. Any device that is in this default state must answer to the default address of zero. Once it has been set to the addressed state, it now has an address greater than 0, but less then or equal to 127. The host is responsible for making sure it does not set a device's address to a value that is already used. From this point on, until a reset or power failure is found, the device will answer only to this address number, and will ignore any other packet.

Something that I questioned often when I was starting to learn this was that if there is more than one device attached at boot up, why doesn't all the devices answer to address zero? Well, we have to remember, only one physical device can be attached to a root hub port at a time. When the port is reset, only the device on this port sees the reset. If this device is a hub, then after reset and the move to the addressed state, it in turn only resets one of its ports at a time. Therefore, only one device should be reset and set to the default state at a time. Please note that on a hub, it is possible to reset each port before sending the set_address packet. Therefore, it is possible to have more than one device in the default state. It is up to the host software to make sure that it does not reset two or more devices without making sure they are in the address state before the next one is reset to the default state. This is also true with the root hub, you should only reset one port at a time. For example, start with the first port, resetting it, start the enumeration process, and setting an address, moving it to the address state. Then move to the next port on the (root) hub until you have found that there are no more ports.

The process from connection to disconnection

When a device is connected, plugged in, or is already plugged in at boot time, the host sends a reset to that port. If the connection is on the root hub, which is the PCI card on the motherboard, the reset is sent via the controller's hardware. If the connection is on an external hub, the host sends a reset packet to the hub in question. A port reset is different on different controller types, and is discussed in later chapters.

Once the reset has completed, the host starts the enumeration process. This process is determined by the host software's developers and may be done differently depending on the wants of the developers. Usually, you want to get as little information as you need, then move the device to the addressed state before you continue on with the enumeration. However, it is perfectly legal to do most standard enumeration requests while the device is in the default state.

For the sake of this book we will get the first few bytes of the device descriptor, reset the port, set the device to the address state, giving it an address of one or higher, then continuing to request the rest of the device's descriptors including getting the complete initial device descriptor. One of these descriptors will be the endpoint descriptor. A device may have more than one, while normally a standard device will not have more than three: the default endpoint, for control transfers; an in endpoint and an out endpoint. A device, such as a mouse, may have only an in endpoint, used for interrupts. Some devices may and can have four or more per configuration.

Each device must have at least one configuration descriptor. After choosing which one, we then have to set that configuration. The device is now ready for use.

When the host senses a device disconnect, the host software will go through the topographical map (topo map) of all the connected devices and see which one is now disconnected. It does so by checking the root hub and any external hubs for the disconnect. It then removes that device's data stored in memory from the topo, unloads the driver, if desired, and continues to monitor for any other connection status changes.

How the PC communicates with the USB

A PC may have one or more USB controllers of a certain type or types. The controller is the actual hardware that your software communicates with to control the USB devices attached. These controllers are often called the Root Hub. The four types of controllers discussed within this book are the UHCI, OHCI, EHCI, and xHCI. Even though each controller is different and your software communicates with it differently, each controller sends the same type of USB data on the cable to the device, no matter the controller type. In theory, the device attached would have no idea what controller type it is attached to. Throughout this book, I will discuss the details of each of these four controllers. Have a look at Appendix F for a brief history on these controllers.

There is actually a fifth USB controller type called the Wireless Host Controller Interface (WHCI) and is used for Wireless Universal Serial Bus (WUSB) communications. For more information on this controller type and its specification, see the whci_r095_interface_spec.pdf file, from the USB-IF web page (See Appendix A).

How the USB communicates with the device

Once the USB controller's registers are set, few of them are changed by the host software. They are, however, read quite often. So, if this is the case, how do we send and receive data on the USB bus?

With the UHCI controller, for example, before we set the "run" bit in the controller's command register, we set up a page of memory, 1024 dwords, as the USB stack for this controller. Each dword points to a data structure that tells the controller what to do. These data structures tell the controller to send or receive data packets from a specified device on the bus. These data structures that the dword above points to is called a Frame with the dword called a Frame Pointer.

This controller allocates one millisecond (mS) for each frame. It follows the frame's data structures, called Queues and Transfer Descriptors, as long as there is enough time left in the frame. Once the 1 mS has elapsed, it moves to the next frame pointer.

In the next few chapters, you will see how the UHCI controller flows from Transfer Descriptor to Transfer Descriptor (and Queue Heads), and knows which one to execute when it comes back to this frame. It is very similar for the OHCI controller, and you will learn how it uses Endpoint Descriptors along with Transfer Descriptors in a later chapter. Following these chapters, you will learn about the EHCI and then the xHCI controllers and stack frames. All four controllers use different ways to communication with a USB device, and are explained in detail. However, don't get confused that this means USB devices will only work with a specific controller. Each controller may use a different form of data structure on the host machine, but all data sent on the USB bus by the controller is the same. Any USB device from version 1.1 to 3.0 will work on any controller, though a device specifically designed for USB version 3.0 and the xHCI controller will have a much faster data transfer rate than the same device plugged into a UHCI or OHCI controller.

The difference in USB speeds

The USB specification states that there are four different speed configurations, low-, full-, high-, and super-speed. The USB 3.0 specification includes all speeds, the 2.0 specification contains up to the high-speed details, while the USB 1.x specification contains the low- and full-speed details. However, don't get the specifications mixed up. Just because a device is a low-speed or full-speed device, doesn't mean it isn't version 3.0 compliant. The same goes for the other way around. A super-speed device can and may work on a low/full speed controller. The version 3.0 doesn't mean that the device is a super-speed device. All super-speed devices must at least enumerate on a low/full speed controller. However, most standard super-speed devices will work on a low/full speed controller without any noticeable difference other than the speed it takes to complete tasks. All low/full speed devices will work on a super-speed controller card with no difference.

The UHCI and OHCI controllers are low- and full-speed controllers, the EHCI controller is a high-speed controller, and the xHCI controller is a super-speed controller.

Other things to know and consider

There are a few things that you should consider when using the USB and writing software. The USB standard states that any cable should be equal to or less than three meters (9.8 feet) in length for low-speed devices, and no longer than five meters (16.4 feet) for full- and high-speed devices. I had a client call one evening to tell me that her new USB mouse, which is a low-speed device, would only work part of the time. When I arrived, I noticed that the cable on the mouse was almost ten feet long. When I plugged in a compliant USB mouse with a six-foot cord, the host now worked fine. The mouse with the 10-foot cord was then plugged into a different host and had the same problematic results.

Unless an external hub is self-powered, it is likely that you will not get a mouse with a ten-foot cord to function properly 100% of the time, if the hub has a ten-foot cord as well. In other words, there should be no more than 9.8 feet of cord between the end device and a powered hub. The device may or may not work depending on the amperage drawn on the hub by other devices and the quality of the device and pathway to the root hub.

See section 11.4.7 of the USB 3.0 specification, table 11-3 for a little more information on wire length.

The design goal of this book

The design goal of this book is to describe how to program the USB hardware, the USB controllers, and to communicate with the attached devices without using any host operating system services.

When the high- and super-speed controllers were introduced, the USB manufactures wanted to allow all low- and full-speed devices to still be usable with the new hardware. Therefore, the EHCI controller is backward compatible with the UHCI and OHCI controllers, and the xHCI controller is backward compatible with all three older controllers.

To make this possible, the EHCI controller contains one or more OHCI or UHCI controllers on board to handle the low- and full-speed devices while the EHCI controller handles the high-speed devices. The OHCI or UHCI controllers on board the EHCI controller are considered the companion controllers. Each companion controller can be independent of the parent controller if the operating system is not parent-controller aware. For example, if there is an EHCI controller with two companion controllers on board, and the operating system isn't EHCI compatible, the operating system will see the two companion controllers as separate controllers and use them as such, never touching the EHCI controller.

Since the EHCI sends all low- and full-speed control to the companion controller(s), you must write a driver for all three controller types, UHCI, OHCI, and EHCI. The xHCI controller makes it a bit easier and eliminates all companion controllers, handling all speeds as a single controller. However, if the operating system does not support the xHCI controller, you can not use lower speed devices with it as you can with the EHCI's companion controllers.

This book will describe how to use and program all four controllers, describing each controller as if it did not have a parent controller, as if it were a stand-alone controller. In the chapters that describe the EHCI controller having companion controllers, I will explain how to pass the control of lower speed devices to these companion controllers.

Since each EHCI companion controller is not aware of its parent controller, I will start with the low/full speed controllers, the UHCI and OHCI controllers. Once you have learned how to use these controllers, you will be able to move on to the EHCI controller, how to use it and how to pass control to the companion controllers if necessary. Once you have learned how to do this, you can move on to the xHCI controller.

The xHCI controller is completely independent of the three previous controllers. If you only want to learn about the xHCI controller, a brief, quick read through of the other three controller chapters is recommended but not required.

Since each controller uses the PCI host bus to communicate with the system, I will describe how to communicate with the PCI to find and set up the controller's base address registers and interrupt number. However, this book will not go into detail about the PCI, other than what is needed to use the USB controllers.

Once each controller is discussed and you know how to set up and communicate with each type of controller, and you understand how to detect attached devices and communicate with them, I will discuss in more detail how to configure and use different devices, along with low- and full-speed devices, high-speed devices, and the new super-speed devices.

For example, see the chapter on USB Mice and USB Keyboards for low-speed devices, the chapter on Mass Storage Devices for full- or high-speed devices. Then I will discuss the new USB 3.0 protocol and describe how to use super-speed devices.

Tested devices

I have tested my research and code on a few devices, though this information and code should work on most any device connected. These devices include multiple HID mice, a HID keyboard/keypad, a few multi-port optionally externally powered hubs, an external 1.44m floppy disk drive, and a few flash/thumb drives.

I have a seven-port external hub that seems odd to have seven ports since seven is not a multiple of two. It is in fact, a four-port hub with one of the ports internally attached to another four-port hub all in the same case. Hence the seven usable ports. When enumerated, you will find a four-port hub and another four port hub attached to one of the first hub's ports.

I will detail a little about how to program and communicate with a printer. However, the instructions within this book will allow you to enumerate the printer and find what type of printer it is, but as with most printers and other devices of this type, they require proprietary software drivers to continue to communicate. Therefore, not much will be detailed about a USB printer, though I will show you how to print a simple image.

Chapter 2 – The PCI Hardware

The USB card must communicate with the system bus through a default and known system. The most widely used is the Peripheral Component Interconnect bus, or the PCI bus. A computer built after 1994 may have a PCI bus, with some of the more modern motherboards containing an integrated USB controller right on the motherboard, though a USB controller that plugs into one of the free PCI slots works just the same. As for the programmer, there is no difference whether the controller is integrated on to the motherboard, or plugged into one of the PCI slots, they each still use the PCI bus.

The new xHCI controller may use the PCIe bus, or the Peripheral Component Interconnect Express bus. For the purpose of this book, and since the PCIe bus is backward compatible with the older PCI bus, we use the PCI technique described here and don't worry about the PCIe's extended capabilities. Any reference to the PCI hardware uses the PCI specification version 2.3 unless otherwise stated.

To set up, control, and read the status of the USB controller, you must be able to read and write to specific I/O addresses on this bus. However, since the PCI interface enables numerous I/O addresses, fixed addresses are not to be relied upon.

If we knew that the USB controller would always use a fixed address for its I/O access, there would be no need to even mention anything about the PCI. This is what the ISA bus was for, right?

What if you install two USB controllers on the same bus? How would you know which is which, and communicate with them? This is where the PCI bus comes in and makes it quite an easy task for us. Once we detect and find the controller(s) on the PCI bus, set up the PCI, we no longer need to worry about the PCI interface, in general.

So how to use the PCI and detect and find all USB controllers on its bus? Well, we need a little background on the PCI first. Each PCI device has what is called a "Configuration Space" and is 256 bytes in length. In this space, each device has signature and configuration information that an operating system or other software can find out what type of device is plugged into this "slot". Below is a table of this configuration space. Since each USB controller uses only header type 0, I will only include this header type.

Table 2-1: Format of PCI Config Space

Format of the PCI configuration space with type 0		
Offset	Size	Description
00h	2	Vendor ID
02h	2	Device ID
04h	2	Command register
06h	2	Status register
08h	1	Revision ID
09h	1	Programming interface
0Ah	1	Sub Class
0Bh	1	Class Code
0Ch	1	Cache line size
0Dh	1	Latency timer
0Eh	1	Header type (type 0 is used here)
0Fh	1	Build in self-test result
10h	4	Base Address 0 (used with OHCI, EHCI, & xHCI)
14h	4	Base Address 1

18h	4	Base Address 2
1Ch	4	Base Address 3
20h	4	Base Address 4 (used with UHCI)
24h	4	Base Address 5
28h	4	CardBus CIS pointer
2Ch	2	Subsystem vendor ID
2Eh	2	Subsystem ID
30h	4	Expansion ROM base address
34h	1	Offset of capabilities list within this space
35h	3	Reserved
38h	4	Reserved
3Ch	1	Interrupt line (0 = none, 1 = IRQ1, etc.)
3Dh	1	Interrupt pin (0 = none, 1 = INTA, etc.)
3Eh	1	Minimum time bus master needs PCI bus ownership
3Fh	1	Maximum latency
40h	192	varies by device

To communicate with the PCI bus hardware and to detect and read this table from each device, you can use one of two methods, PCI Configuration Mechanism 1 or PCI Configuration Mechanism 2. Mechanism 2 is depreciated as of PCI version 2.1 and you should use Mechanism 1. To use Mechanism 1, you simply write a DWORD value containing bitmap flags to the PCI Address port, and then read or write from the Data port. You must read and write DWORD values only and must also be on a DWORD aligned address within the configuration space. The address port is read/written to I/O port 0x0CF8 and the data port is read/written to I/O port 0x0CFC.

Please note that we assume there is a PCI bus on the system and it indeed uses these I/O port values. In FYSOS, I actually check to make sure there is a PCI bus before using these ports. For the purpose of this book, we will assume you have a PCI bus configured to these I/O ports.

The bit fields of the Address port is shown below.

Table 2-2: Format of PCI Address Register

Format of the PCI Mechanism 1 Address Register	
Bits	Description
31	Enable configuration space mapping
30:24	reserved
23:16	Bus number
15:11	Device number
10: 8	Function
7: 2	Configuration register number
1: 0	reserved

In the C language I use the following code to read a byte, word, or dword sized value from the configuration space of a device.

Listing 2-1: Read value from the PCI

```
bit32u read_pci(const bit8u bus, const bit8u dev,
        const bit8u func, const bit8u port,
        const bit8u len) {
```

```
  bit32u ret;
  const bit32u val = 0x80000000 |
    (bus << 16) |
    (dev << 11) |
    (func << 8) |
    (port & 0xFC);
  outpd(0x0CF8, val);
  ret = inpd(0x0CFC + (port & 0x3)); // dword boundaries only
  ret &= (0xFFFFFFFF >> ((4-len) * 8));

  return ret;
}
```

To find a device you simply enumerate the PCI bus, or if multiple PCI buses are installed, enumerate them all.

If you look in the table above, the bus number uses bits 23:16, or 8 bits. This gives a limit of 256 buses. The device number uses 5 bits for a limit of 32 devices per bus, with a function number using 3 bits for a limit of 8 functions per device.

As of the writing of this book, I have not seen more than 2 buses being used on a PCI system. However, on a PCIe system with a xHCI controller, I have seen a bus number of 32.

If we read the 256-byte configuration space for each function on each device on each bus, we should be able to find the USB controller. But how do you know if a device exists on that bus, device number, and function? If a device does not exist on that bus/dev/func configuration, the PCI bus will return a value of `0xFFFF` when reading the Vendor ID value.

Using the code from Listing 2-1, you can read the Vendor ID value on Bus 1, Device 2, Function 3, Port (offset) 0, with the following line of C code.

```
VendorID = read_pci(1, 2, 3, 0, sizeof(bit16u));
```

You can only read dwords and these must be on dword-aligned boundaries. Therefore, the code in Listing 2-1 allows the Port parameter to be any offset, dropping off the bottom two bits, then reading a 1-, 2-, 3-, or 4-byte value. However, this value cannot cross the dword boundary. For example, you cannot read a 4-byte value from offset 2.

Each valid device will have a class code and sub code in its configuration space. Since the USB defines the class code and subclass code of a USB compatible controller to be 0x0C and 0x03 respectively, we simply need to run through all of the devices until we find all that have a class code of 0x0C, and a subclass code of 0x03.

Up to this point, you should be able to find all device functions that have a class and subclass code defined by the USB standard. With this, you should be able to find all USB controllers. I have written a small utility called DETCNTLR.EXE to find and display information about the found devices. This utility and its source code is included on the included disc.

Now that we have this information, what to do with it?

I have a UHCI 2-port root hub on a test machine. When I execute DETCNTLR.EXE on that test machine, I get the following listing.

```
Found a USB compatible device entry.
Bus = 0, device = 7, function = 2 IO Base: 0x00009401, IRQ:   11

Found a USB compatible device entry.
Bus = 0, device = 7, function = 3 IO Base: 0x00009801, IRQ:   11
```

Notice that it is the same device number. A PCI device can have up to eight different functions. Sometimes these functions can be quite different, though most of the time they are similar.

The DETCNTLR.EXE utility also prints the I/O Base and the IRQ number. You will learn about these further in this chapter.

Before you get confused, don't associate the two entries on the same device above with a single UHCI root hub having two ports, one as function 2 and the other as function 3. Each function is capable of two or more ports. You will read further in this chapter why there are two different functions above, with two different I/O Base values, but using the same IRQ value.

An EHCI device may have zero or more UHCI or OHCI controllers, one as function 0 and the next as function 1, with the EHCI controller as function *n*, where *n* is at least the number of UHCI or OHCI controllers, all on the same device. See the section at the end of this chapter for more on the EHCI and its companion controllers.

At this point, we know we have a USB compatible controller at bus 0, device 7, function 2 and another at function 3 on the same device. As far as we are concerned, these are two completely different controllers and should treat them so. The only thing we have to remember is that they just happen to be on the same bus and device, but have different function numbers.

Also, at this point we don't know what type of controller it is. It could be one of four known types: UHCI, OHCI, EHCI, or xHCI. This is why we must now detect what type of controller it is.

Here is where the PCI configuration space shines. The p_interface field in the 256-byte space tells us what type it is. UHCI = 0x00, OHCI = 0x10, EHCI = 0x20, and the xHCI = 0x30.

I have not verified it, nor have I ever seen any documentation stating so, but in my opinion, the values 0x00, 0x10, 0x20, and 0x30 seem to have bits 5:4 indicate the type instead of the whole value indicating the type. If I can find some valid documentation, or other notable source, I will update this in a future edition. The xHCI controller specification states that the super-speed value of 0x30 is for version 3.0. I think it is coincidence that this happened to be the next value used. You could say that 0x20 is version 2.0 for the EHCI controller too, though again, I think it is coincidence.

Execute the DETCNTLR.EXE utility again, this time giving it a command line parameter of /type.

```
C:\MAIN\USB\UTILS>DETCNTRLR /type <enter>
```

Notice that it now displays the type of controller installed on that function. I have installed an additional EHCI with two UHCI controllers on board to show this:

```
Found a USB compatible device entry. (UHCI)
Bus = 0, device = 7, function = 2 IO Base: 0x00009401, IRQ:   11

Found a USB compatible device entry. (UHCI)
Bus = 0, device = 7, function = 3 IO Base: 0x00009801, IRQ:   11

Found a USB compatible device entry. (UHCI)
Bus = 0, device = 9, function = 0 IO Base: 0x0000AC01, IRQ:   9

Found a USB compatible device entry. (UHCI)
Bus = 0, device = 9, function = 1 IO Base: 0x0000B001, IRQ:   5

Found a USB compatible device entry. (EHCI)
Bus = 0, device = 9, function = 2 IO Base: 0xCFDC0000, IRQ:   11
```

Notice that the EHCI controller card is a single device, but has three functions, the first and second as UHCI controllers, and the third as the EHCI controller. Also notice that the first two controllers on the device (device 9) have different IRQ numbers. This is for the function of the EHCI controller. I will go into more detail about this in a later chapter.

As another example, I ran the same utility on a different machine with an OHCI controller on it.

```
Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 0 IO Base: 0xE1005000, IRQ:  11

Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 1 IO Base: 0xE1005100, IRQ:  11

Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 2 IO Base: 0xE1005200, IRQ:  11

Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 3 IO Base: 0xE1005300, IRQ:  11
```

Notice that the I/O Base is 32-bit instead of 16-bit like the UHCI before. This is because the OHCI uses memory-mapped registers. Also note that it has four functions. This particular controller has four ports and uses a single function for each port. For example, function 0 controls port 0, function 1 controls port 1, etc., acting as four separate controllers with one port each.

As one more example, I ran the utility on a machine with an xHCI controller on it, which gave the following results:

```
Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 0 IO Base: 0xE1005000, IRQ:  11

Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 1 IO Base: 0xE1005100, IRQ:  11

Found a USB compatible device entry. (OHCI)
Bus = 0, device = 9, function = 2 IO Base: 0xE1005200, IRQ:  11

Found a USB compatible device entry. (OHCI)
```

```
Bus = 0, device = 9, function = 3 IO Base: 0xE1005300, IRQ:  11

Found a USB compatible device entry. (xHCI)
Bus = 32, device = 0, function = 0 IO Base: 0xF0500004, IRQ:  10
```

Port I/O verses Memory Mapped I/O

Now we need to use a few other items in the PCI configuration space. Each controller uses a base I/O address space, with a number of registers at this base address. This I/O base can be actual Port I/O or memory mapped I/O. The UHCI uses Port I/O, but for the sake of argument, and since the other controllers can and usually do use memory mapped I/O, I will discuss the way to find out which type and how much I/O space is used for each controller.

If you will look back at the first output from DETCNTLR.EXE, the base I/O address was an odd address, ignoring the address for the EHCI controller. Well, you and I know that most port and memory access larger than a byte should be aligned, so why the odd address? The PCI configuration space uses bit 0 to indicate whether this is a port I/O address or a memory mapped I/O address. If the bit is set, as with the UHCI controllers above, it is Port I/O and the address is dword aligned with bits 1:0 of the actual address being zero. In other words, if bit 0 is set, you need to clear out bits 1:0 and use the value as is, as a Port I/O address.

If bit 0 is clear, then it is a memory-mapped address. This time, the address is paragraph aligned, or 16-byte aligned, with bits 31:4 as bits 31:4 of the address. In other words, take the 32-bit value and mask off the bottom 4 bits to get the memory-mapped address. Don't forget to save the bottom 4 bits, since we will need their values. If bits 2:1 = 00, the address is anywhere in the 4gig address space. If bits 2:1 = 01, it is below the 1meg mark, if bits 2:1 = 10, anywhere in the 64-bit address space, and you need to get the high 32-bits of the address from the next base address field in the PCI configuration space. Bit 3 is reserved. Refer to the PCI specification, section 6.2.5.1.

The UHCI uses the BAR (Base Address Register) at offset 0x20 in the PCI configuration space while the OHCI, EHCI, and xHCI all use the BAR at offset 0x10. I do not know why the USB-IF decided to use the 5th BAR instead of the 1st with the UHCI.

Determine the address space required for a function

I don't think very many people know that the PCI hardware has a mechanism for getting the address space size (number of byte-sized registers), of that particular function, so I will spell it out here.

Before we initialize the card, make sure the cards I/O is disabled via the command register. Read the base I/O field, write a value of 0xFFFFFFFF to that base I/O field, read it back saving the value, and then write the original value back to the PCI config space. By doing so, the function will tell us how much I/O address space it requires via the value returned. The value returned is the inverse of the size of the address space.

The following is an example if the function uses port I/O, by masking off bit 0. You should mask off bits 3:0 for a memory mapped I/O device.

```
Range = (bit16u) ((~(returned_value & ~0x1)) + 1);
```

First, mask bit 0, then take the inverse, and finally add one. The remaining 16-bit value is the size, ignoring the upper word of the 32-bit value.

Remember to write the sequence above to the PCI card's configuration space. Don't just write the sequence to your saved buffer in memory. This is a silly mistake that may haunt you until you realize the error.

See the end of the DETCNTLR.C source file for the source code FYSOS uses to get this range.

We are almost done with the PCI configuration. All we have to do now is write a valid base address back to the Base I/O field, for UHCI this is base address 4, write a valid IRQ number to the IRQ field, and make sure the PCI card is "active".

The following is a typical sequence for initializing the PCI configuration space for the UHCI controller.

1. If you have changed the base address your code would like to use, write the new value of the base address I/O field back to its offset in the configuration space.
2. Write 0x00000000 to dword 0x34 and dword 0x38 of the configuration space.
3. Write a valid IRQ number to the IRQ field. By valid number, I mean a free IRQ number that your code has indicated is available.
4. Write 0x0005 to the command field. This enables the bus mastering and I/O access. You would write 0x06 if you were using memory mapped I/O.
5. Write 0x8F00 to word 0xC0 of the configuration space. This is the Legacy Support register of the UHCI specified in version 1.10 of the UHCI specs, turning off all legacy support for keyboards and mice. It is different for the OHCI, EHCI, and xHCI controllers.

Now we are done with the PCI detection, initialize, and set-up. The remaining controller access should be done via the base address and IRQ number we extracted from the configuration space. As far as we are concerned, we can all but forget about the PCI for now.

Except for the following notes on some of the extra fields in the PCI configuration space, you can go on to the next chapter about detecting and setting up the UHCI controller, or skip to Chapter 5 for the OHCI controller, Chapter 7 for the EHCI controller, or Chapter 9 for the xHCI controller.

As for the function of the extra fields in the PCI configuration space, I found a document by VIA Tech, VT83C572, that has a few notes on a few fields. See Appendix A on where to get the

VT83C572.PDF file. The UHCI 1.10 specs have some information about the legacy registers in the PCI configuration space, as do the other controller specifications.

Other fields in the PCI address space

Other fields in the PCI address space that are relevant to USB are, for example on the VT83C572 UHCI controller, are the Misc. Control registers and the Legacy Support register. See the end of Chapter 3 for the VT83C572 Misc. Control registers and other notes on this specific controller.

In the UHCI transfer descriptor, there is a field called MaxLen. It tells the controller the max amount of bytes to transfer. The USB specification says that a maximum of 1023 bytes, for full speed devices, may be transferred at one time. However, the UHCI specs will allow 1280 bytes. Therefore, in the VT83C572 UHCI controller, there is a bit in the first Misc. Control register, bit 3, that indicates if the controller will allow a maximum of 1023 bytes or a maximum of 1280 bytes. It may or may not be beneficial for your driver to check for a controller that supports this bit and 1280 byte transfers. Just because the controller will transfer 1280 bytes per transaction, doesn't mean that the USB device attached will.

Another bit, bit 2, in the first Misc. Control register is used for power management. Setting this bit to 1 will enable power management. See the section later in this chapter.

Bits 2:1 of the second misc. control register are used along with the Legacy Support register at offset 0xC0. Legacy support is used for keyboards and mice when used on a non-USB aware platform, mostly in BIOS software. Have you ever wondered how your USB keyboard was working when running an older DOS version, when you know that DOS did not support USB? That is were the USB legacy support comes in to play. For more on legacy support, see chapter 5, page 38 of the Intel UHCI specification, link found in Appendix A.

Other controllers may have similar misc. control registers and a legacy support register and may be at the same offset as described above. However, to make sure, always get the specifications for that particular brand of controller. It may be different for each controller brand. The information above is just an example.

Offset 0x60 for UHCI, EHCI, and xHCI Controllers

Another example would be the byte at offset 0x60 in the PCI Configuration Space of the UHCI, EHCI and xHCI controllers. This byte is the revision number of the USB Specification that the controller supports. A value of 0x10 indicates that the controller supports version 1.0 of the specification, with a value of 0x00 indicating that it supports a version prior to version 1.0. A value of 0x20 means USB 2.0, and a value of 0x30 means USB 3.0. Your driver should not try to use the controller if you find a value other than 0x10 (UHCI), 0x20 (EHCI), or 0x30 (xHCI) at this location.

The OHCI doesn't use offset 0x60 as explained above. This is probably due to that Intel didn't have anything to do with OHCI, yet had a lot to do with UHCI, ECHI, and xHCI.

EHCI and Companion Controller Function Numbers

When you find an EHCI controller, the PCI function number may be 0 to 7, since eight functions are allowed per PCI device. If the EHCI controller has companion controllers, either in the form of OHCI or UHCI, these controllers must have a function number less than the function number of the EHCI controller.

For example, if an EHCI controller has two companion controllers, the function numbers of each of these controllers may be 0 and 1 while the function number of the EHCI controller may be 2. However, you can not assume that the numbers will always be consecutive or start with zero. For example, a PCI device number may have multiple functions, where three of these functions may be the EHCI and its companion controllers. These numbers may be 3 and 4 for the companion controllers and 7 for the EHCI controller.

One of the EHCI controllers I have has four UHCI companion controllers at functions 0, 1, 2, and 3, with the EHCI controller at function 7. Function numbers 4, 5, and 6 are not used.

You may assume that all companion controllers will be on the same device number and have a smaller function number than the EHCI controller has. The EHCI controller will not be associated with any other UHCI or OHCI controller on another PCI device number. Those controllers may be stand-alone controllers are be associated with another EHCI controller.

If the EHCI controller has a function number of zero, this indicates that there are no companion controllers and that this EHCI controller only supports high-speed devices. However, please remember that an external high-speed USB hub may be used on one of the root hub's ports for low- and full-speed devices.

The PCI and Power Management

If the attached controller supports Power Management, it will have a PCI Power Management Extended Capabilities register set in the PCI address space. The list of Extended Capabilities starts at the offset given by the byte at offset 0x34 in the PCI address space. For example, if the value of this byte is 0x50, then byte offset 0x50 within the PCI address space starts the Extended Capabilities List.

One of the items within this list, will be the Power Management register set. For an example, see the xHCI's list in Chapter 9, Tables 9-12 and 9-13 for how this list is made.

The Power Management Register set consists of two 16-bit words, with the second word containing the following fields.

Table 2-3: The PCI PM Status and Control Register

PCI PM Status and Control Register		
Bits	Access	Description
1:0	RW	Power State: D3 = 11b, D0 = 00b
2	RO	Reserved
3	RO	No Soft Reset
7:4	RO	Reserved
8	RWS	PM Enable Messages
12:9	RO	Data Select
14:13	RO	Data Scale
15	RWC	PME Status

The Power State can be a state from D3 to D0, D0 being the maximum powered state, while D3 being the minimum power state.

It is the job of your PCI driver to check that bits 1:0 is in D0 state. When in the D0 state, the device is fully powered, the driver can read and write from the device's Memory Mapped or Port IO Registers, and the device is fully functional.

If the power state is other than D0, your PCI driver must write a value of 00b to bits 1:0 to power the device, usually you also write a 1b to bit 15 to clear PME Messages being sent. Then, waiting at least the required 10 mS, read back the value making sure bits 1:0 are now 00b. Some device may require a little more time for the transition, even though the PCI PM[3] specification states this transition should be no more than 10mS.

A PCI device must allow the D0, and $D3_{cold}$ and $D3_{hot}$ power states. The D1 and D2 states are optional. The $D3_{cold}$ state is when there is no physical power to the device, meaning the power has been disconnected. The $D3_{hot}$ state is when there is power connected to the device, it just needs to be set to a different power state using the PM Status and Control Register listed above, to use this power.

Chapter 3 – The UHCI Hardware

This chapter describes how to set up the UHCI host controller, a minimal stack frame, and start the controller. You should have read chapter 2 to know how to get the base I/O address and IRQ number from the PCI and be familiar with port I/O addressing.

The UHCI specification states that the UHCI controller will use Port I/O addressing, however this doesn't mean that a controller manufacturer will always conform to this statement. You will need to check bit 0 in the Base I/O address to be sure. Since I have never seen a UHCI controller use anything other than Port I/O addressing, I will assume Port I/O addressing from here on for the UHCI controller access.

A majority of the information in this section, including most of the tables is readily available from the Intel UHCI specification. This information is included here only for reference, and to clear up a few things that may be a little confusing. To get the original tables, see Appendix A for information on how to get the specification file.

The UHCI USB controller has a defined set of I/O registers starting at the Base Address described in Chapter 2. These registers are defined in the following table.

Table 3-1: The UHCI controllers I/O registers

The UHCI controllers I/O registers			
Offset	Size	Access[1]	Description
00h	2	R/W	USBCMD: USB Command Register
02h	2	R/WC	USBSTS: USB Status Register
04h	2	R/W	USBINTR: USB Interrupt Enable
06h	2	R/W[2]	FRNUM: Frame Number
08h	4	R/W	FRBASEADD: Frame List Base Address
0Ch	1	R/W	SOFMOD: Start Of Frame Modify
0Dh	3	RO	Reserved
10h	2	R/WC[2]	PortSC1: Port 1 Status/Control
12h	2	R/WC[2]	PortSC2: Port 2 Status/Control
[1]RO = Read Only, WO = Write Only, R/W = Read/Write, R/WC = Read/Write Clear [2]These are word write-able only. Byte writes are unpredictable.			

If this UHCI controller is a Companion controller to an EHCI controller, you may need to make sure the Configure Flag of the EHCI controller is clear or your UHCI code may not see its own controller. See Chapter 7 for more on this Configuration Flag and Companion controllers.

Some of the bits in the PortSCx registers are WC. If you are unfamiliar with the term WC, or Write Clear, this means you write a 1 to that bit to clear it. Writing a zero to a bit that is a WC bit has no effect on the bit.

In the following tables, I will give detail on the fields that you will need to learn about, and little or no detail on the fields that we will not discuss and are most likely not needed for the purpose of this book. Again, see the Intel specification referred to before for more information on these fields.

The command register USBCMD tells the controller what we want to do and how we want to do it. Bits 7:6 and 2:0 are the most important at the moment and will be discussed below.

Table 3-2: The UHCI Command Register

UHCI controller: Command (Base + 00h) (Word) (R/W)	
Bit	Description
15:8	Reserved
7	Max Packet (MAXP). 1 = 64 bytes, 0 = 32 bytes.
6	Configure Flag (CF)
5	Software Debug (SWDBG). 1 = debug mode, 0 = normal mode.
4	Force Global Resume (FGR)
3	Enter Global Suspend Mode (EGSM)
2	Global Reset (GRESET)
1	Host Controller Reset (HCRESET)
0	Run/Stop (RS) 1 = run, 0 = stop.

Bit 7 selects the maximum packet size that can be used for full-speed bandwidth reclamation at the end of a frame. This value is used by the Host Controller to determine whether it should initiate another transaction based on the time remaining in the Start of Frame (SOF) counter.

Bit 6 is simply a semaphore to let the host set this bit to indicate that it has configured the controller. This bit has no function as far as the controller is concerned. Setting or clearing this bit does not change the status of the controller.

Bits 5:3 are used for software debugging and resume to scheduling. At this time, you only need to know that these bits are here and should set them all to zero when writing to this register.

Bit 2 is used for global resetting. When this bit is set, the host controller sends the global reset signal on the USB and then resets all its logic, including the internal hub registers. The hub registers are reset to their power-on state. This bit needs to be reset to zero by your code after a minimum of 10 mS.

Bit 1 is the host controller reset. When this bit is set, the host controller resets its internal timers, counters, and state machines to their initial value. Any transaction currently in progress on bus is immediately terminated. The host controller clears this bit when the reset process is complete. We will discuss more about this bit further in this chapter on resetting the controller.

Bit 0 is the Run/Stop bit. When set to a 1, the host controller proceeds with execution of the schedule. The Host Controller continues execution as long as this bit is set. When this bit is cleared the host controller completes the current transaction on the USB and then halts. The HCHALTED bit in the status register indicates when the host controller has finished the transaction and has entered the stopped state. The host controller clears this run/stop bit if a fatal error occurs, including consistency check failure and PCI bus errors.

Table 3-3: The UHCI Status Register

UHCI controller: Status (Base + 02h) (Word) (R/WC)	
Bit	Description
15:6	Reserved
5	Controller Halted
4	Host Controller Process Error
3	Host System Error
2	Resume Detect
1	USB Error Interrupt
0	USB Interrupt (USBINT)

A few notes on this register; Remember that WC is Write Clear, a write of 1 to this bit, clears the bit. I have found, though the specification doesn't mention it, that bit 5 is read-only. A write to it has no effect. I will discuss more on the bits in this register later in this chapter.

Table 3-4: The UHCI Interrupt Enable Register

UHCI controller: Interrupt Enable (Base + 04h) (Word) (R/W)	
Bit	Description
15:4	Reserved
3	Short Packet Interrupt Enable
2	Interrupt on Complete (IOC) Enable
1	Resume Interrupt Enable
0	Timeout/CRC Interrupt Enable

The bit settings of the Interrupt Enable register are set or cleared just before normal schedule execution. See a further section in this chapter for more on these settings.

Table 3-5: The UHCI Frame Number Register

UHCI controller: Frame Number (Base + 06h) (Word) (R/W)	
Bit	Description
15:11	Reserved
10:0	Frame List Current Index/Frame Number

This register contains the current frame number. It is usually set to zero when the schedule is first started. During schedule execution, you can read this register to find which frame the schedule is currently executing. All writes must be word writes, but you can read with byte reads, though who would want to do that? See the section on controller initialization for more on this register.

Table 3-6: The UHCI Frame Base Register

UHCI controller: Frame Base (Base + 08h) (DWord) (R/W)	
Bit	Description
31:12	Base Address (See note below)
11:0	Reserved. Must be written all zeros

This register holds the physical memory address of the 4k sized stack frame. This register has bits 11:0 cleared to zero with bits 31:12 corresponding to memory address bits 31:12. With this in mind, this means that the 4k stack frame must be 4k aligned. See the section on controller initialization for more on this register.

Table 3-7: The UHCI Start of Frame Register

UHCI controller: Start of Frame (SOF) (Base + 0Ch) (Byte) (R/W)	
Bit	Description
7	Reserved
6:0	SOF Timing Value

This register is used to modify the value in the generation of SOF timing. Only the 7 least significant bits are used. The default value is 64 and is the value we will use throughout this book. At this time, there is no need to set it differently.

Table 3-8: The UHCI Port Status Register

UHCI controller: Port Status/Control (Base + 1xh) (Word) (R/W)	

Bit	Description
15:13	Reserved
12	Suspend (R/W) 1 = Port is in a suspend state. This bit should not be written to a 1 if global suspend is active (bit 3 in USBCMD). Bit 2 and bit 12 of this register define the hub states as follows: 12:2 state x0 Disable 01 Enable 12:2 state (cont.) 11 Suspend
11:10	Reserved (see end of chapter for a note on this field)
9	Port Reset (R/W) 1 = Port is in Reset. Software sets this bit to reset the port and clears it after a predefined time of 50mS. See later in this chapter on the reset.
8	Low Speed Device attached (RO). 1 = A low speed device is attached, else a full speed device.
7	Reserved (R0) Always read as 1.
6	Resume Detect (R/W) 1 = Resume detected on port.
5:4	Line Status (RO) These bits reflect the D+/D- signals.
3	Port Enable/Disable Change (R/WC) 1 = Port Enable/Disable has changed.
2	Port Enable/Disable (R/W) 1 = Enable Port.
1	Connect Status Change (R/WC) 1 = Change in Current Connect Status. A disconnect or connection was detected.
0	Current Connect Status (RO) 1 = Device is present on port.

The UHCI host controller, also known as the root hub, has two or more ports for down-stream devices. Each port has a register of the above table and is independent of each other. See the section on device status detection and enumeration further in this chapter for more information on these port values. The first port, PORTSCx (where x = zero) is at Base + 10h while the second is at Base + 12h, and any remaining are 2 bytes after the last one.

UHCI Host Controller Detection

Since the PCI bus already told us that there is a UHCI controller at this base, why should we take the time and effort to detect the controller? To be completely independent of the host bus, the UHCI driver should detect the controller rather than relying on its host bus. Future hardware may have the USB on a different bus type that isn't so accommodating.

To detect the controller, we will attempt to reset it and then see what happens to the port values.

There are two bits in the Command register to reset the controller. One is the Global Reset and one is the Host Controller Reset. These two forms of reset are similar, but have a few differences you will need to know. The global reset sends reset to all the USB devices connected to this root hub. This means that any device downstream will see the reset. It is also the job of the software to reset this bit to zero after 10mS. The Host Controller Reset, or HC Reset, only resets the internal state of the controller, which effects bits 8 and 3:0 on the Port Status/Control registers. The controller resets bit 1 to zero when it is done after a HC Reset.

With this in mind, and knowing that after global reset, all registers are set to their default state, do a global reset and check for these default values. The following snippet of code comes from the FYSOS UHCI driver. The driver calls this code expecting a TRUE or FALSE condition on return, returning TRUE if a UHCI controller was detected on this base address. The mdelay() routine simply delays for a given amount of milliseconds.

Listing 3-1: UHCI Host Controller Reset

```
bool det_uhci_root(bit16u base, bool io_space) {

 if (io_space) {
   // do a global reset (5 times 10mS each)
   for (int i=0; i<5; i++) {
     outpw(base+UHCI_COMMAND, 0x0004);
     mdelay(11); // wait 10+ mS
     outpw(base+UHCI_COMMAND, 0x0000);
   }

   // is the command register its default value of 0x0000 ?
   if (inpw(base+UHCI_COMMAND) != 0x0000) return FALSE;
   // is the status register its default value of 0x0020 ?
   if (inpw(base+UHCI_STATUS) != 0x0020) return FALSE;
   // The status register is write clear, let's clear it out.
   outpw(base+UHCI_STATUS, 0x00FF);

   // is the SOF register its default value of 0x40
   if (inpb(base+UHCI_SOF_MOD) != 0x40) return FALSE;

   // if we set bit 1 in the Command register, after a specified
   // time, the controller should reset it to 0.
   //  (I guessed the 42ms max value)
   outpw(base+UHCI_COMMAND, 0x0002);
   mdelay(42);  // wait 42 mS
   if (inpw(base+UHCI_COMMAND) & 0x0002) return FALSE;
 } else {
   // is memmapped space and is ignored for this book.
 }

 return TRUE;
}
```

Set up the UHCI controller

Now that we know we have a UHCI controller, we can start to set up the controller and our USB stack.

The controller has already been reset by the detection routine before, however there are a few more items we need to set up before we have a working USB stack schedule.

First, let's set the Interrupt Enable register. We will set all four bits so that the controller will interrupt on a short packet, a completion of a frame, a resume, and a timeout/CRC error interrupt. See the next chapter for a description on what these four interrupts are.

```
outpw(base+UHCI_INTERRUPT, 0x000F);
```

Now make sure the Frame Number Register is zero.

```
outpw(base+UHCI_FRAME_NUM, 0x0000);
```

Now set the Frame List Base Address register to point to our stack frame. For current purposes we will assume that we already have a valid stack frame at address 0x12345000. The FYSOS code actually calls a function to allocate and initialize a stack frame, returning the address. See the next chapter for a description on how to make a valid stack frame and the items it uses. For example purposes, write 0x12345000 to this register.

```
outpd(base+UHCI_FRAME_BASE, 0x12345000);
```

The Start of Frame register should already be set to 0x40 (64d), but to be sure.

```
outpb(base+UHCI_SOF_MOD, 0x40);
```

Setting up and handling the bits in the Port Status/Command register(s) is enough for a new section, therefore, see the next section on the setting of the port register and detecting the device status.

You are probably wondering why I didn't do the Status and Command registers first. The Command register has the Run/Stop bit and we have to have a valid stack frame and host controller state before we can set the schedule to *Run*. Also, the status register should be cleared just before you start the schedule so that you don't have any residue bits from a previous schedule. Remember that the Status register is Write/Clear. To clear the register, write 1's to each bit. I write 0xFFFF to the register, so that I get all the bits. This way if there ever is another bit specified in an updated specification, I already have it taken care of. The controller doesn't complain that I wrote to the reserved bits.

```
outpw(base+UHCI_STATUS, 0xFFFF);
```

At this point, we have set up all of the HC registers but the Command register. The Command register has three bits that we are concerned with at the moment. For now, we will clear bit 7. Later, when we detect a full speed device, we will set it for 64-byte max packet size on full-speed reclamation. Bit 6 is a "we have configured everything" flag. It has absolutely no effect on the controller. The UHCI specification calls it a semaphore flag, and it is just that, a flag that we set to tell the driver that we have set everything and are ready to start the schedule.

Finally, set bit 0. This tells the controller to start the schedule.

As for the UHCI hardware, we have completely set it up. The next thing to do is to start sending and receiving packets from the schedule. After a section or two on detecting device status and other device related items, I will get back to this topic on how to send and receive

packets to and from the schedule. For now, let's move on to the Port Status/Control register and device detection.

The UHCI Port Status/Control Register

When the user plugs or unplugs a device into/from one of the root hub's ports, that port's status bits are set or cleared depending on the current status.

> At bootup, if a device is already attached (plugged in), the status of this register is identical to if you had just attached the device during normal operation, even though the device was attached before the machine was turned on.

Just after boot up, an empty port will contain a value of 0x058A (0000_0101_1000_1010b), though see a following note on some of the bits in this value.

Bits 15:13 should be zero. Bit 12 should be zero, suspend disabled. Bits 11:10 are reserved, however, if you notice, on my UHCI controller, bit 10 is set. The specification does not say that these two bits should be zero, they are just reserved. With this in mind, the default value of this register in this current state may be different on different controllers due to these two bits. See the notes at the end of this chapter for more on these two bits.

Bit 9 should be zero, not in reset. Bit 8 will be undefined since there is nothing attached. This will also adjust the default value of different controllers. This bit may be set or clear with no definition, when a device is not attached.

Bit 7 always reads as one. This is a perk that will allow us to test to see how many ports are on this controller. See later in this chapter for more on this bit.

Bit 6 should be zero, since no resume was detected. Bits 5:4 should also be zero since no device is attached. Hence, the D+ and D- lines can't be up.

Bit 3 is the port enable changed bit and will be set by the controller so that the host controller's driver will detect the change and read bit 2, the port enable bit. On an empty port, this bit should be zero. An empty port cannot be enabled.

Bit 1 should be set by the controller so that the host controller's driver will detect the change and read bit 0, the connect status bit. On an empty port, this bit should be zero, no connection.

The driver should notice that the connection change bit (bit 1) is set. Your code should clear it by writing a one to it, then check the current connect status bit (bit 0). On an empty port, notice that the bit is clear. Since we have just booted the machine, we know that a connection change was not made so we will ignore the change, though making sure we clear bit 1 (by writing a 1 to it). While you are at it, clear the enable change bit, bit 2.

From this state, when a device is plugged in, the port's value should now read as 0x05A3 (0000_0101_1010_0011b), again remembering the undefined bits stated before. If a device is attached, the connection status change bit is now set, stating that we have had a connection change, with the current connection bit set stating that we now have a device attached. Clear bit 1 by writing a one to it.

> Please see Appendix D, "Port Connection Status Change" for an important note about connect status changes.

Resetting a port

The next step is to reset the port. Resetting a port on an external hub is different than resetting a root hub port. For external hubs, please see the chapter on external hubs and resetting their ports.

To reset a root hub port, you set bit 9 in the port's register, wait 50mS, clear the bit, and then wait 10mS for the recovery time. The USB specification says that a root hub port reset should be at least 50mS, however, this does not have to be a continuous 50mS. It can be five continuous cycles of 10mS each, though the minimum cycle should be 10mS each.

The reason for the 10mS cycle is that an external hub only needs 10mS to reset. For a root hub port reset, just call the reset routine five times, making sure there is less than 3mS between calls.

All low- or full-speed devices that I have seen reset correctly with this technique.

Now that we have a device attached and reset, we can read bit 8 to see if we have a low- or full-speed device and set our driver accordingly. Remember that a high- or super-speed device will attach as a full-speed device on a UHCI controller. The next step is to enable the port.

Enabling the port

Enabling a port on an external hub is different than enabling a root hub port. Please see the chapter on external hubs for enabling their ports.

The USB 2.0 specification states that after a port is successfully reset, the controller will enable the port and set the enable bit. However, the UHCI controller does not do this. The UHCI specification states that ports can be enabled by host software only. This is because the UHCI was created before USB 2.0 was released. Therefore, we must enable the port ourselves.

To enable the port, write 0xF to bits 3:0. This clears bit 3 and bit 1, while the write to bit 0 is ignored, and setting bit 2, enabling the port. However, the port is not instantly enabled. It may take a few bit-times to enable. A 16-iteration loop, polling bit 2 until set, waiting 10mS between each iteration will do nicely. If the bit becomes set while within this loop, the port has been enabled. However, if after the 16-iterations and bit 2 does not become set, then there has been an error enabling the port. I haven't seen this error yet, but a defective device may cause the port to not enable.

Listing 3-2: UHCI Port Reset

```
bit16u val;
int ret = UNSUCCESSFULL;

uhci_set_rh_bit(base, port, UHCI_PORT_PR);
mdelay(USB_TDRSTR);  // at least 50 ms for a root hub
uhci_clr_rh_bit(base, port, UHCI_PORT_PR);
mdelay(USB_TRSTRCY);  // hold for 10ms (reset recovery time)

for (int i=0; i<10; i++) {
  val = inpw(base + port);

  // if bit 0 is clear, nothing attached, don't enable
  if (!(val & UHCI_PORT_CCS)) {
    ret = SUCCESS_RESET;
    break;
  }

  // if either enable_change or connection_change,
  // clear them and continue.
  if (val & (UHCI_PORT_PEC | UHCI_PORT_CSC)) {
    uhci_clr_rh_bit(base, port,
                    (UHCI_PORT_PEC | UHCI_PORT_CSC));
    continue;
  }

  // if the enable bit is set, break.
  if (val & UHCI_PORT_PE) {
    ret = SUCCESS_RESET;
    break;
  }

  // else, set the enable bit
  uhci_set_rh_bit(base, port, UHCI_PORT_PE);
}
```

Now that the port has been enabled, the driver knows that there is a device on this port, we can have the driver enumerate the device attached. This is discussed in the next chapter. For now, we will discuss a few other items pertaining to the UHCI port hardware.

Detecting multiple ports on a root hub

I want to get back to bit 7 always being read as 1. I have written a few routines for detecting multiple ports on a root hub, with bit 7 always being set as a main factor. The routine I have kept, does a nice job though has one drawback, it clears bits 3 and 1 before we get to our device detection code. This means that after a reset, then this *port-count* detection, the two change status bits are clear. In FYSOS, I have taken this into account at initialize time and check for bit 0 whether bit 1 is set or clear.

The FYSOS code calls the routine below, incrementing the port offset by two, until this routine returns FALSE, incrementing a counter each time it returns TRUE.

Listing 3-3: UHCI Port Present

```
bool uhci_port_pres(bit16u base, bit8u port) {

 // if bit 7 is 0, not a valid port
 if ((inpw(base+port) & 0x0080) == 0) return FALSE;

 // try to clear bit 7
 outpw(base+port, inpw(base+port) & ~0x0080);
 if ((inpw(base+port) & 0x0080) == 0) return FALSE;

 // try to write/clear it
 outpw(base+port, inpw(base+port) | 0x0080);
 if ((inpw(base+port) & 0x0080) == 0) return FALSE;

 // if we write a 1 to bits 3:1, do they come back as zero
 outpw(base+port, inpw(base+port) | 0x000A);
 if ((inpw(base+port) & 0x000A) != 0) return FALSE;

 // should be able to assume this is a valid port if we get here
 return TRUE;
}
```

You should now be ready to enumerate each device on an attached port. After a few more notes, we are ready to go on to the next chapter.

Miscellaneous UHCI Notes

On the Intel 430TX and later motherboards, the two reserved bits, bits 11:10 are actually used for over-current conditions. Bit 11 is used to indicate that there was a change in the over-current condition and is Read/Write Clear while bit 10 is Read-only and indicates whether there is an over-current condition. However, now for the reason why bit 10 is set. Intel chose to have bit 10 set to indicate normal operation while being clear indicated an over-current condition.

Also, note that if you have a VIA UHCI Host Controller with a model number of VT83C572 or VT82C586, these chipsets may be unsupported due to problems in the input/output (I/O) port aliasing that can only be resolved in the chips themselves.

The UHCI specification states that the PCI Base register, the register that holds the base address of the UHCI's Operational registers, has bits 4:1 zero. This means that the Operational Register set must be 32-byte aligned. However, since the PCI allows it to be only 4-byte aligned (Port I/O addressing), be sure to allow for this and calculate bits 4:2 into the calculated address. i.e.: Don't assume bits 4:2 are zeros.

I only know of VIA Technologies, Inc. UHCI Host Controllers having the Misc. Control Registers, offset 40h and 41h in the PCI Address space. However, they may exist in other manufacturer's controllers. I include them here for interest sake. See Appendix A for information on where to get the VIA document of the table below.

Someday I may try to find this controller, plug it in, then see if I can get it to fail, as mentioned above, and even see if I can play with the Power Management portion, using bit 2 in the table below.

Table 3-9: The VIA VT83C572 Misc. Control Registers

CIA VT83C572 Misc. Control Register (0x40) (Byte) (R/W)	
Bit	Description
7	PCI Memory Command Option 0 - Support Read Line, Read Multiple, Write, & Invalidate 1 - Only support Memory Read & Write Commands
6	Babble Option 0 - Automatically disable port when EOF babble occurs 1 - Don't disable babbled port
5	PCI Parity Check Option 0 - Disable PERR generation 1 - Enable parity check and PERR generation
4	Reserved
3	USB Data Length Option 0 - Support TD length up to 1280 1 - Support TD length up to 1023
2	USB Power Management 0 - Disable USB power management 1 - Enable USB power management
1	DMA Option 0 - 16DW burst access 1 - 8DW burst access
0	PCI Wait State 0 - Zero wait 1 - One wait
CIA VT83C572 Misc. Control Register (0x41) (Byte) (R/W)	

7:3	Reserved
2	Trap Option 0 - Set trap 60/64 status bits w/o checking enable bits 1 - Set trap 60/64 status bits only when trap 60/64 enable bits are set
1	A20gate Pass Through Option 0 - Use pass through A20GATE command sequence 1 - Don’t pass through Write I/O port 64
0	Reserved

The VIA Technologies, Inc. VT6212/ VT6212L also has these misc. registers, and also includes four more for its UHCI companion controllers.

Chapter 4 – The UHCI Stack

By this chapter, you should know how to detect the UHCI host controller through the PCI configuration space and physical detection at the port I/O level. You should also know how to detect that a device is present on one of the ports on the host controller, also known as the root hub. You should also now know how to point the root hub to our 4k stack, though we haven't discussed the actual format of this stack, and now be able to start the schedule.

If you are comfortable with this up to this point, we will start on the actual UHCI communication protocol and how to detect the device or devices already attached. If you are not ready to continue on, please go back and study the previous chapters.

The UHCI Stack

The UHCI stack consists of 1,024 dword entries on a 4k boundary in memory. Each of these entries is called a Frame Pointer. The host controller's scheduler reads the current frame pointer, pointed to by the Current Frame Number register and executes the frame that it points to. At 1mS, the host controller stops executing that frame, increments the Current Frame Number register, wrapping to 0 at 1024, and starts executing the next frame. This means that each frame will be executed approximately once a second, every 1.024 seconds to be exact, or at 1 Hz. Take a quick look at Figure 4-1 on page 4-7 for an example. Hurry back.

At 1 Hz, this seems very slow, doesn't it? However, this is where the term *stack* comes into play. Each frame pointer may, and most of the time does, point to a queue that can have an unlimited number of stack items. The host controller executes this queue until the 1mS time is up. After the host controller has executed all other frames in the cycle and has returned to this frame, it resumes where it left off 1024 milliseconds ago. An explanation of how it keeps track of this position within a frame is explained later in this chapter.

Let's look at the Frame List, the 4k of memory that you placed the address of into the Frame Base register of the Host Controller. Again, it is 1,024 dword-sized entries. Each dword frame entry has the following format.

Table 4-1: The UHCI Frame Pointer

bits 31:4	3	2	1	0
Frame Pointer	0	0	Q	T

The Frame Pointer entry is a 32-bit pointer to another location in memory that most likely is a Queue mentioned before. Please note that all pointers point to memory addresses that are paragraph (16-byte) aligned, so the Frame Pointer field is the high 28 bits of the address, while the low 4 bits are assumed zero. When reading this field, read the full dword frame entry and simply mask off the last 4 bits. The host controller uses this alignment to an advantage. It uses the four low bits in the frame entry as flags.

Since there are 1,024 dwords, or 1,024 Frame Pointers in the UHCI's schedule, this means that there are 1,024 frames. Each frame pointer points to a single frame. At first execution of a frame, the controller starts with the first item. As it processes an item, it updates the pointers within the items so that when the controller returns to this frame, it knows were to continue execution. See later in this chapter for more information on this technique.

Bits 3:2 are reserved and should be written as zeros. Bit 1 is the queue flag. If this bit is set, the address this entry points to is a Queue. If this flag is clear, the address is a Transfer Descriptor. See below for more on these two terms. The last flag, the T flag, is the Terminate flag. If this bit is clear, the pointer points to a valid Queue or Transfer Descriptor. If this bit is set, this frame entry is not to be executed. The host controller simply waits for the 1mS time to lapse, then moves on to the next entry.

Transfer Descriptors

A Transfer Descriptor is an eight-dword block of memory with the format listed below.

Table 4-2: The UHCI Transfer Descriptor

bits 31:4								3	2	1	0
Link Pointer								0	D	Q	T
31:30	29	28:27	26	25	24	23:16	15:11	10:0			
R	SPD	CERR	LS	IOS	IOC	Status	R	ActLen			
31:21		20	19	18:15		14:8		7:0			
MaxLen		R	D	EndPt		Device address		PID			
Buffer Pointer											
Reserved											
Reserved											
Reserved											
Reserved											

A Transfer Descriptor can point to another Transfer Descriptor or to a new Queue via the Link Pointer field. A Transfer Descriptor contains all the data it needs to make a successful transfer, with each field described next.

Note that all pointers in any field are 32-bit, so all memory used by any field must be below the 4Gig limit. Therefore, on 64-bit machines, make sure to reserve space below this limit for the UHCI controller.

Table 4-3: The UHCI Transfer Descriptor Definition

Transfer Descriptor Definition		
Name	HC Access	Description
T	RO	**Terminate**. If set to 1, this tells the controller that this Transfer Descriptor is the last in the linked list, and that the Link Pointer field is not valid.
Q	RO	**QH/TD Select**. This tells the controller what type of item the Link Pointer points to. 0 = Transfer descriptor 1 = Queue
Vf	RO	**Depth/Breadth Select**. This field is only valid when this Transfer Descriptor is in a Queue, and tells the controller to either continue with the next item pointed to by Link Pointer (Depth first) or go back to the last Queue and process the next horizontal item (Breadth first). 0 = Breadth first. 1 = Depth first.
LP	RO	**Link Pointer**. This is bits 31:4 of the link pointer, pointing to the next item in the list. Bits 3:0 are assume zero. To get this field, read the whole dword and mask off the lower four bits.
ActLen	R/W	**Actual Length**. The controller will fill this field with the actual length of the data transferred minus 1. When 8 bytes are requested, this value will be 0x07 on a successful transfer, and

<table>
<tr><td></td><td></td><td colspan="2">contain the actual amount transferred, minus 1 (0 = 1 byte was transferred).</td></tr>
<tr><td>R</td><td>RO</td><td colspan="2">Reserved</td></tr>
<tr><td>Status</td><td>R/W</td><td>Bit</td><td>Description</td></tr>
<tr><td></td><td></td><td>23</td><td>Active. Set this bit to 1 to enable execution of this Transfer Descriptor. The controller will clear this bit after execution of this Transfer Descriptor.</td></tr>
<tr><td></td><td></td><td>22</td><td>Stalled. Set to 1 by the controller if a stall handshake is encountered.</td></tr>
<tr><td></td><td></td><td>21</td><td>Data Buffer Error. Set to 1 by the controller if an overrun or underrun is encountered.</td></tr>
<tr><td></td><td></td><td>20</td><td>Babble Detected. Set to 1 by the controller if a babble was detected.</td></tr>
<tr><td></td><td></td><td>19</td><td>NAK Received. Set to 1 by the controller if a NAK packet was detected.</td></tr>
<tr><td></td><td></td><td>18</td><td>CRC/Time Out Error. Set to 1 by the controller if a CRC error or a timeout error was detected.</td></tr>
<tr><td></td><td></td><td>17</td><td>Bitstuff Error. Set to 1 by the controller if a bitstuff error was detected.</td></tr>
<tr><td></td><td></td><td>16</td><td>Reserved</td></tr>
<tr><td>IOC</td><td>R/W</td><td colspan="2">Interrupt on Complete. This bit tells the controller to fire an interrupt when end of frame is encountered, if the IOC bit is set in the Interrupt Enable register. End of frame is considered when the 1mS time has elapsed, not when the entire frame is complete.</td></tr>
<tr><td>ISO</td><td>R/W</td><td colspan="2">Isochronous Select. This bit tells the controller that this is an Isochronous Transfer Descriptor.</td></tr>
<tr><td>LS</td><td>R/W</td><td colspan="2">Low Speed Device. Your driver sets this bit if the endpoint is a low speed device.</td></tr>
<tr><td>C_ERR</td><td>R/W</td><td colspan="2">Error Count. Your driver sets this field to 11b. When an error is detected, the controller will decrement this field and try again. If zero is reached, it will fail the transaction and set the corresponding status bits. This field can be set to 11b, 10b, 01b, or 00b. However, the controller will continue trying to execute the transaction indefinitely with an error count of 00b.</td></tr>
<tr><td>SPD</td><td>R/W</td><td colspan="2">Short Packet Detect. This tells the controller to fire an interrupt if the size of the packet it transfers is less than the max packet size we specify in the Max Length field if the SPD bit in the Interrupt Enable register is set.</td></tr>
<tr><td>R</td><td>RO</td><td colspan="2">Reserved</td></tr>
<tr><td>PID</td><td>RO</td><td colspan="2">Packet Identification. This tells the controller what kind of packet this is, IN (0x69), OUT (0xE1), or SETUP (0x2D).</td></tr>
<tr><td>ADDR</td><td>RO</td><td colspan="2">Device Address. At enumeration time, your driver assigns a value of 1 to 127 to each device attached.</td></tr>
</table>

EndPt	RO	**Endpoint**. Each device may contain up to 16 endpoints. However, I have not seen a device have more than three, the default endpoint, the in endpoint, and the out endpoint. This does not mean that it cannot have more.
D	RO	**Data Toggle**. This is used to make sure that the controller receives the data in the correct sequence. In a control packet, the first Transfer Descriptor (SETUP), set this bit to 0, in the second (IN/OUT), set this bit to 1, in the third, set this bit to 0, and so on. A control transfer will always have the last Transfer Descriptor's data toggle bit set. On a bulk transfer, each packet's toggle bit will be set to the opposite of the one before, the packet of the same direction.
R	RO	Reserved
MaxLen	RO	**Maximum Length**. Your driver writes to this field the maximum number of bytes to transfer (for this transfer descriptor only), minus 1. To transfer 8 bytes, you write 7 to this field. Write 0 to transfer 1 byte.
Buff	RO	**Buffer Pointer**. The physical address of the buffer to transfer. This buffer can be byte aligned.

A few more notes on the previous fields.

If you have a low-speed device, you cannot transfer more than 8 bytes at a time, therefore you should place 0x07 in the Maxlen field. This field is to make sure that the controller doesn't copy any bytes beyond the end of your buffer. However, if you do place a value larger than 0x07, when using a low-speed device and receiving an 8-byte packet, you will get a SPD interrupt if the SPD bit is set. Usually this field is set to the remaining size of the buffer or to the size of the max packet allowed (minus 1), whichever is smaller.

When creating a transfer descriptor, you should set the status bits to 0x80, telling the controller that this transfer descriptor should be executed and also clearing the return status bits.

Remember that a frame period includes all Queues and Transfer Descriptors that it can execute in the 1mS time frame and the IOC interrupt is not fired until then. Do not expect an IOC interrupt after each Transfer Descriptor with the IOC bit set. If the IOC bit is set, an interrupt is fired even if the active bit is clear in any transfer descriptor it finds. The controller will only fire the interrupt once at end of frame, no matter how many IOC bits it found set.

When the controller is finished executing a Transfer Descriptor, and is in a Queue, it copies dword 0 of the Transfer Descriptor to the second dword in the last Queue encountered.

If you notice, the PID field is actually two 4-bit fields. The first field is the PID value, while the second field is the inverse of. For example, 0x2D is 0010_1101b. Notice that the high nibble is 0010b which is 0x2, and the low nibble is the inverse, 1101b, 0xD.

The last four dwords of a Transfer Descriptor are reserved, and the controller ignores any values in these fields. However, the UHCI specification says that the controller will not modify these fields. Therefore, you can use these fields to store data that is specific to this Transfer

Descriptor, or any other data you would like. An example would be a signature value that corresponds with a Task ID or a Current Thread ID to make sure you have the right Transfer Descriptor. The FYSOS USB driver uses these fields for short packet information.

Queues

Most of the time, a frame pointer entry will point to a Queue. A UHCI Queue is two dword-sized frame pointer entries. Therefore, please note that the controller only specifies the first two dwords in the table below. Since all Queues must be 16-byte aligned, the FYSOS UHCI driver allocates four dwords and uses the last two for stack insertion.

Table 4-4: The UHCI Queue

bits 31:4	3	2	1	0
Head Link Pointer	0	0	Q	T
Element Link Pointer	0	R	Q	T
Previous Link Pointer*				
Not used*				

*Not specified in the UHCI specification. Used by the FYSOS driver.

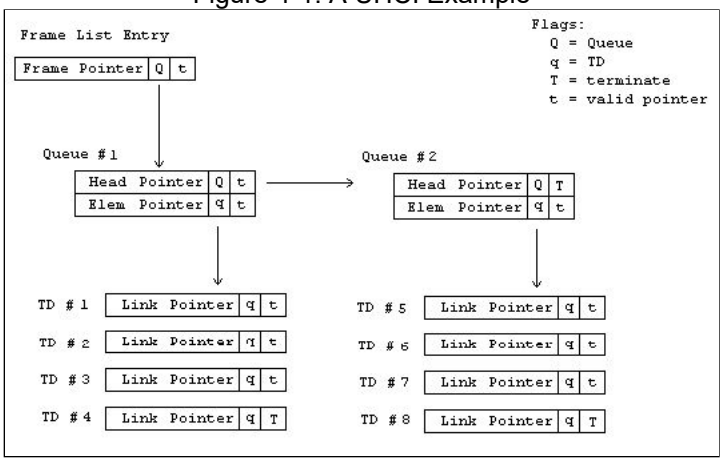
Again, the two link pointers are 32-bit pointers with the low four bits assumed zero. Both sets of Q and T fields are identical to the same fields in the Transfer Descriptor explained before, while the R field is simply a reserved field.

When a frame pointer entry, another Queue, or a Transfer Descriptor, points to a Queue, the Element Link Pointer is executed first. The controller treats this pointer just like a frame pointer entry. If the Q flag is set, the pointed to address is a Queue, if it is clear, the address points to a Transfer Descriptor. If the T bit in the Element Link Pointer of the Queue is set, it ignores the Element Link Pointer and moves to the Head Link Pointer.

However, there is a slight difference in the two addresses. When the controller is done executing a Transfer Descriptor, it will change the Element Link Pointer in the just executed queue to point to the next item in the frame. The next item's address is found in the Link Pointer of the Transfer Descriptor just executed. It also copies the Q and T bits from this Transfer Descriptor it just executed. This is how it knows what to execute the next time around. The controller does not modify the first dword of a Queue.

This is also where the Depth/Breadth flag of the Transfer Descriptor comes into play. If this flag is set in the Transfer Descriptor, and there is enough time left in this frame, the controller gets the next Transfer Descriptor entry from the Element Link Pointer of the Queue and continues execution. However, if the flag is clear, the controller gets the address from the Head Link Pointer and executes it. This may be a bit confusing if you have never seen it done before. To help explain it, see the example below.

Figure 4-1: A UHCI Example



I didn't put the D flag in the first dword of the Transfer Descriptors so that I could use the same figure as an example of both Depth and Breadth execution.

In the example, the controller has just incremented the Current Frame Number Register and now has read in the next Frame Pointer. It first checks the T flag, if it is set, it does nothing until the 1mS elapses. If it is clear, it then checks the Q flag to see if it needs to read in 8 bytes for a queue or 32 bytes for a Transfer Descriptor. This example uses a Queue, so the controller reads in 8 bytes. We are now at Queue #1. The controller checks the T flag of the second dword of this queue, the Element Pointer. It is clear, so it checks the Q flag to see how many bytes to read in from the address pointed to by Element Pointer. It is a Transfer Descriptor, so it reads 32 more bytes. We are now at TD #1. The controller executes this Transfer Descriptor. When it is done, it does a dword copy of the first dword in this Transfer Descriptor (TD #1) and writes it to the last executed Queue's Element Pointer dword, clearing the D field before the write but noting its value. The reason for this is that the controller always reads the queue to find out what to do next. Since the Element Pointer of the Queue is always updated with a pointer to the next Transfer Descriptor or Queue, we should always have a valid pointer in the Queue's Element Pointer.

Now we have executed one Transfer Descriptor. This is where the Depth/Breadth bit comes in to play. If the bit was set, meaning Depth First, the controller continues with the Element Pointer in the Queue, which now points to TD #2. If the bit is clear, meaning Breadth First, the controller backs up to the Head Pointer of the Queue, checks the T bit, and if clear, loads the Queue or Transfer Descriptor from the location it points to and continues execution. Once the controller executes the Head Link Pointer part of the Queue, it will not go back to the Element Link Pointer of that Queue until the next execution of this frame, approximately 1,023 mS later.

After each Transfer Descriptor execution, the controller checks to see if it has enough time to execute the next Transfer Descriptor. If it does, it continues on. If it does not, it waits for the 1mS to expire and then moves to the next Frame List Pointer.

Since the Element Pointer of each Queue was updated with the next item, in this case a Transfer Descriptor, when the controller makes it back to this frame, it doesn't execute the previously executed Transfer Descriptors since the Element Link Pointer now points to the next Transfer Descriptor to be executed.

For example, if the controller ran out of time after executing TD #1 and after copying the Link Pointer of TD #1 to the Element Pointer of Queue #1, the next time the controller makes it to this Queue, the Element Link Pointer is now pointing to TD #2 since the Link Pointer of TD #1 pointed to TD #2.

With this in mind, you don't have to worry if the controller is going to finish all of the Transfer Descriptors before the 1mS time expires. You can be sure that when the controller comes back to this frame, it will start with the next item to execute, right where it left off.

At the end of the frame, when the 1mS time expires, if any of the Transfer Descriptors had the IOC bit set, and the IOC bit in the Enable Interrupt register was set, the controller will fire an interrupt.

Even though the controller will execute a list of Transfer Descriptors pointed to by the Frame List Pointer, meaning no queue was use, without complaint, what happens if the controller could not finish all the Transfer Descriptors in the allotted time? The controller will not modify the Frame List Pointer to point to the next Transfer Descriptor to be executed. The next time the frame is executed, it will run through the previously executed Transfer Descriptors to get to the current one to be executed. There is nothing wrong with that, it just wastes time. That is why it is the norm to use queues.

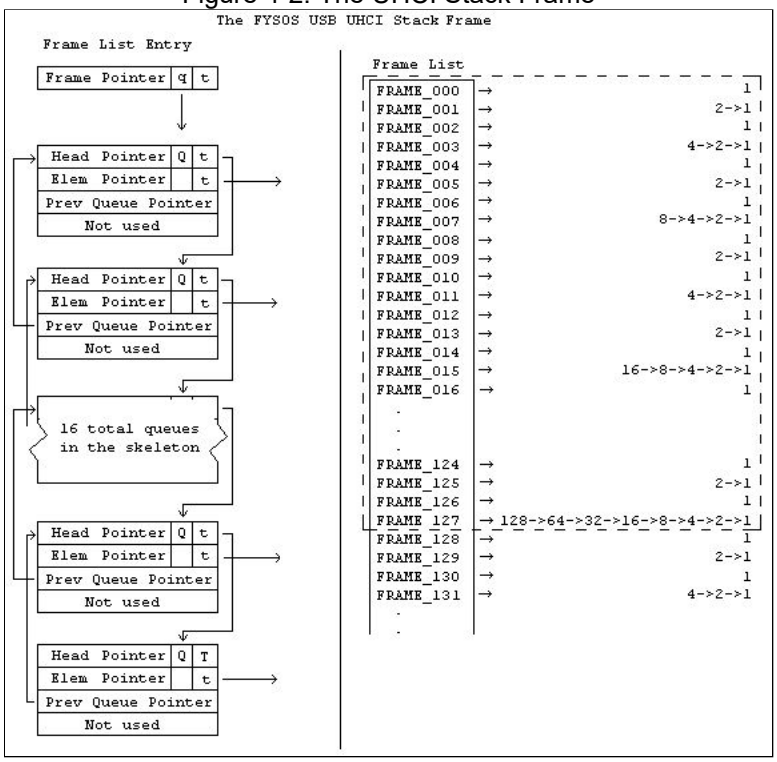
I recommend that you have at least one Queue for each frame, and point the Frame List Pointer to this Queue. This way, if the controller cannot execute all of the items in this frame within the allotted time, the controller only needs to load this Queue before it loads the next item to execute, rather than all of the other items before it.

The description on the previous page probably didn't make much sense, did it? It only makes sense if you have worked on the UHCI before. So, let's get to some examples, show how to use them, then come back to this technique so that it will make more sense.

The FYSOS stack frame

The figure below is a representation of the stack frame that the FYSOS UHCI driver uses for each UHCI controller. It has a frame list pointer that points to a frame with the items on the left, while the table on the right will be explained a little later in this chapter. Remember that the two last dwords in each queue is not used by the controller, but by the driver to create a mechanism to insert and remove queues from the stack.

Figure 4-2: The UHCI Stack Frame



We have 16 Queues that will always be present in our stack and for the purpose of this illustration, we will name them starting with the top Queue, Q128, Q64, Q32, Q16, Q8, Q4, Q2, Q1, QFS, QLS, QISO, Qresv0, Qresv1, Qresv2, Qresv3, and Qresv4. Using this naming convention, Q128 will be executed once every 128mS, Q64 will be executed once every 64mS, and so forth, with Q1 being executed every frame, or each millisecond. Each queue points to the next queue in the stack. Q128 points to Q64, points to Q32, and so on, with Q1 pointing to the full speed queue, QFS, which points to the low speed queue (QLS).

The right half of the illustration lists the Frame List pointers, starting with FRAME_000 as the first Frame List Pointer in the Frame List, FRAME_001 as the second, and so on. The numbers on the right are the Queue numbers they point to. For example, FRAME_007 points to Queue number 8, which in turn points to Queue number 4, which points to Queue number 2, which points to Queue number 1. At FRAME_128, the sequence repeats, then again every 128th frame after.

Using that illustration, notice that the first frame points to the queue named Q1. Therefore, when the controller executes the first frame, Frame List Pointer zero, only anything queued in Q1, and below, will be executed. When the controller moves to the next frame, Frame List Pointer 1 points to the queue named Q2. Only anything queued in Q2 and Q1, since Q2's Head Pointer points to Q1, and below will be executed.

When the controller moves to the third frame, only Q1 and below will be executed. However, when the controller moves to the forth frame, anything in the Q4 Queue will now be executed along with Q2, Q1, and anything below it. Do you see the idea here? We place a queue or a transfer descriptor in one of the Queues above to only execute at a specific time interval. For example, a HID (Human Interface Device) such as a mouse may ask the driver to query a response every 16mS. Rather than having to write up a timer and lengthy code to poll the CPU clock, all we need to do is place a queue or transfer descriptor in the Q16 queue, and the queue or transfer descriptor will be executed every 16mS, no clock to worry about and no

polling of hardware. Once the mouse's transfer descriptor is executed, your driver can remove the old queue or transfer descriptor, insert a new queue or transfer descriptor in the same head queue, Q16, and it will be executed the next 16mS interval.

Figure 6-1 in the OHCI chapter has a similar technique and may be a useful image to help you visualize this technique here.

Notice that there is a broken lined box around the first 128 frame pointers in Figure 4-2. Since we don't need time intervals larger than 128mS, the schedule starts over again. This gives eight individual schedules of 128mS intervals.

This type of scheduling in called Interrupt Scheduling. However, notice that there are a few Queues at the bottom that are called QFS and QLS. The QFS is the Full Speed Queue while the QLS is the Low Speed Queue. These queues are used for control and bulk scheduling. I will detail the control type in the next few sections with the bulk type later in this chapter. The last five queues in the schedule (illustrated in Figure 4-2) are reserved for future use, mainly to occupy 16 queues of 16 bytes each, or 256 bytes.

Since the 16 queues are to remain in the schedule and never removed, we use 8 bytes after each Queue to allow insertion and removal of an unlimited amount of queues in each main queue in the schedule. Each of these sixteen queues are called Queue Heads and use a linked list to add or remove other queues. To insert a queue, simply find the last queue in the desired Queue Head and point its Queue Head Pointer to the address of your queue, and point the previous pointer member of your new queue to the Queue Head Pointer of the last queue found.

To remove a queue, simply use the previous pointer member in the desired queue to be removed to find the previous queue in this queue chain. Then place the head pointer value of your queue in the found previous queue's head pointer member. Now using that same head pointer value, you can find the address of the next queue in the chain and write the previous queues address in the previous pointer member.

See the listing below for an example of inserting a queue in a queue chain.

Listing 4-1: UHCI Queue Insertion

```
// This inserts a queue into the specified queue for the skeleton
// type = 0 = Q128, 1 = Q64, etc.
void insert(struct QUEUE_HEAD *queue, const bit8u type) {
struct QUEUE_HEAD *cur_queue = (struct QUEUE_HEAD *) SKELETON;
cur_queue = &cur_queue[type];
while ((cur_queue->horz_ptr & QUEUE_HEAD_T) == 0)
  cur_queue = (struct QUEUE_HEAD *)
                (cur_queue->horz_ptr & QUEUE_HEAD_PTR_MASK);
cur_queue->horz_ptr = (bit32u) queue | QUEUE_HEAD_Q;
queue->horz_ptr = QUEUE_HEAD_T;
queue->last_ptr = (bit32u) cur_queue;
}
```

And the listing below for an example of removing a queue from a queue chain.

Listing 4-2: UHCI Queue Removal

```
// removes a queue from the skeleton
void remove_frame(struct QUEUE_HEAD *queue) {
((struct QUEUE_HEAD *) queue->last_ptr)->horz_ptr =
                                queue->horz_ptr;
if ((queue->horz_ptr & QUEUE_HEAD_T) == 0)
```

```
 ((struct QUEUE_HEAD *)
  (queue->horz_ptr & ~QUEUE_HEAD_PTR_MASK))->last_ptr =
                                        queue->last_ptr;
}
```

To make sure that you do not interrupt the controller, you must make sure that the controller isn't currently executing any of the queues in this queue head chain. There are a few ways to do this. One way is to stop the controller, and then start it again, but that will mess up the timing of the interrupt schedule.

You could set the T flag in every other frame list pointer, or every odd pointer. Then check to see if the current frame being executed is one of these frames. This has its drawbacks though. You cannot place something in the Q1 queue and expect it to execute once a frame, because the UHCI is now half as slow as it was. It will only execute every other frame.

How the UHCI handles the Stack

When the UHCI registers are set up to the desired state and the driver sets the run bit in the COMMAND register, the controller retrieves the Frame List Pointer from the frame list. It does this by taking the 32-bit physical address from its Frame Address register and retrieving the dword value at the offset of the value in the Frame Num register. If it were to be done in the C language, it would look like this:

```
FrameListPointer = *((bit32u*) (frame_address + (frame_num * 4));
```

The controller first checks bit 0 to see if the pointer value is valid. If bit 0 is set, the controller waits until the 1mS time elapses. If bit 0 is clear, it takes the whole dword value, masks off the low nibble, and reads in a specific amount of memory. To find out how much, it checks bit 1. If bit 1 is set, it is a Queue and reads in 8 bytes. If bit 1 is clear, it is a Transfer Descriptor and the controller reads in 32 bytes. The controller then starts the frame.

> The specification doesn't indicate that it must check the Queue bit and read only 8 bytes if a queue is desired. My guess is that a well-built controller would read in 32 bytes no matter the pointer type. This would drastically simplify and speed up the process. The only catch would be if the queue were located in the last paragraph in physical memory.

The controller stacks each queue it finds. Hence the name, USB stack. When the controller finds a queue, it copies the Head Queue Pointer field to the current position on the stack, increments the stack, then does the same thing for the Element Link Pointer field. In other words, it pushes the two dwords onto its stack. It then starts by executing the current dword on the top of the stack. If it points to a queue, it repeats the operation. If it points to a transfer descriptor, it loads the 32 bytes into memory and executes the transfer descriptor. If the execution was successful, it copies the link pointer of the transfer descriptor, as a dword value, to the current stack. If bit 0 is set, or bit 0 is clear and the Vf bit is clear, it pops off the current stack item and executes the last pushed item. If bit 0 is clear and the Vf flag is set, it repeats the process with the next transfer descriptor.

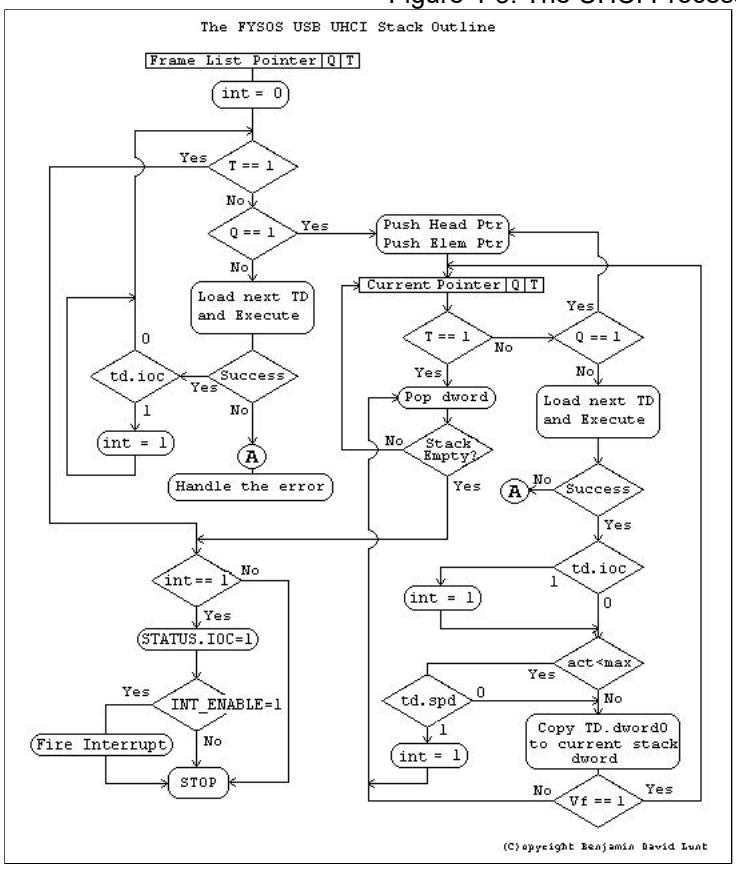
See the outline below for more information on what transpires every millisecond for a single frame. Please note that error checking and checking that enough time is left is not included.

Summary

To continue with the UHCI information in this book, skip to Part 2 to start enumerating the root hub.

Since enumeration is similar for all controllers, with only the data structures being different, if you feel that you have a good grasp on the UHCI hardware, and would like to learn about one or more of the other three controllers, I suggest that you turn to the next chapter for the OHCI hardware, or a further chapter on one of the other two controller types. If you are only after the UHCI part of this book, then please move on to Part 2.

Figure 4-3: The UHCI Process



Chapter 5 – The OHCI Hardware

The following chapter describes how to set up the OHCI Host controller, a minimal stack frame, and start the controller. You should have read Chapter 2 to know how to get the base I/O address and IRQ number from the PCI and be familiar with port I/O and memory mapped I/O addressing.

The OHCI specification states that the OHCI controller will use Memory Mapped I/O addressing, however this doesn't mean that a controller manufacturer will always conform to this statement. You will need to check bit 0 in the Base I/O address to be sure. Since I have never seen an OHCI controller use anything other than Memory Mapped I/O addressing, I will assume Memory Mapped I/O addressing from here on for the OHCI controller access.

A majority of the information in this section, including most of the tables, is readily available from the OHCI specification. I am including this information here only for reference, and to clear up a few things that may be a little confusing. To get the original tables, see Appendix A for information on how to get the specification file.

The OHCI USB controller has a defined set of I/O registers starting at the base address acquired in Chapter 2 of this book. These registers are defined in the following table and are called the Operational Registers.

Table 5-1: The OHCI controller's I/O registers

The OHCI controllers I/O registers			
Offset	Size	access[1]	Description
00h	4	RO	HcRevision
04h	4	R/W	HcControl
08h	4	Mixed	HcCommandStatus
0Ch	4	R/WC	HcInterruptStatus
10h	4	R/W	HcInterruptEnable
14h	4	R/W	HcInterruptDisable
18h	4	R/W	HcHCCA
1Ch	4	RO	HcPeriodCurrentED
20h	4	R/W	HcControlHeadED
24h	4	R/W	HcControlCurrentED
28h	4	R/W	HcBulkHeadED
2Ch	4	R/W	HcBulkCurrentED
30h	4	RO	HcDoneHead
34h	4	R/W	HcFmInterval
38h	4	RO	HcFmRemaining
3Ch	4	RO	HcFmNumber
40h	4	R/W	HcPeriodicStart
44h	4	RO	HcLSThreshold
48h	4	Mixed	HcRhDescriptorA
4Ch	4	R/W	HcRhDescriptorB
50h	4	Mixed	HcRhStatus
54h	4	R/W	HcRhPortStatus[0]
...			...
54h+4*NDP[2]	4	R/W	HcRhPortStatus[NDP-1]

[1] RO = Read Only, WO = Write Only, R/W = Read/Write, R/WC = Read/Write Clear
[2] **N**umber of **D**ownstream **P**orts

If this OHCI controller is a Companion controller to an EHCI controller, you may need to make sure the Configure Flag of the EHCI controller is clear or your OHCI code may not see its own controller. See Chapter 7 for more on this Configuration Flag and Companion controllers.

If you are unfamiliar with WC, or Write Clear, this means you write a one to that bit to clear it. Writing a zero to a bit that is a WC bit, has no effect on the bit.

In the following tables, I will give details on the fields that you will need to learn about, and little or no detail on the fields that we will not discuss and are most likely not needed for the purpose of this book. See the specification file ohci.pdf, from the USB-IF web page (See Appendix A), for more information on these fields.

A note on resetting the host controller, which is discussed a bit later in this chapter but is pertinent here. There are two different forms of resetting. The first is to reset the controller itself using bit 0 of the HcCommandStatus register, which resets the operational registers. It does not reset the root hub or send the reset signal to the downstream ports. The second form of resetting is to put the controller into the USBReset state via the operational mode in the HcControl register. This reset will reset the root hub and send the reset signal to all downstream ports. This reset will not, however, modify any of the controller's operational registers.

Table 5-2: The OHCI HcRevision Register

OHCI controller: HcRevision (Base + 00h)		
Bit	access	Description
31:9	RO	Reserved
8	RO	Legacy
7:0	RO	Version in BCD. Example: 10h = 1.0

The HcRevision register simply tells you what specification version this controller supports. I have not seen anything other than version 1.0 though don't assume it will always be this version. Bits 31:8 are reserved while bits 7:0 hold the value in BCD format.

Most of the controllers I have seen have bit 8 set, which returns a value of 0x00000110 in this 32-bit register. Bit 8 is used to indicate that the controller has legacy support. In other words, the controller will emulate the mouse and the keyboard via the hardware so that DOS and other OHCI/USB unaware systems can still use the keyboard and mouse.

Table 5-3: The OHCI HcControl Register

OHCI controller: HcControl (Base + 04h)		
Bit	Access	Description
1:0	R/W	Control Bulk Service Ratio
2	R/W	Periodic List Enable
3	R/W	Isochronous Enable
4	R/W	Control List Enable
5	R/W	Bulk List Enable
7:6	R/W	Host Controller Functional State
8	R/W	Interrupt Routing
9	R/W	Remote Wakeup Connected
10	R/W	Remote Wakeup Enable
31:11	R	Reserved

Bits 1:0 are the ratio between control and bulk Endpoint Descriptors and can hold a value of 0, 1, 2, or 3, with a default value of 0 after reset.

Bit 2 is used to enable the periodic, or interrupt list,
Bit 3 is used to enable the isochronous list,
Bit 4 is used to enable the control list, and
Bit 5 is used to enable the bulk list.

When any of these bits are set to a one from a zero, the respective list is executed on the next SOF (start of frame), no more than 1mS later.

Bits 7:6 are the current functional state of the controller. A value of 00b means the controller is in reset, a value of 01b is in resume state, 10b is in operational state (run in the UHCI sense), and 11b is in the suspend state.

Bit 8 is used for interrupt routing. Your driver should clear this bit to make sure that all interrupts are sent to the normal host interrupt handling hardware and subsequently, to your IRQ. If you set this bit, the interrupt will go to the System Management interrupt.

Bit 9 indicates whether the controller supports remote wakeup.
Bit 10 enables remote wakeup if bit 9 is set.

Table 5-4: The OHCI HcCommandStatus Register

OHCI controller: HcCommandStatus (Base + 08h)		
Bit	Access	Description
0	R/W	Host Controller Reset
1	R/W	Control List Filled
2	R/W	Bulk List Filled
3	R/W	Ownership Change Request
15:4	RO	Reserved
17:16	RO	Scheduling Overrun Count
31:18	RO	Reserved

A bit written to this register, as a 1 will change to a 1, while a bit written as a zero will leave that bit unchanged.

Bit 0 is use to reset the host controller. Writing a 1 to this bit initiates a software reset. The controller will clear this bit after it is done with the reset, no more than 10uS later.

Bit 1 is set by software to tell the controller that your driver has added a transfer descriptor to an endpoint descriptor in the control list.

Bit 2 is set by software to tell the controller that your driver has added a transfer descriptor to an endpoint descriptor in the bulk list.

Bit 3 is set by software to tell the controller your driver wants a change of ownership. This bit is used to fire a System Management interrupt, if the controller is SMM-based. See section 6.5.8 of the specification for more on this bit.

Bits 17:16 are incremented by the controller each time a scheduling overrun is encountered.

Table 5-5: The OHCI HcInterruptStatus Register

OHCI controller: HcInterruptStatus (Base + 0Ch)		

Bit	Access	Description
0	R/WC	Scheduling Overrun
1	R/WC	Writeback Done Head
2	R/WC	Start of Frame
3	R/WC	Resume Detected
4	R/WC	Unrecoverable Error
5	R/WC	Frame Number Overflow
6	R/WC	Root Hub Status Change
29:7	RO	Reserved
30	R/WC	Ownership Change
31	RO	Zero. Remains zero.

Bit 0 is set by the controller when the schedule of the current frame overruns after the update of HccaFrameNumber (see the HcHCCA register later) and will also cause the HcCommandStatus:SOC bit to be incremented.

Bit 1 is set by the controller when it has written HcDoneHead to HccaDoneHead. The controller will not modify the HccaDoneHead field again until this bit is cleared.

Bit 2 is set by the controller at each start of frame after the update of the HccaFrameNumber.

Bit 3 is set by the controller after a resume is detected.

Bit 4 is set by the controller when it detects an unrecoverable error.

Bit 5 is set by the controller when bit 15 of the HcFmNumber register changes from a 0 to a 1 or a 1 to a 0, after HccaFrameNumber has been updated.

Bit 6 is set by the controller when there is a change in the HcRhStatus register or any of the HcRhPortStatus registers.

Bit 30 is set by the controller when the HcCommandStatus:OCR bit is changed.

Remember that you clear any of these bits by writing a 1 to that bit.

Table 5-6: The OHCI HcInterruptEnable Register

OHCI controller: HcInterruptEnable (Base + 10h)		
Bit	Access	Description
0	R/W	Scheduling Overrun
1	R/W	Write back Done Head
2	R/W	Start of Frame
3	R/W	Resume Detected
4	R/W	Unrecoverable Error
5	R/W	Frame Number Overflow
6	R/W	Root Hub Status Change
29:7	RO	Reserved
30	R/W	Ownership Change
31	R/W	Master Interrupt Enable

Your driver sets any of these bits to 1 when you want the respective interrupt to be enabled, and the Master Interrupt Enable (MIE) bit is set. Clearing the MIE bit will disable all interrupts. Writing a 0 to any bit leaves that bit unchanged. To clear a bit, you write a 1 to the corresponding bit in the HcInterruptDisable register described next.

Table 5-7: The OHCI HcInterruptDisable Register

OHCI controller: HcInterruptDisable (Base + 14h)		
Bit	Access	Description
0	R/W	Scheduling Overrun
1	R/W	Writeback Done Head
2	R/W	Start of Frame
3	R/W	Resume Detected
4	R/W	Unrecoverable Error
5	R/W	Frame Number Overflow
6	R/W	Root Hub Status Change
29:7	RO	Reserved
30	R/W	Ownership Change
31	R/W	Master Interrupt Disable

Writing a 1 to any of these bits will disable the respective interrupt, and clearing the corresponding bit in the HcInterruptEnable register. Writing a 1 to bit 31, the Master Interrupt Disable (MID) bit, will disable all interrupts. Writing a 1 to bit 31 to disable all interrupts does not clear any other bits. It simply disables all interrupts. Writing a zero to any bit is ignored and the bit is left unchanged. Reading this register will return the HcInterruptEnable register's value.

Table 5-8: The OHCI HcHCCA Register

OHCI controller: HcHCCA (Base + 18h)		
Bit	Access	Description
7:0	RO	Reserved
31:8	R/W	Host Controller Communication Area

This register holds the 32-bit physical address of the Host Controller Communication Area, or HCCA.

This address must be aligned on a specific alignment. To find this alignment, write 0xFFFFFFFF to this register, then read it back. The alignment will be the amount of zeros in the low order bits. For example, if bits 8:0 are zero, then a minimum alignment of 512 must be used. An easy way to see this is to find the lowest order set bit and clear all higher order bits. For example, if after writing 0xFFFFFFFF to this register, then reading back a value of 0xFFFFF000, clear the high order bits leaving the lowest order set bit to get 0x00001000. To do this, simply take the 1's compliment (a bitwise NOT operation) of the returned value and add a 1. This new value is the minimum allowed alignment.

The Host Controller Communication Area, or HCCA, is a 256-byte block of memory where the controller and driver communicate. It holds the periodic list (interrupt list), the current frame number, and the DoneHead value.

Table 5-9: The OHCI Host Controller Communications Area

OHCI controller: Host Controller Communication Area Format		
Offset	Size	Description
0x00	128	HccaInterruptTable
0x80	2	HccaFrameNumber
0x82	2	HccaPad1
0x84	4	HccaDoneHead

0x88	116	Reserved

The 116 reserved bytes probably should actually be 120 to fill up the 256-byte block. I think what happened was that the writers of the OHCI specification had another 4-byte field for some reason, then later deleting it and forgot to update the reserved byte count. Since the HccaPad1 field actually has a '1' in its name, shows that this is more of a possibility. There would be little need to append a '1' to a field name unless there were multiple fields with the same name.

The 128 bytes at offset 0x00, are thirty-two 32-bit pointers, pointing to a list of endpoint descriptors and makes up the periodic (interrupt) list. You initialize these 32 pointers to point to 32 endpoint descriptors respectively.

The word at offset 0x80 is the current frame number and is updated just before the controller starts processing the periodic (interrupt) list. It also writes a zero to the HccaPad1 field.

The dword at offset 0x84 is the HccaDoneHead field. When the controller reaches the end of the frame and its deferred interrupt register is zero, it writes the current value of HcDoneHead to this dword location. It then generates an interrupt if the corresponding interrupt bit is enabled in the HcInterruptEnable register. This dword will not be modified by the controller until the HcInterruptStatus:WriteBackDoneHead bit is cleared by software. If bit 0 of the dword written was set, then there was an unmasked bit in the HcInterruptStatus register when the controller wrote to this field.

The 116 (or 120) bytes at offset 0x88 are reserved for use by the controller. I haven't seen whether these bytes really get modified or not, so don't assume they will be untouched by the controller. It is unnecessary, but it might be a good idea to clear these bytes once before you initialize the controller.

Table 5-10: The OHCI HcPeriodCurrentED Register

OHCI controller: HcPeriodCurrentED (Base + 1Ch)		
Bit	Access	Description
3:0	RO	Reserved
31:4	RO	Period Current ED

This register is a 32-bit physical address of the current periodic (interrupt) list endpoint descriptor being executed.

In Table 5-10 above, and the next few tables, bits 3:0 are always zero so that you must have a 16-byte alignment for all endpoint descriptors.

Table 5-11: The OHCI HcCurrentHeadED Register

OHCI controller: HcControlHeadED (Base + 20h)		
Bit	Access	Description
3:0	RO	Reserved
31:4	R/W	Control Head ED

This register is a 32-bit physical address of the head of the control list and points to the first endpoint descriptor of the list.

Table 5-12: The OHCI HcControlCurrentED Register

OHCI controller: HcControlCurrentED (Base + 24h)		
Bit	Access	Description
3:0	RO	Reserved

31:4	R/W	Control Current ED

This register is a 32-bit physical address of the current endpoint descriptor of the control list and points to the current endpoint descriptor being executed.

Table 5-13: The OHCI HcBulkHeadED Register

OHCI controller: HcBulkHeadED (Base + 28h)		
Bit	Access	Description
3:0	RO	Reserved
31:4	R/W	Bulk Head ED

This register is a 32-bit physical address of the head of the bulk list and points to the first endpoint descriptor of the list.

Table 5-14: The OHCI HcBulkCurrentED Register

OHCI controller: HcBulkCurrentED (Base + 2Ch)		
Bit	Access	Description
3:0	RO	Reserved
31:4	R/W	Bulk Current ED

This register is a 32-bit physical address of the current endpoint descriptor of the bulk list and points to the current endpoint descriptor being executed.

Table 5-15: The OHCI HcDoneHead Register

OHCI controller: HcDoneHead (Base + 30h)		
Bit	Access	Description
3:0	RO	Reserved
31:4	RO	Done Head

This register is a 32-bit physical address of the last completed transfer descriptor after it was added to the done queue. This register is read-only. Your driver should use the DoneHead member of the HCCA.

Table 5-16: The OHCI HcFmInterval Register

OHCI controller: HcFmInterval (Base + 34h)		
Bit	Access	Description
13:0	R/W	Frame Interval (defaults to 0x2EDF)
15:14	RO	Reserved
30:16	R/W	FS Largest Data Packet
31	R/W	Frame Interval Toggle

The Frame Interval specifies the interval between two consecutive SOF's in bit times. It defaults to 0x2EDF (11,999) but can be different due to BIOS settings. Your driver should read this value, saving it before resetting the controller. Resetting the controller will reset this field to 0x2EDF. Your driver has the option of restoring the saved value. If you need to change this value, you may only increment or decrement it by 1, not more than once every 6 frames, while the controller is in the running state.

Your driver should calculate the largest amount of data, in bits, any transaction could process before an overrun is given, and store it in the Full Speed Largest Data Packet field. This field is calculated with a value that leaves enough bit times for end of frame processing. The OHCI specification suggest that you take the frame interval value, subtract the maximum bit count for the end of frame processing (210), then multiply by 6/7^{ths}.

```
FSMaxPacketSize = ((FrameInterval - MAXIMUM_OVERHEAD) * 6) / 7;
```

Your driver must set the Frame Interval Toggle bit when a new frame interval value has been stored into the Frame Interval field.

Table 5-17: The OHCI HcFrameRemaining Register

OHCI controller: HcFrameRemaining (Base + 38h)		
Bit	Access	Description
13:0	RO	Frame Remaining
30:14	RO	Reserved
31	R/W	Frame Remaining Toggle

The Frame Remaining field is decremented by the controller at each frame. When this field reaches zero, it is reloaded with the value in HcFmInterval:FrameInterval above.

The Frame Remaining Toggle bit is loaded with the value of HcFmInterval:FrameIntervalToggle bit when the Frame Remaining field reaches zero. This is used by your driver for synchronization between this register and the HcFmInterval register.

Table 5-18: The OHCI HcFrameNumber Register

OHCI controller: HcFrameNumber (Base + 3Ch)		
Bit	Access	Description
15:0	RO	Frame Number
31:16	RO	Reserved

The Frame Number field is incremented by the controller each time the HcFmRemaining register is re-loaded. This field rolls over to 0x0000 after 0xFFFF is incremented. This field will be written to the HCCA:FrameNumber field after incremented and will set the StartofFrame bit in the HcInterruptStatus register.

Table 5-19: The OHCI HcPeriodicStart Register

OHCI controller: HcPeriodicStart (Base + 40h)		
Bit	Access	Description
15:0	R/W	Periodic Start
31:16	RO	Reserved

This register is used to tell the controller when to give priority to the interrupt list over the control or bulk lists. When the HcFmRemaining register counts down to the value in the Periodic Start field, the controller gives priority to the periodic (interrupt) list over the control and bulk lists. Therefore, your driver should set this accordingly to the amount of interrupt driven devices you currently have attached.

The OHCI specification says that this value is usually 10% “off from” HcFmInterval, whatever that means, but gives a typical value of 0x3D67. If the default value for the frame interval is 0x2EDF, then the 0x3D67 will never be reached. I think this may just be a typo and the typical value should have been 0x2D67. An older well know operating system uses the value 0x2A2F which is right at 10% less than the default frame interval of 0x2EDF.

Table 5-20: The OHCI HcLSThreshold Register

OHCI controller: HcLSThreshold (Base + 44h)		

Bit	Access	Description
11:0	R/W	Low Speed Threshold (Defaults to 0x0628)
31:12	RO	Reserved

This register is set to a value where when a low speed transaction is about to take place, if the HcFrameRemaining register is less than this register, the transaction is not executed. This register has a default value of 0x0628.

The OHCI Controller's Root Hub

The remaining registers are for controlling the controller's root hub. Every controller has a root hub with one or more ports, not to exceed fifteen. These registers give information about the root hub, how many ports are attached, along with control of each port. There are two descriptor registers, a status register, and a register for each port.

Table 5-21: The OHCI HcRhDescriptorA Register

OHCI controller: HcRhDescriptorA (Base + 48h)		
Bit	Access	Description
7:0	RO	**Number of Downstream Ports** supported by the root hub. At least 1 and no more than 15.
8	R/W	**Power Switching Mode** 0 = all ports are powered at the same time. 1 = each port is powered individually.
9	R/W	**No Power Switching** 0 = ports are power switched. 1 = ports are always powered when the controller is powered on.
10	RO	**Device Type**
11	R/W	**Over Current Protection Mode** 0 = over-current status is reported collectively for all downstream ports. 1 = over-current status is reported on a per-port basis.
12	R/W	**No Over Current Protection** 0 = over-current status is reported collectively for all downstream ports. 1 = over-current is not supported.
23:13	RO	Reserved
31:24	R/W	**Power On To Power Good Time**

This register is used to find out how many downstream ports are available, and for setting up power switching.

To disable power switching and have all ports powered all the time, write a 1 to bit 9. To be able to power each port separately, write a zero to bit 9, and a 1 to bit 8. Now each port can set its power by setting bit 8 in its respective register or by the Port Power Control Mask in the HcRhDescriptorB register.

The Device Type field is always zero since the root cannot be part of a compound device.

The over-current protection function is similar to power switching. You can have each individual port report its own over-current status, or have over-current status reported collectively over all downstream ports. Clear bit 12 and set bit 11 to have each port report its own over-current status. Set bit 12 otherwise.

The Power On To Power Good Time field specifies the amount of milliseconds to wait after powering on a port and before you access the port. The value is a count of 2mS. If this field's value is 16, you should wait 32mS after setting port power and before accessing the port.

The OHCI specification says that this field is Read/Write. However, I can't image why it would allow write access. Just by writing a smaller value to this field, shouldn't mean that the port will be ready sooner. However, after trying it, a write is

allowed. You should read this value and store it as one of the first things your driver does incase this value actually does get over-written.

The OHCI controller (NEC uPD9210) that is included with the uPD720100A EHCI controller sets this register to 0xFF. I don't know if this was a "who cares" value, or if they actually wanted 510mS for this value. However, just a note, be sure to mask off the high part of the shift when retrieving this value or you could be waiting for a very long time, nearly 50 days in fact.

```
int potpgt;       /* define potpgt as an integer */
potpgt = (HcRhDescriptorA >> 24);            /* returns 0xFFFFFFFF  */
    potpgt = (HcRhDescriptorA >> 24) & 0xFF;  /* returns 0x000000FF    */
```

Table 5-22: The OHCI HcRhDescriptorB Register

OHCI controller: HcRhDescriptorB (Base + 4Ch)		
Bit	Access	Description
15:0	R/W	DR: Device Removable
31:16	R/W	PPCM: Port Power Control Mask

This register is used to indicate if an attached device is removable and has a port power mask when power-switching mode is on.

The Device Removable field is a 16-bit field indicating whether a port is removable or not. Bit 0 is reserved, while bit 1 indicates port 1 is or is not removable. Bit 2 indicates port 2, and so on. If a bit is 1, the device is not removable.

The Port Power Control Mask field is a 16-bit field indicating whether a port uses global power mode or individually controlled with that port's bit 8 value. Bit 0 is reserved, while bit 1 represents port 1, and so on. If a bit is set, that port's power is controlled by bit 8 in its port status register. If a bit is cleared, that port is controlled by the global power mode. This 16-bit field is invalid if power switching mode is off.

Table 5-23: The OHCI HcRhStatus Register

OHCI controller: HcRhStatus (Base + 50h)		
Bit	Access	Description
0	R	**Local Power Status** (always read as 0)
	W	**Clear Global Power** If power switching is off (is a 0), writing a 1 turns power off to all ports. If power switching is on, turns off power to all ports who's bit is clear in the Port Power Control Mask in HcRhDescriptorB register. A write of zero to this bit has no effect.
1	RO	**OCI: Over Current Indicator** This bit is set when there is an over-current condition and over-current global reporting is on.
14:2	RO	Reserved
15	R	**Device Remote Wakeup Enable** 0 = Connect Status Change is not a remote wakeup event. 1 = Connect Status Change is a remote wakeup event.

	W	**Set Remote Wakeup Enable** 0 = has no effect. 1 = sets device remote wakeup enable.
16	R	**Local Power Status Change** (always 0)
	W	**Set Global Power** If power switching is off (is a 0), writing a 1 turns power on to all ports. If power switching is on, turns on power to all ports who's bit is clear in the Port Power Control Mask in HcRhDescriptorB register. A write of zero to this bit has no effect.
17	R/WC	**Over Current Indicator Change** The controller will set this bit when a change has occurred to the Over Current Indicator field above. This field is Write Clear, meaning writing a 1 to this bit clears it, writing a 0 has no effect.
30:18	RO	Reserved
31	W	**Clear Remote Wakeup Enable** Writing a 1 clears Device Remove Wakeup Enable above. Writing a zero has no effect.

The OHCI Controller's Port Register

Table 5-24: The OHCI HcRhPortStatus Register

OHCI controller: HcRhPortStatus[n] (Base + 54h + (n * 4))		
Bit	Access	Description
0	R	**Current Connect Status** This bit indicates the current connect status of this port. 0 = no device attached. 1 = device attached. Note: This bit will always read as 1 if the corresponding 'removable' bit is set in the HcRhDescriptorB register.
	W	**Clear Port Enable** Writing a 1 to this bit clears the Port Enable Status. Writing a zero to this bit has no effect.
1	R	**Port Enable Status** This bit indicates whether this port is enabled or disabled. 0 = disabled. 1 = enabled.
	W	**Set Port Enable** Write a 1 to this bit to enable the port. If **Current Connect Status** is cleared, writing a 1 to this bit will set Connect Status Change below. This indicates that your driver tried to enable a disconnected port. Writing a zero has no effect.
2	R	**Port Suspend Status** This bit indicates whether the port is in the resume or in the suspend state. 0 = port is not suspended. 1 = port is suspended.
	W	**Set Port Suspend** Write a 1 to this bit to set **Port Suspend Status**. If **Current Connect Status** is cleared, writing a 1 to this bit will set **Connect Status Change**. This indicates that your driver tried to enable a disconnected port. Writing a zero has no effect.
3	R	**Port Over Current Indicator** 0 = no over-current condition for this port. 1 = an over-current condition was detected for this port. This bit is only valid if over-current protection mode is set on a per-port basis.
	W	**Clear Suspend Status** 0 = has no effect. 1 = initiates a resume if Port Suspend Status above is set.
4	R	**Port Reset Status** This bit is set when the port is currently in the reset state. It is cleared when the Port Reset Status Change bit below is set.

		0 = port is not in reset 1 = port is in reset
	W	**Set Port Reset** Setting this bit resets the port. If **Current Connect Status** is cleared, writing a 1 to this bit will set **Connect Status Change**. This indicates that your driver tried to enable a disconnected port. Writing a zero has no effect.
7:5	RO	Reserved
8	R	**Port Power Status** This bit indicates the current power status for this port, regardless of the current power switching mode. This bit is read as 1 if power switching is off.
	W	**Set Port Power** Write a 1 to this bit to set the port's port, if port power switching mode is in per port mode.
9	R	**Low Speed Device Attached** 0 = a full speed device is attached 1 = a low speed device is attached This field is only valid if **Current Connect Status** is set.
	W	**Clear Port Power** Writing a 1 to this bit clears the ports power status, turning off power to this port, if port power switching mode is in per port mode. Writing a zero to this bit has no effect.
15:10	RO	Reserved
16	R/WC	**Connect Status Change** This bit is set whenever a connect or disconnect occurs. Writing a 1 to this bit clears this bit while writing a zero has no effect. If **Current Connect Status** is zero when writing a 1 to **Set Port Reset**, **Set Port Enable**, or **Set Port Suspend**, the controller sets this bit to indicate to the driver to re-evaluate the connect status state of the port.
17	R/WC	**Port Enable Status Change** This bit is set whenever a hardware event causes the **Port Enable Status** bit to clear. Writing a 1 to this bit clears this bit while writing a zero has no effect. 0 = no changed in port enable status 1 = there was a change in the port enable status
18	R/WC	**Port Suspend Status Change** The controller sets this bit when the full resume sequence has finished. Writing a 1 to this bit clears this bit while writing a zero has no effect. 0 = resume has not completed 1 = resume has completed
19	R/WC	**Port Over Current Indicator Change**

		This bit is only valid if over-current conditions are reported on a per-port basis. The controller sets this bit when it changes the **Port Over Current Indicator** bit. Writing a 1 clears this bit while writing a zero has no effect. 0 = no change in **Port Over Current Indicator** 1 = **Port Over Current Indicator** bit has changed
20	R/WC	**Port Reset Status Change** The controller sets this bit when the 10mS reset is complete. Writing a 1 to this bit clears this bit while writing a zero has no effect. 0 = port reset is not complete 1 = port reset is complete
31:21	RO	Reserved

Please see Appendix D, “Port Connection Status Change” for an important note about connect status changes.

OHCI Host Controller Detection

Since the PCI bus already told us that there is an OHCI controller at this base, why should we take the time and effort to detect the controller? To be completely independent of the host bus, the OHCI driver should detect the controller rather than relying on its host bus. Future hardware may have the USB on a different bus type that isn't so accommodating.

To detect the controller, we will attempt to reset it and then see what happens to the register values.

First, before we reset the controller, we can read the HcRevision register. If it does not return 0x10 in the lower 8 bits, there is no reason to continue.

If the OHCI controller is on the PCI bus and is set to memory mapped access, and most are, you must read and write each register as a 32-bit dword. Byte and word reads and writes may return undefined results. Case in point, when I was writing my driver, the compiler I used, compiled the following C/C++ code "incorrectly", adding an unwanted optimization. The value of 0xF0000000 is the I/O base address for this example.

```
volatile struct S_FOO *foo = (struct S_FOO *)0xF0000000;
if ((foo->bar & 0x000000FF) != 0x10) return 0;
```

Compiled into the following assembly code:

```
xxxxxx00 B8000000F0      mov          eax,0F0000000h
xxxxxx05 8A10          mov          dl,[eax]
xxxxxx07 80FA10         cmp          dl,byte 10h
```

Notice that the compiler only read a byte from 0xF0000000 instead of reading the whole dword as expected. The compiler optimized the code and read a byte instead of reading a dword and masking the high order bits. Make sure that the compiler you use does not optimize the read to a byte read, by either turning off optimizations, or call an external function to read the dword, or a number of other techniques to make sure this optimization doesn't happen.

After we have read the HcRevision register and verified it is 0x00000010, masking off the high three bytes, we now can reset the controller and check a few of the registers for known values.

We use the HcCommandStatus register to reset the OHCI controller by setting bit 0, HostControllerReset. This bit will initiate a software reset of the controller, which after no more than 10uS, the controller will finish the reset and clear bit 0. Unlike the UHCI reset, the OHCI reset will not send a reset signal to any of the downstream ports. However, the operational state of the controller will be in the suspend state, so no transactions will take place. You can place the operational state of the controller to the USBReset state, which will send the reset signal to all downstream ports, setting it back to the USBSuspend state after 50mS.

After we set bit 0, to initiate the reset, we wait a maximum of 30uS checking to see if the controller has cleared the bit, indicating that it is done with the reset. If after the 30uS, bit 0 is still set, we return FALSE, indicating that there is not a valid controller on this address.

If we make it this far, we can assume that the controller has been reset. Before we return TRUE, we check two registers for default values. First, the functional state bits in the HcControl register. If bits 7:6 are not 11b, we return FALSE. Then if the HcFmInterval:FmInterval register is 0x2EDF we return TRUE.

The code below doesn't include nor did I mention it here, but if you remember in the section before, you will need to save and possibly restore the HcFmInterval value when resetting the controller.

The listing below is a modified snippet of code, which comes from the OHCI driver of FYSOS. The driver calls this code expecting a TRUE or FALSE condition on return, returning TRUE if an OHCI controller was detected on this base address. The udelay() routine simply delays for a given amount of microseconds (uS).

Listing 5-1: OHCI Host Controller Reset

```
bool det_ohci_root(const bit32u base) {
  // first see if the revision register contains 1.0
  //  (the version of the specs we support)
  if ((usb_read_op_regs(base, HCRevision) & 0x000000FF) != 0x10)
    return FALSE;
  // do the reset
  usb_write_op_regs(base, HCCommandStatus, (1<<0));
  int time = 30;
  while (usb_read_op_regs(base, HCCommandStatus) & (1<<0)) {
    if (!--time)
      return FALSE;
    udelay(1);
  }

  // after doing the wait, the HC should be in SUSPEND state
  if ((usb_read_op_regs(base, HCControl) & HC_CONTROL_HCFS) !=
                                      HC_CONTROL_HCFS)
    return FALSE;

  // The Frame Interval should be set at 2EDFh (11,999d)
  if ((usb_read_op_regs(base, HCFmInterval) & 0x00003FFF) !=
                                            0x2EDF)
    return FALSE;

  // if we get here, we found a valid HC
  return TRUE;
}
```

Set up the OHCI controller

Now that we know we have an OHCI host controller, we can start to set up the controller and our stack.

There are a few things we need to do before we can start sending transactions on the bus. We will need to allocate and set up the HCCA memory block, the control, bulk, and periodic/interrupt lists, as well as the remaining registers of the OHCI.

First, we will allocate and initialize the Host Controller Communication Area, or HCCA. Remember that the HCCA address must be aligned and to get that alignment, we use the HcHCCA register. The technique to get the alignment was explained earlier in this chapter where the HcHCCA register is defined. The following pseudo-code shows how to allocate the correctly aligned physical address for the HCCA. Remember that your allocation scheme must return a physical address.

```
 HcHCCA = 0xFFFFFFFF // write 0xFFFFFFFF to the HcHCCA register
 Return = HcHCCA     // read it back and store in 'Return'
   Size = ~Return + 1 + HCCA_SIZE
Address = ALLOCATE(Size)
Aligned = (Address + ~Return) & Return
```

Where Address will be the address to save for your memory free scheme and Aligned is the correctly aligned memory block to store your Host Controller Communication Area data block. However, there is a catch. As long as the alignment value read from HcHCCA, the value of Return, is equal to or "less than" 0xFFFF0000, which is an alignment of 0x00010000 or less, you should be okay with the above technique. An alignment of 0x00010000 will allocate a minimum of 64k + 256 bytes of memory. This 64k+ of memory is a low cost to get 256 bytes allocated on the correct alignment. Each bit of alignment after this doubles the amount of memory you must allocate to fulfill the alignment request. For example, a return value of 0xFFF80000 returns a memory allocation of 512k, a half a megabyte for which you will only use 256 bytes of that chunk. However, I have not seen a controller that returns anything other than 0xFFFFFF00, which is a 256-byte alignment. This doesn't mean that there won't be a controller that does, I just haven't seen one yet.

For now, clear all the bytes in your HCCA data structure. In the next chapter, I will show how to set up this structure.

Now that we have our HCCA data block set, we can now set up the controller's operational registers.

Setting the OHCI Controllers Operational Registers

Since the HcRevision register is read only, and we have already verified that it is a known value, we will skip this register.

Writing 0x00000000 to the HcControl register will send the reset signal to all downstream ports. Then we need to wait a minimum of 50mS as specified in the USB specifications, writing 0x000000C0 to it to stop the reset. This write will also disable the execution of each list, set the Control/Bulk ratio to 1:1, make sure the controller is not in the operational mode, disable remote wakeup, and make sure all interrupts are routed to the normal host bus interrupt mechanism.

Writing 0xA7782EDF to the HcFmInterval register sets the frame interval to the default 0x2EDF, sets the full speed max packet size field to 0x2778 which is (((0x2EDF-210)*6)/7), and sets bit 31 telling the controller we have written a new value to this register.

Writing 0x00002A2F to the HcPeriodicStart register tells the controller to start giving priority to the interrupt list after 1,200 frames, or 10% of the frame interval, (0x2EDF * 1/10).

Reading from the HcRhDescriptorA register will tell us how many downstream ports there are. Save for later. Then clearing bit 9 and setting bit 8 will tell the controller that we will manage the power for each port.

Set each corresponding bit in the HcRhDescriptorB PortPowerControlMask register to the count of downstream ports we have, starting with bit 17 for the first port, 18 for the second, and so on.

Write the predetermined correctly aligned Host Controller Communication Area data block's physical address to the HcHCCA register.

Write the physical address of the first Endpoint Descriptor of the control list to the HcControlHeadED register, write the physical address of the first endpoint descriptor of the bulk list to the HcBulkHeadED register, remembering that both require paragraph alignment.

> We haven't setup or discussed Endpoint Descriptors or Transfer Descriptors yet, though this write would be the physical address of the first Endpoint Descriptor, discussed in the next chapter. Same with the register below.

Write 0x00000000 to both the HcControlCurrentED and HcBulkCurrentED registers. This will tell the controller to load into the HcControlHeadED and HcBulkHeadED registers respectively, the next time either list is enabled, making sure to start with the first endpoint descriptor in the list.

Write 0x000006B0 to the HcControl register. This will enable the bulk and control lists, set the controller in the operational mode, and enable remote wakeup.

Write 0x00008000 to the HcRhStatus register. This will enable the remote wakeup.

Write 0x00000004 to the HcInterruptStatus register. This will clear the start of frame bit.

Write 0xC000005B to the HcInterruptEnable register. This will allow a scheduling overrun, a write to the DoneHead register, a resume detect, an unrecoverable error detect, a root hub status change, and an ownership change to all fire an interrupt. Setting bit 31 allows these interrupts to fire.

Don't forget to set the IRQ handler and unmask the IRQ to allow hardware interrupts for this controller.

At this point, all but each port register is set up properly. We have power switching on so that we can turn on or off each port through each port's register, interrupts are ready to fire, and the controller is in operational mode.

As for the OHCI hardware, we have completely set it up. The next thing to do is to start sending and receiving packets from the schedule. For now, let's move on to the Port Status/Control register and device detection. We will come back to sending and receiving packets in the next chapter.

The OHCI Port Status/Control Register

Now we need to set power on, reset and enable each port register, and detect if a device is attached.

Please note that a non-powered port will not show a Current Connect Status.

Just after boot up, and after we set up the controller as explained before, an empty port may contain the value 0x00000000. The Current Connect Status is zero, the port enable status is zero, port is not in reset, and the device speed bit is invalid.

Unlike the UHCI hardware, the OHCI hardware tells us how many downstream ports we have. Therefore, we don't have to go through the process of detecting how many ports.

Powering the port

There are four different ways to power the port, and are listed below.

1. Set the NoPowerSwitching bit in the HcRhDescriptorA register. This will give all ports power as long as the controller is powered on. With this technique, the power will always be on, and the SetPortPower bit in the port register is ignored.
2. Clear the NoPowerSwitching and PowerSwitchingMode bits in the HcRhDescriptorA register. Then set the SetGlobalPower bit in the HcRhStatus register. This will turn on all power to all ports. This technique allows you to turn off all ports by setting the ClearGlobalPower bit.
3. Clear the NoPowerSwitching bit and set the PowerSwitchingMode bit. Then when you set the SetGlobalPower bit, it will power all ports that have their bit cleared in the PortPowerControlMask portion of the HcRhDescriptorB register.
4. Clear the NoPowerSwitching bit and set the PowerSwitchingMode bit. Then set all bits in the PortPowerControlMask portion of the HcRhDescriptorB register that you want to have individual power control. Then to set and clear the power for each port, using the SetPortPower and ClearPortPower bits in the port register.

Table 5-25: OHCI Port Power Options

Port Power Options					
No Power Switching Bit	Power Switching Mode	Set Global Power	Port Power Control Mask[n]	Port[n] Set Port Power	
1	undefined	All ports are powered			
0	0	1	All ports are powered		
0	0	0	All ports are not powered		
0	1	1	0	Port is powered	
0	1	undefined	1	0	Not powered
0	1	undefined	1	1	Powered

Resetting the port

The next step is to reset the port. Resetting a port on an external hub is different than resetting a root hub port. See the chapter on external hubs for resetting their ports.

> Remember that a non-powered port will not reset.

If the CurrentConnectStatus bit is set, we can attempt to reset the port. To reset a root hub port, you set the SetPortReset bit in the port's register. When the controller is done with the reset, it will set the PortResetStatusChange bit indicating it is finished. Writing a 1 to this bit will clear the bit. If you try to reset a port that has the CurrentConnectStatus bit clear, instead of resetting the port, it will set the ConnectStatusChange bit instead. So before you wait indefinitely for the PortResetStatusChange bit to be set, check that the ConnectStatusChange bit isn't set.

Listing 5-2: OHCI Port Reset

```
bit32u val = 0;
if (!ohci_set_port_power(root, port, TRUE))
  return NONSUCCESSFUL;

// only try to reset it if the CCS bit is set after powered
if (usb_read_op_regs(root->base, HCRhPortStatusOff) &
                                  HC_PORT_STATUS_CCS) {
   usb_write_op_regs(root->base, HCRhPortStatusOff,
                                   HC_PORT_STATUS_PRS);
   for (int i=0; i<10; i++) {
     mdelay(USB_TRSTRCY);  // hold for USB_TRSTRCY ms
     // The reset done bit will be set when the reset is done
     val = usb_read_op_regs(root->base, HCRhPortStatusOff);
     // The ConnectStatusChange bit will be set and
     // CurrentConnectStatus Cleared if no device attached.
     if ((val & (HC_PORT_STATUS_CSC | HC_PORT_STATUS_CCS |
              HC_PORT_STATUS_PRSC)) == HC_PORT_STATUS_CSC)
       // no device attached, clear the bit
       return SUCCESS_RESET;
     // return if successful reset and enable
     if ((val & (HC_PORT_STATUS_CCS | HC_PORT_STATUS_PES |
                               HC_PORT_STATUS_PRSC)) ==
        (HC_PORT_STATUS_CCS | HC_PORT_STATUS_PES |
                                 HC_PORT_STATUS_PRSC)) {
       return SUCCESS_RESET;
     }
   }
 }
 return NONSUCCESSFUL;
```

Now that we have a device attached and reset, we can read the LowSpeedDeviceAttached bit to see if we have a low- or full-speed device and set our driver accordingly.

The next step is to enable the port.

> If you see that the CurrentConnectStatus bit is set, then try to reset the port, and the PortResetStatusChange bit never becomes set, check to make sure the Device Removable bit isn't set in the HcRhDescriptorB register. If the removable bit is mistakenly set, the CurrentConnectStatus bit will also be set for a port that may be

disconnected. Therefore, if you set the SetPortReset bit, it will remain set indefinitely. A read from the port will return 0x00000111.

Enabling the port

Enabling a port on an external hub is different than enabling a root hub port. Please see the chapter on external hubs for enabling their ports.

The USB specifications 2.0, section 7.1.7.5, note “After the reset, the hub port will transition to the Enabled state.” The OHCI specification doesn’t specifically state that the enable bit will be set after a successful reset, and it does not have to. This is because the OHCI was created before USB 2.0 was released.

However, every OHCI controller that I have seen, sets the enable bit after a successful reset, which is USB 2.0 compliant, even though the OHCI was built on the USB 1.1 specification.

There will be times that you will need to enable the port. For instance, the controller will clear the enable bit if there is an over-current condition. You will need to remedy the condition and re-enable the port.

To enable the port, you write a 1 to the SetPortEnable bit, bit 1 in the port register. As trying to reset the port, if you try to enable a port that has the CurrentConnectStatus bit clear, instead of enabling the port, it will set the ConnectStatusChange bit instead.

Summary

Now that we have successfully set up the controller and each of the root hub's ports, it is time to enumerate the devices attached. First you will need to learn how to set up the OHCI stack schedule and each individual list, along with how the controller follows and executes these lists. Please see the next chapter for this technique.

Chapter 6 – The OHCI Stack

By this chapter, you should know how to detect the host controller through the PCI configuration space and physical detection at the port I/O level. You should also know how to detect that a device is present on one of the ports on the host controller, also known as the root hub.

If you are comfortable with this up to this point, we will start on the actual OHCI communication protocol and how to detect the device or devices already attached. If you are not ready to continue on, please go back and study the previous chapters.

The OHCI Stack

The OHCI stack has three lists, the periodical list, also called the interrupt list, the control list, and the bulk list. The periodical list is in the Host Controller Communication Area structure, also known as the HCCA structure. All interrupt packets use this list. This list is also where the isochronous endpoint descriptors reside.

All interrupt endpoint descriptors should be before any isochronous endpoint descriptors since the controller will stop executing this list as soon as it finds an isochronous endpoint descriptor if the IsochronousEnable bit is clear in the HcControl register.

The control and bulk lists are pointed to by the HcControlHeadED and HcBulkHeadED registers. These two lists must contain at least one endpoint descriptor each.

The host controller's scheduler starts with the control and bulk lists. The controller needs to find the next endpoint descriptor to process, so it reads the value from the HcControlCurrentED register. If it is zero, it reads the value from the HcControlHeadED register. Then the controller looks at the ControlListEnable bit, if it is set, it then looks at the ControlListFilled bit in the HcCommandStatus register. If it is not set, it moves to the bulk list and does the same checks on that list.

The controller uses the ControlBulkServiceRatio value in the HcControl register and an internal count to see which list it should retrieve the next endpoint descriptor. If the internal count for non-empty processed control endpoint descriptors is more than the ratio for the non-empty count of processed bulk endpoint descriptors, the controller will switch to the bulk list. The controller will monitor the internal count for both control and bulk non-empty endpoint descriptors and switch back and forth as appropriate. These internal counters are preserved across frame boundaries.

Once the control and bulk lists are empty, or if the HcFmRemaining register value reaches the HcPeriodicStart register value, the controller will move to the periodic list.

The controller will process any endpoint descriptors in the periodical list until it finds the end of the list, finds an isochronous endpoint descriptor with isochronous endpoint descriptors disabled in the HcControl register, or calculates that no more packets can be processed in the time remaining within the frame.

When the controller gets to this point, this is considered one frame. This means that each frame will be executed approximately once a second, every 1.024 seconds to be exact, or at 1 Hz.

When it moves to the next frame, it starts over with the same process by processing any endpoint descriptors that were not processed in the previous frame due to not enough time in that frame. The controller uses the HcControlCurrentED, HcBulkCurrentED, and HcPeriodicCurrentED registers to find the next endpoint descriptor to process, in the order explained before.

Now that we know where the lists are located, and how each list is processed, we need to create valid lists to occupy the memory allocated.

Endpoint Descriptors

First I'll explain the format of the OHCI Endpoint Descriptor. This endpoint descriptor is 16 bytes in size, or 4 dwords, and has the format listed in the table below.

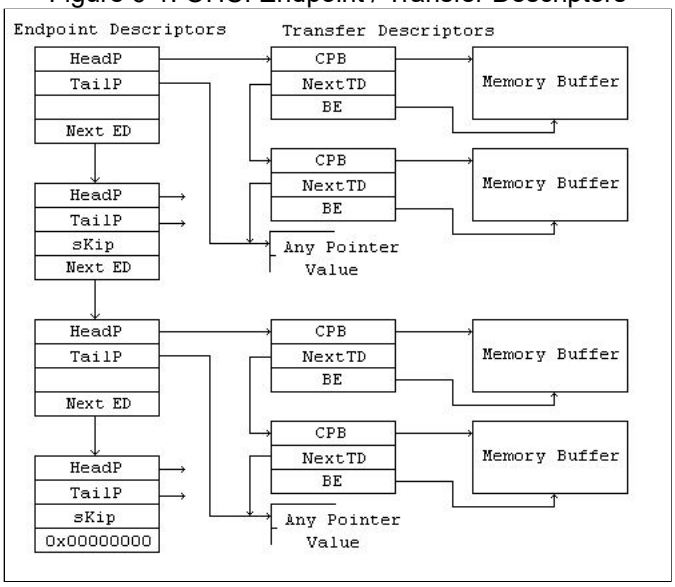
Table 6-1: OHCI Endpoint Descriptor Format

31:27	26:16	15	14	13	1 1 2 1	10:7	6 5 4	3	2	1	0
--	MPS	F	K	S	D	EN	FA				
TD Queue Tail Pointer (TailP)								--			
TD Queue Head Pointer (HeadP)								0		C	H
Next Endpoint Descriptor (NextED)								--			

An endpoint descriptor can point to a linked list of transfer descriptors and is in a linked list itself. If the TailP field's value and the HeadP field's value are identical, this endpoint descriptor does not point to a list of transfer descriptors. It is an empty endpoint descriptor. If the NextED field is non-zero, it points to the next endpoint descriptor in the list. The values in these three fields are to be paragraph aligned with bits 3:0 of each dword assumed zero. Therefore, each entry contains bits 31:4 of the physical address of the item it points to. You simply read the whole dword and mask off the least significant four bits. You cannot assume that the last four bits will be zero in the 2nd and 4th dwords, and it uses the lower 4 bits in the 3rd dword as flags. See Table 6-2 on page 6-4 for the descriptions of each field of an Endpoint Descriptor.

See Figure 6-1 below for a visual look at the Endpoint and Transfer Descriptor linked list.

Figure 6-1: OHCI Endpoint / Transfer Descriptors



In the figure above, there are four Endpoint Descriptors, each pointing to a list of Transfer Descriptors. The first and third Endpoint Descriptors point to a list of two Transfer Descriptors each, while the second and fourth are skipped. Each of the two sets of Transfer Descriptors have two each, moving half of the data per Transfer Descriptor. So that the controller knows when to stop executing transfer descriptors, the TailP pointer in the Endpoint Descriptor and the NextTD pointer in the last Transfer Descriptor, point to the same place in memory. When the controller updates the HeadP value with the value from last Transfer Descriptor's NextTD field, the newly updated HeadP value and the TailP value will now be equal, indicating that there are no more transfer descriptors to process. Note that this value could be any value as long as it is the same value. It is simply a technique for the controller to know when it needs to stop

executing TD's in the current Endpoint Descriptor and move to the next Endpoint Descriptor. Once the controller has reached the last Endpoint Descriptor, it knows to stop executing ED's, since the NextED field is now 0x00000000.

Table 6-2: OHCI Endpoint Descriptor Definition

Endpoint Descriptor Definition		
Name	HC Access	Description
FA	RO	**Function Address**. This is the device's address given at enumeration time. Seven bits containing a value of 0 to 127.
EN	RO	**Endpoint Number**. Four bits containing 0 to 15.
D	RO	**Direction**. This two-bit field specifies which direction the data flow will be. If a value of 00b or 11b is used, the direction is determined from the current Transfer Descriptor, allowing multiple directions within this endpoint descriptor. 00b = get direction from TD 01b = OUT direction 10b = IN direction 11b = get direction from TD Note: If you have a SETUP packet in one of the transfer descriptors, you must use the transfer descriptor's direction field and set this field to 00b or 11b.
S	RO	**Speed**. Indicates the speed of the endpoint. 0 = Full speed 1 = Low speed
K	RO	**sKip**. When this bit is set, the controller skips this endpoint descriptor and any transfer descriptors linked to it, and moves to the next endpoint descriptor.
F	RO	**Format**. This bit indicates whether this is a Control/Bulk/Interrupt (F = 0) or an Isochronous endpoint descriptor (F = 1).
MPS	RO	**Max Packet Size**. This indicates the maximum data transfer per transfer descriptor within this endpoint descriptor. Example: Set to 8 for all low-speed devices.
TailP	RO	**Tail Pointer**. This points to the last Transfer Descriptor in the list. If the HeadP and TailP fields point to the same place, then there are no transfer descriptors left in the list.
H	R/W	**Halted**. The controller will set this bit to indicate that there was an error in processing one of the transfer descriptors within this endpoint descriptor.
C	R/W	**Toggle Carry**. This bit holds the toggle bit of the last completed transfer descriptor's toggle bit (lower bit of the data toggle field). This bit is not used for Isochronous transfers.

HeadP	R/W	**Head Pointer**. This field points to the next transfer descriptor to be executed. When the controller retires that transfer descriptor, it copies the next transfer descriptor's address to this field.
NextED	RO	**Next Endpoint Descriptor**. If this field is non-zero, it points to the next endpoint descriptor in the endpoint descriptor list.

All transfer descriptors in the linked list that this endpoint descriptor points to use the same attributes of this endpoint descriptor. This means that all transfer descriptors linked with this endpoint descriptor must communicate with the same device and endpoint, in the same direction (if the D field is 01b or 10b), have the same max packet size, and be of the same format.

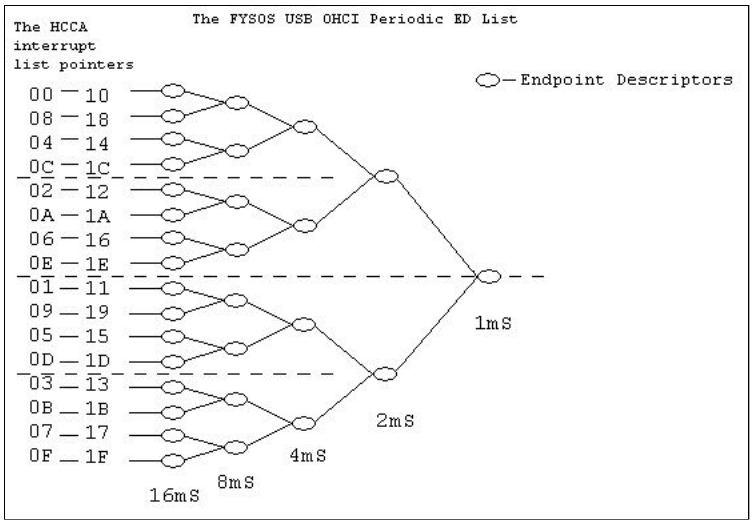
Now that we know what an endpoint descriptor is, we need to create the control, bulk, and periodic lists. The control and bulk lists must have at least one endpoint descriptor per list. Since each list is a linked list of endpoint descriptors, you can add as many as you would like, as they are needed. The FYSOS driver has a list of 16 each, and each endpoint descriptor points to the next endpoint descriptor in its respective list with the last having a value of zero in the NextED field. Remember that each endpoint descriptor must be paragraph (16-byte) aligned.

The periodic list is a little more difficult. If you remember, the periodic list is in the HCCA data structure and this structure had to be specifically aligned with the alignment calculation described in an earlier chapter.

Once we have a correctly aligned buffer for the HCCA, we can fill it with valid data. The first 128 bytes are not endpoint descriptors, but are 32 dwords, each a physical address pointer to an actual endpoint descriptor.

You might think that we just need a memory block with enough room to hold 32 endpoint descriptors, similar to the control or bulk list. However, this is our interrupt and isochronous list, and devices that are interrupt driven or use isochronous data packets, usually have a specific interval between packet transfers. Therefore, we will create a chain of endpoint descriptors that when we insert a transfer descriptor, we can insert one so that it gets executed every 16mS, 8mS, 4mS, 2mS, or every single frame, 1mS. The example in the OHCI specification has one more leaf, allowing an interval to be every 32mS. However, FYSOS simply creates 16 pointers, with five leafs, with the first 16 pointing to these leafs. The second 16 pointers are duplicate pointers and point to the same leaf items as the first 16.
The figure below shows the FYSOS Periodical List.

Figure 6-2: OHCI Periodic ED List



See Figure 3-4 of the OHCI specification for a similar, more detailed figure.

On the left side, you will see the pointer offsets. The “00 - 10” figure means that the 1st and 17th dwords in the periodic list of the HCCA point to the first endpoint descriptor in the 16mS column, the “08 - 18” figure means that the 9th and 25th dwords point to the 2nd endpoint descriptor, and so on.

If you divide the first column of endpoint descriptors in half, half on top, and half on bottom (indicated by the dotted line), you will notice that the 1st and 17th pointer in the HCCA periodic list, points to the 1st endpoint descriptor in the top half. The 2nd and 18th pointers, point to the 1st endpoint descriptor in the bottom half. Then divide the two halves in half again (shown with shorter dotted lines). The 3rd and 19th pointers point to the 1st endpoint descriptor in the 2nd half of the first original half, while the 4th and 20th pointers point to the 1st endpoint descriptor in the 2nd half of the 2nd original half. Do you see the pattern?

With this in mind, you should see that by linking a transfer descriptor with an endpoint descriptor in column 1, it will get processed every 16mS. Linking a transfer descriptor with an endpoint descriptor in column 2, every 8mS. If you link a transfer descriptor with an endpoint descriptor in column 5, that transfer descriptor will be executed every millisecond.

To get the controller to process the transfer descriptor as shown on the previous page, you will also have to process the return information, and update the transfer descriptor before the next frame interval for that transfer descriptor. If you do not, your transfer descriptor will only be processed once, even though it is still linked to the endpoint descriptor.

Using the following code, you can create the periodic list described previously.

Listing 6-1: OHCI Schedule Creation

```
bit8u ohci_ed_leaf0[16] =
 { 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15, };
bit8u ohci_ed_leaf1[ 8] =
 { 16, 20, 18, 22, 17, 21, 19, 23 };
bit8u ohci_ed_leaf2[ 4] =
 { 24, 26, 25, 27 };

 // (column 0)
 for (j=0; j<16; j++) {
```

```
 hcca.InterruptTable[j]    =
 hcca.InterruptTable[j+16] = &ed_table[ohci_ed_leaf0[j]];
}

// (column 1)
for (j=0; j<7; j++)
  ed_table[j].nexted = &ed_table[ohci_ed_leaf1[j]];

// (column 2)
for (j=0; j<3; j++)
  ed_table[j+16].nexted = &ed_table[ohci_ed_leaf2[j]];

// (column 3)
ed_table[28].nexted = ed_table[29].nexted = &ed_table[30];

// (column 4)
ed_table[30].nexted = 0x00000000;
```

Of course during the creation, you will need to clear/set each field within each endpoint descriptor. Simply setting the Halt and sKip bits will make sure that the controller does not execute the endpoint descriptor. Not yet, anyway.

Transfer Descriptors

Transfer Descriptors are what do the actual work. They are used for the actual data transfer. There are two kinds of transfer descriptors. One being the Standard Transfer Descriptor used for interrupt, control, and bulk transfers, and the other is called a Isochronous Transfer Descriptor, used for isochronous transfers or ISO transfers. For now we will only use the Standard Transfer format. See Appendix D and Part 5 of this book for information on ISO transfers.

The standard transfer descriptor is 16 bytes, 4 dwords, and has the following format.

Table 6-3: OHCI Transfer Descriptor Format

31:28	27 26	25 24	23 21	20 19	18:04	3	2	1	0
CC	EC	T	DI	DP	R	--			
Current Buffer Pointer (CBP)									
Next Transfer Descriptor (NextTD)						0			
Buffer End (BE)									

The transfer descriptor contains the physical addresses of the first byte and last byte to be transferred, the physical offset of the next transfer descriptor, and other flags used before processing, and updated after processing. The two buffer pointers can be byte aligned, while the Next Transfer Descriptor pointer must be paragraph (16-byte) aligned. The following table describes each field.

Table 6-4: OHCI Transfer Descriptor Definition

Standard Transfer Descriptor Definition		
Name	HC Access	Description
R	RO	**bufferRounding**. This bit is used for short packet detect, and set by the driver. 0 = no short packet allowed. 1 = the last packet may be smaller than requested without returning an error.
DP	RO	**Direction/PID**. If the direction field in the endpoint descriptor was 00b or 11b, then this 2-bit field is used for direction/PID. 00 = SETUP 01 = OUT 10 = IN 11 = reserved
DI	RO	**DelayInterrupt**. If this field is non zero, but less than 111b, then the controller will wait DelayInterrupt frames before it fires the interrupt. See later for a note on this field.
T	R/W	**DataToggle**. This is the data toggle that is compared with the PID value (DATA0/DATA1) of a successful transmission/reception of a data packet. 0x = The data toggle is acquired from the toggleCarry field in the endpoint descriptor. The low order bit of this field is ignored. 1x = The data toggle is acquire from the low order bit of this field (x).

<table>
<tr><td>EC</td><td>R/W</td><td colspan="3">ErrorCount. The controller increments this value each time there is an error. If the error count is 2 and another error occurs, the error is recorded in the ConditionCode field and the transfer descriptor is placed in the done queue. If 2 or fewer errors occur, then this field is cleared before it is placed in the done queue.</td></tr>
<tr><td>CC</td><td>R/W</td><td colspan="3">ConditionCode. This field contains the error condition of the attempted transaction associated with this transfer descriptor after it is processed. Before it is processed, your driver should write 14 or 15 to this field.</td></tr>
<tr><td></td><td></td><td>0000</td><td>NoError</td><td>The transfer descriptor completed with no detectable errors.</td></tr>
<tr><td></td><td></td><td>0001</td><td>CRC</td><td>Last data packet from the endpoint contained a CRC error.</td></tr>
<tr><td></td><td></td><td>0010</td><td>BitStuffing</td><td>Last data packet from the endpoint contained a bit stuffing error.</td></tr>
<tr><td></td><td></td><td>0011</td><td>Data Toggle Mismatch</td><td>Last data packet from the endpoint had a data toggle that did not match what was expected.</td></tr>
<tr><td></td><td></td><td>0100</td><td>Stall</td><td>A stall was detected.</td></tr>
<tr><td></td><td></td><td>0101</td><td>Device not Responding</td><td>Device did not respond to a token (IN) or did not proved a handshake (OUT)</td></tr>
<tr><td></td><td></td><td>0110</td><td>PID Check Failure</td><td>Check bits on PID from endpoint failed on data PID (IN) or handshake (OUT)</td></tr>
<tr><td></td><td></td><td>0111</td><td>Unexpected PID</td><td>Received PID was not valid when encountered or PID is not defined.</td></tr>
<tr><td></td><td></td><td>1000</td><td>Data Overrun</td><td>The amount of data returned by the endpoint was more than either the size of the maximum data packet allowed or the remaining buffer size.</td></tr>
<tr><td></td><td></td><td>1001</td><td>Data Underrun</td><td>The endpoint returned less than max packet size.</td></tr>
<tr><td></td><td></td><td>1010</td><td>Reserved</td><td></td></tr>
<tr><td></td><td></td><td>1011</td><td>Reserved</td><td></td></tr>
<tr><td></td><td></td><td>1100</td><td>Buffer Overrun</td><td>During an IN, the controller received data from the endpoint faster than it could be written to the systems memory.</td></tr>
<tr><td></td><td></td><td>1101</td><td>Buffer Underrun</td><td>During an OUT, the controller could not receive data from the systems memory fast</td></tr>
</table>

				enough to keep up with the USB data rate.
		111x	**Not accessed**	Your driver sets the high three bits to tell the controller that this transfer descriptor has not been accessed yet.
CBP	R/W	**CurrentBufferPointer**. This field contains the physical address of the first byte of the buffer to be transferred, or zero if no data to be transferred.		
NextTD	R/W	**NextTD**. This points to the physical address of the next transfer descriptor in the list. This field is not necessarily used to tell the controller that this is the last transfer descriptor. The HeadP and TailP fields in the endpoint descriptor are used instead.		
BE	R	**BufferEnd**. This field contains the physical address of the last byte in the buffer to be transferred by this Transfer Descriptor.		

The 18-bit field in the first dword in Table 6-3, marked as '--', can be used by your driver however you would like. The controller is not to modify this field. You may use it for driver flags, or other information. The FYSOS driver actually allocates 32 bytes for every transfer descriptor and uses the second 16 bytes for driver flags and data.

You should set the DelayInterrupt field in the Transfer Descriptor to 7 (111b) for all Setup and Data Transfer Descriptors, then setting it to zero (000b) for the Status Transfer Descriptor. This will tell the controller to update the HcDoneHead register. See Chapter 12 for more about this technique.

Before we get to the controller's process of execution, there are a few more things we need to do.

The remaining parts of the HCCA need to be set to zero before you put the controller in the operational state. The HccaFrameNumber and HccaDoneHead fields should be zero and the HccaPad1 field should be zero.

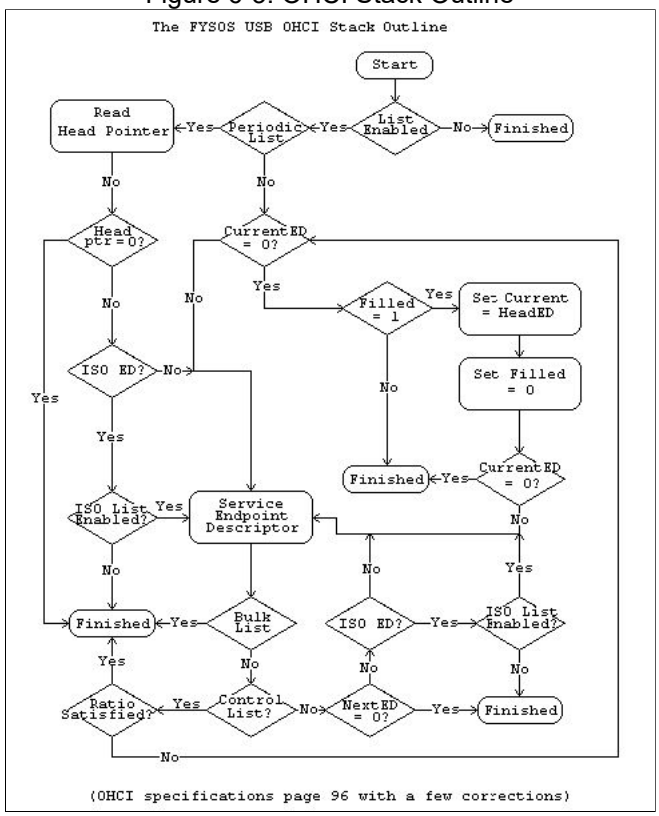
The HccaPad1 register may be set to a non-zero value since when the controller updates the HccaFrameNumber field it will clear the HccaPad1 field. This way you can monitor when the HccaFrameNumber is changed. However, I don't know why you would want to do this. You can simply read the HccaFrameNumber field and wait for it to change just as easy as reading the HccaPad1 field and waiting for it to change.

See Figure 6-3 below on how the OHCI controller handles a single frame. This outline is directly from page 96 of the OHCI specification with a few minor typing errors fixed in this outline.

Summary

If you feel that you have grasped this information well enough, then you are ready to continue on. If you would like to learn about the EHCI controller, go on to the next chapter. However, if you would like to go directly to how to enumerate the OHCI's root hub, then go directly to Chapter 12 in Part 2 of this book.

Figure 6-3: OHCI Stack Outline



Chapter 7 – The EHCI Hardware

The following information pertains to version 1.0 of the EHCI specifications dated 12 March 2002. In August of 2008, Intel released version 1.1 titled “Energy-efficiency extensions for the EHCI specification”. I will not go into much, if any detail about version 1.1. If you would like to see the additions, please see the EHCIv1-1_Addendum.pdf file from the USB-IF web page (See Appendix A).

That addendum adds items to currently reserved bits in many of the controller’s registers, including changing the HCIVERSION register from 0x0100 to 0x0110. If your driver finds a controller with version 1.1, and does not support that version, then when reading and writing to any register, do not modify any reserved bits. In other words, when writing to a register with reserved bits, always read the register first, modify the desired documented bits, and then write it back so that you leave the reserved bits unchanged. If one of the reserved bits is now Write-Clear, your driver won’t hurt anything by clearing the bit (writing a 1 since you read a 1).

When I first saw the EHCI controller, I have to admit, I asked myself which ports were the high-speed ports and which were the low- and full-speed ports and why are they not marked? The answer--they all are. The EHCI controller controls all the ports passing ownership to the companion controllers for low- and full-speed devices. Continue with the next section for an explanation.

Explanation of companion controllers

Before we get into the EHCI controller, there are a few misconceptions that I would like to clear up. There has been a lot of documentation and other sources saying that the ECHI controller is version 2.0 of the USB, while the UHCI and OHCI controllers are version 1.10. This is not the case. Version 2.0 of the USB specification is still compatible with the OHCI and UCHI controllers, as USB version 1.10 is still compatible with the EHCI controller. Version 2.0 of the USB specification simply adds information for use with the EHCI controller. Each controller should still support version 1.10 and all devices are version 2.0 compatible, though they may not have all the capabilities explained in version 2.0. Only high-speed devices use the new information in version 2.0 and the EHCI controller.

With this in mind, how about some more information to confuse you? The EHCI controller will not communicate with low- and full-speed devices. The EHCI controller will only work with high-speed devices. So, does this mean if you have an EHCI controller, you can only use high-speed devices? No. The EHCI controller has zero or more OHCI or UHCI controllers, called companion controllers, on board to handle low- and full-speed devices. When the EHCI controller finds that a non high-speed device has been attached to one of its root hub ports, it passes control of that port to one of the companion controllers. Which companion controller it is sent to, is set up in one of the EHCI registers. Your driver can determine which controller has control of which root hub port by reading certain fields in certain registers described within this chapter.

After a low- or full-speed device is disconnected from an EHCI's root hub port, the EHCI controller monitors control of that port until another device is again attached. It then regains control if it is a high-speed device or passing control to the companion controller if it is a low- or full-speed device.

Please note that an EHCI controller may have no companion controllers, and therefore would only accept high-speed devices. However, this is a rare and mainly non-existent setup used on embedded devices and the sort. However, you can plug a high-speed external hub into that port, then plug low- and full-speed devices into that external hub.

If a low- or full-speed external hub is attached to one of the EHCI's root hub ports, and a high-speed device is attached to one of the external hub's ports, that device will only function in low- or full-speed depending on the speed of the external hub. A device can only function in a speed less than or equal to the speed of the hub it is attached to.

If you recall, in the chapter on the PCI hardware, any companion controllers will be seen as lower numbered functions on the same PCI device. In most BIOS's, there is an option to "disable" the use of the EHCI controller if the host operating system does not support EHCI. All the BIOS does, is set up the EHCI controller to pass all connections to the companion controllers. Any high-speed device will be seen as a full-speed device and may or may not be able to function past enumeration. This was added so that non-EHCI compatible operating systems could still use the companion controllers. These operating systems would see the companion controllers as standalone controllers and would not need to do anything with the EHCI.

Now that we have the role of the EHCI controller, let's continue with the set up of the hardware.

This chapter describes how to set up the EHCI host controller, a minimal stack frame, and start the controller. You should have read Chapter 2 to know how to get the base I/O address and IRQ number from the PCI and be familiar with memory mapped and port I/O addressing.

A majority of the information in this section, including most of the tables is readily available from the Intel EHCI specification. This information is included here only for reference, and to clear up a few things that may be a little confusing. To get the original tables, see Appendix A for information on how to get the specification file.

The EHCI USB controller has two sets of I/O registers, with the first one starting at the base address described in Chapter 2, and the second one a few bytes after, though its address is calculated via a register in the first set.

Host Controller Capability Registers

The first set is the Host Controller Capability Registers and has the following format. All registers are read-only.

Table 7-1: EHCI Capability Registers

The EHCI Controllers Capability Registers		
Offset	Size	Description
00h	1	CAPLENGTH: Total length of these registers
01h	1	Reserved
02h	2	HCIVERSION: Interface Version Number
04h	4	HCSPARAMS: Structural Parameters
08h	4	HCCPARAMS: Capability Parameters
0Ch	8	HCSP-PORTROUTE: Companion port route description

These registers specify the limits, restrictions, and capabilities of and are used as parameters for the EHCI and are explained below.

Table 7-2: The EHCI CAPLENGTH Register

EHCI controller: CAPLENGTH (Base + 00h)		
Bit	Access	Description
7:0	RO	This byte value is used to find the second set of registers, the Operational Registers.

The Operational Registers, the second set of registers, are located at BaseAddress plus this byte's value. The value of this register is usually 0x10, but could be 0x14 if there are nine or more root hub ports. A controller on one of my test machines has this value as 0x20.

In theory, the amount of memory mapped address space that the host controller needs, would be 148 bytes. If the Capability Registers took 0x14 (20d), the Operational Registers took 0x44 (68d) not counting the ports, and you could have a total of 15 ports using 4 bytes each, this gives 20+68+(15*4). However, to get the actual amount that the PCI has reserved for this card, see the "Determine the Address Space Required for a Function" section in Chapter 2. It is always a good idea to get the size from the PCI as described in that chapter. A device could have Debug-Port capabilities, which may locate the debug registers after the 148 bytes calculated above.

Table 7-3: EHCI Reserved

EHCI controller: reserved (Base + 01h)		
Bit	Access	Description
7:0	RO	Reserved

This byte is reserved for future use.

Table 7-4: The EHCI HCIVERSION Register

EHCI controller: HCIVERSION (Base + 02h)		
Bit	Access	Description
15:0	RO	Interface version (in BCD)

This 16-bit register holds the version number of the interface this controller supports. Versions 0x0095, 0x0096, 0x0100 and 0x0110 are supported, and are Binary Coded Decimals of 0.95, 0.96, 1.00 and 1.10 respectively.

I have seen a VIA VX800/VX820 EHCI controller that has a version value of 0x0095, even though the data sheet for that motherboard says it should be 0x0100. Always use the 0x0095 version in this case.

Table 7-5: The EHCI HCSPARAMS Register

EHCI controller: HCSPARAMS (Base + 04h)		
Bit	Access	Description
3:0	RO	N_PORTS
4	RO	Port Power Control (PPC)
6:5	RO	Reserved
7	RO	Port Routing Rules
11:8	RO	Number of Ports per Companion Controller (N_PCC)
15:12	RO	Number of Companion Controllers (N_CC)
16	RO	Port Indicators (P_INDICATOR)
19:17	RO	Reserved
23:20	RO	Debug Port Number (optional)
31:24	RO	Reserved

The N_PORTS field is the number of downstream ports on the root hub. This is the physical count of ports, no matter how many companion controllers or which controller has control. A value of zero is undefined. Why would you have a controller without any ports? With 4 bits, the maximum count of ports on the root hub is 15.

The Port Power Control (PPC) bit indicates that your driver may control the power to each port. A value of 1 means that you can turn on or off the power to each port via a bit in that port's register. A value of zero means that all ports are hardwired to have power on.

The Port Routing Rules bit indicates to the driver the method of how the downstream ports are divided among the companion controllers. If this bit is 1, the ports are divided indicated by the HCSP-PORTROUTE register. If this bit is zero, then using bits 11:8 of this register, the first N_PCC number of ports are on the first companion controller, the next N_PCC number of ports are on the second, and so on.

The Number of Ports per Companion Controller (N_PCC) field tells how many downstream ports are controlled by each companion controller. For instance, if there are 4 total ports and this field contains the value 2, then the first companion controller will control the first 2 ports, while the second controls the last two ports. If the value in this field is 3 and there are a total of 7 downstream ports, then the first controller will control the first 3 ports, the second controller will control the next 3 ports, and the last controller will control the last single port. It doesn't specify in the EHCI specification, but this field may be invalid if the Port Routing Rules bit is set.

The Number of Companion Controllers (N_CC) field specifies how many companion controllers are associated with this EHCI controller. If this field is zero, there are no companion controllers and this EHCI controller only supports high-speed devices.

The Port Indicators (P_INDICATOR) bit, if set, indicates that this controller supports port indicator lights and that you can control them.

The Debug Port Number field tells your driver which of the downstream ports is the debug port. If it is non-zero, then it is the 1-based number of the debug port. If it is zero, this controller does not support the debug port option.

If a debug port is indicated, then this port can be used to monitor the bus activity of that port, using the debug controller. The debug controller uses a separate register space and when initialized, will have control of that port. See Appendix C in the EHCI v1.0 specification, page 133.

Table 7-6: The EHCI HCCPARAMS Register

EHCI controller: HCCPARAMS (Base + 08h)		
Bit	Access	Description
0	RO	64-bit Addressing Capability
1	RO	Programmable Frame List Flag
2	RO	Asynchronous Schedule Park Capability
3	RO	Reserved
7:4	RO	Isochronous Scheduling Threshold
15:8	RO	EHCI Extended Capabilities Pointer (EECP)
31:16	RO	Reserved[1]
[1]Bits 19:16 are used in version 1.10		

The 64-bit Addressing Capability bit indicates that the controller supports 64-bit addressing in the data structures. If this bit is clear, all the data structures use the 32-bit addressing versions. If this bit is set, they use the 64-bit versions. The 64-bit versions simply add enough 32-bit dwords to the end of the 32-bit version to store the high 32-bits of each 64-bit address.

Don't confuse this bit with the address of these I/O registers. This bit has nothing to do with the size of the address in the PCI address space. The bit in the lower nibble of the base address in the PCI configuration space indicates whether the address to these I/O registers is 64-bit.

The Programmable Frame List Flag bit indicates that the controller supports smaller frame list sizes. If this bit is set, then your driver may create a frame list that is smaller than 1,024 elements using the USBCMD register. If this bit is clear, your frame list must be 1,024 elements. Either way, your frame list must be page aligned.

The Asynchronous Schedule Park Capability bit indicates that the controller supports the park feature for high-speed queue heads in an Asynchronous Schedule. Using the USBCMD register, you can enable this by the Asynchronous Schedule Park Mode Enable and the Asynchronous Schedule Park Mode Count fields.

The Isochronous Scheduling Threshold field indicates when your driver can reliably update the isochronous schedule.

The EHCI Extended Capabilities Pointer field is used as an offset within the PCI configuration space for this device, to the Extended Capabilities list. If this field is zero, the list is non-existent. If this field is non-zero, then it should be greater than or equal to 0x40 and will point to the list. For the EHCI, there should be two dwords listed. The first is a pointer to the next, while the second holds flags to tell the controller whether or not to issue a SMI (System Management Interrupt) when certain fields of the EHCI controller change. This support is mainly used by the BIOS for legacy support and Power Management.

Table 7-7: The EHCI HCSP-PORTROUTE Register

EHCI controller: HCSP-PORTROUTE (Base + 0Ch)		
Bit	Access	Description
3:0	RO	Port0 routing value
7:4	RO	Port1 routing value
...	...	...

35:32	RO	Port8 routing value
...	...	...
59:56	RO	Port14 routing value
63:60	RO	Reserved

Depending on the value in the CAPLENGTH register above, this field will be either 32-bits or 64-bits in size. This field has at most 15 nibble-sized fields indicating which companion controller a given port is routed to. The first nibble indicates which companion controller the first port is routed to, and so on. These fields are only valid if bit 7 in the HCSPARAMS register is set.

Technically, the field above could be non-existent and the CAPLENGTH field could have a value of 0x0C. Since these registers are read-only and bit 7 in the HCSPARAMS register could be zero, there would be no reason to reserve the space for the HCSP-PORTROUTE field. However, I would suspect that a manufacturer would follow the norm and leave 32- or 64-bits for this field, used or not.

Host Controller Operational Registers

The second set of controller registers, is the Host Controller Operational Registers and has the following format. The first seven registers are in the core power well, while the last two are in the auxiliary power well.

The power wells are used when the host goes to sleep. The core power well may remain powered and the auxiliary well may be dropped.

Table 7-8: The EHCI Operational Registers

The EHCI Controllers Operational Registers		
Offset	Size	Description
00h	4	USBCMD: USB Command
04h	4	USBSTS: USB Status
08h	4	USBINTR: USB Interrupt Enable
0Ch	4	FRINDEX: USB Frame Index
10h	4	CTRLDSSEGMENT: 4G Segment Selector
14h	4	PERIODICLISTBASE: Frame List Base Address
18h	4	ASYNCLISTADDR: Next Asynchronous List Address
1Ch-3Fh	36	Reserved
40h	4	CONFIGFLAG: Configured Flag Register
44h	4	PORTSC[]: Port Status/Control

These registers are used to control the EHCI controller and the root hub and are explained next. Each register is a 32-bit dword and should only be accessed with 32-bit reads and writes.

Table 7-9: The EHCI Command Register

EHCI controller: USBCMD: USB Command Register		
Bit	Access	Description
0	R/W	Run/Stop (RS)
1	R/W	Host Controller Reset (HCRESET)
3:2	R/W / RO	Frame List Size
4	R/W	Periodic Schedule Enable
5	R/W	Asynchronous Schedule Enable
6	R/W	Interrupt on Async Advance Doorbell
7	R/W	Light Host Controller Reset
9:8	R/W / RO	Asynchronous Schedule Park Mode Count
10	RO	Reserved
11	R/W / RO	Asynchronous Schedule Park Mode Enable
15:12	RO	Reserved
23:16	R/W	Interrupt Threshold Control
31:24	RO	Reserved

The Run/Stop bit is set by your driver to start the scheduler or cleared to stop the scheduler. To stop the scheduler, write a zero to this bit, and then wait for the HCHalted bit in the USBSTS register to be set. You should not set this bit unless the HCHalted bit is also set.

The Host Controller Reset bit is used to reset the controller. All of the Operational Registers are reset to their default states and any transaction is halted, but the USB reset is not driven on the downstream ports. None of the PCI registers are affected by this reset. Ownership of each

port reverts back to the companion controllers. Once the reset is finished, the host controller will clear this bit. Your driver should make sure the HCHalted bit in the USBSTS register is set before you try to reset the controller.

The Frame List Size field is only valid if the Programmable Frame List Flag is set in the HCCPARAMS register, and specifies the size of the frame list. This allows smaller frame lists if the hardware resources are small. The values are listed in the table below
.

Table 7-10: EHCI: Size of Frame List

Value	Size of Frame List
00b	1024 elements at 4096 bytes, default value
01b	512 elements at 2048 bytes
10b	256 elements at 1024 bytes
11b	Reserved (32 elements in EHCI version 1.1)

The Periodic Schedule Enable bit simply is set by your driver to enable the Periodic list which uses the PERIODICLISTBASE register and the Asynchronous Schedule Enable bit is set to enable the Asynchronous list which uses the ASYNCLISTADDR register.

The Interrupt on Async Advance Doorbell bit is a way for the controller to tell the driver that it has advanced the Async state. If you set the corresponding bit in the Command register, the controller will set the Interrupt on Async Advance status bit in the USBSTS register, then if the Interrupt on Async Advance Enable bit is set in the USBINTR register, the controller will fire an interrupt on the next interrupt threshold.

The Light Host Controller Reset bit is optional and if implemented and a 1 is written to it, will reset the Operational Registers, but will not reset any of the root hub ports nor will it change the companion controller ownership status. The controller will clear this bit when the light reset is done. To see if this reset is implemented, set this bit and then directly read it back. If you read a zero, this reset is not implemented.

> I have found that this reset, if implemented, takes just about a single microsecond to complete. Therefore, to see if it is implemented, you will need to make sure that you read the bit back less than a microsecond after setting it. Another technique for checking for implementation would be to set one of the effected registers to a known, non-default value, setting this bit, then checking for the same value in that effected register.

The Asynchronous Schedule Park Mode Count field is valid and R/W if the Asynchronous Park Capability bit in the HCCPARAMS register is set, else this field is invalid and read only. If this field is valid, it is the amount of successive transactions the controller is allowed to process from a high-speed queue head before continuing traversal of the schedule. Values 1 to 3 are allowed, while a value of zero produces undefined behavior.

The Asynchronous Schedule Park Mode Enable bit is valid and R/W if the Asynchronous Park Capability bit in the HCCPARAMS register is set, else this field is invalid and read only. If this bit is valid, and when set, this Enables Park mode.

The Interrupt Threshold Control field is used to tell the controller how many micro-frames to process before it can fire an interrupt. This field should not be written to unless the HCHalted bit is set in the USBSTS register. Valid values for this register are listed below while invalid values will have undefined results.

Table 7-11: EHCI: Maximum Interrupt Interval

Value	Maximum interrupt interval

0x00	Reserved
0x01	1 micro-frame
0x02	2 micro-frames
0x04	4 micro-frames
0x08	8 micro-frames (default) (1mS)
0x10	16 micro-frames (2mS)
0x20	32 micro-frames (4mS)
0x40	64 micro-frames (8mS)

The EHCI Status register has the format listed below.

Table 7-12: The EHCI Status Register

EHCI controller: USBSTS: USB Status Register		
Bit	Access	Description
0	R/WC	USB Interrupt (USBINT)
1	R/WC	USB Error Interrupt (USBERRINT)
2	R/WC	Port Change Detect
3	R/WC	Frame List Rollover
4	R/WC	Host System Error
5	R/WC	Interrupt on Async Advance
11:6	RO	Reserved
12	RO	HCHalted
13	RO	Reclamation
14	RO	Periodic Schedule Status
15	RO	Asynchronous Schedule Status
31:16	RO	Reserved

The USB Interrupt bit is set by the controller on the completion of a transaction that had the IOC bit set or when a short packet was detected.

The USB Error Interrupt bit is set by the controller when the completion of a transaction resulted in an error. If the transaction also had the IOC bit set, then the USB Interrupt bit is also set.

The Port Change Detect bit is set by the controller when any of the following occur: 1) Any port that has the port owner bit clear, and has any port bit change from zero to one. 2) When the Force Port Resume bit goes from zero to a one due to a J-K transition detected on a suspended port. 3) When the Connect Status Change bit is set after system software has relinquished ownership of a connected port by writing a one to the Port Owner bit.

The Frame List Rollover bit is set by the controller every time the frame list index rolls over from its maximum size to zero. The maximum size is determined by the Frame List Size field in the USBCMD register. For example, if the frame list size is 512, the host controller sets this bit every time bit 12 of the FRINDEX register toggles. If the frame list size is 1024, then every time bit 13 toggles.

The Host System Error bit is set when a serious error occurs during a host system access. In a PCI system, this bit would be set due to a PCI Parity error, PCI Master Abort, or a PCI Target Abort. When the controller sets this bit, it also clears the Run/Stop bit in the USBCMD register so that no further serious errors can occur.

The Interrupt on Async Advance bit is set by the controller when the Interrupt on Async Advance Doorbell bit is set in the USBCMD register and the asynchronous schedule is

advanced.

The HCHalted bit is set by the controller after it has stopped executing as a result of the Run/Stop bit being cleared by software or by the controller due to an error.

The Reclamation bit is set by the controller when it detects an empty Async schedule.

The Periodic Schedule Status bit is set by the controller when the Periodic schedule is enabled. This schedule is not automatically enabled due to software setting the Periodic Schedule Enable bit in the USBCMD register. Therefore, your driver should watch for this bit to become set before it assumes the Periodic Schedule is being processed. When you disable the Periodic Schedule, your driver should watch for this bit to become clear before it assumes the schedule is disabled. Therefore the schedule is not enabled until both bits are set, and it is not disabled until both bits are clear.

The Asynchronous Schedule Status bit is set by the controller when the Asynchronous schedule is enabled. This schedule is not automatically enabled due to software setting the Asynchronous Schedule Enable bit in the USBCMD register. Therefore, your driver should watch for this bit to become set before it assumes the Asynchronous Schedule is being processed. When you disable the Asynchronous schedule, your driver should watch for this bit to become clear before it assumes the schedule is disabled. Therefore the schedule is not enabled until both bits are set, and it is not disabled until both bits are clear.

Table 7-13: The EHCI Interrupt Enable Register

EHCI controller: USBINTR: USB Interrupt Enable Register		
Bit	Access	Description
0	R/W	USB Interrupt Enable
1	R/W	USB Error Interrupt Enable
2	R/W	Port Change Interrupt Enable
3	R/W	Frame List Rollover Enable
4	R/W	Host System Error Enable
5	R/W	Interrupt on Async Advance Enable
31:6	RO	Reserved

The USB Interrupt Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the USBINT bit in the USBSTS register becomes set. Your driver clearing the USBINT bit acknowledges this interrupt.

The USB Error Interrupt Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the USBERRINT bit in the USBSTS register becomes set. Your driver clearing the USBERRINT bit acknowledges this interrupt.

The Port Change Interrupt Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the Port Change Detect bit in the USBSTS register becomes set. Your driver clearing the Port Change Detect bit acknowledges this interrupt.

The Frame List Rollover Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the Frame List Rollover bit in the USBSTS register becomes set. Your driver clearing the Frame List Rollover bit acknowledges this interrupt.

The Host System Error Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the Host System Error Status bit in the USBSTS register becomes set. Your driver clearing the Host System Error Status bit acknowledges this interrupt.

The Interrupt on Async Advance Enable bit is set by your driver to enable an interrupt at the next interrupt threshold when the Interrupt on Async Advance bit in the USBSTS register becomes set. Your driver clearing the Interrupt on Async Advance bit acknowledges this interrupt.

Table 7-14: The EHCI Frame Index Register

EHCI controller: FRINDEX: Frame Index Register		
Bit	Access	Description
13:0	R/W	Frame Index
31:14	RO	Reserved

The Frame Index field is updated every micro-frame (every 125mS) and is used to select an item in the Periodic List. Depending on the size of the frame list, using the Frame List Size field in the USBCMD register, all of bits 13:0 may not be used.

If the frame list size is 1,024 elements, then Frame Index uses bits 12:3, if the frame list size is 512 elements, then it uses bits 11:3, if the frame list size is 256 elements, then it uses bits 10:3. The low order bits, bits 2:0, are the micro-frame index. This way, each frame entry gets processes 8 times, or once for each micro-frame in that frame entry.

Since this may be confusing for you, here is another way to look at the same dword value of this register when the frame size is 1024, 512, and 256 respectively.

Table 7-15: EHCI: Frame Index Register: 1024

EHCI controller: FRINDEX: Number of elements = 1024 (4096 bytes)		
Bit	Access	Description
2:0	R/W	Micro-frame Index. Counts from 0 to 7. Rolls over at 8, and increments Frame Index below.
12:3	R/W	Frame Index. Counts from 0 to 1023. Rolls over at 1024, and toggles bit 13 below.
13	R/W	Toggles from 1 to 0 or 0 to 1 each time the Frame Index above rolls over.
31:14	RO	Reserved

Table 7-16: EHCI: Frame Index Register: 512

EHCI controller: FRINDEX: Number of elements = 512 (2048 bytes)		
Bit	Access	Description
2:0	R/W	Micro-frame Index. Counts from 0 to 7. Rolls over at 8, and increments Frame Index below.
11:3	R/W	Frame Index. Counts from 0 to 511. Rolls over at 512, and toggles bit 12 below.
12	R/W	Toggles from 1 to 0 or 0 to 1 each time the Frame Index above rolls over.
13	R/W	Undefined. See note below.
31:14	RO	Reserved

Table 7-17: EHCI: Frame Index Register: 256

EHCI controller: FRINDEX: Number of elements = 256 (1024 bytes)		
Bit	Access	Description
2:0	R/W	Micro-frame Index. Counts from 0 to 7. Rolls

		over at 8, and increments Frame Index below.
10:3	R/W	Frame Index. Counts from 0 to 255. Rolls over at 256, and toggles bit 11 below.
11	R/W	Toggles from 1 to 0 or 0 to 1 each time the Frame Index above rolls over.
13:12	R/W	Undefined. See note below.
31:14	RO	Reserved

In my tests to see what happens to bits 13:12 when the frame size is 256, I have found that these bits are undefined. I tried setting both bits, and once read back, only bit 12 was set. After letting the schedule run for a few seconds, both bits were clear on one test, while bit 13 was set on another.

You shouldn't need to write to the Frame Index Register except at initialization time, where you write 0x00000000 to this register. You must write to this register using a dword write only. In other words, you must write all 4 bytes with one dword write.

Version 1.1 of the EHCI specifications allows for frames with only 32 elements. It operates just as if it were a 1024 element list, but uses modulo 32 on the frame list register. Therefore, the controller goes through the 32 elements 32 times while the frame list register counted to 1024 once.

Table 7-18: The EHCI Ctrl/Data Structure Segment Register

EHCI controller: CTRLDSSEGMENT: Ctrl/Data Structure Seg Register		
Bit	Access	Description
31:0	R/W	Control Data Structure Segment

If the 64-bit Addressing Capability bit in the HCCPARAMS register is set, the CTRLDSSEGMENT register is used as the most significant address bits [63:32] for the 64-bit address when combined with the PERIODICLISTBASE register and the ASYNCLISTADDR register. Therefore, if you were using 64 bit addresses, you would use the register pair CTRLDSSEGMENT:PERIODICLISTBASE to point to the periodical list. Same for the asynchronous list, you would use CTRLDSSEGMENT:ASYNCLISTADDR. However, you must use the same value for both lists. That is, you must have both lists within the same 4Gig memory block.

Writes must be dword writes. If the 64-bit Addressing Capability bit in the HCCPARAMS register is clear, this register is R/O and will return 0x00000000.

Table 7-19: The EHCI Periodic List Address Register

EHCI controller: PERIODICLISTBASE: Periodic List Address Register		
Bit	Access	Description
11:0	R/W	Reserved, and are undefined during runtime
31:12	R/W	Base Address. Bits 31:12

The PERIODICLISTBASE register holds the 4k aligned physical address of the periodic list. If the 64-bit Addressing Capability bit in the HCCPARAMS register is set, the CTRLDSSEGMENT register is used as the most significant address bits [63:32] for 64-bit addresses. Writes must be dword writes.

Table 7-20: The EHCI Async List Address Register

EHCI controller: ASYNCLISTADDR: Async List Address

Register		
Bit	Access	Description
4:0	R/W	Reserved
31:5	R/W	Base Address. Bits 31:5

The ASYNCLISTADDR register holds the 32 byte aligned physical address of the asynchronous list. If the 64-bit Addressing Capability bit in the HCCPARAMS register is set, the CTRLDSSEGMENT register is used as the most significant address bits [63:32] for 64-bit addresses. Writes must be dword writes.

Table 7-21: The EHCI Configure Flag Register

EHCI controller: CONFIGFLAG: Configure Flag Register		
Bit	Access	Description
0	R/W	Configure Flag (CF)
31:1	RO	Reserved

The CONFIGFLAG register should be the last register you write to, after you have set up the controller. This flag, when set, gives the EHCI controller control of each root hub port. The EHCI controller will not see a device connection until this bit is set.

When this bit is clear, this flag gives the companion controllers control of each port depending on how you set up the port routing logic.

If the BIOS is EHCI aware, but your Operating System is not, the BIOS may have a setting to clear this bit, allowing the companion controllers to act as stand-alone controllers.

Table 7-22: The EHCI Port Status and Control Register

EHCI controller: PORTSC: Port Status and Control Register		
Bit	Access	Description
0	RO	Current Connect Status[1]
1	R/WC	Connect Status Change[1]
2	R/W[2]	Port Enabled/Disabled[1]
3	R/WC	Port Enable/Disable Change[1]
4	RO	Over-current Active
5	R/WC	Over-current Change
6	R/W	Force Port Resume[1]
7	R/W	Suspend[1]
8	R/W	Port Reset[1]
9	RO	Reserved
11:10	RO	Line Status[1]
12	R/W, RO	Port Power (PP)
13	R/W	Port Owner
15:14	R/W, RO	Port Indicator Control[1]
19:16	R/W	Port Test Control
20	R/W	Wake on Connect Enable (WKCNNT_E)[1]
21	R/W	Wake on Disconnect Enable (WKDSCNNT_E)[1]
22	R/W	Wake on Over-current Enable (WKOC_E)[1]
31:23	RO	Reserved
[1]These bits are zero, no matter connection status, if bit 12, Port Power, is zero. [2]This bit is actually RO. See note below.		

The PORTSC register(s) are used for controlling and receiving status from each port on the root hub. To determine how many ports there are on the root hub, read the HCSPARAMS register and mask off all but bits 3:0. With this in mind, there can be no more than 15 ports on a root hub.

Why only 15 when a 4-bit field will hold 16 values? A value of zero means that there are no ports attached.

To access any of these 15 possible ports, you read or write a dword to (OperationalBase + (0x44 + (4 * port_num))), where port_num is a zero based port number. For example, the following code snippet could be used to read from a specified port as long as the pointers were physical address pointers.

Listing 7-1: EHCI Port Read

```
// Read a dword from specified port
//     base = 32-bit physical address of operation registers
// port_num = zero based port number to read from
bit32u read_port(const bit32u base, const int port_num) {
    bit32u *port = (bit32u *) (base + 0x44);
    return port[port_num];
}
```

Explanation of the PORTSC register

The Current Connect Status bit, when set by the controller, indicates that there is a device attached. If the Port Power bit is zero, this bit will be zero, no matter the connection status.

The Connect Status Change bit, when set by the controller, indicates that a connection change has taken place. Your driver must be careful with this status change.

Please see Appendix D, "Port Connection Status Change" for an important note about connect status changes.

The Port Enabled/Disabled bit is actually read-only, even though the specification says otherwise. A port can only be enabled by the controller after a successful reset, and then only if the device attached is a high-speed device. Your driver cannot set this bit to enable the port. This bit is read to indicate whether the port is enabled or not. A port will be disabled if an error occurs such as an over-current condition.

The Over-Current Active bit indicates that there has been an over-current condition on this port. Once the over-current condition has been removed, the controller will clear this bit. The Over-current Change bit indicates that there has been a change to the over-current condition on this port.

The Force Port Resume bit is used to drive the resume signal to the down stream device. If the port is not in the suspend state and is enabled, your driver setting this bit to 1 will have undefined results.

The EHCI specification quotes that "if the port is not suspended (*Suspend* and *Enabled* bits are a one)...", which seems to contradict itself. However, this same error is in version 1.1 with additions made to this very bit's description. Not everyone's as perfect as I am. :~)

The Suspend bit is set by your driver to place the port in the suspend state. If a transaction is currently in progress, the suspend will not happen until the transaction is complete. Therefore, this bit will remain zero until the transaction is complete. Writing a zero to this bit has no effect. To take the port out of suspend, you set the Force Port Resume bit. If the port is not enabled, this bit may cause undefined behavior when set.

The Port Reset bit is used to reset and possibly enable the port. Your driver writes a one to this bit while at the same time writing a zero to the Port Enabled/Disabled bit. After the minimum of 50mS for a root hub, your driver then writes a zero to this bit. Then 2mS later, if the attached device is a high-speed device, the enable bit will set. If the enable bit doesn't set after the 2mS, the attached device is a full- or low-speed device. See later in this chapter on more about resetting a root hub port.

The Line Status bits are used to see if the device is a low-speed device. If, before reset, these bits are 01b, then there is no need to do a high-speed reset. Release ownership of the port to one of the companion controllers. If any other value, do a high-speed reset, then if the enable bit is set 2mS after the reset is complete, you have a high-speed device. If not, release ownership. Remember that these bits are undefined if port power is not on.

The Port Power bit is to switch power on or off for this port. If the Port Power Control (PPC) bit in the HCSPARAMS register is set, you may use this bit to power on or off the port. If the PPC bit is clear, this bit is read-only and port power is always on. If your driver sets this bit to a one from a zero, you must wait a minimum of 20mS before you can use the port.

The Port Owner bit is used to tell the controller who has the ownership of this port. This bit unconditionally goes to zero when your driver changes bit 0 in the CONFIGFLAG from a zero to a 1. This bit is 1 when bit 0 in the CONFIGFLAG is zero. A one in this bit tells the controller that the attached device is not a high-speed device and to release ownership to one of the companion controllers. A zero in this bit tells the controller that this is a high-speed device and to keep ownership of the port.

When the controller detects a disconnect, the EHCI regains ownership of the port, while the companion controller sees the disconnect and acts normally, completely unaware of the EHCI controller.

The Port Indicator Control bits have no effect if the P_INDICATOR bit in the HCSPARAMS register is zero. If it is 1, then these two bits have the encoding listed below.

Table 7-23: EHCI: Port Indicator Color

Value	Meaning
00b	Port indicators are off
01b	Amber
10b	Green
11b	Undefined

The USB 2.0 specification, section 11.5.3, detail the port indicator usage. Amber usually means error, while green usually means fully operational. The blinking of an indicator is used to gain the attention of the user. Blinking amber usually means hardware attention while blinking green usually means software attention. Remember that these indicators reflect the status of the port, not the device attached to it.

The Port Test Control bits are used to set a port into a certain hardware test mode. An EHCI controller may only have one port under test at one time and must be a port owned by the EHCI controller, the PortOwner is zero. In regular operation, these bits should remain zero at all times. See the USB 2.0 specification, section 7.1.20 for more information on port testing.

The Wake on Connect Enable, Wake on Disconnect Enable, and Wake on Over-current Enable bits are used to tell the controller to issue a wake-up event if either of these events occur while in a suspend state.

EHCI Host Controller Detection

Since the PCI bus already told us that there is an EHCI controller at this base, why should we take the time and effort to detect the controller? To be completely independent of the host bus, the EHCI driver should detect the controller rather than relying on its host bus. Future hardware may have the USB on a different bus type that isn't so accommodating.

To detect the controller, we will attempt to reset it and then see what happens to the register's values.

There are two bits in the command register to reset the controller. One is the Light Host Controller Reset bit and its function is optional. I discussed how to determine if it is supported earlier in this chapter. This reset will reset some of the operational registers, but leave the CONFIGFLAG register as is and will not affect the PORTSC registers. This is so that you can do a reset of the controller, without affecting the relationship of the ports and the companion controllers.

Since we have not yet set up the controller, and this is our first time for a reset, we should not use the light reset. We need a reset that will reset the entire controller.

The Host Controller Reset (HCRESET) bit in the command register will reset the entire controller including the relationship between the root hub ports and the companion controllers, with all ownership passed to the companion controllers. The controller resets this bit to zero when the reset is complete.

With this in mind, and knowing that after the host reset, all registers are set to their default state, do the reset and check for these default values. The following modified snippet of code comes from the EHCI driver of FYSOS. The driver calls this code expecting a TRUE or FALSE condition on return, returning TRUE if an EHCI controller was detected at this base address. The mdelay() routine simply delays for a given amount of milliseconds (mS).

Listing 7-2: EHCI Host Controller Reset

```
bool det_ehci_root(const bit32u base) {
 bit32u timeout = EHCI_RESET_DELAY;

 // make sure the run/stop bit is clear
 ehci_write_oper_reg(base, EHC_OPS_USBCommand,
       ehci_read_oper_reg(base, EHC_OPS_USBCommand) & ~(1<<0));

 // set the hcreset bit
 ehci_write_oper_reg(base, EHC_OPS_USBCommand, (1<<1));
 while (ehci_read_oper_reg(base, EHC_OPS_USBCommand) & (1<<1)) {
  if (--timeout == 0) {
    putstr("\nEHCI HC did not reset in EHCI_RESET_DELAY mS.");
    return FALSE;
  }
  mdelay(1);  // 1ms
 }

 if ((ehci_read_caps_reg(base, EHC_CAPS_HCCParams) & (1<<2)) &&
  (ehci_read_oper_reg(base, EHC_OPS_USBCommand) != 0x0080B00))
    return FALSE;
 if (!(ehci_read_caps_reg(base, EHC_CAPS_HCCParams) & (1<<2)) &&
  (ehci_read_oper_reg(base, EHC_OPS_USBCommand) != 0x0080000))
    return FALSE;
 if (ehci_read_oper_reg(base, EHC_OPS_USBStatus) != 0x00001000)
    return FALSE;
```

```
 if (ehci_read_oper_reg(base, EHC_OPS_USBInterrupt)
                                    != 0x00000000)
   return FALSE;

 if (ehci_read_oper_reg(base, EHC_OPS_FrameIndex) != 0x00000000)
   return FALSE;
 if (ehci_read_oper_reg(base, EHC_OPS_CtrlDSSegemnt)
                                    != 0x00000000)
   return FALSE;
 if (ehci_read_oper_reg(base, EHC_OPS_ConfigFlag) != 0x00000000)
   return FALSE;

 return TRUE;
}
```

I have yet to find a resource that states how long the controller will take to reset. One of my test machine's controllers takes less than a microsecond, while another takes 3.4mS. I would assume that if the controller has not reset after 20mS (EHCI_RESET_DELAY in Listing 7-2), it is not a valid controller, and should return FALSE.

The code in Listing 7-2 checks for the following items to be true. If any of them are not true, the function returns FALSE to the caller.

- The Interrupt Threshold Control field in the command register is 08h, and if the Asynchronous Park Capability bit in the HCCPARAMS register is set, the park enable is 1 and the park count is 3. If it is clear, the park enable is zero and the park count is zero.
- The HcHalted bit in the USBSTS register is set.
- The Interrupt Enable register is zero.
- The Frame Index register is zero.
- The Control Data Structure Segment register is zero.
- The Configure Flag register is zero.

If any of the items listed above are false, then we did not find a valid controller, and the function returns FALSE.

Set up the EHCI controller

Now that we know we have an EHCI controller, we can start to set up the controller and our stack and schedule.

The controller has already been reset by the detection routine in Listing 7-2, however there are a few more items we need to set up before we have a working USB stack schedule.

First, set the interrupt enable register. We will set the first three bits so that the controller will interrupt on a short packet and completion of a frame, an error with transaction, and a port change interrupt.

```
ehci_write_oper_reg(base, EHC_OPS_USBInterrupt, 0x07);
```

Now make sure the frame number register is zero.

```
ehci_write_oper_reg(base, EHC_OPS_FrameIndex, 0);
```

Now set the frame list base address for both the periodic list and the asynchronous list. For current purposes we will assume that we already have a valid stack frame at address PERIODIC_BASE, and a valid asynchronous list at ASYNC_LIST. The FYSOS code actually calls a function to allocate and initialize the stack frame, returning the address. I will discuss later how to create a valid EHCI stack frame and its use.

```
ehci_write_oper_reg(base, EHC_OPS_PeriodicListBase,
                                    PERIODIC_BASE);
ehci_write_oper_reg(base, EHC_OPS_AsyncListBase, ASYNC_LIST);
```

If the controller is capable of using 64-bit addresses, we need to set the high dword portion of the address in the CTRLDSSEGMENT register. For the sake of this example, we use addresses below the 4gig mark, so write zero to the register.

```
ehci_write_oper_reg(base, EHC_OPS_CtrlDSSegemnt, 0x00000000);
```

Setting up and handling the bits in the Port Status/Command register(s) is enough for a new section, therefore, see the next section on the setting of the port register and detecting the device status.

You are probably wondering why I didn't do the Status and Command registers first. The Command register has the Run/Stop bit and we have to have a valid stack frame and controller state before we can set the schedule to *Run*. Also, the status register should be cleared just before you start the schedule so that you don't have any residue bits from a previous schedule. Remember that the status register is Write/Clear. To clear the register, write 1's to each bit that should be cleared. I write 0x0000003F to the register, so that I get all the bits.

```
ehci_write_oper_reg(base, EHC_OPS_USBStatus, 0x0000003F);
```

At this point, we have set up all of the controller's registers but the Command register and the Configuration register. We need to make sure that the interrupt threshold is set to 8 micro-frames. This means that the controller will not fire an interrupt until 8 micro frames have passed. This means that the controller, if needed, can fire an interrupt no more than once a frame, or once a millisecond. We also need to enable the asynchronous and periodic lists, make sure the frame size field is set for 1,024 elements and set the run bit.

```
ehci_write_oper_reg(base, EHC_OPS_USBCommand, 0x00080031);
```

Finally, we set bit 0 in the configure register to tell the controller to take ownership of all root hub ports.

```
ehci_write_oper_reg(base, EHC_OPS_ConfigFlag, 1);
```

As for the EHCI hardware, we have completely set it up. The next thing to do is to start sending and receiving packets from the schedule. After a discussion on detecting device status and other device related items, I will get back to this topic on how to send and receive packets to and from the schedule. For now, let us move on to the Port Status/Control register and device detection.

The EHCI Port Status/Control Register

Before we can do anything with the port, we must make sure it has power. If the Port Power Control bit in the HCSPARAMS register is set to 1, meaning all ports have power switching and can be powered by a bit in the port, we will need to set the Port Power bit in the port register. If the Port Power Control bit is clear, then all ports are hardwired to power on. The controller will not report the connection status, line status, or any other status unless there is power to the port. After we set the Port Power bit from a 0 to a 1, we must wait no less than 20mS for the port to have power, and before we read from it.

When the user plugs or unplugs a device into/from one of the root hub's ports, that port's status bits are set or cleared depending on the current status.

> At bootup, if a device is already attached (plugged in), the status of this register is identical to if you had just attached the device during normal operation, even though the device was attached before the machine was turned on.

The EHCI specification says that the default value for a root hub port is 0x00002000 or 0x00003000 depending on the power control of the port. I don't necessarily agree. The default value is dependent on many items. For example, if the Port Power Control bit in the HCSPARAMS register is set to 1, then by default the power to that port is off. The Line Status bits in the port are undefined if the power is off. If the power is on, these two bits could be one of four values if a device is attached, undefined if a device is not attached.

Which brings us to the Current Connect Status and Connect Status Change bits. If a device is attached, these two bits will reflect that.

> Please see Appendix D, "Port Connection Status Change" for an important note about connect status changes.

Another bit that will be dependent of its default value is the Port Owner bit. If your driver has set the Configured flag, then this bit will be zero otherwise it will be 1.

If the Current Connect Status bit is set we have a device attached. Next task would be to reset and enabled it so that we can start the enumeration process.

Resetting and Enabling the Port

Before we reset the device, remember that the EHCI controller can only communicate with high-speed devices. If we know that the attached device is a low-speed device, there is no need to perform an EHCI reset. Simply pass ownership of the port to the companion controller by setting the Port Owner bit. The companion controller will see the Connect Status Change bit, and then its driver should act accordingly.

To see if the attached device is a low-speed device, read the Line Status bits, making sure we have powered the port as explained before. If the value returned is 01b, we have a low-speed device. If the value is not 01b, we may have a full- or high-speed device, so perform the EHCI reset on this port.

Before resetting the port, the HcHalted bit in the USBSTS register must be zero. Since you cannot write a zero to this bit, you must start the schedule by setting the Run/Stop bit in the USBCMD register. Make sure that the schedule enable bits are clear if you have not yet set up a valid schedule.

To reset the port, set the Port Reset bit and clear the Port Enabled/Disabled bit at the same time. The port itself only needs 10mS to reset. However, since this is a root hub port, the USB specification says that we must send the reset signal for a minimum of 50mS. This doesn't have to be a continuous 50mS, though if it is not, it must be a consecutive of no less than 5 sets of no less than 10mS each, with no more than 3mS in between sets.

In other words, you may reset the port 5 times, using a reset duration of no less than 10mS each, consecutively with no more than 3mS between loops.

An EHCI driver may use the snippet of code below to reset the port.

Listing 7-3: OHCI Port Reset

```
ehci_write_oper_reg(root->base, HCPortStatusOff, (1<<8));

// wait for the bit to be set (note: may never exit!)
while (!(ehci_read_oper_reg(root->base, HCPortStatusOff)
                                        & (1<<8)))
  ;

mdelay(USB_TDRSTR);  // at least 50 ms for a root hub
ehci_write_oper_reg(root->base, HCPortStatusOff, 0);

// wait for bit to actually be cleared (note: may never exit!)
while (ehci_read_oper_reg(root->base, HCPortStatusOff)
                                        & (1<<8))
  ;

mdelay(USB_TRSTRCY); // reset recovery time
```

After the reset we can read the Port Enabled/Disabled bit. If it is set, then the attached device is a high-speed device. If it is cleared, then it is a full-speed device and we need to release ownership of the port by setting the Port Owner bit. Your driver cannot enable a port by writing a 1 to the Port Enabled/Disabled bit. The controller will only set this bit if a successful reset, a device is attached, and that device is a high-speed device.

Detecting multiple ports on a root hub

Fortunately, Intel learned there lesson and gave the EHCI a register to hold the count of root hub ports, so that your driver does not have to detect the count like the UHCI controller.

To get the count of downstream root hub ports, simply read the N_PORTS field from the HCSPARAMS register, by reading the dword and masking off all but the least significant nibble.

In the next chapter, I will explain how to send and receive packets on the EHCI controller.

Chapter 8 – The EHCI Stack

By this chapter, you should know how to detect the EHCI host controller through the PCI configuration space and physical detection at the port I/O level. You should also know how to detect that a device is present on one of the ports on the host controller, also known as the root hub. You should also now know how to point the root hub to our stack and start the schedule.

If you are comfortable with this up to this point, we will start on the actual USB communication protocol and how to detect the device or devices already attached. If you are not ready to continue on, please go back and study the previous chapters.

The EHCI Stack

The EHCI stack combines some of the features of the UHCI and some of the features of the OHCI. Like the OHCI, the EHCI uses an asynchronous linked list of queues for control and bulk transfers, instead of a list of frame pointers like the UHCI. However, the EHCI does use a frame list similar to the UHCI for a periodic list.

The Periodic List is used for all isochronous and interrupt transfers, while the Asynchronous List is used for all control and bulk transfers. The Periodic List is similar to the UHCI frame list, it has a number of frame list pointers, called Frame List Element Pointers, and point to four types of items that include Isochronous Transfer Descriptors, Queue Heads, Split Transaction Isochronous Transfer Descriptors, and Frame Span Traversal Nodes. The Asynchronous List is used for all control and bulk transfers and is simply a round-robin list of Queue Heads.

The host controller's scheduler reads the current frame pointer, pointed to by the Frame Index register and executes the frame that it points to. At 1mS, the host controller stops executing that frame, increments the Frame Index register, wrapping to 0 at the last entry, and starts executing the next frame. This means, if we have a frame size of 1,024 entries, each frame will be executed approximately once a second, every 1.024 seconds to be exact, or at 1 Hz. The current frame entry to read is calculated by taking the PERIODICLISTBASE register, using the CTRLDSSEGMENT register for 64-bit addressing, and add the FRINDEX dword index.

These frame list elements have the format below with the size of the list indicated by the Frame List Size field in the USBCMD register.

Table 8-1: Frame List Link Pointer

bits 31:5	4:3	2:1	0
Frame List Link Pointer	0	Type	T

The Frame List Link Pointer is the high 27 bits of a 32-bit physical address pointer to one of the four types mentioned. Since bits 4:0 are used for other items, the physical pointer must be 32-byte aligned. To read the address from the field, simply read the whole 32-bit dword and mask off the bottom 5 bits. To write to the field, make sure the address is 32-byte aligned, setting the lower 5 bits according to the fields defined below, then write the whole 32-bit dword. Bits 4:3 should always be zero.

Please note that even though most data structures in the EHCI are 32-byte aligned, page 31, section 3 in the EHCI specification states that any data structure that can be read or written by the controller must not cross a 4k boundary. Since some data structures are larger than 32 bytes, you must be sure that they do not cross a 4k boundary. See the first of Chapter 10 for an example of memory alignment and boundary crossing.

The Type field is a 2-bit field indicating which type of item the list pointer is pointing to.

Table 8-2: EHCI: Frame List Item Type

Type Field Value Definitions	
00b	Isochronous Transfer Descriptor (iTD).
01b	Queue Head (QH).
10b	Split Transaction Isochronous Transfer Descriptor (siTD).
11b	Frame Span Traversal Node (FSTN).

The T bit is the same as the UHCI controller, if set, the pointer is not valid and the controller should continue with the next frame list entry at next Start of Frame.

Queue Heads and Transfer Descriptors will explained shortly, with Isochronous Transfers explained in Appendix D and Part 5.

The Asynchronous Linked List is a linked list of queues. Since the operational register ASYNCLISTADDR is a pointer to the next queue head, this implies that the list is simply a round-robin type list. In other words, the asynchronous list is simply a number of Queue Head's in a linked list, with the last queue head pointing to the first queue head. Each queue head may point to any of the types identified in Table 8-2 above.

Since the controller starts with the Periodic List first, when either the Periodic List is completed, empty, or disabled, the controller moves on to the Asynchronous List. Therefore, you should not place any isochronous or periodic transfers in the Asynchronous List.

Transfer Descriptors

There are two different types of Transfer Descriptors. The Isochronous Transfer Descriptor is used for audio, video, and other continuous data transfers. I will discuss these types of transfers in Appendix D and Part 5 of this book.
For now, we will discuss the Queue Element Transfer Descriptor, which allows control and bulk transfers. Any reference to a transfer descriptor in this chapter is considered a reference to a Queue Element Transfer Descriptor.

A Queue Element Transfer Descriptor, further known as a transfer descriptor, is where the real data transfer takes place and can only be used with a Queue Head as an element of the Queue Head, hence the name. In other words, after completion of a transfer descriptor, the completed and updated transfer descriptor is overlaid on to the Queue Head. After the description of a transfer descriptor below, I will show you what this means.

A transfer descriptor can transfer up to 20,480 bytes using five buffers of 4,096 bytes each. These buffers can be a continuous amount of memory occupying a physically continuous 20,480 bytes or spread among five different buffers. Each buffer used, not counting the first, must start on a physical page (4k) and occupy the whole page unless it is the last buffer. I will provide an example once I define the elements of the transfer descriptor.

A transfer descriptor is a 32-byte block of memory with the format listed below. Please note that I added the last five elements for use with 64-bit addressing.

Table 8-3: EHCI: Queue Head Transfer Descriptor

bits 31:5						4:1	0
Next qTD Pointer						0	T
Alternate Next qTD Pointer						0	T
31	30:16	15	14:12	11:10	9:8	7:0	
dt	Total Bytes to Transfer	IOC	C_Page	C_ERR	PID	Status	
31:12				11:0			
Buffer Pointer (Page 0)				Current Offset			
Buffer Pointer (Page 1)				Reserved			
Buffer Pointer (Page 2)				Reserved			
Buffer Pointer (Page 3)				Reserved			
Buffer Pointer (Page 4)				Reserved			
High 32-bits of 64-bit Buffer Pointer 0							
High 32-bits of 64-bit Buffer Pointer 1							
High 32-bits of 64-bit Buffer Pointer 2							
High 32-bits of 64-bit Buffer Pointer 3							
High 32-bits of 64-bit Buffer Pointer 4							

When the controller is finished executing this transfer descriptor, it will modify some of the areas mark reserved along with a few other areas, then insert, or overlay the modified transfer descriptor on to the Queue Head. Please see the Queue Head description later in this chapter for which parts are modified.

Table 8-4: EHCI: Transfer Descriptor

Transfer Descriptor Definition		
Name	HC Access	Description
T	RO	**Terminate**. If set to 1, this tells the controller that this transfer descriptor is the last in the

		linked list, and that the Next qTD Pointer field is not valid.
Next qTD Pointer	RO	**Next qTD Pointer**. This is bits 31:5 of the link pointer, pointing to the next transfer descriptor in the list. Bits 4:0 are assumed zero. To get this field, read the whole dword and mask off the lower five bits.
Alt Next qTD Pointer	RO	**Alternate Next qTD Pointer**. This is bits 31:5 of the link pointer, pointing to the next transfer descriptor in the list after a short packet detected. Bits 4:0 are assumed zero.
dt	R/W	**dt**. This is the Data Toggle bit. The use of this bit is determined by the Data Toggle Control bit in the Queue Head. The data toggle is used to make sure that the controller receives the data in the correct sequence. In a control packet, the first transfer descriptor (SETUP), set this bit to 0, in the second (IN/OUT), set this bit to 1, in the third, set this bit to 0, and so on. A control transfer will always have the last transfer descriptor's data toggle bit set. On a bulk transfer, each packet's toggle bit will be set to the opposite of the one before.
Total Bytes to Trans-fer	R/W	**Total Bytes to Transfer**. This field is set by software to indicate how many bytes to transfer, with a max setting of 0x5000. This field is decremented by the controller for every byte that is transferred.
IOC	RO	**Interrupt on Complete**. This bit tells the controller to fire an interrupt at the next interrupt threshold.
C_Page	R/W	**Current Page**. This field indicates which page entry is currently being used. Software should normally set this field to zero.
C_ERR	R/W	**Error Counter**. Your driver sets this field to 11b. When an error is detected, the controller will decrement this field and try again. If zero is reached, it will fail the transaction and set the corresponding status bits. This field can be set to 11b, 10b, 01b, or 00b. However, the controller will continue trying to execute the transaction indefinitely with an error count of 00b.
PID	RO	**Packet Identification Code**. This tells the controller what kind of packet this is, IN (01b), OUT (00b), or SETUP (10b).
Status	R/W	

Bit	Description
7	**Active**. Set to 1 to enable execution of this transfer descriptor. The controller will clear this bit after execution of this transfer descriptor.
6	**Halted**. Set to 1 by the controller if a serious error is encountered.
5	**Data Buffer Error**. Set to 1 by the

			controller if an overrun or underrun is encountered.
		4	**Babble Detected**. Set to 1 by the controller if a babble was detected.
		3	**Transaction Error (XactErr)**. Set to 1 by the controller if the host did not receive a valid response from the device.
		2	**Missed Micro-Frame**. Used for low- and full-speed endpoints.
		1	**Split Transaction State (SplitXstate)**. Used for low- and full-speed endpoints.*
		0	**Ping State**. If the transfer is a high-speed OUT transfer, then this field is zero for an OUT, or 1 for a PING PID. If the transfer is a non high-speed OUT transfer, this bit is set if an error occurred.
Current Offset	R/W	**Current Offset**. This field is the current offset within the Current Page (starting page) of the first byte to transfer.	
Buffer Pointer page X	RO	**Buffer Pointer (Page X)**. The physical Page address of the buffer to transfer. See the text on the next page for more details on these two fields.	
64-bit Pointer	RO	**64-bit Buffer Pointer (Page X)**. The high 32-bits of the physical address of the buffer to transfer.	

The Next qTD Pointer is used to point to the next transfer descriptor within this Queue Transfer. It should be a physical address aligned on a 32-byte boundary with the T field cleared. If the T field is set, the Next qTD Pointer field is ignored and this is the last transfer descriptor to processes within this queue.

The Alternate Next qTD Pointer is used when a short packet is detected. This is where Intel got their act together. With the UHCI, if a short packet was detected, it was up to the software driver to make sure the correct transfer descriptor was executed next. Now, with the EHCI controller, your driver can include a pointer to the transfer descriptor, usually the STATUS descriptor, and the controller will stop executing the transfer descriptors in the Next qTD Pointer list and go directly to the STATUS transfer descriptor. Again, this pointer must be a physical address aligned on a 32-byte boundary and is ignored if the T bit in this dword is set.

The Total Bytes to Transfer field is set to the amount of bytes you would like to transfer. Unlike the UHCI, you set this field to 1 for 1 byte, 2 for 2 bytes, etc. The controller will decrement this field for every byte transferred. If this field is non-zero after completion, the actual amount of bytes transferred is the original amount minus the current field value.

Your driver uses the C_Page, Buffer Pointer, and Current Offset fields to indicate to the controller where the buffers are to make the transfer.

You can use up to five different buffers each no more than 4,096 bytes in length. Each buffer, not including the first, must start on a page boundary. The controller uses the C_Page field to know which Buffer Pointer to use as the Page address, and uses the Current Offset field as an offset within that page.

A typical initialization would be to set the C_Page field to zero, write a physical address to the first dword used for the first Buffer Pointer, and write the physical page address of the next

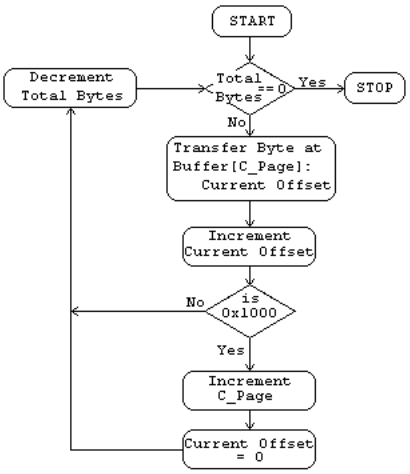
pointers. Here is pseudo-code example:

```
C_Page = 0
qTD.dword0x0C = physical address of first byte in buffer
qTD.dword0x10 = (qTD.dword0x0C & ~0xFFF) + 0x1000
qTD.dword0x14 = qTD.dword0x10 + 0x1000
qTD.dword0x18 = qTD.dword0x14 + 0x1000
qTD.dword0x1C = qTD.dword0x18 + 0x1000
```

If you will notice, a 32-bit write to dword 0x0C of the physical address of the buffer will write the page value and current offset as one write. A physical address is the page address plus the offset within that page.

The outline below explains what the controller might do with these fields as it executes the transfer. I say, "might do" since different controller manufacturers may have different techniques as long as the results are the same.

Figure 8-1: EHCI: Buffer Address



If you will notice in the outline above, the first byte of the first buffer may be byte-aligned. However, every page after must be page aligned and occupy the whole page.

You may use a single buffer of 0x5000 bytes that is physically continuous. In these examples, the second page offset will simply be the next page from the previous page offset.

As mentioned before, each transfer descriptor can transfer 20,480 bytes, which is 0x5000, or 5 pages of data in one transfer. However, since there are only 5 page entries, if the start of your buffer isn't page aligned, then you can not transfer a total of 20,480 bytes in this transfer descriptor.

For an example, let's say that the first byte of your buffer is at 0x51A3E823. The controller would start with the first page buffer (C_Page = 0) which would be 0x51A3E000, with the Current Offset field containing 0x0823. As soon as 0x7DD bytes were transferred (0x1000 minus 0x823 = 0x7DD), the C_Page index would increment, the Current Offset field would roll over to zero, and the controller would start to use the next Buffer Pointer (Page entry). If the controller can only transfer 4,096 bytes per buffer pointer entry, this will only leave an allowed transfer of 18,397 bytes, not 20,480 bytes (0x7DD + (4 * 0x1000)).

Therefore, make sure that you do not place in the Total Bytes to Transfer field a value that is more than the allotted buffer you allocate for the transfer.

Queue Heads

The EHCI Queue Head is where the device information is stored for transfer descriptors, such things as maximum packet size, device speed, endpoint, and device address. Each transfer descriptor linked with a queue head uses the same device data stored in the queue head.

As noted earlier, after the completion of a transfer descriptor, the controller copies the resulting transfer descriptor to the correct location in the queue head, or overlays it on the queue head. This is so that the controller can return to the correct place and so that your driver can view the last transfer descriptor processed whether there was a successful transfer or an error with one of the transfer descriptors.

Before execution, a Queue Head needs only four dwords of memory. However, for the transfer descriptor overlay, you will need enough room for the four dwords, plus the size of the overlay. Therefore, make sure that your schedule of queue heads is far enough apart in memory, that the overlay does not overwrite the next queue head. The FYSOS driver allocates 96 bytes, or 24 dwords for the EHCI Queue Heads. That is 4 for the Queue Head, 13 for the overlay, 1 for queue insertion, and 6 more for future use.

An EHCI Queue Head has the format outlined below. Please note that I have included the Transfer Descriptor overlay with the included fields for transfer descriptor completion status in bold.

Table 8-5: EHCI: Queue Head

bits 31:5	4:3	2:1	0
Queue Head Horizontal Link Pointer	0	Typ	T

31:28	27	26:16	15	14	13:12	11:8	7	6:0
RL	C	Maximum Packet	H	dtc	EPS	EndPt	I	Addr

31:30	29:23	22:16	15:8	7:0
Mult	Port Number*	Hub Addr*	µFrame C-Mask*	µFrame S-mask

bits 31:5	4:0
Current qTD Pointer	0

bits 31:5	4:1	0
Next qTD Pointer	0	T
Alternate Next qTD Pointer	**NakCnt**	T

31	30:16	15	14:12	11:10	9:8	7:0
dt	Total Bytes to Transfer	IOC	C_Page	C_ERR	PID	Status

31:12	11:0
Buffer Pointer (Page 0)	Current Offset
Buffer Pointer (Page 1)	Reserved / **C-prog-mask***
Buffer Pointer (Page 2)	**S-bytes*** / **FrameTag***
Buffer Pointer (Page 3)	Reserved
Buffer Pointer (Page 4)	Reserved
High 32-bits of 64-bit Buffer Pointer 0	
High 32-bits of 64-bit Buffer Pointer 1	
High 32-bits of 64-bit Buffer Pointer 2	
High 32-bits of 64-bit Buffer Pointer 3	
High 32-bits of 64-bit Buffer Pointer 4	

*These fields are used exclusively to support split transactions to USB 2.0 Hubs

When the controller is finished executing the current transfer descriptor linked by this queue head, it will overlay the resulting transfer descriptor as indicated before. The table below defines the layout of a Queue Head without the transfer descriptor overlay.

Table 8-6: EHCI: Queue Head Definition

Queue Head Definition		
Name	HC Access	Description
T	RO	**Terminate**. If set to 1, this tells the controller that this transfer descriptor is the last in the linked list, and that the Next qTD Pointer field is not valid.
Typ	RO	**Type**. These two bits indicate to the controller what type of item the Link Pointer points to. See Table 8-2 on page 8-2.
Queue Head Horz Link Pointer	RO	**Queue Head Horizontal Link Pointer**. This is bits 31:5 of the link pointer, pointing to the next item in the list. Bits 4:0 are assumed zero. Usually, as with the Asynchronous List, each Queue Head Horizontal Pointer points to another Queue Head with the last in the list pointing to the first.
Addr	RO	**Device Address**. This is the address given to the device at enumeration time.
I	RO	**Inactivate on Next Transaction**. This field, when set, tells the controller to set the Active bit to zero if the queue head is in the Periodic schedule and is a low- or full-speed endpoint.
EndPt	RO	**Endpoint**. This is the endpoint number of the current function. If this is a Control transfer, this would be set to zero.
EPS	RO	**Endpoint Speed**. This indicates to the controller the speed of the endpoint. 00b = Full Speed (12Mbs) 01b = Low Speed (1.5Mbs) 10b = High Speed (480 Mbs) 11b = reserved
dtc	RO	**Data Toggle Control**. This bit indicates where the controller should get the data toggle bit. If set, it should get the data toggle from the incoming transfer descriptor. If clear it should get the data toggle from the overlay in the queue head from the previous transfer descriptor.
H	RO	**Head of Reclamation List Flag**. This bit indicates to the controller that this is the head or first queue head of the reclamation list.
Max Packet Length	RO	**Max Packet Length**. This indicates to the controller the maximum packet size in bytes the endpoint can transfer.
C	RO	**Control Endpoint Flag**. When set, this indicates to the controller that this is a non high-speed control transfer.
RL	RO	**Nak Count Reload**. Used by the controller to

		reload the Nak Counter Field.
µFrame S-mask	RO	**Interrupt Schedule Mask**. This field should be zero when in the Asynchronous schedule, and non-zero when in the Periodic schedule as an interrupt endpoint. This is a bit mask to indicate to the controller on which micro-frame to execute.
µFrame C-Mask	RO	**Split Completion Mask**. Used when in the Periodic schedule and is a low- or full-speed endpoint.
Hub Addr	RO	**Hub Address**. Used when the endpoint is a low- or full-speed device on an external hub in a split transaction.
Port Num	RO	**Port Number**. Used when the endpoint is a low- or full-speed device on an external hub in a split transaction.
Mult	RO	**High-Bandwidth Pipe Multiplier**. Set to a 1, 2, or 3, to indicate to the controller how many transactions to be issued for this endpoint per micro-frame.

You may be asking yourself why there are references to low- and full-speed devices in these high-speed Transfer Descriptors since the EHCI controller will pass all low- and full-speed control to the companion controllers. This is because there could be a high-speed external hub attached to the root hub's port, which means that the EHCI controller will retain control. However, there could be a low- or full-speed device attached to one of the external hub's ports and you are sending packets to one of those devices. For the purpose of this book, I will assume that all packets are sent to the device attached to the root hub's port, and this device is not an external hub. External hubs are discussed in a later chapter.

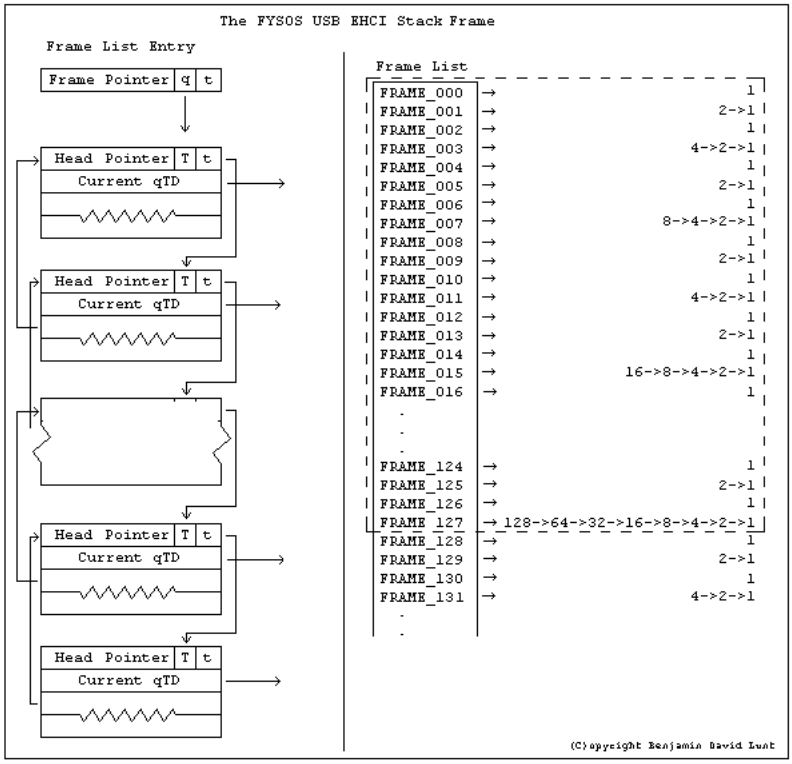
Controller Schedule

As mentioned before, the controller will start with the Periodic Schedule. Once this schedule is complete, due to an empty schedule for example, the controller will move to the Asynchronous Schedule. To find out where to begin in this schedule, it reads the ASYNCLISTADDR register to find the current queue head to execute. Since this schedule is a round-robin type schedule, the current queue head pointer should always be valid. The controller will run the Asynchronous Schedule until the time of end of frame.

The FYSOS Periodic Stack Frame

The figure below is a representation of the Periodic Stack Frame that the FYSOS USB driver uses for each EHCI controller's Periodic Schedule. It is very similar to the UHCI's periodic scheduler.

Figure 8-2: EHCI Stack Frame



We have 8 queues that will always be present in our stack and for the purpose of this illustration, we will name them starting with the top queue, Q128, Q64, Q32, Q16, Q8, Q4, Q2, and Q1. Using this naming convention, Q128 will be executed once every 128mS, Q64 will be executed once every 64mS, and so forth, with Q1 being executed every frame, or each millisecond. Each queue points to the next queue in the stack. Q128 points to Q64, points to Q32, and so on, with Q1's pointer set as invalid (T = 1).

Using the illustration on the previous page, notice that the first frame points to the queue named Q1. Therefore, when the controller executes the first frame, Frame List Pointer zero, only anything queued in Q1 will be executed. When the controller moves to the next frame, Frame List Pointer 1 points to the queue named Q2. Only anything queued in Q2 and Q1, since Q2's Head Pointer points to Q1, and below will be executed.

When the controller moves to the third frame, again, only Q1 will be executed. However, when the controller moves to the forth frame, anything in the Q4 queue will now be executed along with Q2, and Q1. Do you see the idea here? Place a transfer descriptor in one of the queues above to only execute at a specific time interval. For example, a device may ask the driver to query a response every 16mS. Rather than having to write up a timer and lengthy code to poll the CPU clock, all we need to do is place a transfer descriptor in the Q16 queue. That transfer descriptor will be executed every 16mS. No clock to worry about and no polling of hardware. Once the device's transfer descriptor is executed, your driver can remove the old transfer descriptor, insert a new transfer descriptor in the same queue head, Q16, and it will be executed again at 16mS.

Notice that there is a broken lined box around the first 128 frame pointers in the figure on the previous page. Since we don't need time intervals larger than 128mS, the schedule starts over again. With a schedule size of 1024, this gives eight individual schedules of 128mS intervals. This type of scheduling in called Interrupt Scheduling.

The 8 queues, labeled Q128 to Q1 as above never get removed from the periodic list. Each queue remains and when the list is empty, doesn't point to any other queue. For example, if the list is empty, Q1's T bit is set, meaning this is the last queue in this frame.

As stated earlier, one of the dwords in the scratch area of an FYSOS Queue Head buffer, is used for inserting queues into a linked list. We use these 4 bytes after each queue to allow insertion and removal of an unlimited amount of queues in each main queue in the schedule. Each of these 8 queues is called Queue Heads and uses a linked list to add or remove other queues. To insert a queue, simply find the last queue in the desired queue head and point its queue head pointer to the address of your queue, and point the previous pointer member of your new queue to the queue head pointer of the last queue found.

To remove a queue, simply use the previous pointer member in the desired queue to be removed to find the previous queue in this queue chain. Then place the head pointer value of your queue in the found previous queue's head pointer member. Now using that same head pointer value, you can find the address of the next queue in the chain and write the previous queues address in the previous pointer member.

Examples of inserting and removing queues in a queue chain are listed below.

Listing 8-1: EHCI Queue Insertion

```
// This inserts a queue into the specified queue for the skeleton
// type = 0 = Q128, 1 = Q64, etc.
bool insert(struct QUEUE_HEAD *queue, const bit8u type) {
  struct QUEUE_HEAD *cur_queue = (struct QUEUE_HEAD *) SKELETON;
  cur_queue = &cur_queue[type];
  while ((cur_queue->horz_ptr & QUEUE_HEAD_T) == 0)
    cur_queue = (struct QUEUE_HEAD *)
                (cur_queue->horz_ptr & QUEUE_HEAD_PTR_MASK);
  cur_queue->horz_ptr = (bit32u) queue | QUEUE_HEAD_Q;
  queue->horz_ptr = QUEUE_HEAD_T;
  queue->last_ptr = (bit32u) cur_queue;

  return TRUE;
}
```

Listing 8-2: EHCI Queue Removal

```
// removes a queue from the skeleton
void remove_frame(struct QUEUE_HEAD *queue) {
  ((struct QUEUE_HEAD *) queue->last_ptr)->horz_ptr =
                                    queue->horz_ptr;
  if ((queue->horz_ptr & QUEUE_HEAD_T) == 0)
    ((struct QUEUE_HEAD *) (queue->horz_ptr & ~
            QUEUE_HEAD_PTR_MASK))->last_ptr = queue->last_ptr;
}
```

To make sure that you do not interrupt the controller, you must make sure that the controller isn't currently executing any of the queues in this queue head chain. There are a few ways to do this. One way is to stop the controller, and then start it again, but that will mess up the timing of the interrupt schedule.

Inserting and removing queues from the Asynchronous Schedule is handled quite differently and explained in Chapter 13.

To initialize the EHCI's periodic schedule, no matter the size of the schedule, always initialize the first 128 entries, then copying these 128 entries to the next 128 entries until the schedule is full. We need the first Frame List entry to point to Q1, the second to Q2, the third to Q1, the fourth to Q4, etc., just as the table describes. To make this a little easier to understand, imagine the following.

- Set each entry to point to Q1. Then,
- Set each 2nd entry to point to Q2. Then,
- Set each 4th entry to point to Q4. Then,
- Set each 8th entry to point to Q8. Then,
- Set each 16th entry to point to Q16. Then,
- Set each 32nd entry to point to Q32. Then,
- Set each 64th entry to point to Q64. And finally,
- Set each 128th entry (the last entry) to point to Q128.

This can easily be done with the following pseudo code, where SIZE_OF_LIST is set to 128, 256, 512, or 1024.

Listing 8-3: EHCI Schedule Creation

```
        k = 0;
        for (j = 1; j < 128; j <<= 1)
                for (i = (j - 1); i < 128; i += j)
                        for (l = 0; l < SIZE_OF_LIST; l += 128)
                                frame_pointer[l + i] = address of Q[1 << k]
                                frame_pointer[l + i] bit 0 = 1;
    endfor
  endfor
                k++
endfor
```

In the pseudo code above, if counter 'k' equals zero, then the address of Q[1 << 0] is the address of Q1. We also set bit zero of each pointer to 1 since the schedule is empty.

I am sure there are other numerous ways to get the same results, though the above seems to explain it a bit better.

The FYSOS Asynchronous Stack Frame

The FYSOS's asynchronous stack frame is simply 16 Queue Heads linked together to form a round robin type list.

Listing 8-4: EHCI Async Schedule Creation

```
for (i = 0; i < 16; i++)
                    async_queue_head[i] = address of async_queue_head[i+1]
endfor
async_queue_head[15] = address of async_queue_head[0]
async_queue_head[0] bit H = 1;
```

Each queue head points to the next queue head with the last queue head pointing to the first queue head. The first queue head also has bit H set, indicating that it is the Head of the full-speed reclamation list.

Inserting and removing queues from the Asynchronous Schedule is explained in Chapter 13.

How the EHCI handles the Stack

When the EHCI registers are set up to the desired state and the driver sets the run bit in the COMMAND register, the controller retrieves the value from the Periodic List Base register and starts processing the Periodic or Interrupt List. It does this by taking the 32-bit physical address from PERIODICLISTBASE register and retrieving the dword value at the offset of the value in the FRMINDEX register. If it were to be done in the C language, it would look like this:

```
FrameListPointer = *(bit32u *) (frame_address + (frame_num * 4));
```

The controller first checks bit 0 to see if the pointer value is valid. If this pointer is valid, it will load and execute the item or items in this frame.

If the pointer is valid, and once the last item is executed, indicated by bit 0 being set in the Next Link Pointer field, or the Periodic List is disabled, or the Periodic list is empty, the controller will move to the Asynchronous List.

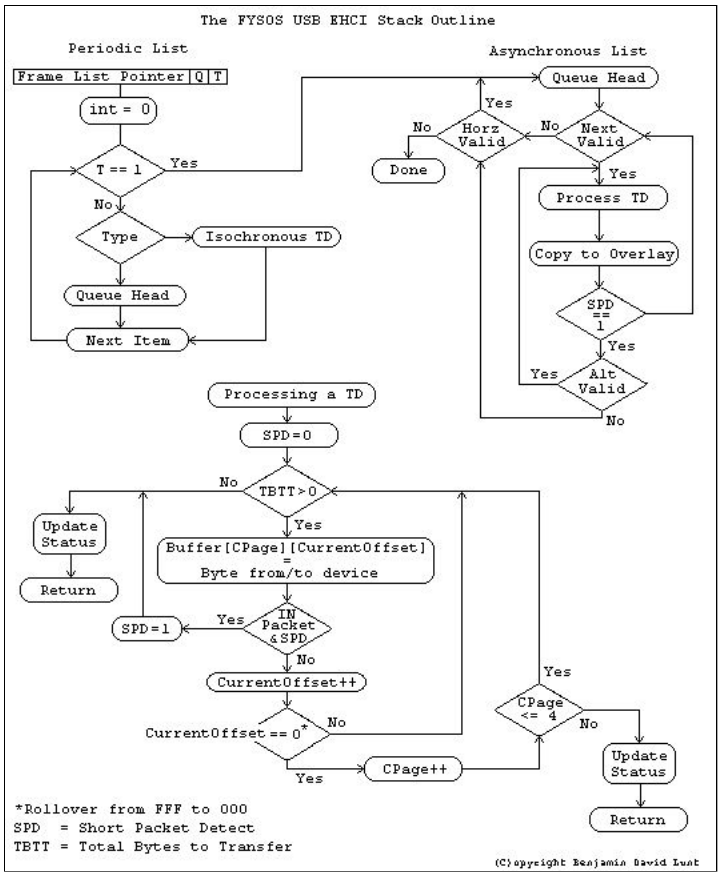
The Asynchronous List is where the Control and Bulk items are executed. This list is a simple round-robin list of queue heads. The controller will execute through this round-robin list until end of frame is reached.

See the outline on the next page for more information on what transpires every millisecond for a single frame. Please note that error checking and checking that enough time is left is not included.

The steps outlined are for a typical controller function. The outline below may or may not be how each controller operates, and is simply to show you a visual way that the controller may process the schedules. For example, the controller may or may not check that CPage is less than or equal to four after the increment.

To continue with the EHCI information in this book, skip to Chapter 13 to start enumerating the root hub. However, if you feel that you have a good grasp on the EHCI hardware, I suggest that you go directly to Chapter 9 and learn the new Super-speed xHCI controller's hardware. Once you have a good grasp on these controllers, then turn to Part 2 of this book to continue with learning how to enumerate the root hub devices with the UHCI, then the OHCI, then the EHCI, then finally the xHCI.

Figure 8-3: The EHCI Stack Outline


The FYSOS USB EHCI Stack Outline
Periodic List
Frame List Pointer
Q
T
int = 0
T == 1
Yes
No
Type
Isochronous TD
Queue Head
Next Item
Asynchronous List
Queue Head
Yes
No
Horz Valid
No
Next Valid
Done
Yes
Process TD
Copy to Overlay
SPD == 1
Yes
Yes
Alt Valid
No
Processing a TD
SPD=0
No
TBTT>0
Yes
Update Status
Return
Buffer[CPage][CurrentOffset] = Byte from/to device
Yes
IN Packet & SPD
SPD=1
No
CurrentOffset++
CurrentOffset == 0*
No
Yes
CPage++
Yes
CPage <= 4
No
Update Status
Return
*Rollover from FFF to 000
SPD = Short Packet Detect
TBTT = Total Bytes to Transfer
(C)opyright Benjamin David Lunt

Chapter 9 – The xHCI Hardware

The following information pertains to revision 1.00 of the Intel xHCI specifications dated 21 May 2010 (see note below) and describes how to set up the xHCI Host controller, a minimal stack frame, and start the controller. You should have read Chapter 2 to know how to get the base I/O address and IRQ number from the PCI and be familiar with port I/O and memory mapped I/O addressing.

The xHCI specification states that the xHCI controller will use Memory Mapped I/O addressing, however this doesn't mean that a controller manufacturer will always conform to this statement. You will need to check bit 0 in the Base I/O address to be sure. Since I have not seen a xHCI controller use anything other than Memory Mapped I/O addressing, I will assume Memory Mapped I/O addressing from here on for the xHCI controller access.

Please note that at time of this writing, most controllers were compatible with version 0.96 of the xHCI specifications. Version 1.0 was released and dated 21 May 2010, and was still mostly compatible with version 0.96. Also, please note that there were a few controllers released with a revision number of 0.95, and it was not 100% compatible with version 0.96. Also note version 1.10 has now been released. I will mention the differences as they arise.

A majority of the information in this section, including most of the tables is readily available from the xHCI specification. I am including this information here only for reference, and to clear up a few things that may be a little confusing. To get the original tables, see Appendix A for information on how to get the specification file.

The xHCI USB controller has two defined sets of I/O registers with the first set starting at the base address acquired in Chapter 2.

Host Controller Capability Registers

The first set is the Host Controller Capability Registers and has the format listed below. All registers are read-only.

Table 9-1: xHCI Capability Registers

The xHCI Controllers Capability Registers		
Offset	Size	Description
00h	1	CAPLENGTH: Total length of these registers
01h	1	Reserved
02h	2	HCIVERSION: Interface Version Number
04h	4	HCSPARAMS1: Structural Parameters 1
08h	4	HCSPARAMS2: Structural Parameters 2
0Ch	4	HCSPARAMS3: Structural Parameters 3
10h	4	HCCPARAMS1: Capability Parameters 1
14h	4	DBOFF: Doorbell Offset
18h	4	RTSOFF: Runtime Register Space Offset
1Ch	4[1]	HCCPARAMS2: Capability Parameters 2
	n	Reserved (CAPLENGTH - 0x20 in bytes)
[1]xHCI Version 1.10 only. Previous versions have the reserved area start at 0x1C.		

These registers specify the limits, restrictions, and capabilities and are used as parameters for the xHCI and are explained below.

Table 9-2: The xHCI CAPLENGTH Register

xHCI controller: CAPLENGTH (Base + 00h)		
Bit	Access	Description
7:0	RO	This byte value is used to find the second set of registers, the Operational Registers.

The Operational Registers, the second set of registers, are located at BaseAddress plus this byte's value. The value of this register is usually 0x20 to round to the next paragraph. A controller on one of my test machines has this value as 0x20, as expected, and uses a total of 8,192 bytes of memory mapped address space.

? Please note that with version 1.10, this value may not be 0x20 since this is the reserved area. However, this doesn't mean that the reserved area can't be zero bytes long.

? In theory, the maximum amount of memory mapped address space that the host controller needs, would be 37,936 bytes. If the Capability Registers took 0x20 (32d), the Operational Registers took 0x400 (1,024d), a maximum of 255 ports using 16 bytes each, and the runtime register space with a maximum of 1,024 interrupters, not counting alignment padding, this gives a maximum size of 32+1024+(255*16)+(32+(1024*32)). (This does not count the space used for the Doorbell Array). However, to get the actual amount that the PCI has reserved for this card, see the "Determine the Address Space Required for a Function" section in Chapter 2. It is always a good idea to get the size from the PCI as described in that chapter.

Table 9-3: Reserved

xHCI controller: reserved (Base + 01h)		
Bit	Access	Description
7:0	RO	Reserved

This byte is reserved for future use.

Table 9-4: The xHCI HCIVERSION Register

xHCI controller: HCIVERSION (Base + 02h)		
Bit	Access	Description
15:0	RO	Interface version (in BCD)

This 16-bit register holds the version number of the interface this controller supports. As of this writing, versions 0x0095, 0x0096, 0x0100, and 0x0110 are supported and are Binary Coded Decimals of 00.95, 00.96, 01.00, and 01.10 respectively.

Table 9-5: The xHCI HCSPARAMS1 Register

xHCI controller: HCSPARAMS1 (Base + 04h)		
Bit	Access	Description
7:0	RO	Number of Device Slots (MaxSlots)
18:8	RO	Number of Interrupters (MaxIntrs)
23:19	RO	Reserved
31:24	RO	Number of Ports (MaxPorts)

The MaxSlots field is the maximum number of device context structures and doorbell array entries the controller can support. Valid values are 1 to 255.

The MaxIntrs field specifies the number of 32 byte Interrupter Register Sets that are used in the Runtime Register Space. Valid values are 1 to 1024.

The MaxPorts field is the number of downstream ports on the root hub. Actually, it is the number of the highest port allowed, though since it is *one* based, this would be the count of ports included. With 8 bits, the maximum count of ports on the root hub is 255.

Please note that this value may not reflect the actual physical count of downstream ports, aka sockets. You must check the Supported Protocol Capability entries in the Extended Capabilities section. For example, if you have a value of 4 in this field, this may mean you have 2 USB 3.0 ports and 2 USB 2.0 ports. However, there are only 2 physical ports allowing both USB 3.0 and USB 2.0 protocols each. See the section on Supported Protocol Capability later in this chapter.

Table 9-6: The xHCI HCSPARAMS2 Register

xHCI controller: HCSPARAMS2 (Base + 08h)		
Bit	Access	Description
3:0	RO	Isochronous Scheduling Threshold (IST)
7:4	RO	Event Ring Segment Table Max (ERST Max)
20:8	RO	Reserved
25:21	RO	Max Scratchpad Buffers (High 5 bits)[1]
26	RO	Scratchpad Restore (SPR)
31:27	RO	Max Scratchpad Buffers (Low 5 bits)
[1]xHCI Version 1.10 only. Previous versions have this field reserved.		

The Isochronous Scheduling Threshold field is used to tell software how far ahead in frames that it can modify a TRB (Transfer Request Block) and still have it executed correctly. For example, this tells the software that you may or may not modify the next TRB item in the frame since the hardware may or may not execute it correctly due to memory caches. It is safe practice to not modify the next TRB after the current TRB.

The Event Ring Segment Table Max field tells software the maximum value supported by the Event Ring Segment Table Base Size registers. The max size allowed is 2 to the power of this field. For example, if this field is 8, the max value supported is 2^8 or 256.

The Scratchpad Restore bit is only used if scratchpad save/restore buffers are supported. If they are supported, and bits 25:21 & 31:27 are non-zero, then this bit tells software that the memory allocated for the scratch pad buffers is to be maintained across power events if this bit is set. If this bit is cleared, the software may free and then reallocate the scratchpad memory buffers between power events.

The Max Scratchpad Buffers fields, when combined, is the amount of memory buffers that software must allocate for use by the controller. If this amount is zero, then no memory is required or allocation needed. The size of these buffers along with their alignment is retrieved from the PageSize register in the Operational Registers. See a later section for more about the Scratchpad Buffers and their use.

Table 9-7: The xHCI HCSPARAMS3 Register

xHCI controller: HCSPARAMS3 (Base + 0Ch)		
Bit	Access	Description
7:0	RO	U1 Device Exit Latency
15:8	RO	Reserved
31:16	RO	U2 Device Exit Latency

These two fields each respectively represent the count of microseconds a root hub will take to transition from the U1 to U0 or U2 to U0 power state. Valid values for bits 7:0 are 0 to 10 microseconds and for bits 31:16, 0 to 2,047 microseconds.

Table 9-8: The xHCI HCCPARAMS1 Register

xHCI controller: HCCPARAMS1 (Base + 10h)		
Bit	Access	Description
0	RO	64-bit Addressing Capability (AC64)
1	RO	BW Negotiation Capability (BNC)
2	RO	Context Size (CSZ)
3	RO	Port Power Control (PPC)
4	RO	Port Indicators (PIND)
5	RO	Light HC Reset Capability (LHRC)
6	RO	Latency Tolerance Messaging Capability (LTC)
7	RO	No Secondary SID Support (NSS)
8	RO	Parse All Event Data (PAE)[1]
9	RO	Stopped - Short Packet Capability (SPC)[1]
10	RO	Stopped EDTLA Capability (SEC)[1]
11	RO	Continuous Frame ID Capability (CFC)[1]
15:12	RO	Maximum Primary Stream Array Size (MaxPSASize)
31:16	RO	xHCI Extended Capabilities Pointer (xECP)
[1]xHCI Version 1.10 only. Previous versions have this reserved.		

The 64-bit Addressing Capability bit tells software that the high order 32-bits of 64-bit addressing is valid (bit set) or is to be ignored (bit cleared). If this bit is set, the controller will use all 64-bits for memory addressing. If this bit is cleared, software must ignore the high order 32-bits of each 64-bit address data structure pointer.

The BW Negotiation Capability bit indicates whether the controller implements Bandwidth Negotiation.

The Context Size bit tells software to use 64-byte (bit set) or 32-byte (bit clear) Context data structures. This bit does not apply to Stream Contexts.

The Port Power Control bit indicates whether each port will always be powered (bit clear) or if each port can be powered on its own (bit set).

The Port Indicators bit indicates controlling of the port indicator lights. If clear, port indicator controlling is not available. See bits 15:14 of the Port Control register for more information.

The Light HC Reset Capability bit indicates whether the Light Host Controller Reset function of the USBCMD register is available.

The Latency Tolerance Messaging Capability bit indicates whether the controller supports Latency Tolerance Messaging (LTM). If this bit is set, LTM is supported.

The No Secondary SID Support bit indicates whether the controller supports Secondary Stream IDs. If this bit is set, Secondary Stream IDs are not supported.

The Parse All Event Data bit indicates that if set, all TRB's are parsed while advancing to the next Transfer Descriptor. This bit is used when a Short Packet is detected.

The Short Packet Capability bit indicates that this controller is capable of issuing a Stopped-Short Packet Completion code.
The Stopped EDTLA Capability bit indicates that this controller implements the Stopped EDTLA field in a Stream Context. This bit must be set for all 1.10 versions.

The Continuous Frame ID Capability bit indicates that this controller is capable of matching the ID's of consecutive ISO transfer descriptors.

The Maximum Primary Stream Array Size (MaxPSASize) field indicates the maximum size the Primary Stream Array is allowed. The value in this field is one less than the power of 2 that is supported. i.e.:

Max Primary Stream Array Size = $2^{MaxPSASize+1}$

Valid values for MaxPSASize are 1 to 15. Therefore, if this field contains 7, then Max Primary Stream Array Size would be 256.

Max Primary Stream Array Size = $2^{7+1} = 2^8 = 256$

The xHCI Extended Capabilities Pointer (xECP) field indicates the existence of the Capabilities List. If this field is non-zero, it is the offset in 32-bit dwords from the I/O Base of the Capability Register Set, to the offset of the Capabilities List. For example, if this field is 0x80, then the first Capability List item would be at BASE + 0x0200. See later for more on the format of this list.

The xECP pointer should never be zero. There must be a Port Protocol List included so that you can determine which port register is of what speed.

Table 9-9: The xHCI Doorbell Offset Register

xHCI controller: DBOFF (Base + 14h)		
Bit	Access	Description
1:0	RO	Reserved

31:2	RO	Doorbell Array Offset

This register defines the offset from the Base of the Capability Registers to the Doorbell Array. Be sure to read the whole 32-bit register clearing bits 1:0 to get the offset. The Doorbell Array Offset field is the high 30 bits of this 32-bit register.

The xHCI specification versions 1.00 and 1.10 may be a little confusing. The register definition states the offset in dwords while the Doorbell Array Offset field states the dword offset. If you read the whole 32-bit dword and clear out the bottom two bits, you will have the byte offset to the array. If you read in the Doorbell Array Offset field, you will have the offset in dwords to the array, just backward to the specification.

Table 9-10: The xHCI Runtime Register Base Offset Register

xHCI controller: RTSOFF (Base + 18h)		
Bit	Access	Description
4:0	RO	Reserved
31:5	RO	Runtime Register Space Offset

This register defines the offset from the Base of the Capability Registers to the Runtime Registers. Be sure to read the whole 32-bit register clearing bits 4:0 to get the offset. The Runtime Register Space Offset field is the high 27 bits of the 32-bit register.

Table 9-11: The xHCI HCCPARAMS2 Register

xHCI controller: HCCPARAMS2[1] (Base + 1Ch)		
Bit	Access	Description
0	RO	U3 Entry Capability (U3C)
1	RO	ConfigEP Command Max Exit Latency too Large (CMC)
2	RO	Force Save Context Capability (FSC)
3	RO	Compliance Transition Capability (CTC)
4	RO	Large ESIT Payload Capability (LEC)
5	RO	Configuration Information Capability (CIC)
31:6	RO	Reserved
[1]xHCI Version 1.10 only.		

The U3 Entry Capability bit indicates whether the root hub ports support port Suspend Completion notification.

The Configure EP Command Max Exit Latency too Large bit indicates whether a Configure End Point command is capable of generating a Max Exit Latency too Large compatibility error. Also used by the CME bit in the USBCommand Register.

The Force Save Context Capability bit indicates that the Save State command shall store any cached Slot, End Point, Stream, or other context to memory.

The Compliance Transition Capability bit indicates that the Root Hub ports support the Compliance Transition Enabled bit.

The Large ESIT Payload Capability bit indicates that the controller supports ESIT Payloads greater than 48k in size.

The Configuration Information Capability bit indicates that the controller supports the extended configuration information of the Configuration Value, Interface Number, and Alternate settings

field in the Input Control Context block.

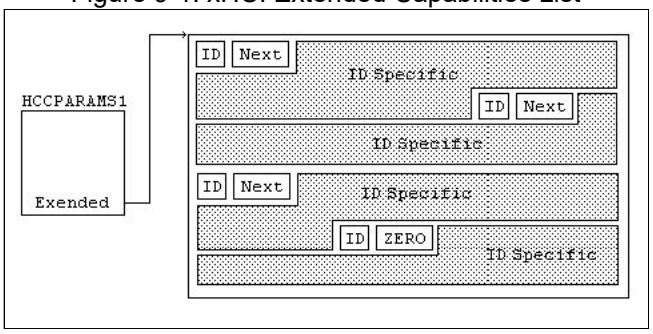
Please note that the HCCPARAMS2 register is only supported by version 1.10 controllers.

Host Controller Extended Capabilities List

The xHCI controller has a list of extended capabilities using a method similar to the PCI extended capabilities. Once you have the offset of this list by using the Extended Capabilities Pointer found in the HCCPARAMS1 register described previously, you read in the 32-bits at this offset. The first 8 bits give an ID of 0 through 255, while the next 8 bits give an offset in dwords, from the current position to the next dword offset. The remaining 16-bits and any bytes before the next offset are ID specific.

The Figure below shows an example of this list. In this example, the HCCPARAMS1 register points to the list with four entries. The first three have 'Next' pointers where as the last has this field as zero, indicating no more entries.

Figure 9-1: xHCI Extended Capabilities List



The following table shows the format of a single entry in this list.

Table 9-12: Format of xHCI Extended Capabilities List Entry

xHCI controller: Base + Current Entry		
Bit	Access	Description
7:0	RO	ID. 0 - 255
15:8	RO	Next Offset in dwords. i.e.: If current List Entry is at offset Base + 0x100 and this field contains 0x10, then the next List Entry would be at Base + 0x100 + (0x10 * 4).
31:16 *n* bytes	Varies	ID capability specific.

If the Next Offset field is zero, then this entry is the last entry. Valid Capability IDs are listed in Table 9-13 below.

Table 9-13: Valid Extended Capabilities List IDs

ID	Name	Description	Size
0	Reserved		
1	USB Legacy Support	Provides the Pre-OS or BIOS support. If present, must be first in list.	8
2	Supported Protocol	Enumerates the protocols and revisions supported by this controller.[1]	12
3	Extended Power Management	Extended Power Management.[1]	n
4	I/O Virtual-ization	Hardware virtualization support.	Up to 1280
5	Message Interrupt	Message Interrupt Support.[2]	n
6	Local Memory	Local Memory Support.	Up to 4TB

7-9	Reserved		
10	USB Debug Capability	Debug Capability Support.	56
11-16	Reserved		
17	Extended Message Interrupt	Extended Message Interrupt Support.[2]	
18-191	Reserved		
192-255	Vendor Specific		

[1]At least one entry with this ID is required.

[2]At least the Message Interrupt or Extended Message Interrupt is required.

xHCI USB Legacy Support

Unlike the older controllers where the Legacy support was in the PCI configuration space, however similar to the EHCI controller, the xHCI controller has it in its own similar Legacy Support configuration space. It is 64-bits in size, or two DWORDs, one right after the other.

If this USB Legacy Support Extended Capabilities List entry is found, it must be the first item in the list and requires another 32-bit dword to follow this dword, called the Legacy Support Control and Status Register.

Tables 9-14 and 9-15 describe the first and second dwords of the legacy support.

Table 9-14: Format of xHCI USB Legacy Support: First DWORD

xHCI USB Legacy Support: First DWORD		
Bit	Access	Description
7:0	RO	ID (0x01)
15:8	RO	Next Offset (**0x04**). Example: Base + xECP + Current (0x00) + (**0x04** * 4)
16	R/W	HC BIOS Owned Semaphore Bit. Default = 0. • BIOS sets this bit to 1 to indicate to system software that the BIOS has ownership of the controller. • It will clear this bit to 0 if system software sets bit 24 below.
23:17	R/W	Reserved and must be preserved if written to.
24	R/W	System Software Owned Semaphore Bit. Default = 0. • System software should set this bit to 1 when it wants ownership of the controller. • The BIOS will clear bit 16.
31:25	R/W	Reserved and must be preserved if written to.

The Next Offset field is the count of DWORDs until the next item in the list. For example, if we are currently at base offset 0x00500 in the list and the value in this field is 0x04, the next item in the list will be 4 DWORDs, 16 bytes from 0x00500, or 0x00510.

Please note that this field does not point to the remaining dword in the USB Legacy Support data. The remaining Legacy data is the very next DWORD after this one. Therefore, the value in this field must be at least 0x02 and usually is 0x04.

The two Owned Semaphore bits, bits 16 and 24, are used by the BIOS. The BIOS will set bit 16 to tell system software that the BIOS has ownership of the controller. When the operating system is ready to take over the ownership of the controller, the system software should set bit 24. The BIOS will see this, relinquish ownership and clear bit 16 indicating that it has.

If the BIOS does not support the xHCI controller, bit 16 will be clear. It is good practice to set bit 24 even if the BIOS doesn't support this controller.

The dword following the first dword before is described in the following table.

Table 9-15: Format of xHCI USB Legacy Support: Second DWORD

xHCI USB Legacy Support: Second DWORD		
Bit	Access	Description
0	R/W	USB SMI Enable. Default = 0
3:1	R/W	Reserved and must be preserved if written to.

4	R/W	SMI on Host System Error Enable. Default = 0
12:5	R/W	Reserved and must be preserved if written to.
13	R/W	SMI on OS Ownership Enable. Default = 0
14	R/W	SMI on PCI Command Enable. Default = 0
15	R/W	SMI on BAR Enable. Default = 0
16	RO	SMI on Event Interrupt. Default = 0
19:17	R/W	Reserved and must be preserved if written to.
20	RO	SMI on Host System Error. Default = 0
28:21	R/W	Reserved and must be written as zeros.
29	R/WC	SMI on OS Ownership Change. Default = 0
30	R/WC	SMI on PCI Command. Default = 0
31	R/WC	SMI on BAR. Default = 0

The second DWORD displayed above is used by the BIOS. Your software should not need to use its values. Please see page 410 of the xHCI specification v1.10 for more on these fields.

To stop BIOS Legacy support, and have full ownership of the controller, you should set bit 24 in the first DWORD, and then wait for bit 16 to become clear. If you do not follow this process, the BIOS might interfere with some of the devices, resetting them when it finds that the device is not in the state it thought it should be.

Again, if the BIOS doesn't support the xHCI controller, and most BIOS' may not until USB 3.0 is more common, setting bit 24 won't hurt anything, and bit 16 will already be clear.

xHCI USB Supported Protocol Capability

Every xHCI controller must support at least one USB Protocol, while most may support two. The allowed protocols are USB 3.0 and USB 2.0. One of the controllers I have has two physical downstream ports, called sockets, and two Protocol Support entries in the Extended Capabilities section. The first entry is the USB 3.0 entry describing the first two ports, and the second entry describing the same two ports as USB 2.0. Therefore, both sockets support USB 2.0 and USB 3.0 protocols.

The table below describes the format of a listed protocol entry.

Table 9-16: Format of xHCI USB Protocol Support

xHCI USB Protocol Support (DWORD 0)		
Bit	Access	Description
7:0	RO	ID (0x02)
15:8	RO	Next Offset
23:16	RO	Minor Revision number (The 0 of USB 3.0)
31:28	RO	Major Revision number (The 3 of USB 3.0)
xHCI USB Protocol Support (DWORD 1)		
Bit	Access	Description
31:0	RO	Name String (Big Endian) (“USB ”)

xHCI USB Protocol Support (DWORD 2)		
Bit	Access	Description
7:0	RO	Compatible Port Offset
15:8	RO	Compatible Port Count
27:16	RO	Protocol Defined
31:28	RO	Protocol Speed ID Count

xHCI USB Protocol Support (DWORD 3)		
Bit	Access	Description
4:0	RO	Slot Type[1] (0-31) (currently only zero is supported)
31:5	RO	Reserved and Preserved
[1]xHCI Version 1.10 only.		

The Name String and Major and Minor Number fields indicate which protocol the ports use. If these fields have “USB “ and 2.0, they support the USB 2.0 Specification protocol.

The Compatible Port Offset field is the starting port register set number that supports this protocol, with the Compatible Port Count field indicating the count, starting with this port number, of the consecutive ports that are included in this protocol.

The Protocol Defined field is specific for the protocol defined. Protocol 3.0 uses this for a Hub Depth field. Protocol 2.0 uses bit 17 to indicate whether these ports supports high-speed only, along with a few other fields including Hub Depth. See section 7.2.2.1.3 in the xHCI specification, version 1.10.

The Protocol Speed ID Count field indicates how many more dwords follow (not counting the reserved field at DWORD 3) that contains Protocol Speed information. Please see section 7.2 in the xHCI specification for more information.

To find out exactly which ports use what protocol and how many actual ports there are, see the next section.

Port Routing and Control

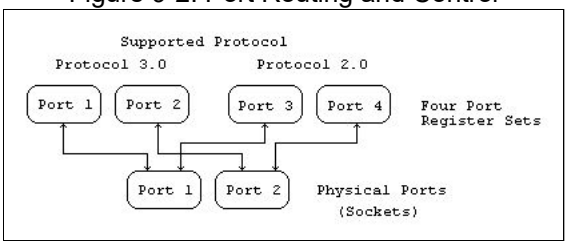
Each USB 3.0 protocol Root Hub port may have a USB 2.0 protocol device attached to it, though only one protocol can be used at a given time on a given port. Each protocol also uses a separate set of Port Control Registers.

Please see Chapter 21 on Super-Speed external hubs where both register sets are used at the same time for the same socket.

For example, one of the controllers I have has two physical ports, also called sockets, but the value in the HCSPARAMS1 register states 4 ports. Since the Supported Protocol entries in the Extended Capability space define 2 ports using protocol 3.0 and 2 more using protocol 2.0, we actually have only 2 sockets, each supporting both protocols. However, since the Port Control registers must be different for each protocol, we have 4 Port Control Register sets. The first two sets for the 3.0 protocol and the second two sets for the 2.0 protocol, for this specific card.

In the figure below we have four ports specified in the HCSPARAMS1 register, each supporting protocol USB 3.0 and USB 2.0, with two actual sockets.

Figure 9-2: Port Routing and Control



Just as an example, if the first Supported Protocol Count was two USB 3.0 supported entries, and the second Supported Protocol Count was three USB 2.0 supported entries, you would have three physical downstream ports (sockets). The first two may support both USB 3.0 and USB 2.0, while the third might only support USB 2.0. In other words, the first two sockets may support all speeds, while the third may only support high-, full-, and low-speed, depending on the High-Speed Only bit. The Offset and Count fields will identify which socket supports what protocol. See the xHCI specification, section 4.19.7 for more information.

The Offset and Count fields in the Supported Protocol Capability fields defined previously, tell you which Port Register Set supports what protocol. For example, if the USB 3.0 protocol entry has a 1 for the offset and a 2 for the count, the first two register sets will support the USB 3.0 protocol. If the USB 2.0 protocol entry has a 3 for the offset and a 2 for the count, the third and forth register sets support the USB 2.0 protocol.

Host Controller Operational Registers

The next set of host controller registers are the Host Controller Operational Registers, residing CAPLENGTH bytes (usually 32) past the initial base of the controllers I/O address space, and has the format described below. Remember that the Offset value is Base + CAPLENGTH + Offset.

Table 9-17: The xHCI Operational Registers

The xHCI Controllers Operational Registers		
Offset	Size	Description
00h	4	USBCMD: USB Command
04h	4	USBSTS: USB Status
08h	4	PAGESIZE: Page Size
0Ch-13h	8	Reserved
14h	4	DNCTRL: Device Notification Control
18h	8	CRCR: Command Ring Control
20h-2Fh	16	Reserved
30h	8	DCBAAP: Device Context Base Address Array Pointer
38h	4	CONFIG: Configure
3Ch-3FFh	964	Reserved
400h-13FFh	n*16	Port Register Set(s)

These registers are used to control the xHCI controller and the root hub and are explained below. Each register should only be accessed with 64-bit read and writes[4] for the 64-bit registers, and 32-bit read and writes for the 32-bit registers.

Table 9-18: The xHCI Command Register

xHCI controller: USBCMD: USB Command Register		
Bit	Access	Description
0	R/W	Run/Stop (RS)
1	R/W	Host Controller Reset (HCRST)
2	R/W	Interrupter Enable (INTE)
3	R/W	Host System Error Enable (HSEE)
6:4	R/W	Reserved and Preserved
7	R/W / RO	Light Host Controller Reset (LHCRST)
8	R/W	Controller Save State (CSS)
9	R/W	Controller Restore State (CRS)
10	R/W	Enable Wrap Event (EWE)
11	R/W	Enable U3 MFINDEX Stop (UE3S)
12	R/W	Stopped Short Packet Enable (SPE)[1]
13	R/W	CEM Enable (CME)[1]
31:14	R/W	Reserved and Preserved
[1]xHCI Version 1.10 only.		

The Run/Stop bit is set by your driver to start the scheduler or cleared to stop the scheduler. To stop the scheduler, write a zero to this bit, and then wait for the HCHalted bit in the USBSTS register to be set. You should not set this bit unless the HCHalted bit is also set. The controller should halt within 16mS after clearing this bit.

The Host Controller Reset bit is used to reset the controller. All of the Operational Registers are reset to their default values and any transaction is halted, but the USB reset is not driven on any USB2 downstream ports. A Hot or Warm Reset is driven on all USB3 downstream ports. None of the PCI registers are affected by this reset. Once the reset is finished, the host controller will clear this bit. Your driver should make sure the HCHalted bit in the USBSTS register is set before you try to reset the controller.

The xHCI controllers I have tested, all resets completed in an average of 24uS when nothing is attached. See later in this chapter for more on this.

Please see the Host Controller Detection section later in this chapter on port power status after a reset.

The Interrupter Enable bit enables interrupts to occur at its next interrupt threshold. Clear this bit to disable interrupts.

The Host System Error Enable bit allows an interrupt to fire when the HSE bit in the Status register is set. To acknowledge the interrupt, clear the HSE bit.

If a field is Reserved and Preserved, remember to write back the same values that you read from the register. This means for every write, you must first read the register so that you preserve these bits.

The Light Host Controller Reset bit is optional and if implemented and a 1 is written to it, it will reset the Operational Registers, but will not reset any of the root hub ports. The controller will clear this bit when the light reset is done. The Light HC Reset Capability bit in the HCCPARAMS1 register is set if this reset is implemented.

The Controller Save State bit is used to store the state of the controller in one of the scratch buffers you allocated at initialize time. Make sure the HCHalted bit is set, then set this bit to save the controller state. The Save State Status bit in the Status register indicates when the save is complete. See the Save and Restore Controller State section later in this chapter for more information.

The Controller Restore State is to restore the last saved state of the controller. Again, the HCHalted bit in the Status register should be set before you attempt a restore. The Restore State Status bit in the Status register indicates when the restore is complete. See the Save and Restore Controller State section later in this chapter for more information.

The Enable Wrap Event bit, when set, will trigger an interrupt every time the MFINDEX register goes from 0x3FFF to 0x0000.

The Enable U3 MFINDEX Stop bit is used when the controller's root hub is in the U3, Disconnected, Disabled, or Powered-off state.

The Stopped Short Packet Enable bit indicates that the controller should generate a Stopped-Short Packet Completion code on a short packet detect.

The CEM Enable bit indicates that the controller may return a Max Exit Latency too Large error when a Configure Endpoint Command is received and errors.

Table 9-19: The xHCI Status Register

xHCI controller: USBSTS: USB Status Register		
Bit	Access	Description

0	RO	HCHalted
1	R/W	Reserved and Preserved
2	R/WC	Host System Error (HSE)
3	R/WC	Event Interrupt (EINT)
4	R/WC	Port Change Detected (PCD)
7:5	R/W	Reserved and Zero'd
8	RO	Save State Status (SSS)
9	RO	Restore State Status (RSS)
10	R/WC	Save/Restore Error (SRE)
11	RO	Controller Not Ready (CNR)
12	RO	Host Controller Error (HCE)
31:13	R/W	Reserved and Preserved

The HCHalted bit is set by the controller after it has stopped executing as a result of the Run/Stop bit being cleared by software or by the controller due to an error.

Please see the note in the definition of the previous register for information on Reserved and Preserved fields.

The Host System Error bit is set by the controller when a serious error has occurred. The controller will then clear the Run/Stop bit to prevent other errors to follow.

The Event Interrupt bit is set by the controller when an Interrupt Pending bit of any Interrupter is set. When clearing the Interrupter's Interrupt Pending bit, you should clear this bit first to prevent a race condition.

The Port Change Detect bit is set by the controller when any port on the root hub has any port bit change from zero to one. This is so that system software may watch one bit instead of many bits when looking for port changes.

If a field is Reserved and Zero'd, remember to write zeros to this field. Therefore, if new bits are defined in a later specification, they might be *Clear on Write* bits, and you will not clear them by writing a 1 to them.

The Save State Status bit indicates that the controller is currently saving its internal state. When system software sets the Controller Save State bit in the Command Register, this bit becomes set. When the controller has successfully saved the state, it will clear this bit. Be sure to check bit 10 below.

The Restore State Status bit indicates that the controller is currently restoring its internal state. When system software sets the Controller Restore State bit in the Command Register, this bit becomes set. When the controller has successfully restored the state, it will clear this bit. Be sure to check bit 10 below.

The Save/Restore Error bit indicates that an unsuccessful Save or Restore was attempted. If this bit is set, an error occurred during a save or restore.

At the time of the 1st Edition writing, there was no known controllers that supported save and restore. One of the controllers I have allows the save bit to be set, but never modifies the Save State Status bit. After correspondence with the manufacturer, the controller's save and restore was not completed due to it being a pre-release board.

At the time of this 2nd Edition writing, a newly acquired controller used for testing,

supporting specification version 1.00, supports save and restore.

The Controller Not Ready bit indicates that system software should not write to any Doorbell or Operational registers, except the Status Register, until this bit becomes clear. The controller sets this bit when it is in reset.

When reading this register for the first time, this bit may be set, so always read it a second time when reading it for the first time after power is applied.

The Host Controller Error bit indicates that an internal error has occurred and the controller should be reset and reinitialized.

Table 9-20: The xHCI Page Size Register

xEHCI controller: PAGESIZE: Page Size		
Bit	Access	Description
15:0	RO	Page Size
31:16	RO	Reserved

The Page Size register indicates to system software the page size supported by the controller. To find the page size supported, use the formula 2^(x+12) where x is the bit location of the set bit. For example, if bit 0 is set, then this controller supports 2^(0+12) = 2^12 = 4096 byte page sizes. If bit 2 is set, this controller supports 2^(2+12) = 2^14 = 16384 byte page sizes.

I wondered why Intel/USB-IF decided to create this register this way. Clearly 2^27, which is the largest page size allowed, fits in 32 bits. Why didn't Intel just place the 32-bit page size in this register? I think it may be that in future implementations, multiple page sizes may be supported. For example, if both bits 0 and bits 2 above were set, then you could choose between the page sizes. Now you would need a register to tell the controller which page size to use. Being that the second half of this register is reserved, this would be a perfect place for that. Multiple bits allowed in the lower half to indicate supported page sizes, and a single bit allowed in the upper half to indicate to the controller which page size to use. Also, if this register were never meant to be modified, it would be in the Capabilities register set, not the Operational register set. Just my opinion though, whether that means anything or not :-).

An easy way to calculate the page size, using the C Language, would be:

```
page_size = (read_register(PAGE_SIZE) & 0xFFFF) << 12;
```

Table 9-21: The xHCI Device Notification Control Register

xHCI controller: DNCTRL: Device Notification Control Register		
Bit	Access	Description
0	R/W	Notification Enable 0 (N0)
...	R/W	...
15	R/W	Notification Enable 15 (N15)
31:16	RO	Reserved and Preserved

This register is used to enable or disable reporting of the reception of Device Notification Transaction Packets. There are 16 bits for 16 notification types. To enable a notification, set the bit. Currently, only type N1, FUNCTON_WAKE, is allowed. This register must be written to as a dword. Byte writes are undefined.

Table 9-22: The xHCI Command Ring Control Register

xHCI controller: CRCR: Command Ring Control Register		
Bit	Access	Description
0	R/W	Ring Cycle State (RCS)
1	R/WC	Command Stop (CS)
2	R/WC	Command Abort (CA)
3	RO	Command Ring Running (CRR)
5:4	R/W	Reserved and Preserved
63:6	R/W	Command Ring Pointer

The Ring Cycle State bit is used to tell the controller to start with a set bit or a clear bit in the Command Ring's Cycle State. See the section on the Command Ring usage for more information on this bit.

The Command Stop bit is used to stop the controller from executing the command ring. The controller will continue with the current command, then stop processing the Command Ring with a Command Ring Stopped event.

The Command Abort bit will immediately stop the controller executing the command ring. The controller will place a Command Ring Stopped event in the Event Ring.

The Command Ring Running bit simply identifies that the ring is running. This bit is set when the Run/Stop bit is set, and is cleared when the Run/Stop bit is cleared, or either the Command Stop or Command Abort bits are set.

The Command Ring Pointer is the high order 58 bits of the 64-bit address of the command ring, with bits 5:0 assumed zero. See the next chapter for a definition of this ring.

Table 9-23: The xHCI Device Context Base Address Array Register

xHCI controller: DCBAAP: Device Context Address Array Register		
Bit	Access	Description
5:0	R/W	Reserved and Zero'd
63:6	R/W	Device Context Base Address Array Pointer

This register is the 64-byte aligned physical address of the Device Context Address Array.

Table 9-24: The xHCI Configure Register

xHCI controller: CONFIG: Configure Register		
Bit	Access	Description
7:0	R/W	Max Device Slots Enabled (MaxSlotsEn)
8	R/W	U3 Entry Enable (U3E) [1]
9	R/W	Configuration Information Enable (CIE)[1]
31:10	R/W	Reserved and Preserved
[1]xHCI Version 1.10 only.		

This register is used to enable the Device Slots. Writing a value to bits 7:0 indicates the maximum amount of slots to enable. You should not modify this register if the Run/Stop bit is set.

The U3 Entry Enable bit indicates to the controller that it should assert the PLC flag when a Root Hub port transitions to the U3 state. Check bit 0 in the HCCPARAMS2 register for support of this bit.

The Configuration Information Enable bit indicates to the controller that it should initialize the Configuration Value, Interface Number, and Alternate Settings field in the Input Control Context block when it references a Configure Endpoint command. Check bit 5 in the HCCPARAMS2 register for support of this bit.

Explanation of the Save/Restore Operation

If the controller supports Save/Restore, it can be used to save and restore the state of the controller for when you power down the controller for sleep or hibernation.

To save the controller's state, read the USBCMD, DNCTRL, CRCR, DCBAAP, and CONFIG Operational registers and the IMAN, IMOD, ERSTSZ, ERSTBA, and ERDP Runtime Registers and save their state to memory. Then read and save all port register sets on the Root Hub. Finally, set the Controller Save State bit in the Command Register.

The controller will save its internal state and will indicate that it is done with the operation by clearing the Save State Status bit in the Status Register.

To restore the controller's state, set the Controller Restore State bit, wait for the Restore State Status bit to be clear, then restore the other registers you saved above.

The memory used by the Scratch Pad Buffers must remain intact. Since powering down might destroy the contents of the RAM, you can save this memory to disk via a paging mechanism, or simply save it to disk without paging, and when powering back up, restore the memory from disk before you set the Restore bit.

A few reasons you might want to save and restore the controller state, besides power management, could be that your platform has a few USB devices hardwired to the system, meaning they will always be present and in the same state. Therefore, the first time you boot the system, your code could enumerate these fixed devices, then save the state, storing the result to disk. Then the next time you booted, you simply restore the state, send a few TRB's, and the devices are up and going.

Be sure to check the Scratchpad Restore bit in the HCSPARAMS2 register. If it is set, the save will use the Scratchpad buffers. If it is clear, the save will use the internal memory of the controller, probably a flash memory card.

As of the first edition of this book, I didn't know of any xHCI controllers that supported the Save/Restore State operation. However, as of this writing, there are a few controllers that now support this feature, one being Renesas' uPD720202 controller.

Example of the Memory Mapped I/O Space

For an example of the memory mapped address space used, I have included below, the layout of one of the xHCI controllers I have used.

Table 9-25: xHCI Memory Mapped Address Space Usage

xHCI Memory Mapped Address Space Usage Example (8,192 bytes)		
Offset	Size	Description
0x0000	32	Host Controller Capability Registers including the 4 bytes padding for paragraph alignment.
0x0020	1,024	Host Controller Operational Registers including the 964 bytes reserved.
0x0400	64	Room for four 16-byte Port Register Sets.
0x0440	192	Reserved. Unused except for alignment for next item.
0x0500	256	Host Controller's Extended Capabilities List.
0x0600	32	Runtime Register Set start.
0x0620	256	Eight 32-byte Interrupter Register Sets.
0x0720	224	Reserved. Unused except for alignment for next item.
0x0800	128	Door Bell Array of thirty-two 32-bit entries.
0x0880	6,016	Reserved. Padding to a total space of 8,192 bytes.

Most likely, the Runtime Register Set and Door Bell Array offsets are hard coded on the controller so that it is a simple task to add ports and Interrupter Sets without having to rewire these offsets when other cards are manufactured. Most likely all controllers will be very similar to this outline and may use this same amount of space.

The Root Hub's Port Status and Control Registers

The port registers are used for controlling and receiving status from each port on the root hub. To determine how many ports there are on the root hub, read the HCSPARAMS1 register looking at bits 31:24. With this in mind, there can be no more than 255 ports on a root hub.

The xHCI controller uses four 32-bit registers to control its root hub's ports. The first set starts at offset 0x400 in the Operational Register space, with the next port's set at 0x410, etc., with 256 total supported register sets. However, since the MaxPorts field in the HCSPARAMS1 register only allows 255, the last set will not be used.

To access any of these 255 possible port sets, you read or write 16 bytes from/to (OperationalBase + (0x0400 + (16 * port_num))), where port_num is a zero based port number.

Table 9-26: The xHCI Port Register Set

XHCI controller: Port Register Set		
Offset	Name	Description
0x00	PORTSC	Port Status and Control
0x04	PORTPMSC	Port Power Management Status and Control
0x08	PORTLI	Port Link Info
0x0C	PORTHLPMC	Port Hardware LPM Control[1]
[1]xHCI Version 1.10 only. Reserved in version 1.00 or before.		

Explanation of the PORTSC Register

The Port Status and Control Register, is used to control the Root Hub's ports, and is the same format for both USB 3.0 and USB 2.0 protocol ports.

Until the Run/Stop bit is set and the HCHalted bit is clear, you should not write to this register and this register will not show connection status changes.

Table 9-27: The xHCI Port Status and Control Register

xHCI controller: PORTSC: Port Status and Control Register		
Bit	Access	Description
0	RO	Current Connect Status[1]
1	R/WC	Port Enabled/Disabled[1]
2	R/W	Reserved and Zero'd
3	RO	Over-current Active
4	R/W	Port Reset[1]
8:5	R/W	Port Link State[1]
9	R/W	Port Power
13:10	RO	Port Speed
15:14	R/W	Port Indicator Control[1]
16	R/W	Port Link State Write Strobe[1]
17	R/WC	Connect Status Change[1]
18	R/WC	Port Enable/Disable Change[1]
19	R/WC[2]	Warm Port Reset Change[1]
20	R/WC	Over-current Change[1]
21	R/WC	Port Reset Change[1]
22	R/WC	Port Line State Change[1]
23	R/WC[2]	Port Config Error Change[1]
24	RO[2]	Cold Attach Status[1] (RsvdZ in version 0.96)
25	R/W	Wake on Connect Enable
26	R/W	Wake on Disconnect Enable
27	R/W	Wake on Over-current Enable
29:28	R/W	Reserved and Zero'd
30	RO	Device Removable
31	R/WC[2]	Warm Port Reset
[1]These bits are 0, no matter the connection status, if bit 9, Port Power, is 0. [2]These bits are reserved and zeroed on USB 2.0 protocol ports.		

The Current Connect Status bit, when set by the controller, indicates that there is a device attached. If the Port Power bit is zero, this bit will be zero, no matter the connection status.

The Port Enabled/Disabled bit shows the enabled status of the port. A port can only be enabled by the controller after a successful reset. Your driver cannot set this bit to enable the port. This bit is read to indicate whether the port is enabled or not. A port will be disabled if an error occurs, such as an over-current condition. System software setting this bit will disable the port and you must reset the port to re-enable it, re-enumerating the device(s) attached.

The Over-Current Active bit indicates that there has been an over-current condition on this port. Once the over-current condition has been removed, the controller will clear this bit.

The Port Reset bit is used to reset and possibly enable the port. Depending on the protocol used for this port, system software resets this port using the technique defined in the specified

protocol specification. Resetting a USB 2.0 port is slightly different that resetting a USB 3.0 port. See later in this chapter on resetting a root hub port.

The Port Link State field is used to set the port's power state. This is used to set the port to the U0, U1, U2, and U3 state. At this point I will not go into much detail about the link state of a port. This may be left for another section or chapter all together. Please note that you must also write a 1 to the Port Link State Write Strobe bit to be able to write to this field. You may also read from this field at any time to see the current link state of the port. To use the port, this field must be in the U0 state, a value of 0000b.

The Port Power bit is to switch power on or off for this port. If the Port Power Control (PPC) bit in the HCCPARAMS1 register is set, the Port Power bit will turn on or off the power to this port. If the PPC bit is clear, the port is always powered. However, since different controllers will control power differently, this bit must be set to one to use the port. If this bit is zero, the port will not show a connect status, nor read any values from the port no matter the state of the PPC bit in the HCCPARAMS1 register. If your driver sets the Port Power bit to a one from a zero, you must wait a minimum of 20mS before you can use the port.

If a physical port has more than one Protocol Port Control Register set, power will be applied to the port if the Port Power bit is set in either register set. e.g., if you set the port's power to on in the USB3 register set, the USB2 register set will show the same power status.

The Port Speed field indicates the speed of the attached device. If the port is using the USB2.0 protocol, this field is invalid until after the reset. The following table defines the values for this field. Please note that full-speed is first, not low- then full-speed as you might think.

Table 9-28: The xHCI Port Speed values

xHCI controller: PORTSC: Port Speed			
Value	Speed	Bit Rate	Protocol
0	Undefined		
1	full-speed	12 Megabytes/Sec	USB 2.0
2	low-speed	1.5 Megabits/Sec	USB 2.0
3	high-speed	480 Megabits/Sec	USB 2.0
4	super-speed	5 Gigabits/Sec	USB 3.0
5-15	Undefined		

The Port Indicator Control bits have no effect if the PIND bit in the HCCPARAMS1 register is zero. If it is 1, then these two bits have the following encoding.

Table 9-29: xHCI: Port Indicator Control

Value	Meaning
00b	Port indicators are off
01b	Amber
10b	Green
11b	Undefined

The USB 2.0 specification, section 11.5.3, detail the port indicator usage. Amber usually means error, while green usually means fully operational. The blinking of an indicator is used to gain the attention of the user. Blinking amber usually means hardware attention while blinking green usually means software attention. Remember that these indicators reflect the status of the port, not the device attached to it.

The Port Link State Write Strobe bit indicates to the controller that the value in bits 8:5 of the value written will be written to the port. If this bit is set, the controller takes the value of bits 8:5

as a written value to the Port Link State. If this bit is clear, it ignores bits 8:5 of the value written to this register.

> This feature is required so that the controller doesn't see a need for a transition, when written to. If you read the register, then write it back without the feature of this bit, the controller will see it as a transition command and transition to that state, even if it is the same state.

The Connect Status Change bit, when set by the controller, indicates that a connection change has taken place. Your driver must be careful with this status change.

> Please see Appendix D, "Port Connection Status Change" for an important note about connect status changes.

The Port Enable/Disable Change bit, when used with a USB 2.0 protocol port, will become set when a change in the Enable State is found, usually when an enabled port becomes disabled.

The Warm Port Reset Change bit, when used with a USB 3.0 protocol port, indicates that the Warm Port Reset is complete. Software resets this bit to zero by writing a one to it. This bit remains reserved when used with a USB 2.0 protocol port.

The Over-current Change bit indicates that there has been a change to the over-current condition on this port.

The Port Reset Change bit is set by the controller when the Port Reset bit transitions from a one to a zero due to a reset complete.

The Port Link State Change bit indicates that there has been a change in the Link State of the port. Refer to section 4.23.5 of the xHCI specification for more information.

The Port Config Error Change bit indicates that the port failed to configure its link partner.

The Cold Attach Status bit, when used with a USB 3.0 protocol port, indicates that the attached device did not train correctly due to lack of power, wrong power state, or another problem. In other words, a USB 3.0 protocol device is attached, but failed to move to the enabled state. This bit remains reserved when used with a USB 2.0 protocol port.

The Wake on Connect Enable, Wake on Disconnect Enable, and Wake on Over-current Enable bits are used to tell the controller to issue a wake-up event if either of these events occur while in a suspend state.

The Device Removable bit indicates to system software if the device attached to this port is removable. If set, the device is non-removable.

> Please note that the name of this bit, Device Removable, would indicate that if the bit was set, the device is removable. However, it is just the opposite. If this bit is set, the device is non-removable.

The Warm Port Reset bit, when used with a USB 3.0 protocol port, is set by software to start the Warm Port Reset sequence as defined in the USB3 specification. The Port Reset, Port Reset Change, and Warm Reset Change bits are used to indicate the completion of the reset. This bit remains reserved when used with a USB 2.0 protocol port. See later in this chapter for port reset using this bit.

Explanation of the PORTPMSC Register

The Port Power Management Status and Control Register definition depends on the protocol used. For the USB2 protocol, this register is used for the Link Power Managed states and defined next.

Table 9-30: The xHCI PM Port Status and Control Register (USB2)

XHCI controller: PORTPMSC: PM Port Status/Control Register (USB2)		
Bit	Access	Description
2:0	RO	L1 Status
3	R/W	Remote Wake Enable
7:4	R/W	Host Initiated Resume Duration[1]
15:8	R/W	L1 Device Slot
16	R/W	Hardware LPM Enable
27:15	R/W	Reserved and Preserved
31:28	R/W	Port Test Control
[1]This field is renamed in xHCI Version 1.10, but does the same thing.		

The L1 Status bit is read only and indicates the status of an L1-based suspend request. Its values are listed in the table below.

Table 9-31: xHCI: L1 Status Field

Value	Meaning
0	Invalid. This field should be ignored if zero.
1	Success. Port successfully transitioned to L1.
2	Not Yet. Device is has not transitioned yet.
3	Not Supported. Device doesn't support transition.
4	Timeout/Error. Device failed to respond to the transition or an error occurred.
5-7	Reserved

The Remote Wake Enable bit is set by software to enable the device for remote wake from the L1 state.

The Host Initiated Resume Duration field is used to set the duration the controller will drive the resume if it initiates and exit from L1. Values 0 to 15 are supported, and is interpreted as:

$$\text{Duration} = 50 + (x * 75)$$

where x is the value of this field.

The L1 Device Slot field is used to indicate to the hardware which device slot this port will use to generate the LMP Token packet. A value of zero indicates no device present.

The Hardware LPM Enable bit when set to 1, indicates that the hardware controlled LPM is enabled for this port. This field is not used and is reserved if the Hardware LMP Capability is not supported. To see if Hardware LPM is supported, check bit 19 of the USB 2.0 Protocol Capabilities register. See the section on Extended xHCI USB Supported Protocol Capabilities earlier in this chapter.

Bits 27:15 should be preserved when writing to them. For example, when you read this register, the values read for bits 27:15 should be written back unchanged.

The Port Test Control field is used for Port Test Control. Software may write a value to this field to generate a specific test mode. This field is not used if the port is in the powered state. See page 320 of the xHCI, ver 1.10, specification for more information.

For the USB3 protocol, this register is used for the Super-speed timeouts and defined in Table 9-32 below.

Table 9-32: The xHCI PM Port Status and Control Register (USB3)

xHCI controller: PORTPMSC: PM Port Status/Control Register (USB3)		
Bit	Access	Description
7:0	R/W	U1 Timeout
15:8	R/W	U2 Timeout
16	R/W	Force Link PM Accept
31:17	R/W	Reserved and Preserved

The U1 Timeout field indicates the delay in microseconds before U1 inactivity is determined and initiating a U1 entry. Allowed values are 0x00 to 0x7F, and 0xFF. A value of 0xFF disables the port from initiating a U1 entry. A value of 0x01 indicates 1uS, 0x02 indicates 2uS, and so on.

The U2 Timeout field indicates the delay in microseconds before U2 inactivity is determined and initiating a U2 entry. Allowed values are 0x00 to 0xFF. A value of 0xFF disables the port from initiating a U2 entry. This field differs in that a value of 0x01 indicates 256uS, 0x02 indicates 512uS, 0x03 indicates 768uS, and so on.

The Force Link PM Accept bit tells the port to generate a Set Link Function LMP with the Force LinkPM Accept bit set.

Explanation of the PORTLI Register

The Port Link Information Register definition depends on the protocol used. For the USB2 protocol, this register is reserved and preserved.

Table 9-33: The xHCI PORTLI Register (USB2)

xHCI controller: PORTLI: Link Info Register (USB2)		
Bit	Access	Description
31:0	R/W	Reserved and Preserved

For the USB3 protocol, this register is used to report the Link Error Count. For information on Link Error Counting, refer to section 10.14.2.5 of the USB 3.0 specification.

Table 9-34: The xHCI PORTLI Register (USB3)

xHCI controller: PORTLI: Link Info Register (USB3)		
Bit	Access	Description
15:0	RO	Link Error Count
19:16	RO	Rx Lane Count (RLC)
23:20	RO	Tx Lane Count (TLC)
31:24	R/W	Reserved and Preserved

The Link Error Count field reports the number of link errors detected. The controller will reset this field to zero on a Chip Hardware Reset, a Port Reset, or when Current Connect Status is set.

The Rx and Tx Lane Count fields identify the number of Receive or Transmit lanes negotiated by the port. Zero based, 0 - 15, representing lanes 1 through 16.

Explanation of the PORTHLPMC Register

The Port Hardware LPM Control Register definition depends on the protocol used. For the USB2 protocol, it is used to automatically send an LPM token to the downstream device.

Table 9-35: The xHCI PROTHLPMC Register (USB2)

xHCI controller: PROTHLPMC : Port Hardware LPMC Register (USB2)		
Bit	Access	Description
1:0	RWS	Host Initiated Resume Duration Mode (HIRDM)
9:2	RWS	L1 Timeout
13:10	RWS	Best Effort Service Latency Deep (BESLD)
31:14	R/W	Reserved and Preserved

The HIRDM duration value must be 0 (as default) or 1.

The L1 Timeout can be a number from zero to 255. A value of zero is the default and represents a 128 uS timeout. A value of 1 to 255 represents a (value * 256) uS timeout. In other words, if the value is greater than zero, multiple that number by 256 to get the count of microseconds to timeout.

The BESLD is used to indicate how long the recipient device is to wait after a U2 exit. See Table 13 on page 267 of the xHCI specification, version 1.10.

For the USB3 protocol, this register is reserved and preserved.

Table 9-36: The xHCI PROTHLPMC Register (USB3)

xHCI controller: PROTHLPMC : Port Hardware LPMC Register (USB3)		
Bit	Access	Description
31:0	R/W	Reserved and Preserved

Explanation of the Host Runtime Register Set

The Host Runtime Register set includes the Microframe Index, which can be read to determine the current frame number, and up to 1,024 Interrupter Register Sets. To determine where the Host Runtime Register Set resides, see the RTSOFF register in the Capabilities Register set. To determine how many Interrupter Register Sets there are, see the HCSPARAMS1 register in the Capabilities Register Set. The table below shows the format of the Host Runtime Register Set.

Table 9-37: The xHCI Host Runtime Register Set

The xHCI Host Runtime Register Set		
Offset	Size	Description
0x0000	4	Microframe Index Register.
0x0004	28	Reserved and Zero'd.
0x0020	32	Interrupter Register Set 0 (IR_0)
0x0040	32	Interrupter Register Set 1 (IR_1)
0x0060	32	Interrupter Register Set 2 (IR_2)
...	...	...
0x8000	32	Interrupter Register Set 1023 (IR_1023)

The Controller does not have to reserve all 0x8020 bytes if it does not use all interrupters. For example, it the controller only has 15 interrupters, the controller only has to reserve 512 bytes for this register set. The DoorBell Array, or any other Register Set, could follow at Runtime_Register_Set + 0x200.

The Microframe Index register is incremented by the controller every 125 microseconds while the Run/Stop bit is set. In other words, the controller increments this counter at the beginning of each microframe. This allows system software to see which microframe is currently being processed.

Since each frame has 8 microframes, simply mask off bits 2:0 to get the current frame number. Bits 13:3 will be incremented every 1 millisecond.

A controller must support at least one interrupter, which is called the Primary Interrupter. If a controller supports more than one, each interrupter after the first is called a Secondary Interrupter. The Primary Interrupter starts at offset 0x0020, with each remaining supported interrupter following the previous one. The Interrupter Register Set uses eight 32-bit registers, each must be read and written as 32-bit registers, and has the following format.

Table 9-38: The xHCI Interrupter Register Set

xHCI controller: Interrupter Register Set		
offset	Access	Description
0x00	R/W	Interrupter Management Register
0x04	R/W	Interrupter Moderation
0x08	R/W	Event Ring Segment Table Size
0x0C	R/W	Reserved and Preserved
0x10	R/W	Event Ring Segment Table Base Address
0x18	R/W	Event Ring Dequeue Pointer

Note that the last two registers are 64-bits. If the current hardware platform is unable to write 64-bits at once, system software may write 32-bits to the low dword then 32-bits to the high dword.

The Interrupter Management register contains two applicable bits and shown below.

Table 9-39: The xHCI Interrupter Management Register

xHCI controller: Interrupter Management Register		
bit	Access	Description
0	R/WC	Interrupt Pending (IP)
1	R/W	Interrupt Enable
31:2	R/W	Reserved and Preserved

The Interrupt Pending bit is set by the controller when there is an interrupt pending for this interrupter. This bit should be cleared by system software before another interrupt can take place.

It is very important in what method and sequence you use to clear this bit and the Status Register's Interrupt bit. The controller will not fire another interrupt until this bit is cleared. However, you must not clear the Status Register's Interrupt bit before this IP bit since the controller may fire an interrupt before you have a chance to clear the IP bit. Then you will miss the interrupt since the bit was already set. See a later chapter for more information on this technique.

I found that with one of the controllers I have, you must clear the Interrupter's IP bit and the Status Register's Interrupt bit before you read any of the other registers. The controller needs system software to acknowledge the interrupt before anything else is done. I had the interrupt acknowledge code after the IRQ code and the controller would not fire another interrupt. As soon as I moved the Acknowledge interrupt code to the beginning of the IRQ, the controller would continue to fire interrupts when expected.

The Interrupt Enable bit is set by system software to allow this Interrupter to cause an interrupt at the next interrupt threshold.

The Interrupter Moderation register contains two fields and is described below.

Table 9-40: The xHCI Interrupter Moderation Register

XHCI controller: Interrupter Moderation Register		
Bit	Access	Description
15:0	R/W	Interrupt Moderation Interval
31:16	R/W	Interrupt Moderation Counter

The Interrupt Moderation Interval field is simply a value, in 250nS intervals, to tell the controller to wait before it fires the next interrupt. After you clear the Interrupt Pending bit, the controller will use the Interrupt Moderation Counter field to count down to zero. Once this field is zero, the next interrupt can take place. This is to give a little time between interrupts. If you put a value of zero in the Interrupt Moderation Interval field, this operation is overridden and interrupts take place as soon as they are generated.

The specification, rev 1.0 dated 21 May 2010, has the second formula on page 306 wrong. So does version 1.10, page 328. Use the formula below.

To calculate the value in the Interval field with a given interrupts-per-second value, use the following formula:

$$\text{Interval} = 1 / (250\text{x}10^{-9} \text{ x interrupts_per_sec})$$

For example, if you want the interrupt to generate no less than once every millisecond, you would place a value of 3906 in this field.

$$\text{Interval} = 1 / (250\text{x}10^{-9} \text{ x } 1024)$$

If you want to find how many times a second a given interval value would allow the controller to generate an interrupt, use the formula below.

$$\text{Interrupts per second} = 1 / (250\text{x}10^{-9} \text{ x Interval})$$

Again, a value of zero in the Interval field disables this feature and interrupts are processed as soon as they are generated.

You should not place a value less than 500 into the Interval field other than zero. A value of 500 would be 8000 interrupts per second.

I don't know why anyone would want to, but you can write a value to the Counter field at any given time. Maybe to make sure the interrupt isn't processed until a specific time after the currently executed code, or sooner than currently programmed to do so, or maybe a one time different Moderation Interval value is desired without having to change the Interval field twice, once before and once after.

The Event Ring Segment Table Size register contains two fields and is as follows.

Table 9-41: The xHCI Event Ring Segment Table Size Register

XHCI controller: Event Ring Segment Table Size Register		
Bit	Access	Description
15:0	R/W	Event Ring Segment Table Size
31:16	R/W	Reserved and Preserved

The Event Ring Segment Table Size field is the count of segments in this Interrupters event ring. The max size is calculated with bits 7:4 of the HCSPARAMS2 Capabilities register. If this value is zero, this interrupter's event ring is disabled. The primary event ring should not be disabled. See Chapter 10 for more on Event Rings and how these segments are used.

The Event Ring Segment Table Base Address register contains two fields and is as follows.

Table 9-42: xHCI Event Ring Segment Table Base Address Register

XHCI controller: Event Ring Segment Table Base Address Register		
Bit	Access	Description
5:0[1]	R/W	Reserved and Preserved
63:6	R/W	Event Ring Segment Table Base Address (63:6)
[1]In version 0.96, this is bits 3:0. Version 1.00+ uses bits 5:0		

The Event Ring Segment Table Base Address field is the high order 58 bits of the 64-bit physical address of the Event Ring Segment Table. To simplify, simply read the whole 64-bit register and mask off the lower six bits to get the 64-bit physical address.

When writing to this register, be sure to preserve the lower six bits, since they may be used in a later release.

If your system does not allow 64-bit memory accesses, you may read or write the lower 32-bit address first, then the higher 32-bit address. Make sure to do it in that order.

Writing to this register, “starts” the Advancement State of the ring. You should write this register after writing to previous registers. See Chapter 10 for more information.

The Event Ring Dequeue Pointer register contains three fields and is as follows.

Table 9-43: The xHCI Event Ring Dequeue Pointer Register

XHCI controller: Event Ring Dequeue Pointer Register		
Bit	Access	Description
2:0	R/W	Dequeue ERST Segment Index
3	R/W	Event Handler Busy
63:4	R/W	Event Ring Dequeue Pointer (63:4)

The Dequeue ERST Segment Index field is used to accelerate the checking of an Event Ring Full condition. This field can be 0.

The Event Handler Busy bit is set by the controller when it sets the Interrupt Pending bit. Then once your handler is finished handling the event ring, you clear it by writing a ‘1’ to it, usually when writing the next field.

The Event Ring Dequeue Pointer field is the high order 60 bits of the 64-bit physical address of the next item in the Event Ring.

System software should not write the same value to the Event Ring Dequeue Pointer field consecutively unless the ring is full or empty.

See Chapter 10 for more on the Event Ring Registers.

Explanation of the Doorbell Register Set

The Doorbell Register Set is used to tell the controller that there is something in the command ring to process or one of the Device Slots has related work to perform. To determine where the Doorbell Register Set resides, see the DBOFF register in the Capabilities Register set. To determine how many Doorbell Registers there are, see the HCSPARAMS1 register, bits 7:0, in the Capabilities Register Set, remembering to add one for the existence of the Command Ring.

The Doorbell Register Set is an array of up to 256 dwords each 32-bits in length. The first is the Command Doorbell Register while the remaining registers are used one for each Device Slot allowed. Figure 9-2 below shows the layout.

Figure 9-3: The xHCI Doorbell Register Set

Command Doorbell Register
Device Slot #1 Doorbell
Device Slot #2 Doorbell
Device Slot #3 Doorbell
...
Device Slot #*max* Doorbell

The Command Ring Doorbell is as follows.

Table 9-44: The xHCI Command Doorbell Register

XHCI controller: Command Doorbell Register		
Bit	Access	Description
7:0	R/W	Target
31:8	R/W	Reserved and Zero'd

The Target field should be zero when writing to the Command Doorbell. In fact, the actual writing of the Command Doorbell is all that really needs to be done to "ring" this doorbell. However, the specification has left room for vendors to specify specific terms for the Command Doorbell. Therefore, you should always write a zero to this 32-bit register when not performing vendor specific tasks.

The format of a Device Slot Doorbell register is a little different and is described below
.

Table 9-45: The xHCI Device Slot Doorbell Register

XHCI controller: Device Slot Doorbell Register		
Bit	Access	Description
7:0	R/W	Target
15:8	R/W	Reserved and Zero'd
31:16	R/W	Doorbell Stream ID

The Target field is used to indicate what notification the "ringing" of the doorbell is generating. The table below indicates what values to use.

Table 9-46: Doorbell Target Notification values

Value	Definition
0	Reserved for Command Doorbell
1	Control EP Enqueue Pointer Update

2	EP 1 OUT Enqueue Pointer Update
3	EP 1 IN Enqueue Pointer Update
4	EP 2 OUT Enqueue Pointer Update
5	EP 2 IN Enqueue Pointer Update
...	
30	EP 15 OUT Enqueue Pointer Update
31	EP 15 IN Enqueue Pointer Update
32-247	Reserved
248-255	Vendor Specific

If I were using the C programming language, I would calculate the target value for a specific Endpoint, use the following line:

```
TargetValue = 2 + (ZeroBasedEndpoint * 2) + (is_out_ep ? 0 : 1);
```

The Doorbell Stream ID field is used by endpoints that use Streams and indicates which stream the Doorbell is targeting. Values 0, 65534, and 65536 are reserved and should not be written to this field.

See Chapter 10 for more on using Doorbells.

Detecting and Resetting the Host Controller

Since the PCI bus already told us that there is an xHCI controller at this base, why should we take the time and effort to detect the controller? To be completely independent of the host bus, the xHCI driver should detect the controller rather than relying on its host bus. Future hardware may have the USB on a different bus type that isn't so accommodating.

To detect the controller, we will attempt to reset it and then see what happens to the register's values.

There are two bits in the command register to reset the controller. One is the Light Host Controller Reset bit and its function is optional. To determine if it is supported, read bit 5 of the HCCPARAMS1 register. If set, this reset is supported and will reset some of the operational registers, but will not affect the PORTSC register sets. This is so that you can do a reset of the controller, without affecting the status of the ports.

Since we have not yet set up the controller, and this is our first time for a reset, we should not use the light reset. We need a reset that will reset the entire controller.

The Host Controller Reset (HCRST) bit in the command register will reset the entire controller including the root hub ports. The controller resets this bit to zero when the reset is complete.

After a Host Controller reset, power is applied to all ports no matter the port power bit in HCCPARAMS1. However, a controller might not apply power until a write to the MaxSlotsEnabled field in the Config Operations Register. This is so the ports will not consume power until the driver is loaded and ready.

With this in mind, and knowing that after the host reset, all registers are set to their default state, do the reset and check for these default values. The following modified snippet of code comes from the xHCI driver of FYSOS. The driver calls this code expecting a TRUE or FALSE condition on return, returning TRUE if an xHCI controller was detected on this base address. The mdelay() routine simply delays for a given amount of milliseconds (mS).

Listing 9-1: xHCI Host Controller Reset

```
bool det_xhci_root(const bit32u base) {
  // make sure the run/stop bit is clear
  xhci_write_oper_reg(base, xHC_OPS_USBCommand,
        xhci_read_oper_reg(base, xHC_OPS_USBCommand) & ~(1<<0));

  // wait for the HcHalted bit to be set. The specification
  //  says 16ms, we will wait a few more to make sure
  bit32u timeout = 20;
  while
    (!((xhci_read_oper_reg(base, xHC_OPS_USBStatus) & (1<<0))) {
    if (--timeout == 0) {
      putstr("\nXHCI HC did not Halt within 20 ms.");
      return FALSE;
    }
    mdelay(1);  // 1ms
  }

  // set the hcreset bit
  xhci_write_oper_reg(base, xHC_OPS_USBCommand, (1<<1));

  // wait for this bit and the CtlrNotReady bit to be clear
  // The controller I have takes about 24 microseconds to reset.
```

```
 timeout = XHCI_RESET_DELAY; // more than enough
 while ((xhci_read_oper_reg(base, xHC_OPS_USBCommand) & (1<<1))
   || (xhci_read_oper_reg(base, xHC_OPS_USBStatus) & (1<<11))) {
   if (--timeout == 0) {
     printf("\nXHCI did not reset in %imS.", XHCI_RESET_DELAY);
     return FALSE;
   }
   mdelay(1);  // 1ms
 }
 mdelay(USB_TRSTRCY);

 // check the defaults of the Operational Registers
 if (xhci_read_oper_reg(base, xHC_OPS_USBCommand) != 0)
   return FALSE;
 if (xhci_read_oper_reg(base, xHC_OPS_USBDnctrl) != 0)
   return FALSE;
 if (xhci_read_oper_reg64(base, xHC_OPS_USBCrcr) != 0)
   return FALSE;
 if (xhci_read_oper_reg64(base, xHC_OPS_USBDcbaap) != 0)
   return FALSE;
 if (xhci_read_oper_reg(base, xHC_OPS_USBConfig) != 0)
   return FALSE;
 return TRUE;
}
```

I have yet to find a resource that states how long the controller will take to reset. One of my test machine's controllers, when no devices are attached, takes about 24uS. However, when a super-speed device is attached, it takes up to 43mS. I would assume that if the controller has not reset after 100mS (XHCI_RESET_DELAY in listing above), it is not a valid controller.

The code in Listing 9-1 checks for the following items to be true. If any of them are not true, the function returns FALSE to the caller.

- The Command register should be zero.
- The HcHalted bit in the Status register is set. If a device is attached, the Port Change Detect bit may be set, so ignore this bit.
- The Device Notification Control register should be zero.
- The Command Ring Control register should be zero.
- The Device Context Base Address should be zero.
- The Configure register should be zero.

If any of these items are false, then we did not find a valid controller, and the function call returns FALSE.

Set up the xHCI controller

Now that we know we have an xHCI controller, we can start to set up the controller and our schedule. The controller has already been reset by the detection routine shown before, however there are a few more items we need to set up before we have a working schedule.

> This section mentions data structures that have not yet been discussed. These data structures are detailed in the next chapter. For now, simply know that the following registers need physical addresses to allocated memory for these structures, and in your driver these structures should already be initialized correctly.

We need to write a 64-bit physical address of our Context data structure to the Device Context Base Address Array Pointer Register. The address must be 64-byte aligned.

```
xhci_write_oper_reg64(base, xHC_OPS_USBDcbaap,
                                    ThePhysicalAddress);
```

The Command Ring Control Register also needs a 64-bit address to allocated memory that is 64-byte aligned. Also, make sure that the lower command bits are all clear, with the cycle bit set since this is the Producer.

```
xhci_write_oper_reg64(base, xHC_OPS_USBCrcr,
                                  ThePhysicalAddress | 1);
```

The Configure register needs to be set to the amount of slots we will support. In this case, simply write the maximum allowed slots from the HCSPARAMS1 register.

```
xhci_write_oper_reg(base, xHC_OPS_USBConfig, MaxSlots);
```

The Device Notification Control register needs to have bit 1 set. Currently, this is the only bit allowed.

```
xhci_write_oper_reg(base, xHC_OPS_USBDnctrl, (1<<1));
```

Within the Runtime Register Set is the Interrupt Register Set. At this time is when you would need to set up this set. Again, read the next chapter for information on this process.

> Please note that all data structures have an alignment requirement and also must not cross a page boundary. See the next chapter for more information on alignment and page boundaries.

At this point, the controller is ready to run the schedule. Setting the Run/Stop bit, the Interrupter Enable bit, and the Host System Error Enable bit in the Command register, will start the schedule. Be sure to wait for the HCHalted bit to become clear before you proceed to add things to the schedule.

```
xhci_write_oper_reg(base, xHC_OPS_USBCommand,
                                  (1<<3) | (1<<2) | (1<<0));
```

Setting up and handling the bits in the Port Status/Command Register Set is enough for a new section, therefore, see the next section on the setting of the port registers and detecting the device status.

As for the xHCI hardware, we have completely set it up. The next thing to do is to start sending and receiving packets from the schedule. After a discussion on detecting device status and other device related items, I will get back to this topic on how to send and receive packets to and from the schedule.

For now, let us move on to the Port Status/Control Register Set and device detection.

The xHCI Port Status/Control Register Set

Before we can do anything with the port, we must make sure it has power. If the Port Power Control bit in the HCCPARAMS1 register is clear, then all ports are hardwired to power on. If the Port Power Control bit is set to 1, meaning all ports have power switching and can be powered by a bit in the port, we will need to make sure the Port Power bit in the port register is set. The controller will not report the connection status or other status unless there is power to the port. However, since different controllers handle this differently, you must have the Port Power bit set no matter the status of the HCCPARAMS1:PPC bit. Any controller will not report connections or other states unless bit 9, Port Power Control, is set.

Unlike the EHCI, the xHCI controller has power to the port on by default. However, if it isn't, after we set the Port Power bit from a 0 to a 1, we must wait no less than 20mS for the port to have power, and before we read from it.

When the user plugs or unplugs a device into/from one of the root hub's ports, that port's status bits are set or cleared depending on the current status.

> At bootup, if a device is already attached (plugged in), the status of this register is identical to if you had just attached the device during normal operation, even though the device was attached before the machine was turned on.

If the Current Connect Status bit is set we have a device attached. Next task would be to reset and enabled it so that we can start the enumeration process.

> After reset, the port's PLS field may be in the RxDetect (5d) state. This means that the port is trying to detect if there is a device on this port. Your driver should check this field to be sure it is in the U0 (0d) state by waiting for the transition. If the transition to the U0 state does not happen, you may have a faulting device. If the PLS field is in a different state, you may be able to write 0000b to it to transition to the U0 state. Remember to set the Strobe bit when writing to the PLS field.

Pairing up the USB2 and USB3 port registers

Before we reset the device, remember that an xHCI controller's root hub port can be a USB2 or USB3 protocol port, and will function differently depending on the protocol. If you remember from a previous section, if the root hub supports both USB2 and USB3 protocols, it will have two different port register sets for the same physical port (socket). Depending on the device attached depends on which register set you use.

Please note that if you plug in a USB3 protocol device, such as a super-speed hard drive, the USB3 protocol port register set will show the connection, with the USB2 protocol port possibly also showing the connection. Therefore, if you go through the port list, enumerating the devices starting with the USB3 port registers, once you get to the USB2 port registers, you will see a connection and try to reset the device again. This will reset the USB3 device you already enumerated and try to enumerate it as a USB2 device.

Therefore, you must pair up the USB3 port register sets with the USB2 port register sets and only reset and enumerate one port register set. If the USB3 port reset is unsuccessful, you can then allow the USB2 port reset. If the USB3 port reset is successful, you need to know not to try the USB2 port reset.

The code listed below will show how to pair up the port register sets along with setting any non-pairing USB2 port register sets active.

Remember that a controller may have a USB2 only socket that is not USB3 capable, but a controller must have a USB2 port register set for every USB3 port register set. Therefore, the code below will mark all USB3 port register sets and any non-paired USB2 port register sets active and all of the paired USB2 port register sets inactive, as the default state.

See Listings 14-1 and 14-2 in Chapter 14 on retrieving the protocol type of a port register set. Please note that the listing below assumes you have already retrieved the port protocol type and stored it in a structure as defined.

Listing 9-2: xHCI Paired Protocol

```
// pair up each USB3 port with it's companion USB2 port
for (i=0; i< num_ports; i++) {
  for (k=0; k<num_ports; k++) {
    if ((port_info[k].offset == port_info[i].offset) &&
       ((port_info[k].flags & PROTO_INFO) !=
        (port_info[i].flags & PROTO_INFO))) {
          port_info[i].other_port_num = k;
          port_info[i].flags |= PROTO_HAS_PAIR;
          port_info[k].other_port_num = i;
          port_info[k].flags |= PROTO_HAS_PAIR;
    }
  }
}

// mark all USB3 ports and any USB2 only ports as active,
// marking inactive any USB2 ports that have a USB3 companion
for (i=0; i< num_ports; i++) {
  if (!(port_info[i].flags & PROTO_HAS_PAIR) ||
      ((port_info[i].flags & PROTO_HAS_PAIR) && IS_USB3_PORT(i)))
        port_info[i].flags |= PROTO_ACTIVE;
}
```

The structure and #defines used above are described as the following:

```
struct PORT_INFO {
  bit8u  flags;          // port_info flags
  bit8u  other_port_num;  // 0 based offset to other speed port
  bit8u  offset;     // offset of this port within this protocol
  bit8u  reserved;   // for dword alignment
};

// flags
#define PROTO_INFO      (1<<0) // bit 0: USB3, else USB2
#define PROTO_HSO       (1<<1) // bit 1: USB 2 and HS Only
#define PROTO_HAS_PAIR  (1<<2) // bit 2: has a corresponding port
                               // (is a USB3 and has USB2 port)
                               // clear: doesn't have a
                               //  corresponding port (is a USB2
                               //  port without a USB3)
#define PROTO_ACTIVE    (1<<3) // is the active port of the pair

#define IS_USB3_PORT(x)
                    ((port_info[(x)].flags & PROTO_INFO) == 1)
```

The code in Listing 9-2 assumes that you have already initialized the PORT_INFO structure with the flags and offset members assigned. Again, see Chapter 14 for more information on this.

Resetting and Enabling the Port

The port's register set contains a Port Speed field to tell us which speed the attached device is. However, this field is invalid on the USB2 protocol port register set until after the reset, but is valid on the USB3 port register set before the reset, if a super-speed device is attached. Therefore, we must reset the port before we can use this speed field to determine which port register set to use, the USB3 port register set or the associated USB2 port register set.

Before resetting the port, the HcHalted bit in the USBSTS register must be zero. Since you cannot write a zero to this bit, you must start the schedule by setting the Run/Stop bit in the USBCMD register, assuming you have at least a valid empty schedule.

There is not much difference in resetting a USB2 protocol port than resetting a USB3 protocol port. Using bit 4 is a Hot Reset, while using bit 31 would be a Warm Reset and is only supported on USB3 protocol ports. The difference between the two is a Warm Reset, on a USB3 protocol port, will force the link hardware to enter the Rx.Detect state, indicating that the link detection should be performed again.

When resetting either protocol's port, do a Hot Reset, setting bit 4. The port will reset, then if successful, it will set bit 21, and will show the connection status and enable the port. Once you reset the port, check to see if the connection and enabled status is set. If so, you have a device attached that corresponds to the protocol of that port. For example, if you are resetting a USB2 protocol port register set, and it is successful, you will have a USB 2.0 device attached, either a low-, full-, or high speed device. If you are resetting a USB3 protocol port register set, and it is successful, you have a super-speed device attached. Your driver can not enable a port by writing a one to the Port Enabled/Disabled bit. The controller will only set this bit if a successful reset occurred and a device is attached.

A Hot reset can take as little as 1mS, while a Warm Reset may take a bit longer, in some cases it can take up to 250mS. You can poll bit 21 to see when the Hot Reset is complete. You must use bit 4 on all USB2 protocol port register sets, but may use either bit 4 or bit 31 on a USB3 protocol port register set. Just remember that using a Warm Reset will take longer to perform than a Hot Reset, and you must check bit 19 for a Warm Reset completion.

When checking to see if the port has completed the reset, you must check bit 21 to be set. You cannot assume that the reset is complete when the controller clears bit 4. Clearing bit 4 is only one step of the reset, and does not have to be the last step.

A MSD I have takes less than a millisecond to Hot Reset while the same device takes almost 87mS to Warm Reset.

The FYSOS xHCI driver uses something similar to the snippet of code below to reset a port.

Listing 9-3: xHCI Port Reset

```
int ret = UNSUCCESSFUL_RESET;

// we might need to power the port first.
if ((xhci_read_oper_reg(base, HCPortOff) & (1<<9)) == 0) {
  xhci_write_oper_reg(root->base, HCPortOff, (1<<9));
  mdelay(20);
  if ((xhci_read_oper_reg(base, HCPortOff) & (1<<9)) == 0)
    return BADRESET;  // return bad reset - port would not power
}

// make sure that the status change bits are clear first.
xhci_write_oper_reg(base, HCPortOff, (1<<9) | xHC_CHANGE_BITS);
```

```
// set bit 4 (USB2) or 31 (USB3) to reset the port
bit32u reset_bit =
            (protocol == xHCI_PROTO_USB3) ? (1<<31) : (1<<4);
bit32u reset_chng =
            (protocol == xHCI_PROTO_USB3) ? (1<<19) : (1<<21);

xhci_write_oper_reg(root->base, HCPortOff, (1<<9) | reset_bit);

// wait for bit 19 or bit 21 to be set
int timeout = 500;
while (timeout) {
  if (xhci_read_oper_reg(base, HCPortOff) & reset_chng)
    break;
  timeout--;
  mdelay(1);
}

// if we didn't time-out
if (timeout > 0) {
  // reset recovery time
  mdelay(USB_TRHRSI);

  // if after the reset, the enable bit is non zero,
  //  there was a successful reset/enable
  if (xhci_read_oper_reg(base, HCPortOff) & (1<<1)) {
    // clear the status change bit(s)
    xhci_write_oper_reg(base, HCPortOff,
                                (1<<9) | xHC_CHANGE_BITS);
    // success
    ret = SUCCESS_RESET;
  }
}

// if error resetting USB3 protocol, deactivate this port and
// activate the paired USB2 port.  It will be paired since
// all USB3 ports must be USB2 compatible.
if ((ret != SUCCESS_RESET) && IS_USB3_PORT(port_num)) {
  port_info[port_num].flags &= ~PROTO_ACTIVE;
  port_info[port_info[port_num].other_port_num].flags
                                        |= PROTO_ACTIVE;
}

return ret;
```

Be sure to clear bit 19 and 21 before the reset. If you do not clear these bits first, after setting the reset bit, then checking for these bits to be set, you may get a premature indication of successful reset since the controller may not have cleared them yet as part of the reset.

Detecting multiple ports on a root hub

To get the count of downstream root hub ports, simply read the MaxPorts field from the HCSPARAMS1 register, by reading the dword and masking off all but the most significant byte. Remember that this is the count of register sets, not the actual count of physical ports, also called sockets.

Also remember that most devices will use either the USB2 protocol port set OR the USB3 protocol port set, not both. However, a USB3 external hub will use them both.

Chapter 10 – The xHCI Schedule

By this chapter, you should know how to detect the host controller through the PCI configuration space and physical detection at the port I/O level. You should also know how to detect that a device is present on one of the ports on the host controller, also known as the root hub. You should also now know how to point the root hub to our schedule and start the schedule. Don't worry, we haven't discussed how to set up the schedule yet, just how to point to it.

If you are comfortable with this up to this point, we will start on the actual xHCI communication protocol and how to detect the device or devices already attached. We also discuss the different data structures of the controller and the schedule. If you are not ready to continue on, please go back and study the previous chapters.

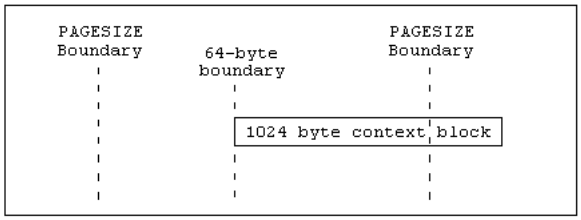
In this chapter, I will assume that all devices are attached to the root hub and the attached device is not an external hub. Discussion of external hubs attached to an xHCI controller are discussed in Chapter 21.

Memory Usage and Alignment

The xHCI requires blocks of system memory to function correctly. These blocks of memory have certain requirements. For example, they must be aligned on a certain byte boundary. Also, a given block of memory must not cross the controller's boundary limits. If the controller has a page size boundary of 4096, that block of memory may or may not cross a 4096 boundary, depending on the type of memory block.

For example, the controller may need a 1024[5] byte block of memory for each Device Context entry. This block of memory must also be 64-byte aligned. See the figure below.

Figure 10-1: Memory Boundaries



Even though the 1024-byte block of memory within that figure is aligned on a 64-byte boundary, it is not legal since it crosses, or spans the controller's page size boundary. When allocating memory for the controller, make sure to align the memory so that it does not span the controller's boundary requirements. The table below shows the requirements as the maximum amount of memory a block will use, the boundary span, and alignment requirements. The PAGESIZE value is calculated from the controller's PageSize register.

Table 10-1: Data Structure Memory Requirements

xHCI controller: Data Structure Memory Requirements			
Data Structure Name	Max Size	Boundary	Alignment
Device Context Index	2048	PAGESIZE	64 bytes
Device Context	2048	PAGESIZE	64 bytes
Input Control Context	64	PAGESIZE	64 bytes
Slot Context	64	PAGESIZE	32 bytes
Endpoint Context	64	PAGESIZE	32 bytes
Stream Context	16	PAGESIZE	16 bytes
Stream Array (Linear)	1 Meg	None	16 bytes
Stream Array (Pri/Sec)	4k/64k	64k	16 bytes
Transfer Ring Segments	64k	64k	16 bytes
Command Ring Segments	64k	64k	64 bytes
Event Ring Segments	64k	64k	64 bytes
Event Ring Segment Table	512k	None	64 bytes
Scratchpad Buffer Array	248	PAGESIZE	64 bytes
Scratchpad Buffers	PAGESIZE	PAGESIZE	PAGESIZE

For an example, take the Device Context structure. It is no more than 2048 bytes in size and must not span a PAGESIZE boundary. If the PAGESIZE is 4096, the Device Context structure must start no higher than offset 2048 within that 4096 page of memory. Command Ring Segments may be up to 64k in size and must not span a 64k boundary. Therefore, if a Command Ring Segment is 64k in size, it must have an alignment of 64k.

The xHCI Device Context Area

The xHCI needs an allocated amount of memory for its Device Context storage area. The physical address of this memory block is stored in the controller's Device Context Base Address Array Pointer (DCBAAP) Register. This memory is called the Device Context Index memory block and is at most 256 64-bit address pointers. Except for the first entry, each entry points to a Device Context memory block.

Since the MaxSlots entry in the HCSPARAMS1 register can hold a max value of 255, one less than the 256 noted above, the xHCI controller uses the first entry in the Device Context Index block for a pointer to the Scratchpad buffers.

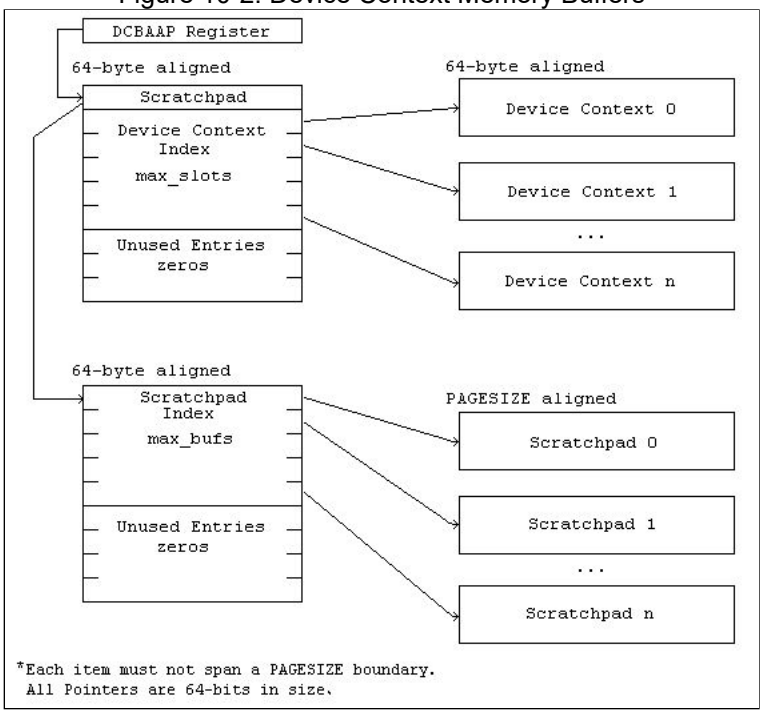
I think the reason the first one is used for the Scratchpad buffers, is that most of the other items in the controller are one-based instead of zero-based. Therefore, they use the 1st or index zero of this block for something else.

If the controller requires scratchpad buffers, as defined in the HCSPARAMS2 register, the first pointer is the 64-byte aligned physical address of the Scratchpad Buffer Index Array.

The Scratchpad Buffer Index Array is at most 31 (for version 1.0) or 1,023 (for version 1.10) 64-bit physical address pointers to PAGESIZE sized buffers. These buffers must be PAGESIZE aligned and will be used by the controller for things like saving and restoring the state of the controller. The count of pointers is from the HCSPARAMS2 register, or the count of scratchpads the controller requires.

The remaining maximum of 255 entries in the Device Context Index are used for Device Context blocks. These entries point to individual blocks of either 1,024 or 2,048 bytes in size, depending on the Context Size bit of the HCCPARAMS1 register. You specify in the Configuration register how many of these blocks the controller can use. Any unused entries in the Index should be zeros. See the figure below for an example.

Figure 10-2: Device Context Memory Buffers



In the figure above, the DCBAAP register points to a 64-byte aligned 2048 byte[6] memory block containing the Scratchpad pointer, a max_slots count of 64-bit Device Context pointers, and the remaining of the memory block as zeros.

Your driver may decide not to use all of the max_slots Device Context entries. Therefore, you may not need all of them. However, if you have allocated and configured to use, for example, 8 slots and decide to modify the controller to only use 4, you must make sure that the last four slots are disabled before you modify the controller's configuration.

Your driver will need to allocate a max_bufs count of PAGESIZE sized, PAGESIZE aligned memory blocks. These buffers are used when the host requires its own memory usage or goes to the suspend state, in other words, the host would like to cut power to the controller. When this happens, your driver will save the controller's state, using a bit in the Command register. When the host decides to resume, your driver will then restore the state, again using another bit in the Command register. Please be sure to check the Scratchpad Restore bit in the HCSPARAMS2 register. If this bit is zero, the controller does not use the Scratchpad buffers to save and restore the controller state.

❓ As of this writing, there are a few more xHCI controllers on the market. The NEC/Renesas uPD720202 supports the Save and Restore feature.

The xHCI Device Context Data Structure

The xHCI controller uses Device Context data blocks to store information about each device. Each device will require a single Device Context. The first structure in a Device Context is for the device itself and holds information such as the speed and device address, things that are common to all endpoints of the device. Then following the first structure is up to 31 more 32- or 64-byte entries, per Device Context Data block.

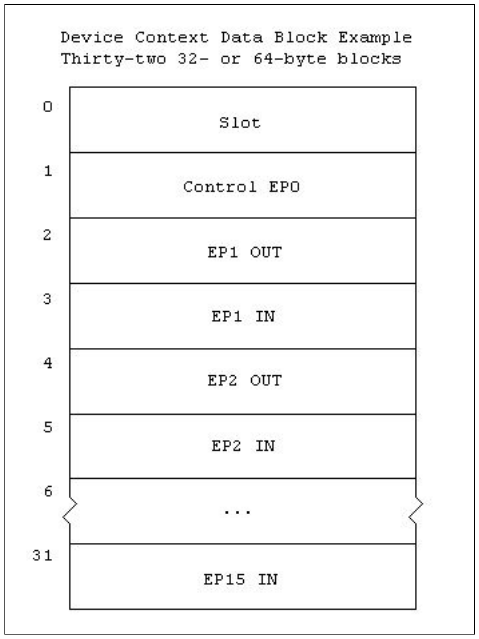
These data blocks are either 1,024 or 2,048 bytes in size depending on the Context Size bit in the HCCPARAMS1 register. The Device Context Data block contains a single Slot Context block, the Control Endpoint Context, and then up to 30 Endpoint Context blocks, or 15 pairs of endpoints.

> The only thing I can think of why the xHCI specification allows for 64 byte context sizes, is that maybe in future implementations there might be the need for more members, obviously. However, for now, they want to save space by using only 32 byte entries, giving the option to have either.

> Since 64 byte context sizes may not be fully used for a while, for the purpose of this book, I will assume all context sizes are 32 bytes from here on out. However, don't assume 32-byte sizes. A few of the controllers I have use 64-byte context structures, though the last 32 bytes are probably unused.

The Slot Context Data Structure contains items that are needed for each transfer such as Device Address and Device Speed. See Figure 10-3 below.

Figure 10-3: Device Context Data Block Example



The first entry, the Slot, holds the common information about the device, while the second entry is the Control Endpoint Slot, and then 15 pairs of Endpoints.

The structure of the first entry is shown below. At initialization/allocation time, system software should initialize all fields to zero in this data structure and any Endpoint Contexts that follow it. This memory is owned by system software until the Address Device command is sent, which initializes this memory. Once the device is addressed, this memory is now owned by the

controller and is considered read-only by system software until the Disable Slot command is sent.

Figure 10-4: xHCI Slot Context Format

31 27	26	25	24	23 20	19 18 17 16 15 8 7 0		
Context Entries	Hub	MTT	RZ	Speed	Route String		
Number of Ports				Root Hub Port Num		Max Exit Latency	
Interrupt Target				RsvdZ	TTT	TT Port Num	TT Hub Slot ID
State	RsvdZ						Device Address
Reserved Opaque[1]							
Reserved Opaque[1]							
Reserved Opaque[1]							
Reserved Opaque[1]							
[1]Used by controller. Do not modify after enabling slot.							

Each of the above eight dwords are explained below.

Table 10-2: xHCI Slot Context: DWORD 0

xHCI controller: Slot Context: DWORD 0	
Bit	Description
19:0	Route String
23:20	Speed
24	Reserved and Zero'd
25	Multi-TT
26	Hub
31:27	Count of Context Entries

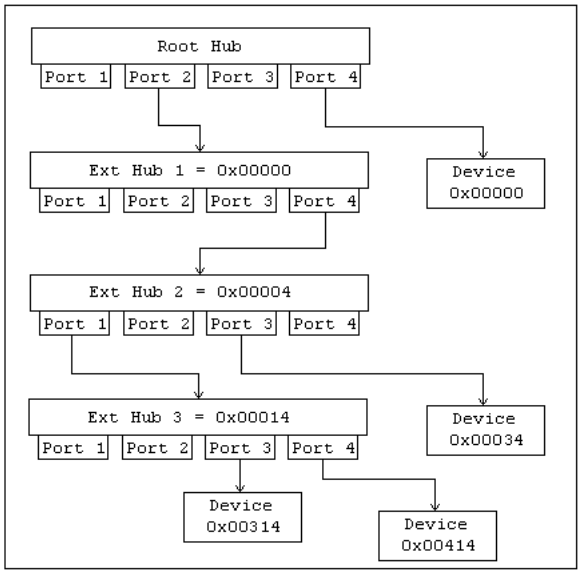
The Route String field is used by hubs to route to the correct port. It is a field of five 4-bit fields.

Table 10-3: Route String

19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
Hub@Tier5	Hub@Tier4	Hub@Tier3	Hub@Tier2	Hub@Tier1

The Route String field is not used for Root Hubs and should be set to all zeros. However, in slots that are used with external hubs, this field is used to point to the port that this hub is on. The Controller uses the Root Hub Port Num field and the Route String field to transverse through the multiple external hubs. See the figure below for an explanation of how this field is built.

Figure 10-5: Route String Example



The Root Hub is not included in the calculation of the Route String. The Route String is calculated and shown below as, RS is Route String, Parent is the device's parent hub, port_num is the 1 based port number of the parent this device is plugged in to, and hub_level is the 1 based hub number, or tier, this device is plugged in to:

RS = Parent->RS | (Parent->port_num << ((hub_level - 1) * 4))

In Figure 10-5 above, the device attached to the last hub's port 3 has a Route String value of 0x00314. If we read it from left to right, this means that to get back to the root hub it will go through port 3, then port 1, then port 4, and finally to root hub port 2. The controller uses the Root Hub Port Num field in the Slot Context to retrieve the last port number in the sequence above.

Please note that since there are only 5 fields in the Route String field, this assumes that you will not have more than 5 external hubs before a device. Also, if an external hub has more than 15 ports and a device is plugged into a port number higher then 15, you must use a value of 15 (0Fh) in that field's position.

Besides, who would plug more than 5 hubs in series, then plug a device into port number 16 in the last hub? That's right, me. I have to try it. □ (If I can ever find a hub that has more than 15 sockets on it.)

Back to the Context Structure. The Speed field is used to indicate the speed of the device. Refer to Chapter 9 Table 9-28 for valid values for this field.

The Multi-TT bit is used with high-speed hubs.

The Hub bit is set by system software if the device attached is a hub.

Note that you won't know if the device is a Hub until you request the device's descriptor. Therefore, the bits in the slot context that refer to hubs do not need to be set until after you have retrieved the hub's descriptor.

The Context Entries field is the count of valid Endpoint Context structures following this Slot Context. This field should be one if all you have is the Control Endpoint defined.

Table 10-4: xHCI Slot Context: DWORD 1

xHCI controller: Slot Context: DWORD 1	
Bit	Description
15:0	Max Exit Latency
23:16	Root Hub Port Number
31:24	Number of Ports

The Max Exit Latency field is given in microseconds and specifies the longest time it may take to wake up all of the links in the path to the device.

The Root Hub Port Number field is the root port number that this device is attached to, either directly or through hubs. Remember that it is a one-based number.

The Number of Ports field is written to by system software as the number of down stream ports on the attached hub. If the device is not a hub, this field should be zero.

Table 10-5: xHCI Slot Context: DWORD 2

xHCI controller: Slot Context: DWORD 2	
Bit	Description
7:0	TT Hub Slot ID
15:8	TT Port Number
17:16	TT Think Time
21:18	Reserved and Zero'd
31:22	Interrupter Target

The TT Hub Slot ID field is the Slot ID of the Full Speed hub this device is attached to if this device is a full- or low-speed device. If this device is not attached to a full-speed hub, then this field should be zero.

The TT Port Number field is the down-stream port number of a full-speed hub this device is attached to if this device is a full- or low-speed device. If this device is not attached to a full-speed hub, then this field should be zero.

The TT Think Time field indicates the full-speed bit times it takes a full-speed hub to proceed to the next transaction. Refer to the hub's Hub Descriptor for more information. If the attached device is not a full-speed hub, then this field should be zero. Valid values are listed below.

Table 10-6: xHCI Slot Context: DWORD 2: TT Think Time

Value	Description
0	TT Requires at most 8 Full Speed bit times.
1	TT Requires at most 16 Full Speed bit times.
2	TT Requires at most 24 Full Speed bit times.
3	TT Requires at most 32 Full Speed bit times.

The Interrupter Target field is the index of the Interrupter that will receive Bandwidth Request Events, Device Notification Events, Ring Underrun or Ring Overrun notifications, generated by this slot.

Table 10-7: xHCI Slot Context: DWORD 3

xHCI controller: Slot Context: DWORD 3	
Bit	Description
7:0	Device Address
26:8	Reserved and Zero'd
31:27	Slot State

The Device Address field is set by the controller to the address assigned by the controller upon a successful completion of a Set Address command.

Depending on the internal organization of the controller, the Device Address may not be unique across all Device Context Structures. However, the combination of Device Address and Root Hub Port Number will always be unique throughout all devices connected to this controller.

The Slot State field is updated by the controller when the device transitions from one state to another. The allowed states are show below.

Table 10-8: xHCI Slot Context: DWORD 3: Slot State

Value	Description
0	Disabled or Enabled
1	Default
2	Addressed
3	Configured
31-4	Reserved

The remaining DWORDs are used by the controller and should be initially set to zero, then remain untouched by system software. If the Context Size bit in the HCCPARAMS1 register is set, then there are eight additional DWORDs in this structure and they too must be initially set to zero and remain untouched.

The xHCI Endpoint Context Data Structure

The xHCI controller uses Endpoint Context data blocks to control each endpoint transaction. For example, the first Endpoint Context structure is the Control Endpoint structure.

The Endpoint Context data blocks follow the Device Context data block they are designed to go with and can be from 1 to 31 entries each depending on the Context Entries field in the Device Context data block.

Figure 10-6: xHCI Endpoint Context Format

31 24	23 16	15	14 10	9	8 7 6 5 4 3	2 1 0
Max ESIT Payload (High)	Interval	LSA	MaxP Streams	Mult	Reserved	EP State
Max Packet Size		Max Burst Size		H	R Type	CErr R
TR Dequeue Pointer Low						Resvd CS
TR Dequeue Pointer Hi						
Max ESIT Payload (Low)		Average TRB Length				
Reserved*						
Reserved*						
Reserved*						
*Used by controller. Do not modify after enabling slot.						

Each of these DWORDs is explained below.

Table 10-9: xHCI Endpoint Context: DWORD 0

xHCI controller: Endpoint Context: DWORD 0	
Bit	Description
2:0	Endpoint State
7:3	Reserved and Zero'd
9:8	Mult
14:10	Max Primary Streams
15	Linear Stream Array
23:16	Interval
31:24	Max Endpoint Service Time Interval Payload (High)[1]
[1]xHCI Version 1.10 only	

The Endpoint State field indicates the current state of the endpoint. Valid values are listed here.

Table 10-10: xHCI Endpoint Context: DWORD 0: EP State

Value	Description
0	Disabled and not operational.
1	Running and may be waiting for a Doorbell or could be processing TD's.
2	Halted due to a halt condition and needs a Reset Endpoint command to be placed in the stopped state.
3	Stopped and not running due to a Reset Endpoint command or recovering from a Halted state.
4	Error due to a TRB error.
7-5	Reserved

The Mult field indicates the maximum supported number of bursts within an interval. Valid values are 0, 1, 2, and 3, where number of bursts are one more than value given. For example, a value of 2 allows 3 bursts. This field should be zero for all endpoints other than super-speed Isochronous endpoints.

The Max Primary Streams field indicates the maximum number of primary streams this endpoint allows. If this value is zero, then the TR Dequeue Pointer points to a Transfer Ring, else it points to a Primary Stream Context Array.

The Linear Stream Array bit, when set, disables the Secondary Stream Arrays and the Stream ID shall be a linear index into the Primary Stream Array.

All of the information in this book assumes Transfer Rings and gives little or no more detail about Streams.

The Interval field indicates the period between consecutive request to an endpoint to send or receive data expressed in 125 microsecond increments as used with the following formula:

$$period = 125uS * 2^{interval}$$

For example, an interval value of 5 would give a period of 4000uS.

$$125uS * 2^5 = 4000$$

Valid values are 3 to 11 for low- and full-speed endpoints, and 0 to 15 for high- and super-speed endpoints.

Table 10-11: xHCI Endpoint Context: DWORD 1

xHCI controller: Endpoint Context: DWORD 1	
Bit	Description
0	Reserved and Zero'd
2:1	Error Count
5:3	Endpoint Type
6	Reserved and Zero'd
7	Host Initiate Disable
15:8	Max Burst Size
31:16	Max Packet Size

The Error Count field is set by system software and is a count down of allowed errors before the controller stops trying to process the TD. For example, if you set this field to 3 and an error occurs while processing this endpoint, the controller will decrement this field and try again. When the count goes from one to zero, the controller will then halt the endpoint. Setting this field to zero will process the endpoint indefinitely.

Note that the xHCI processes the Error Count field differently than the EHCI controller does.

The Endpoint Type field is set by system software to the type of endpoint this context will be. Valid values are as follows.

Table 10-12: xHCI Endpoint Context: DWORD 1: EP Type

Value	Description
0	Not valid.
1	Isochronous OUT

2	Bulk OUT
3	Interrupt OUT
4	Control
5	Isochronous IN
6	Bulk IN
7	Interrupt IN

The Host Initiate Disable bit, when clear, enables normal stream operation. You should have this bit set for all examples here.

The Max Burst Size field is set by system software to indicate the maximum number of consecutive transactions that should be processed during the current scheduled event. This value is zero based, therefore, a value of zero in this field represents a burst size of 1, a value of 1 represents a burst size of 2, and so on.

The Max Packet Size field is set by system software to the max packet size for the endpoint. This can be retrieved from the device's Endpoint Descriptor.

Note that even though Super-Speed devices may have a Base 2 Multiple type Max Packet Size value in the Endpoint Descriptor, the field here is the actual count of bytes.

Table 10-13: xHCI Endpoint Context: DWORDs 2 & 3

xHCI controller: Endpoint Context: DWORDs 2 & 3	
Bit	Description
0	Dequeue Cycle State
3:1	Reserved and Zero'd
63:4	TR Dequeue Pointer

The Dequeue Cycle State bit is set to the current cycle bit value in the item pointed to by the TR Dequeue Pointer. The value of the cycle bit is explained in a later section.

The TR Dequeue Pointer field is the high order bits of the 64-bit physical address of the Transfer Ring or Stream Context Array associated with this Endpoint. If MaxPStreams is zero, then this pointer should point to a Transfer Ring.

Table 10-14: xHCI Endpoint Context: DWORD 4

xHCI controller: Endpoint Context: DWORD 4	
Bit	Description
15:0	Average TRB Length
31:16	Max Endpoint Service Time Interval Payload (Low)

The Average TRB Length field is set by system software to the average length of the TRBs processed by this endpoint. This field should not be set to zero.

The Max Endpoint Service Time Interval Payload field, or Max ESIT Payload, when combined with the same named field in the first dword, is set by system software and used with Interrupt and Isochronous endpoints and indicates the maximum total of bytes the controller will transfer for this endpoint's allotted time interval.

The remaining DWORDs are used by the controller and should be initially set to zero, then remain untouched by system software. If the Context Size bit in the HCCPARAMS1 register is set, then there are eight addition DWORDs in this structure and they too must be initially set to zero and remain untouched.

The xHCI Transfer Descriptor

The Transfer Descriptor is simply a set, or chain of Transfer Request Blocks. A TD can be one or more TRB's depending on the intended use. If more than one TRB is used for a single TD, the Chain bit should be set in all but the last TRB. See the next section for the description of Transfer Request Blocks.

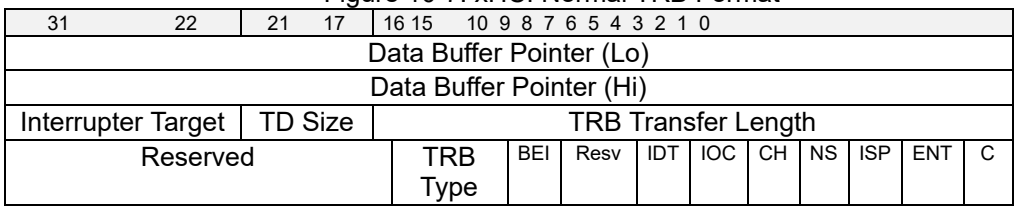
The xHCI Transfer Request Block

The Transfer Request Block, or TRB, is the workhorse of the xHCI schedule. TRB's get placed on Command Rings, Event Rings, Transfer Rings, and other rings, and this is where the data transfer takes place.

When you place a command in the Command Ring, you use a Command TRB, and ring the Command Doorbell. Then the controller places a Command Event TRB on the Event Ring and fires the interrupt. It is very similar for other TRB types.

The Normal TRB is used for mostly Bulk and Interrupt transfers and has the format below.

Figure 10-7: xHCI Normal TRB Format



Each of the DWORDs is explained next.

Table 10-15: xHCI Normal TRB Format: DWORD 0 & 1

xHCI controller: Normal TRB Format: DWORD 0 & 1	
Bit	Description
63:0	Data Buffer Pointer

The Data Buffer Pointer field is a 64-bit physical address of the data for this transaction or it is 8 bytes of immediate data. See the definition of the Immediate Data bit in Table 10-17 below for more on immediate data allowed in this field.

Table 10-16: xHCI Normal TRB Format: DWORD 2

xHCI controller: Normal TRB Format: DWORD 2	
Bit	Description
16:0	TRB Transfer Length
21:17	TD Size
31:22	Interrupter Target

The TRB Transfer Length field indicates the length of the data sent or received. When used as an OUT TRB, this is the length of data to send to the device. If this length is zero, the controller sends a zero size packet and ignores the pointer, or data, in the Data Buffer Pointer field. If this is an IN TRB, this is the amount of data received and is pointed to by the Data Buffer Pointer. Valid values are 0 to 65536.

For an IN TRB's, the Buffer Pointer will always be a pointer even if the length is 8 bytes or less. i.e.: the Immediate Data bit is not allowed for IN TRB's.

The TD Size field is used to indicate the amount of packets remaining in the TD. The Interrupter Target field is used to indicate the Interrupter that will receive events generated by the controller for this TRB. Valid values are zero to MaxIntrs minus 1.

Table 10-17: xHCI Normal TRB Format: DWORD 3

xHCI controller: Normal TRB Format: DWORD 3	
Bit	Description

0	Cycle Bit
1	Evaluate Next TRB
2	Interrupt on Short Packet
3	No Snoop
4	Chain Bit
5	Interrupt on Completion
6	Immediate Data
8:7	Reserved and Zero'd
9	Block Event Interrupt
15:10	TRB Type
31:16	Reserved and Zero'd

The Cycle Bit field is used to mark the Enqueue Pointer of the Transfer Ring. See the section on Transfer Rings for more on the use of this bit.

The Evaluate Next TRB bit tells the controller to evaluate the next TRB before updating the endpoint's state.

The Interrupt on Short Packet bit tells the controller to stop executing this TD, generate a Transfer Event TRB, and advance to the next TD if a short packet is detected. A description of what happens on a short packet is described in Chapter 14.

The No Snoop bit is used with the PCIe hardware and won't be discussed here.

The Chain Bit bit is used to tell the controller that this TRB is associated with the next TRB in the TD. A TD, or Transfer Descriptor is defined as one or more TRB's in a chain. This bit should be zero for the last TRB in the chain.

The Interrupt on Completion bit tells the controller to generate a Transfer Event TRB and asserts an interrupt after completing this TRB. This bit should only be set on a Status Stage TD, when Control TRB's are used.

The Immediate Data bit tells the controller that the 8 bytes in the Data Buffer Pointer field is actually eight or less bytes of packet-data instead of a pointer to that data. If your TD only needs to send eight or less bytes, you don't need to set up a data buffer.

To use this bit:

- This TRB must be the only one in the TD, a chain with only one TRB.
- This must be an OUT TRB.
- The size of the data being sent must be 0 to 8 bytes.
- The TRB Transfer Length field must be less than or equal to eight.

The Block Event Interrupt bit, when set and the IOC bit is set, tells the controller not to generate the intended interrupt for this TRB, though still generates the Transfer Event TRB.

The TRB Type field is set by system software to the type of TRB this will be. See Appendix G for a list of allowed types.

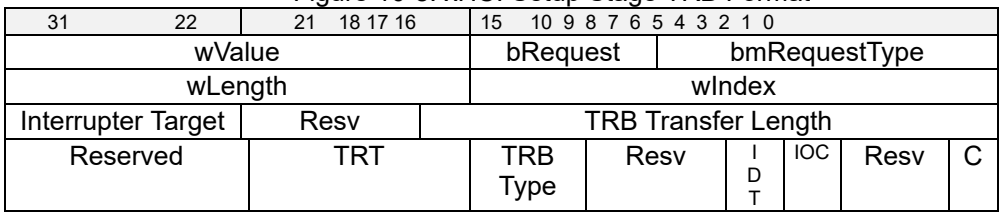
The xHCI Control Transfer Request Block

To send and receive Control type TD's, you must use the Control TRB types, which are the Setup Stage TRB, optional Data Stage TRB's, and the Status Stage TRB. These three TRB's are described within the next few sections.

The Setup Stage Transfer Request Block

When first enumerating a device, as with any controller, you must send control packets to retrieve the device descriptor and other control items. The first packet is the Setup Packet TRB.

Figure 10-8: xHCI Setup Stage TRB Format



Each of the DWORDs is explained below.

Table 10-18: xHCI Setup Stage TRB Format: DWORD 0

xHCI controller: Setup Stage TRB Format: DWORD 0	
Bit	Description
7:0	bmRequestType
15:8	bRequest
31:16	wValue

The bmRequestType, bRequeset, and wValue fields are set to the values of the specific USB request you are requesting. See previous controller type protocols or the USB specification for these values.

Table 10-19: xHCI Setup Stage TRB Format: DWORD 1

xHCI controller: Setup Stage TRB Format: DWORD 1	
Bit	Description
15:0	Windex
31:16	Wlength

The wIndex and wLength fields are also set to the values of the specific USB request you are requesting.

Table 10-20: xHCI Setup Stage TRB Format: DWORD 2

xHCI controller: Setup Stage TRB Format: DWORD 2	
Bit	Description
16:0	TRB Transfer Length
21:17	Reserved and Zero'd
31:22	Interrupt Target

The TRB Transfer Length field should always be set to 8. You are transferring 8 bytes as immediate data. Remember to also set the Immediate Data bit.
The Interrupt Target field is set by system software to the interrupter you would like to receive the Event TRB's. Valid values are 0 to *MaxIntrs* minus 1.

Table 10-21: xHCI Setup Stage TRB Format: DWORD 3

xHCI controller: Setup Stage TRB Format: DWORD 3	
Bit	Description
0	Cycle Bit
4:1	Reserved and Zero'd

5	Interrupt On Completion
6	Immediate Data
9:7	Reserved and Zero'd
15:10	TRB Type
17:16	Transfer Type
31:18	Reserved and Zero'd

The Cycle Bit field is used to mark the Enqueue point of the Transfer Ring. See a later section for more on this subject.

The Interrupt on Completion bit tells the controller to generate a Transfer Event TRB and asserts an interrupt after completing this TRB. This bit should only be set on a Status Stage TRB.

The Immediate Data bit should always be set for Setup TRB's. This indicates that there are 8 bytes of immediate data as the first 64-bits in this TRB.

The TRB Type field should be set to *Setup Stage TRB*. See Appendix G for a list of types.

The Transfer Type field indicates the type and direction of this control transfer.

Table 10-22: Control Transfer Type

Value	Description
0	No Data Stage
1	Reserved
2	OUT Data Stage
3	IN Data Stage

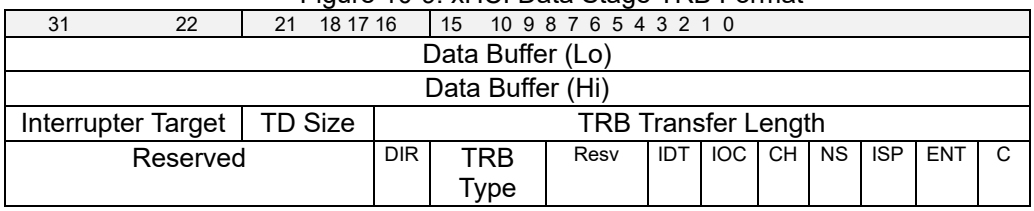
When set to zero, this TRB is a zero length TD. When set to 2, the Data Stage TRB is an OUT TRB, and when set to 3, the Data Stage TRB is an IN TRB.

The Data Stage Transfer Request Block

After the Setup Stage TRB, the request may or may not need a data stage. If the request does not need a data stage, then a Status Stage TRB would follow. See the next section for Status Stage TRB's.

Data Stage TRB's can be one or more TRB's to make up the TD to transfer the requested data. The following section describes a Data Stage TRB.

Figure 10-9: xHCI Data Stage TRB Format



Each of the DWORDs is explained next.

Table 10-23: xHCI Data Stage TRB Format: DWORD 0 & 1

xHCI controller: Data Stage TRB Format: DWORD 0 & 1	
Bit	Description
63:0	Data Buffer Pointer

The 64-bit Data Buffer field is used as the 64-bit physical address to the memory buffer to use for the transfer. If the IDT bit in this TRB is set and this is an OUT TD, this field is up to 8 immediate bytes used for transfer.

The 64-bit address may be byte aligned though an alignment of 64 bytes or 128 bytes will provide better performance.

Table 10-24: xHCI Data Stage TRB Format: DWORD 2

xHCI controller: Data Stage TRB Format: DWORD 2	
Bit	Description
16:0	TRB Transfer Length
21:17	TD Size
31:22	Interrupt Target

The TRB Transfer Length field, when used as an OUT TRB, is the size of the data to transfer. When used as an IN TRB, system software should set this field to the amount of bytes expected to receive by this TRB's transfer. Valid values for this field are 0 to 65536.

The TD Size field is used to indicate the amount of packets remaining in the TD.
The Interrupt Target field is set by system software to the Interrupter you would like to receive the Event TRB's. Valid values are zero to MaxIntrs minus 1.

Table 10-25: xHCI Data Stage TRB Format: DWORD 3

xHCI controller: Data Stage TRB Format: DWORD 3	
Bit	Description
0	Cycle Bit
1	Evaluate Next TRB
2	Interrupt on Short Packet
3	No Snoop
4	Chain Bit

5	Interrupt on Completion
6	Immediate Data
9:7	Reserved and Zero'd
15:10	TRB Type
16	Direction
31:17	Reserved and Zero'd

The Cycle Bit bit is used to mark the Enqueue Pointer of the Transfer Ring. See the section on Transfer Rings for more on the use of this bit.

The Evaluate Next TRB bit tells the controller to evaluate the next TRB before updating the endpoint's state.

The Interrupt on Short Packet bit tells the controller to stop executing this TD, generate a Transfer Event TRB, and advance to the next TD if a short packet is received. A description of what happens on a short packet is described in Chapter 14.

The No Snoop bit is used with the PCIe hardware and won't be discussed here.

The Chain Bit field is used to tell the controller that this TRB is associated with the next TRB in the TD. A TD, or Transfer Descriptor is defined as one or more TRB's in a chain. This bit should be zero for the last TRB in the chain.

The Interrupt on Completion bit tells the controller to generate a Transfer Event TRB and asserts an interrupt after completing this TRB. This bit should only be set on a Status Stage TD.

The Immediate Data bit tells the controller that the 8 bytes in the Data Buffer Pointer field is actually eight or less bytes of packet data instead of a pointer to that data. If your TD only needs to send eight or less bytes, you don't need to set up a data buffer. See the previous section on the Normal TRB for use of this bit.

The TRB Type field is set by system software to the type of TRB this will be. See Appendix G for a list of allowed types.

The Direction bit, when set to 0, is an OUT packet. When set to 1, is an IN packet.

The Status Stage Transfer Request Block

After the Setup Stage TRB or optional Data Stage TRBs, you will need to send a Status Stage TRB. The Status Stage TRB is as follows.

Figure 10-10: xHCI Status Stage TRB Format

31 22	21 18 17 16	15 10 9 8 7 6 5 4 3 2 1 0
Reserved		
Reserved		
Interrupter Target	Reserved	

Reserved	DIR	TRB Type	Resv	IOC	CH	Resv	ENT	C

Each of the DWORDs is explained next.

Table 10-26: xHCI Status Stage TRB Format: DWORD 0 & 1

xHCI controller: Status Stage TRB Format: DWORD 0 & 1	
Bit	Description
63:0	Reserved and Zero'd

The first 64-bits should be zeros.

Table 10-27: xHCI Status Stage TRB Format: DWORD 2

xHCI controller: Status Stage TRB Format: DWORD 2	
Bit	Description
21:0	Reserved and Zero'd
31:22	Interrupt Target

The Interrupt Target field is set by system software to the Interrupter you would like to receive the Event TRB's. Valid values are zero to MaxIntrs minus 1.

Table 10-28: xHCI Status Stage TRB Format: DWORD 3

xHCI controller: Status Stage TRB Format: DWORD 3	
Bit	Description
0	Cycle Bit
1	Evaluate Next TRB
3:2	Reserved and Zero'd
4	Chain Bit
5	Interrupt on Completion
9:6	Reserved and Zero'd
15:10	TRB Type
16	Direction
31:17	Reserved and Zero'd

The Cycle Bit field is used to mark the Enqueue Pointer of the Transfer Ring. See the section on Transfer Rings for more on the use of this bit.

The Evaluate Next TRB bit tells the controller to evaluate the next TRB before updating the endpoint's state.

The Chain Bit field is used to tell the controller that this TRB is associated with the next TRB in the TD. A TD, or Transfer Descriptor is defined as one or more TRB's in a chain. This bit should be zero for the last TRB, which would be this Status TRB, in the chain.

The Interrupt on Completion bit tells the controller to generate a Transfer Event TRB and asserts an interrupt after completing this TRB. This bit should only be set on a Status Stage TRB, this TRB.

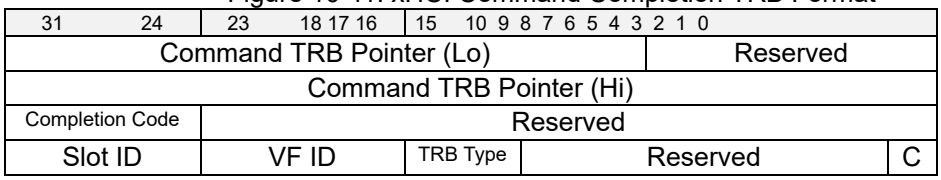
The TRB Type field is set by system software to the type of TRB this will be. See Appendix G for a list of allowed types.

The Direction bit, when set to 0, is an OUT packet. When set to 1, is an IN packet.

The Command Completion Transfer Request Block

When a Command TRB has been executed, and it requested the interrupt, a Command Completion TRB is generated in the Event Ring and has the following format.

Figure 10-11: xHCI Command Completion TRB Format



Each of the DWORDs is explained below.

Table 10-29: xHCI Command Completion TRB Format: DWORD 0 & 1

xHCI controller: Command Completion TRB Format: DWORD 0 & 1	
Bit	Description
3:0	Reserved and Zero'd
63:4	Command TRB Pointer

The Command TRB Pointer field, in most cases, contains a 16-byte aligned, physical address pointer to the TRB that generated this Event TRB. Since this address must be 16-byte aligned, the lower four bits are marked as Reserved and Zero'd.

Table 10-30: xHCI Command Completion TRB Format: DWORD 2

xHCI controller: Command Completion TRB Format: DWORD 2	
Bit	Description
23:0	Reserved and Zero'd
31:24	Completion Code

The Completion Code field is encoded with the completion status of the event. See Appendix G for a list of commands and their completion codes.

Table 10-31: xHCI Command Completion TRB Format: DWORD 3

xHCI controller: Command Completion TRB Format: DWORD 3	
Bit	Description
0	Cycle Bit
9:1	Reserved and Zero'd
15:10	TRB Type
23:16	VF ID
31:14	Slot ID

The Cycle Bit field is used to mark the Enqueue Pointer of the Event Ring. See the section on Transfer Rings for more on the use of this bit.

The TRB Type field is set by system software to the type of TRB this will be. See Appendix G for a list of allowed types.

The VF ID field is only used with Virtual Functions and will not be discussed here.

The Slot ID field, in general, is the ID of the slot associated with the command that generated the event. When the Enable Slot Command is given, the controller places the new Slot ID into this field. There are a few other exceptions. For example, the NOP command will put a value of zero here.

The xHCI Transfer Rings

A ring has a Producer and a Consumer. The Producer places TRBs onto the ring, while the Consumer processes TRBs from the ring. With each ring, system software is either the Producer or the Consumer, while the controller is the other.

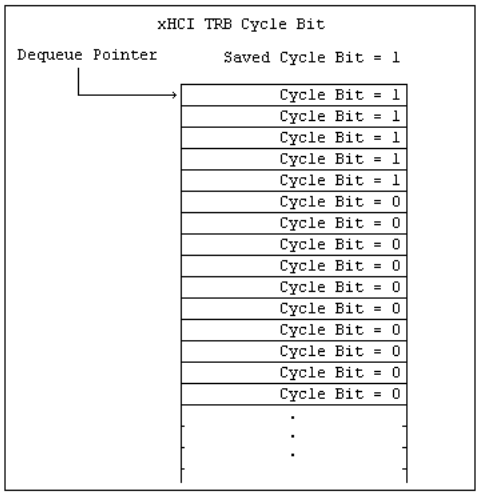
A transfer ring contains blocks of TRBs known as segments. Each ring type has a different technique for keeping track of these segments. However, the way the Producer or Consumer knows where the current pending TRB or the next empty TRB is located is the same for each ring. The Producer and Consumer use an Enqueue Pointer and a Dequeue Pointer. The Producer uses the Enqueue pointer to point to the next empty TRB that it may write to, while the Consumer uses a Dequeue pointer that points to the next TRB to be processed.

The Producer should keep track of the Consumers Dequeue pointer so not to overwrite a TRB. For example, the Producer should keep track of the Dequeue pointer so that, when it desires to add a TRB to the ring, it does not overwrite a pending TRB waiting for the Consumer to process it.

The Cycle bit in each TRB is used to know where the current Enqueue and Dequeue pointers are. When system software is initializing a ring, it should set each TRB's Cycle bit to the same value, usually a one. When the Producer inserts a TRB in the array, it sets its Cycle bit to the current value of its internally saved value.

When the Consumer has finished executing the current TRB, it needs to know if there is another TRB to process. If the Cycle bit of the next TRB is different than this value, there are no more TRBs to execute at this time. However, when the Consumer rolls back to the beginning of the ring, it toggles its own Cycle bit before checking the Cycle bit in the TRB. See the figure below for an example.

Figure 10-12: xHCI Transfer Ring Cycle Bit



In the figure above, the Dequeue Pointer points to the next TRB the Consumer will process, in this case, the first one. When your driver set up this ring and wrote the Dequeue pointer, it also wrote the Cycle bit, which the controller saves as its internal Consumer Cycle Bit setting. In the case of the example above, your driver should have written a '1'. Once the controller is finished with this TRB, it checks the next TRB. Since the next TRB's Cycle bit is 1, it will process that TRB, and so on. When it is done with the fifth TRB in the example above, the Consumer extracts the next TRB, checks the Cycle bit and stops execution since its Cycle Bit does not match the current saved cycle bit. A Transfer Ring, other than an Event Ring, must include a Link TRB as the last TRB in the ring. See Figure 10-13 on the next page for an example.

Each ring processes the Cycle bit a little different when it comes to the end of the ring, or a Link TRB, as in a Command or Transfer Ring. See each section on a particular ring for more information on this Cycle Bit and that particular ring type.

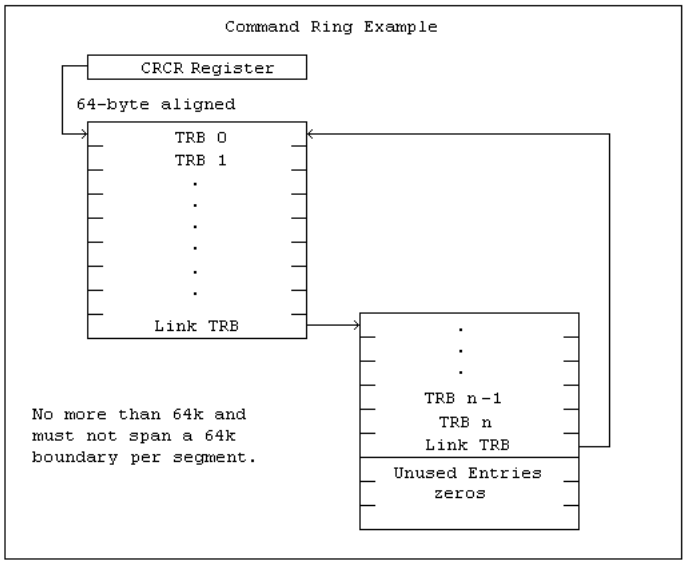
The Command Ring and TD Transfer Ring

The xHCI controller uses a Round-Robin style ring of TRBs to receive commands from the host or to send and receive data from the attached devices. A Command Ring and a TD Transfer Ring are identical other than the specific doorbell that is rung. Therefore, I will explain the Command Ring here.

A Command Ring is pointed to by the controller's Command Ring Control Register. With the Command Ring, system software is the Producer and the controller is the Consumer.

This command ring is simply a list of TRBs with the last one pointing back to the first one using a Link TRB. Therefore, a Link TRB is required in this form of ring. See the figure below for an example of a Command Ring.

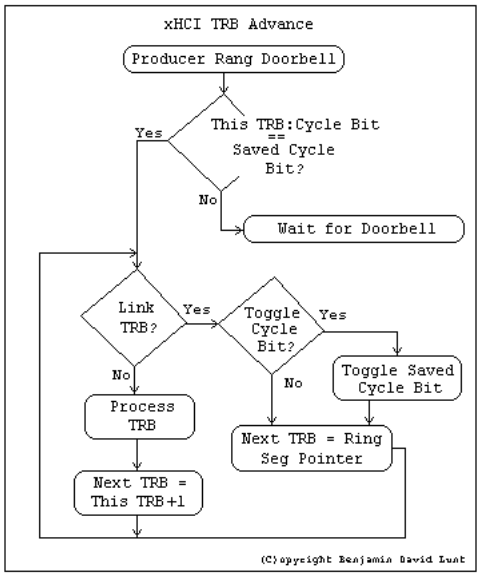
Figure 10-13: Command Ring Example



Each block in the figure above is called a segment. Each segment may not be more than 64k in size and cannot span a 64k boundary. Therefore, if you need more than 64k for your command ring, you must use a Link TRB to point to the next segment.

The Producer keeps track of the position of the current TRB to be processed and the next "empty" TRB that is free to place a command in. Therefore, each time you insert a command, it must not overwrite a command that has not been processed yet. When the Producer inserts a command, or set of commands, it rings the Command doorbell telling the Consumer that at least one Command TRB is ready to be processed. The outline below shows the process of the Cycle bit with more than one segment, using a Link TRB.

Figure 10-14: xHCI Ring TRB Advance with Link TRB



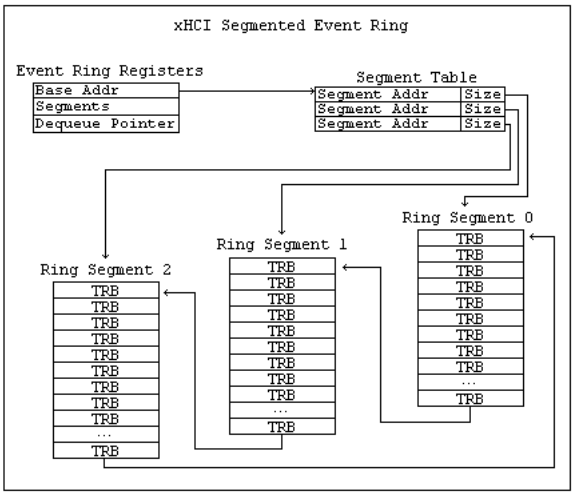
When the Consumer finds a Link TRB, it simply extracts the physical address of the next segment of TRBs. It then starts with that segment. However, before continuing on, it first checks the Toggle Cycle bit within this Link TRB, and if set, toggles the Consumer's saved toggle bit for checking further TRBs.

The Command's Event Ring

The Command's Event Ring is very similar to the Command Ring described before. However, it does not use Link TRBs to point to the next segment. In this type of ring, system software is the Consumer while the controller is the Producer.

The Event Ring uses a Segment Table to specify where the ring resides in memory. The Interrupter Register Set has a pointer to this segment table and its size. Each segment in the table may be of a different size, but has the same rules of memory alignment. It must be less than or equal to 64k in size, must not cross a 64k boundary, and must be 64-byte aligned. The following figure shows and example of an Event Segment Table and its arrays.

Figure 10-15: xHCI Command Ring TRB Advance with Link TRB



The Producer starts with the first TRB position in the first segment. When it needs to advance, it checks the size of this segment. If the advance will exceed the current segment, it checks the size of the segment table. If there is another segment, it loads the address of the next segment and starts at the first TRB position of that segment. If there is not another segment, it goes back to the first segment and the first TRB within that segment.

When you write to the Event Ring Segment Table Base Address Register, the controller will start at the beginning of the ring. Make sure your driver does accordingly.

Segment Tables

The following figure shows the format of the Segment Table shown in Figure 10-15 on the previous page. In the Event Ring's Registers, the Segments field indicates how many of the following 16-byte entries there will be in the Segment Table.

Figure 10-16: xHCI Event Ring Segment Table Entry

31 24	23 18 17 16	15 10 9 8 7 6 5 4 3 2 1 0
Ring Segment Base Address (Lo)		Reserved
Ring Segment Base Address (Hi)		
Reserved		Ring Segment Size
Reserved		

Each of the DWORDs is explained below.

Table 10-32: xHCI Event Ring Segment Table Entry: DWORD 0 & 1

XHCI controller: Event Ring Segment Table Entry: DWORD 0 & 1	
Bit	Description
5:0	Reserved and Zero'd
63:6	Ring Segment Base Address

The Ring Segment Base Address field, contains a 64-byte aligned, physical address pointer to the Event Ring Segment this entry points to. Since this address must be 64-byte aligned, the lower six bits are marked as Reserved and Zero'd.

Table 10-33: xHCI Event Ring Segment Table Entry: DWORD 2

XHCI controller: Event Ring Segment Table Entry: DWORD 2	
Bit	Description
15:0	Ring Segment Size
31:16	Reserved and Zero'd

The Ring Segment Size field indicates how many TRB's can be placed within this segment. Since the max size for a segment is 64k and a TRB is 16 bytes, valid values are 16 to 4096. A minimum of 16 and a maximum of 65536 divided by 16.

A minimum of 16 TRB's is not a limit, as the above paragraph may note. The xHCI specification recommends that a segment have at least 16 TRB's.

Table 10-34: xHCI Event Ring Segment Table Entry: DWORD 3

XHCI controller: Event Ring Segment Table Entry: DWORD 3	
Bit	Description
31:0	Reserved and Zero'd

The last DWORD is used for alignment and possibly future use.

Starting the Communication Process

Now that we have discussed the required items for communication with USB devices via the xHCI controller, it is time to create your driver. The advantage of the xHCI controller is that, in my opinion, Intel finally got it right. You no longer have to know the internal workings of the controller. You now simply send commands and Transfer Descriptors, and receive Event status information. With this technique you no longer have to worry about the speed of the device attached, but more importantly, you don't have to write two or more drivers to handle different speeds as the EHCI controller did. The xHCI does it for you.

If you feel that you have a good grasp on what was discussed in this and earlier chapters, and are ready to get started writing a driver for the xHCI controller, please turn to Chapter 14 in Part 2 of this book. There I will discuss in more detail, the xHCI controller, the new super-speed it includes, and how to setup and use all of the things discussed in this chapter.

Part 2

In Part 2, I will explain how to detect and configure an external attached device. This is called device enumeration. I will discuss the process to enumerate multiple devices using all four controller types.

Each chapter will contain information about a specific controller type; hence there will be four chapters within Part 2 of this book.

Enumeration, as far as the device is concerned, is the same for each type. However, each of the controllers are controlled with different data types and techniques.

Each of the following chapters assume that the device is attached to the root hub and the device attached is not an external hub. External hubs are discussed in a later chapter.

To learn how to enumerate a device using the UHCI, turn the page to Chapter 11. The OHCI enumeration is explained in Chapter 12, the EHCI in Chapter 13, and the new xHCI in Chapter 14.

Chapter 11 – Device Enumeration with the UHCI

In this chapter, I will explain how to enumerate the devices on the UHCI root hub. Enumeration means to number or name one by one, the devices on the root hub. By doing this, we can find all devices attached, give them an address number, and allow them to start communicating with the host.

Here, in this chapter, I will use the UHCI controller for my examples. If you are going to use one of the other controller types, you may skip this chapter and go to the chapter specific for that controller. Those chapters have some of the same information, descriptions, and data tables as this chapter, but I thought it would be confusing to the reader to keep flipping pages, between controllers, to find a packet description, or other identically used items. Therefore, even though there are duplicate descriptions in these chapters, there is no need to flip forward to the OHCI section to find a table, when using UHCI examples, for example.

If you have read one of the other chapters within Part 2, you will find some of the information here in this chapter repeated. This is intentional. These chapters were written independent of each other. One should not have to read one chapter to use the other. With this said, let's go on to the UHCI enumeration.

Let's assume that you already have set up the UHCI root hub hardware, discussed in chapters 3 and 4, and have detected that there is a single device plugged into one of the ports and it has been reset and enabled correctly.

Each device on the hub must be assigned an identification number, also known as a device address. This is so that we can direct a packet to the correct device. A root hub can have at most 127 devices attached. In theory, all these devices can be attached to a single port on the root hub. We must be able to set up each device to execute the packets that are only designated for it and it alone.

The USB specification says you may have 127 devices all on one port of a root hub as long as the other ports are empty. So why did I say, "in theory?" Each device can draw only a specified amount of current from the root hub. If you have too many devices that require an amount of current that together draw more than this allotted amount of current, the UHCI will stop execution and return an error interrupt. Also, you should not use more than about nine feet of cable/hub combination away from a powered hub, whether that is the root hub or an external hub that has an external power source.

Why are there only 127 devices allowed when a 7-bit value will hold 128 values? When a device first gets attached to the bus, it does not have a value assigned to it, but it still must execute the set-up packets. Therefore, a device that has just been attached and has not yet been set up, is in the Default State. In this default state, the device has a device address of zero. Therefore, with the device address value of zero already used, that leaves only 127 values left.

With this in mind, we know that a newly attached device will respond to a device address of zero, so let's make our first packet transfer on the USB stack.

Creating a Queue of Transfer Descriptors

The Operating System, or your driver, has no idea what kind of device has been attached. Therefore, each device has a standard device descriptor, along with other descriptors, that tells the operating system what kind of device it is. The device descriptor tells us a lot about the attached device. For example, we can see if the device complies to the version of the USB specification that our driver supports, it might tell us the class code and sub-class code of the device, as well as what string descriptors it has, giving us a plain text description of the device. The class and subclass codes are probably the most important of these items. For example, if the device is a Human Interface Device (HID) mouse, it should have a class code of 0x03, and the subclass code will be a HID defined protocol code. All HID devices should have a class code of 0x03, while for example, all mass storage devices should have a class code of 0x08. See Appendix I. However, please note that if the class code is zero, the class and subclass codes should be retrieved from the Interface Descriptor, described later in this chapter.

The format of the Standard Device Descriptor is explained here.

Table 11-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	0x12	Size of this descriptor in bytes. For the Standard Device Descriptor, this should be 18 bytes.
1	Type	1	0x01	Device descriptor type. For the Standard Device Descriptor, should be 1.
2	Release Num	2	BCD	USB Specification Release Number in Binary-Coded Decimal. i.e., 2.10 = 0x0210. This field identifies the release of the UBS Specification revision with which the device and its descriptors are compliant with.
4	Device Class	1	Value	The Device Class code assigned by the USB-IF.[1]
5	Sub Class	1	Value	The Device Sub Class code.[1]
6	Protocol	1	Value	Protocol Code.[1]
7	MaxPacketSize	1	Value	Max packet size for endpoint zero (8, 16, 32, or 64)
8	Vendor ID	2	Value	Vendor ID
10	Product ID	2	Value	Product ID
12	Device Rel	2	BCD	Device release number in Binary-Coded Decimal
14	Manufacturer	1	Index	Index of string descriptor[2]
15	Product	1	Index	Index of string descriptor[2]
16	Serial Num	1	Index	Index of string descriptor[2]
17	Configurations	1	Value	Number of possible configurations.

[1] http://www.usb.org/developers/defined_class (Also Appendix I within this book)

[2] A value of zero means no string descriptor

Getting the Device Descriptor of the Device

To get this device descriptor from the device, we must send it a request packet called a Setup packet. The Setup packet is an 8-byte data packet that consists of the format explained here.

Table 11-2: Device Request Packet

Offset	Field	Size	Value	Description
0	Request type	1	Value	Bit 7 Transfer Direction 0 = Host to device 1 = Device to host Bit 6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	Value	The Request desired
2	Value	2	Value	Depends on the request
4	Index	2	Index	Depends on the request
6	Length	2	Count	If a data phase, number of bytes to transfer.

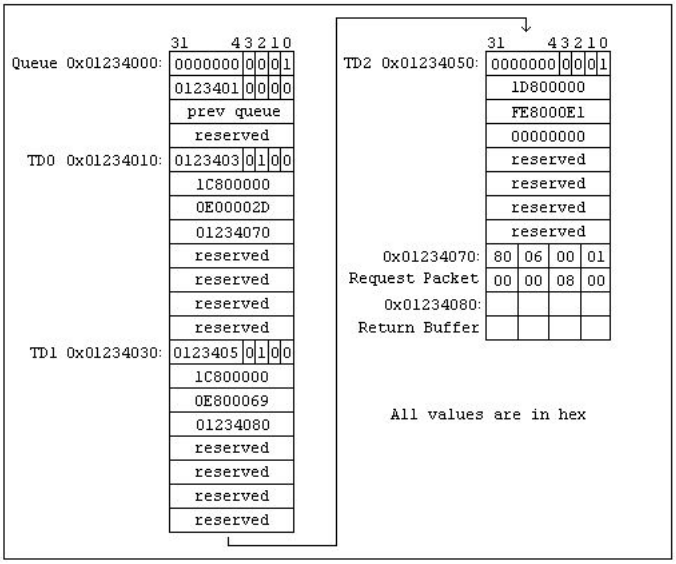
This is the data we will send to a device that has a Device Address value of zero. So how do we send this request to the device and receive the device descriptor data? We have to place a few TDs on to the USB stack. Three to be exact, along with a queue. Please note that we only request the first 8 bytes of the device descriptor at this time. We don't know for sure the size of the descriptor and by getting the first 8 bytes, we can get the actual size from the data returned. Then we can do this again using as many TDs as we need to get the whole descriptor. See later for the values we place in this request packet.

We need a buffer of 128 bytes and it must be paragraph aligned. Since the FYSOS malloc() function always returns paragraph aligned allocated memory, I can simply call it to return a minimum of 128 bytes. For the purpose of this example, let us say that it returns a memory block physically allocated at 0x01234000.

Please note that I used malloc(). You can place these buffers on the stack or even have them be global buffers in your code. However, you must be able to get the physical address of the buffer, not the linear address. If your OS is using paging or another form of linear addressing, be sure to find a way to get the physical address of the buffer. I used malloc() in the examples assuming that the address returned is a physical address.

If we place the queue, the three transfer descriptors, the request packet, and the return buffer all in the same memory buffer, we can do something like the figure below in a UHCI stack frame.

Figure 11-1: UHCI Request



When this Queue gets inserted into the stack frame, the insertion routine will set the Head Link Pointer accordingly, so for the example above, we set the T bit.

We use the element link pointer of the queue and place it in the pointer field, the address of the first transfer descriptor, which in this example is 0x01234010 shifted right 4 bits. Remember that the controller assumes that the lower four bits of the address will be zero and therefore only needs the top 28 bits. We also now make sure the T bit is cleared since the pointer is a valid pointer, and make sure the Q bit is cleared since we point to a transfer descriptor.

The remaining two dwords in the queue are not used while in the stack frame and we don't have to worry about them here.

Now for the Transfer Descriptors. The first transfer descriptor will start at physical address 0x01234010 and must be a SETUP transfer descriptor. The Link Pointer will contain the upper 28 bits of the address of the next transfer descriptor, which will be 32 bytes from this transfer descriptor. The depth/breadth bit is set so that the controller continues with our transfer descriptors instead of moving on to the next Queue Head pointer. The Q bit and T bit are cleared, since the next item is a TD, and is valid.

The SPD field in the second dword will be cleared. Set the C_ERR bits to 11b for 3 error tries, the LS bit (bit 26) to 1 for a low-speed device or cleared if it is a full-speed device, the IOS field to 0, and the IOC field to 0.

Set the Status field (bits 23:16) to 0x80, which sets the active bit and clears the remaining bits. The controller will not execute a transfer descriptor unless the active bit is set. It will clear the active bit after successfully executing this transfer descriptor. Now set the Actual Length to 0x00 to be sure we read it back as zero if it is an unsuccessful transfer.

In the third dword, set the Maximum Length field to 0x07, the size of the request packet minus 1, set the Data Toggle bit to 1, the Endpoint and the Device Address to zero. Set the Packet Identification field to 0x2D since this is a setup packet.

Now set the forth dword to the physical address of the request packet, Table 11-2. This value should be 0x01234070. You may set the remaining dwords to any value you see fit. For now,

set them to zeros so that there is no confusion if you do a memory dump of the USB stack.

That was the setup packet, which is responsible for telling the device that we are requesting its Device Descriptor. Now we need to create another TD telling the controller where to copy the requested bytes.

The second transfer descriptor starts at physical address 0x01234030 and must be an IN transfer descriptor. The rest of the fields are very similar to the first transfer descriptor. In the first dword, the link pointer contains the upper 28 bits of the address of the next transfer descriptor, which is 32 bytes from this transfer descriptor. The Depth/Breadth bit is set, and the Q and T bits are cleared. The SPD field will be cleared, set the C_ERR bits to 11b for 3 error tries, the LS bit (bit 26) to 1 for a low-speed device, zero for full-speed, the IOS field to 0, and the IOC field to 0. Set the Status field (bits 23:16) to 0x80, and set the Actual Length to 0x00 to be sure we read it back as zero if it is an unsuccessful transfer.

Set the Maximum Length field to 0x07, the size of the data we are requesting minus 1, this time clear the Data Toggle bit, and set the Endpoint and the Device Address to zero. Set the Packet Identification field to 0x69 since this is an IN packet. Set the forth dword to the physical address of the data buffer to hold the 8 bytes of the descriptor. This can be any byte-aligned data buffer as long as it is the physical address, though for the above example set it to 0x01234080.

The USB specification states that we must send a zero length OUT packet to the device to tell the device we have received the data. With this in mind, the third transfer descriptor is almost exactly the same as the second except that we change the PID to OUT (0xE1), the Data Toggle bit back to 1, and give it a zero max length (0x7FF). We also need to set the T bit since we don't have any more TDs, and set the IOC bit so that the controller fires an interrupt when it is done with this frame.

Since the controller won't fire the interrupt until end of frame, it really doesn't matter which transfer descriptor you set the IOC bit in as long as you know that the whole queue will complete before the end of the frame. However, if you have the IOC bit in the SETUP packet and the controller doesn't have time to execute the IN packet in this frame, you will receive the interrupt, but no data transfer will have taken place. It is always a good idea to place the IOC bit in the last data packet, but for this example, place it in the status packet. Be sure to see the note below.

As for the Data Toggle bit, for control transfers, you always start with zero, then toggle it for each consecutive transfer descriptor in the queue. The status packet (the last packet) will always be 1.

In the SETUP packet, we send eight bytes of information. Using Figure 11-1, we can see that we set the Request Type field to 0x80. The requested data is coming from the device to the host with a standard request type for a device. The Request is the GET_DESCRIPTOR request, which has a value of 0x06, and the Value field contains two byte values. The high byte contains a value of 0x01 for DEVICE, while the low byte contains 0. These two bytes are written as a 16-bit word in little-endian format. The Index field contains zero, and the Length field contains 8, the number of bytes we are requesting.

Right now we assume that the data will be transferred without error. Usually, your driver won't send the STATUS packet (the third transfer descriptor above) until it actually receives a valid transfer of the device descriptor. However, for the purpose of this demonstration, we assume no errors.

Inserting your queue into the stack

Now we are ready to insert our queue into the UHCI's stack. See the section "The FYSOS Stack Frame" in Chapter 4 on how to insert your queue into the UHCI's stack frame.

Once we have done this, we should wait for the interrupt to occur. If we have the IOC bit in the INTERRUPT_ENABLE register set, the interrupt will fire at the end of the frame after completing the status TD, whether it is successful or not. If you would rather not have an interrupt routine, you can clear the IOC bit in the INTERRUPT_ENABLE register and poll bit 0 of the STATUS register until it gets set.

Now it is time to see if we had a successful transfer. First, check that the status bits in each of the TD's are all zero including the active bit. Also, make sure that the act_len field is 7 for the SETUP packet and IN packet and 0x7FF for the OUT packet.

Once you have checked these fields, you can now assume the data in the buffer at 0x01234080 is filled with the first eight bytes of the device descriptor.

At this point, you are probably wondering why we did not just request the whole 18 bytes of the descriptor at one time. There is no real reason other than preference and one more reason explained below. However, the first byte in the returned buffer tells us how long the device descriptor actually is. Also, the eighth byte is the max_packet_len byte, which tells us how many bytes we can read or write to this endpoint zero with each TD. This tells us how many transfer descriptors we must have to retrieve the complete Device Descriptor. On a full-speed device, the max_packet_len could be 32 or even 64. If so, why request two 8-byte packets, and a single 2-byte packet, when you can request a single packet of 18 or more bytes?

The main reason is that some devices were only expected to work with the Windows Operating System. Win98SE for example, gets the first 8 bytes, does a reset of the port, sets the address of the device, and then finally gets all 18 bytes of the descriptor. Therefore, these few devices don't expect to send more than the first 8 bytes while in the default state, the state before it is given an address, and may not function properly until that sequence is received. It completely defies the given operation of the USB specification, but this is how some devices function. I have not personally seen or used any device that functions in this way, but have read documentation that states this may happen.

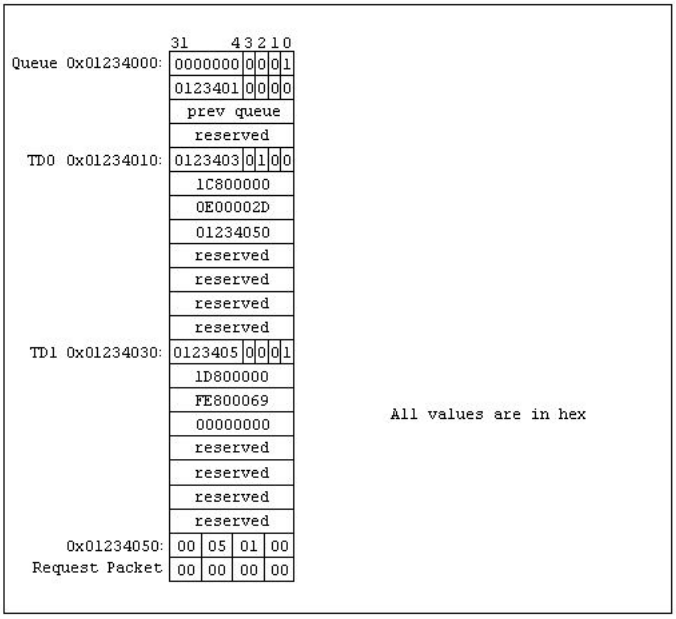
Usually, at this point you know if there is a valid device attached and that it is responding to your requests. Therefore, it is a good idea to reset the device, set its address to the next available address number, and retrieve the whole descriptor.

Setting the Address of the device

After you have successfully reset the device, you should now set the address. Up to this point the device has been in the Default state. Most devices will handle most control requests while in this state, but you must set the address and move the device into the Addressed state to configure the device for use.

To set the address, you send a SETUP packet similar to what we did before, except there is no data transfer. The address number to set the device to is sent in the SETUP packet, there is no data packet, and then a single STATUS packet. In this example, we will set the devices address number to 1, the third byte in the request packet of the SETUP packet.

Figure 11-2: Request Return



Once you have sent the Address Command above, now you can get the whole descriptor using a similar queue as we used to get the first eight bytes. Using the descriptor size, the value in the first byte of the descriptor we got before, and adding or subtracting TD's depending on how many bytes you can receive in a single packet TD, send the request again, remembering to now use the new address number.

Getting the String Descriptor(s) of the Device

As mentioned before, a device may contain three or more string descriptors. A string descriptor is a descriptor of unknown size that contains 16-bit Unicode characters describing the device in human-readable form. I say, unknown size, since each string may be of a different size. Well, then how do you get the size? Just as we did before to get the size of the device descriptor. Get the first eight bytes. Within the first eight bytes of the string descriptor is the descriptor's length. This tells you how many additional bytes to read.

There are two forms of string descriptors. The first is the language identification descriptor. It contains a list of 16-bit word values defining what language(s) this device supports. Once you have the list of LANGID's, as they are called, you can request the remaining string descriptors in any one of these languages, by sending the LANGID in the wIndex field of the Get String Descriptor request packet. To get the list of LANGID's supported, you call the Get String Descriptor request with an index of zero.

Table 11-3: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes
1	Type	1	0x03	String Descriptor
2	Langid[0]	2	Value	Supported Language Code 0
4	Langid[1]	2	Value	Supported Language Code 1
6	Langid[n]	2	Value	Supported Language Code n

The table will hold as many language-codes as it supports. To find out how many, simply take the Length, subtract two, then divide by two. With this in mind, a device can support up to 126 languages. Again, you need to get the first eight bytes of the descriptor to find out how many more bytes to read.

Since the Length field in most descriptors is a byte-sized value, and can hold a maximum value of 255, instead of requesting eight bytes to get the length, then requesting the whole packet, you can request 255 bytes. Assuming 8-byte packets, simply set up a queue with enough 8-byte packets to hold 255 bytes, and set the SPD bit in each TD. The controller will stop executing in the Depth First direction and start on the next Queue Head Pointer at the first Short Packet detected. However, you still need to send the status packet after detecting the SPD interrupt.

Once you have the LANGIDs, you can choose a language you would like to support. Then, each time you request a sting descriptor, place this LANGID code from above in the Langid field in the request packet. A string will have a value of 1 or higher in the index field of the request packet and have the format described here.

Table 11-4: String Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	String	n	xx	Unicode Encoded String.

The string is 16-bit Unicode encoded and must not be zero-terminated. You must get its size from the Length field in the descriptor. Note that the length value includes the first two bytes of the descriptor.

Some of these LANGID codes are:
0x0409 - English - United States
0x0C09 - English - Australian
0x0407 - German - Standard
0x1009 - English - Canadian
0x040C - French - Standard
0x080A - Spanish - Mexican

Notice that the English languages have the low byte as 0x09. This isn't a coincidence. The Primary Language is a 10-bit value in the lower ten bits, with English being 0x009. The sub-language identifier is in the high six bits of the high byte, which is 0x01 for the United States, and 0x03 for Australian. For the complete list of LANGID codes, see the file USB_LANGIDs.pdf, from the USB-IF web page (See Appendix A).

Getting/Setting the Configuration of the Device

Before most devices will function properly, you must request the configuration descriptor(s), decide which one you want to use, and set that configuration. If you recall earlier, the device descriptor shows how many configurations a device may have. Though most devices only have one, a device may have two or more configurations. For example, the first configuration may be used when the device is self-powered, and the second configuration may be used when the device is bus-powered.

The configuration descriptor also contains all the interface descriptors for the device along with each endpoint descriptor for each interface. The Length field contains the size of the configuration header, while the TotalLength field contains the total length of all data returned, the header, each interface, and each endpoint.

You can get the different configurations by giving the index in the request packet.

Table 11-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x02	Config Descriptor
2	TotalLength	2	Value	Total length of all data.
4	NumInterfaces	1	Value	Number of Interfaces.
5	ConfigVal	1	Value	Value for this configuration
6	ConfigString	1	Value	Index for String Descriptor describing this config.
7	Attributes	1	Bitmap	D7 Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	mA	Maximum Power Consumption

To set the desired configuration, you send the SetConfiguration request with the desired ConfigVal in the request header.

Each configuration descriptor returns one or more interfaces. A device may have more than one interface. For example, a numpad with the usual number keys, enter, plus and minus keys, similar to the right hand part of a regular full sized keyboard, may also be used as a pointing device. The first interface may be the keypad's individual keys, while the second interface may be certain keys on the numpad used for direction and button press/release information. You cannot request an interface descriptor as you would a device descriptor using the GetDescriptor request. You retrieve the interface descriptor from the data returned after the configuration descriptor as described earlier.

Table 11-6: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x04	Interface Descriptor
2	Interface Num	1	Value	Number of Interface.
3	Alternate Set	1	Value	Used to select alt. setting.
4	Endpoints	1	Value	Number of Endpoints.
5	Class Code	1	Value	Class Code

6	Sub Class	1	Value	Sub Class Code
7	Protocol	1	Value	Protocol Code
8	Interface	1	Value	Index of String Descriptor describing this interface.

Depending on the interface, each interface may have zero or more endpoints. Usually, the first endpoint is the control endpoint and has an endpoint address of zero. This endpoint is assumed and usually not listed in the configuration data. The second endpoint may be an interrupt endpoint or a bulk endpoint. The Attributes field listed below, specifies what type of endpoint it is, with bit 7 in the Address field specifying which direction, in or out.

Table 11-7: Endpoint Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x05	Endpoint Descriptor
2	Address	1	Value	Endpoint address: Bits 3:0 Endpoint Num 6:4 reserved (0) 7 0 = Out, 1 = In
3	Attributes	1	Bitmap	Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 7:2 Reserved unless it is an ISO endpoing, then: Bits 3:2 Sync type 00 = No Sync 01 = Ascynchronous 10 = Adaptive 11 = Synchronous Bits 5:4 Usage Type 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback 11 = Reserved Bits 7:6 reserved
4	MaxPacketSize	2	Value	Maximum packet size this endpoint is capable of.
6	Interval	1	Value	Interval for polling endpoint data trasfers. Ignored by Bulk and Control, must be 1 for ISO, and can be 1 to 255 for Interrupt.

Using the technique explained in the first part of this chapter, you should be able to get all the descriptors explained to this point. In Part 3, I will explain the process to send and receive data from specific types of devices, such as keyboards, mice, and mass storage devices.

Short Packet Detect with the UHCI Stack

A short packet detect is when the controller receives less bytes than the TD had requested. When there is only one TD or this is the last TD in the transaction, this really isn't an issue. However, what if you have a Max Packet Size of 8 and are requesting 255 bytes? This would mean that you have 32 data TD's in this transaction. Now, what if the controller receives only 18 bytes, as it would on a Device Descriptor request. It would Short Packet Detect on the 3rd TD and it would not be necessary to execute the remaining 29 TD's. Therefore, how do you skip the remaining 29 TD's and go directly to the Status TD?

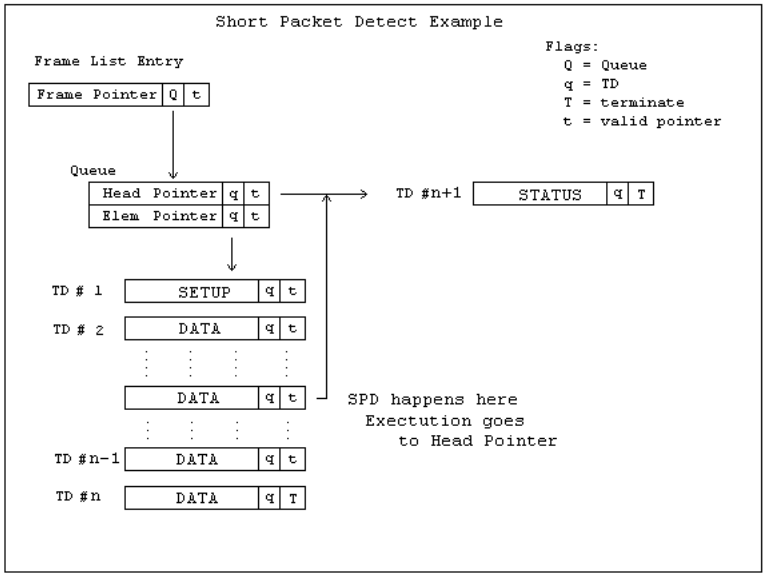
The UHCI gives a technique to skip the remaining data TD's after a Short Packet Detect, or SPD. When using the Depth First bit and while in a queue, on a SPD, the controller will stop executing the TD's in this Queue and move to the Head Link Pointer as if it were executing in Breadth First fashion. It does this because a SPD is considered a non-successful transaction due to not all bytes was transferred as expected. However, as with other errors, the controller does not count down the CERR field.

Therefore, use a single queue for your transaction. Place the Queue on the stack, with the Element Link Pointer member pointing to a list of TD's where the first TD is the Setup TD and the remaining TD's are the Data TD's with the last TD marked with the Terminate bit set. Then in the Queue's Head Link Pointer, place a pointer to the Status TD. If and when a SPD is detected, the remaining TD's after the TD that triggers the SPD will not be executed and the controller will go directly to the Status TD. If a SPD is not detected, all TD's under the Element Pointer will be executed as expected, then the controller will move to the Head Link Pointer and execute the Status TD. See the figure on the next page for an example.

Again, please note that in normal operation, you should not send the Status TD until you have successfully received the data. In other words, you should only send the Status TD after your driver verifies the data. However, for the sake of these examples, I send it within the same Queue, assuming no errors.

The next three chapters explain all of this information for the remaining controller types with the very next chapter, the second of the four, using the OHCI controller, the third using the EHCI controller, and the fourth using the xHCI controller. If you are only interested in the UHCI controller, you may move on to Part 3 and learn how to enumerate different device types.

Figure 11-3: UHCI Short Packet Detect



Chapter 12 – Device Enumeration with the OHCI

In this chapter, I will explain how to enumerate the devices on the OHCI root hub. Enumeration means to number or name one by one, the devices on the root hub. By doing this, we can find all devices attached, give them an address number, and allow them to start communicating with the host.

Here, in this chapter, I will use the OHCI controller for my examples. If you are going to use one of the other controller types, you may skip this chapter and go to the chapter specific for that controller. Those chapters have some of the same information, descriptions, and data tables as this chapter, but I thought it would be confusing to the reader to keep flipping pages, between controllers, to find a packet description, or other identically used items. Therefore, even though there are duplicate descriptions in these chapters, there is no need to flip back to the UHCI section to find a table, when using OHCI examples, for example.

If you have read one of the other chapters within Part 2, you will find some of the information here in this chapter repeated. This is intentional. These chapters were written independent of each other. One should not have to read one chapter to use the other. With this said, let's go on to the OHCI enumeration.

Let's assume that you already have set up the OHCI root hub hardware, discussed in chapters 5 and 6, have detected that there is a single device plugged into one of the ports, and it has been reset and enabled correctly.

Each device on the root hub must be assigned an identification number, also known as a device address. This is so that we can direct a packet to the correct device. A root hub can have at most 127 devices attached. In theory, all these devices can be attached to a single port on the root hub. We must be able to set up each device to execute the packets that are only designated for it and it alone.

The USB specification says you may have 127 devices all on one port of a root hub as long as the other ports are empty. So why did I say, "in theory?" Each device can draw only a specified amount of current from the root hub. If you have too many devices that require an amount of current that together draw more than this allotted amount of current, the UHCI will stop execution and return an error interrupt. Also, you should not use more than about nine feet of cable/hub combination away from a powered hub, whether that is the root hub or an external hub that has an external power source.

Why are there only 127 devices allowed when a 7-bit value will hold 128 values? When a device first gets attached to the bus, it does not have a value assigned to it, but it still must execute the set-up packets. Therefore, a device that has just been attached and has not yet been set up, is in the Default state. In this Default state, the device has a device address of zero. Therefore, with the device address value of zero already used, that leaves only 127 values left.

With this in mind, we know that a newly attached device will respond to a device address of zero, so let's make our first packet transfer on the USB stack.

Creating Endpoint Descriptors and Transfer Descriptors

The Operating System, or your driver, has no idea what kind of device has been attached. Therefore, each device has a standard device descriptor, along with other descriptors, that tells the operating system what kind of device it is. The device descriptor tells us a lot about the attached device. For example, we can see if the device complies to the version of the USB specification that our driver supports, it tells us the class code and subclass code of the device, as well as what string descriptors it has, giving us a plain text description of the device. The class and subclass codes are probably the most important of these items. For example, if the device is a Human Interface Device (HID) mouse, it should have a class code of 0x03, and the subclass code will be a HID defined protocol code. All HID devices should have a class code of 0x03, while for example, all mass storage devices should have a class code of 0x08. However, please note that if the class code is zero, the class and subclass codes should be retrieved from the interface descriptor. The format of the Standard Device Descriptor is as follows.

Table 12-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	0x12	Size of this descriptor in bytes. For the Standard Device Descriptor, this should be 18 bytes.
1	Type	1	0x01	Device descriptor type. For the Standard Device Descriptor, should be 1.
2	Release Num	2	BCD	USB Specification Release Number in Binary-Coded Decimal. i.e., 2.10 = 0x0210. This field identifies the release of the UBS Specification revision with which the device and its descriptors are compliant with.
4	Device Class	1	Value	The Device Class code assigned by the USB-IF.[1]
5	Sub Class	1	Value	The Device Sub Class code.[1]
6	Protocol	1	Value	Protocol Code.[1]
7	MaxPacketSize	1	Value	Max packet size for endpoint zero (8, 16, 32, or 64)
8	Vendor ID	2	Value	Vendor ID
10	Product ID	2	Value	Product ID
12	Device Rel	2	BCD	Device release number in Binary-Coded Decimal
14	Manufacturer	1	Index	Index of string descriptor[2]
15	Product	1	Index	Index of string descriptor[2]
16	Serial Num	1	Index	Index of string descriptor[2]
17	Configurations	1	Value	Number of possible configurations.

[1] http://www.usb.org/developers/defined_class

[2] A value of zero means no string descriptor

Getting the Device Descriptor of the Device

To get this device descriptor from the device, we must send it a request packet called a Setup packet. The setup packet is an 8-byte data packet that consists of the following format.

Table 12-2: Device Request Packet

Offset	Field	Size	Value	Description
0	Request type	1	Value	Bit 7 Transfer Direction 0 = Host to device 1 = Device to host Bit 6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	Value	The Request desired
2	Value	2	Value	Depends on the request
4	Index	2	Index	Depends on the request
6	Length	2	Count	If a data phase, number of bytes to transfer.

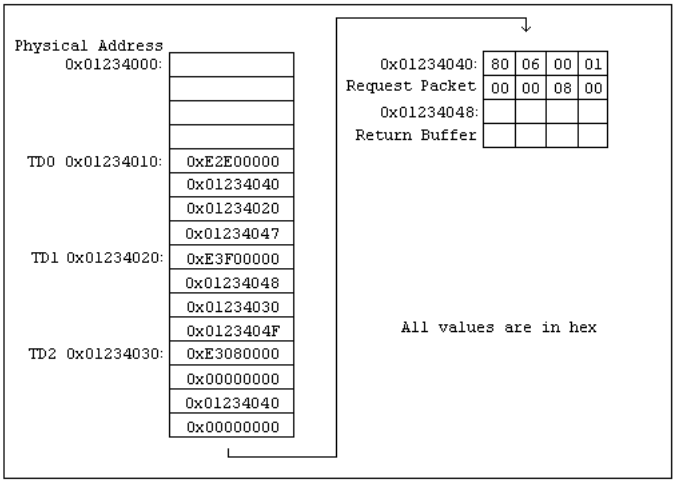
This is the data we will send to a device that has a device address value of zero. So how do we send this request to the device and receive the device descriptor data? We have to place a few transfer descriptors on the USB stack. Three to be exact, inserted into the control list. Please note that we only request the first 8 bytes of the device descriptor at this time. We don't know for sure the size of the descriptor and by getting the first 8 bytes, we can get the actual size from the data returned. Then we can do this again using as many transfer descriptors as we need to get the whole device descriptor. See a bit later for the values we place in this request packet.

We need a buffer that is large enough to hold three transfer descriptors, the 8-byte request packet, and the 8-byte return packet. Since each transfer descriptor is 16 bytes, we need a minimum of 64 bytes and it must be paragraph aligned. Since the FYSOS malloc() function always returns paragraph aligned allocated memory, I can simply call it to return a minimum of 64 bytes. For the purpose of this example, let us allocate 80 bytes, and say that it returns a memory block allocated at 0x01234000. The extra 16 bytes at 0x01234000 will be used in the endpoint descriptor example later in this section.

Note that I used malloc(). You can place these buffers on the stack or even have them be global buffers in your code. However, you must be able to get the physical address of the buffer, not the linear address. If your OS is using paging or another form of linear addressing, be sure to find a way to get the physical address of the buffer. I used malloc() in the examples above assuming that the address returned is a physical address.

If we place the three transfer descriptors, the request packet, and the return buffer all in the same memory buffer, we can do something like the outline below in an OHCI stack frame.

Figure 12-1: OHCI Request



For the first transfer descriptor located at physical address 0x01234010, we set the flags for a SETUP packet and set the condition code to 14. This tells the controller that this transfer descriptor has not been accessed yet. Set the error count to zero. This allows for three tries before it reports an error. You could set this field to one, two, or three. If you set it to two, the controller will only try once more before it returns an error. A setting of zero is typical.

Set the toggle bits to 10b. This tells the controller to use this field (MSb set), instead of the toggle field in the endpoint descriptor. The LSb is zero for a SETUP packet. Set the delay interrupt field to 111b. This will tell the controller to retire the transfer descriptor, but not change the HCCA's DoneHead field or fire an interrupt.

This is a SETUP packet, so set the direction/PID field to 00b. There is no rounding issue. We want all 8 bytes transferred, so clear the rounding bit.

The Current Buffer Pointer field needs to have the physical address of the SETUP packet, in this case 0x01234040. The Buffer End field is calculated as shown on the top of the next page.

```
td->be =
    (td->cbp + smaller_of(size_of_data, max_packet_size) - 1);
```

Unless the Current Buffer Pointer is zero. Then this field needs to be zero also.

The NextTD field needs to have the physical address of the next transfer descriptor. Your code should write the sum of the address of the current transfer descriptor plus the size of your driver's transfer descriptor.

```
td->next_td =
      (current_address + sizeof(struct TRANSFER_DESCRIPTOR));
```

There is nothing wrong with having your driver's transfer descriptor data structure larger than 16 bytes. The FYSOS driver uses a 32-byte structure, or 8 dwords. The first 4 dwords are exactly as the controller requires them, while the remaining 4 dwords are used by the driver for transfer identification and information. The controller doesn't require that the next transfer descriptor be 16 bytes away from the current one, so use as many bytes as you would like. However, make sure that the size of your data structure is a multiple of 16, so that it will be paragraph aligned when you use the above calculation.

That was the Setup packet, which is responsible for telling the device that we are requesting its Device Descriptor. Now we need to create another transfer descriptor telling the device where to copy the requested bytes.

The second transfer descriptor starts at physical address 0x01234020 and must be an IN transfer descriptor. The rest of the fields are very similar to the first transfer descriptor.

The only difference in the first dword of the second transfer descriptor and the first transfer descriptor is that the toggle bits should be 11b, and the direction/PID field should now be 10b, an IN packet. The Current Buffer Pointer should now point to the empty buffer we want the controller to fill, 0x01234048, the Buffer End field to point to 8 minus 1 bytes past that, and the NextTD field to point to the next transfer descriptor.

The USB specification states that we must send a zero length OUT packet to the device to tell the device we have received the data. With this in mind, the third transfer descriptor is almost exactly the same as the second except that we change the Direction/PID field to and OUT (01b). We also change the Data Toggle bits to 11b (which is actually the same since we only had one IN packet), and give it a length of zero, by clearing out both the Current Buffer Pointer and Buffer End fields. We also write 000b to the DelayInterrupt field. This tells the controller to retire the transfer descriptor upon completion and update the DoneHead field in the HCCA.

Right now we assume that the data will be transferred without error. Usually, your driver won't send the STATUS packet (the third transfer descriptor above) until it actually receives a valid transfer of the device descriptor. However, for the purpose of this demonstration, we assume no errors.

Also, if you will note, I set the NextTD field in the last transfer descriptor to physical address 0x01234040. This actually points to the SETUP packet data. This is not an error. The controller doesn't use the NextTD field to know when to stop retrieving transfer descriptors. It uses two fields in the endpoint descriptor for this purpose. Therefore, my code simply calculates the NextTD field with the calculation displayed above for all transfer descriptors, including the last one. This is a must, and you will see why when we create an endpoint descriptor below.

As for the Data Toggle bit, for control transfers, you always start with zero, then toggle it for each consecutive transfer descriptor after that, within the current data transfer. Then, the last transfer descriptor, the STATUS packet, will always be set to 1.

In the SETUP packet, we send eight bytes of information. Using Figure 12-1, we can see that we set the Request Type field to 0x80. The requested data is coming from the device to the host with a standard request type for a device. The Request is the GET_DESCRIPTOR request, which has a value of 0x06, and the Value field contains two byte values. The high byte contains a value of 0x01 for DEVICE, while the low byte contains 0. These two bytes are written as a 16-bit word in little-endian format. The Index field contains zero, and the Length field contains 8, the number of bytes we are requesting.

Inserting your transfer descriptors into the stack

Now we are ready to insert our transfer descriptors into the OHCI's stack. To do this, we can either use one of the already created endpoint descriptors in the corresponding list, for this case it would be the Control List, or we can create a new endpoint descriptor and insert it into the linked list. For the purpose of this demonstration, we will create a new endpoint descriptor.

To create an endpoint descriptor, you will need a minimum of 16 bytes allocated on a 16-byte boundary. If you will remember, I asked that you allocate 16 additional bytes in the malloc() example earlier, and displayed it as the first 16 bytes. This is where we will place our endpoint descriptor.

The endpoint descriptor, located at physical address 0x01234000, holds the device address and endpoint number, along with other data that will be the same for all transfer descriptors the endpoint descriptor links to.

To start with, set the FunctionAddress field to zero. This device is still in the Default state. Also set the EndpointNumber to zero. All control packets go to endpoint zero, which is usually the control endpoint. Since we set the direction bits in the transfer descriptor, we will set the Direction field in the endpoint to either 00b or 11b. I use 00b.

Set the Speed field to either 0 or 1, depending on the speed of the device. The speed of the device, either low- or full-speed, can be determined by bit 9 in the HcRhPortStatus register that this device is plugged in to. For this example, we will set this field to 0, for a low-speed device.

Clear the sKip and Halted bits. If either of these bits are set, the controller will skip this endpoint descriptor and our transfer descriptors will not be executed. Clear the Format bit, since this is a Control, Bulk, or Interrupt endpoint. Also, clear the toggleCarry bit.

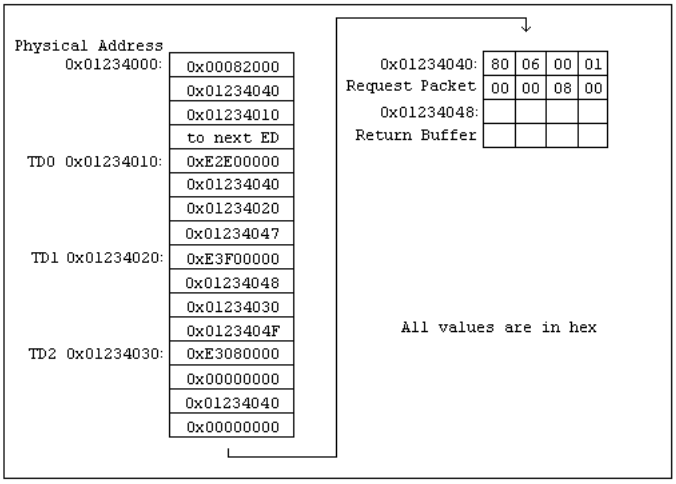
The MaximumPacketSize field should be set to 8 for now. At this time, we don't know what the maximum packet size the device will allow, since we have not received the device's descriptor yet. All devices will handle an 8-byte packet. Once we receive the first 8 bytes of the device's descriptor, the process we are currently requesting, we can retrieve the maximum packet size from the device's descriptor, update the drivers data for this device, and then request 8 or more bytes per request, depending on this value.

Now is where we make the endpoint descriptor point to the transfer descriptors we have already created. We set the HeadP field to point to the physical address of the first transfer descriptor, in our case, 0x01234010. The TailP field should point to the same place that the last transfer descriptors NextTD field points to, in this case physical address 0x01234040. This way, when the controller retires the last transfer descriptor, it compares that transfer descriptor's NextTD field with the endpoint descriptor's TailP field. If they are equal, the controller is finished processing this endpoint descriptor.

The NextED field will be set by the insertion routine described later.

Below is an updated look at the endpoint descriptor and transfer descriptors ready to be inserted in to the control list of our stack frame.

Figure 12-2: Request Return

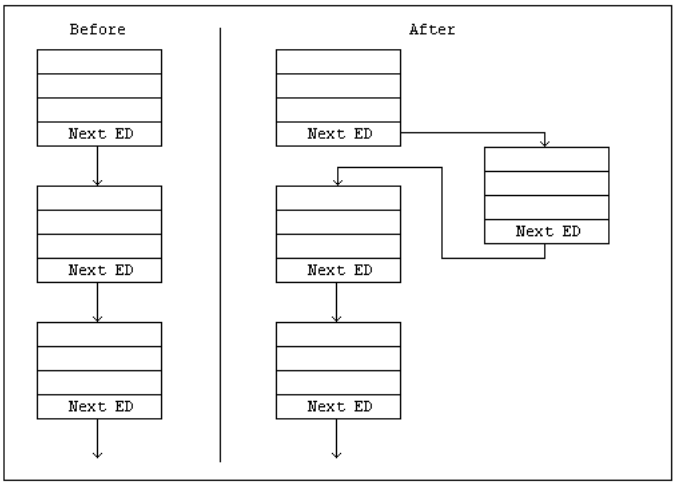


Now this endpoint descriptor is ready to be inserted in to the control list. To do so, you make sure that the control list is not currently being processed, by a number of techniques. If the HcFrRemaining register is less than the HcPeriodicStart register, the controller is processing the periodic list, therefore go ahead with the insertion. However, you will want to make sure that the value in the HcFrRemaining register is not so low that a new frame will start before your code will finish with the insertion.

To be sure that the control list is not being processed while you insert your endpoint descriptor, clear the ControlListEnable bit in the HcControl register. Since the clearing of the bit will not take effect until the next frame starts, you will need to wait for the next start of frame to happen. You can either watch the StartofFrame bit in the HcInterruptStatus register, or simply delay for 1 millisecond after clearing the ControlListEnable bit.

Since the control list is just a linked list of endpoint descriptors, simply link your new endpoint descriptor anywhere in the linked list.

Figure 12-3: ED Insert



If you insert your new endpoint descriptor at the end of the list, make sure to write 0x00000000 to the NextED field in your endpoint descriptor.

After the controller has processed your endpoint descriptor, you may remove it from the list. When you remove the endpoint from the list, simply write the value of the endpoint descriptor's NextED to the previous endpoint descriptor's NextED field. When removing an endpoint descriptor from the end of the list, simply write 0x00000000 to the previous endpoint descriptor's NextED field.

Make sure that you disable the control list, or make sure that the controller isn't processing the control list while you remove the endpoint descriptor.

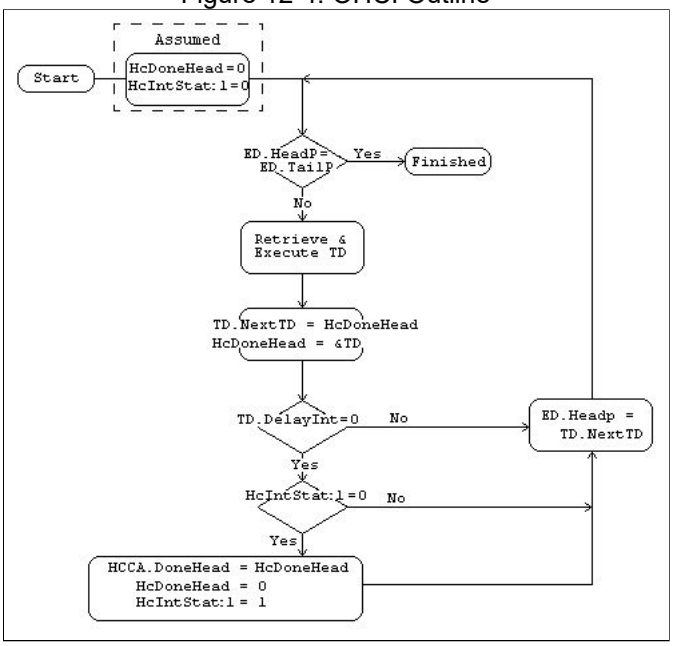
When the Controller Has Processed the Endpoint Descriptor

After the controller has processed the last transfer descriptor, assuming no errors, it will retire all transfer descriptors to a done queue. The done queue is an upside down list of retired transfer descriptors with the first of the list pointed to by the HcDoneHead register. Your driver usually doesn't need to access the controller's HcDoneHead register. Each time the controller retires a transfer descriptor, it writes the value of the HcDoneHead register to the NextTD field in the retired transfer descriptor, then writes the physical address of the transfer descriptor to the HcDoneHead register.

If you will remember in the description before, we set the DelayInterrupt field to zero for the last transfer descriptor. This tells the controller to retire the transfer descriptor as usual, changing the NextTD field and updating the HcDoneHead register as described above. Then it writes the HcDoneHead register's value to the DoneHead field in the HCCA (Host Controllers Communication Area), first checking the WritebackDoneHead bit in the HcInterruptStatus register for zero.

For example, let's walk through what happens when we insert our endpoint descriptor above using the outline below. Please note however, that this is a simplified version of what the controller really does. This outline will define what the controller will do with the example endpoint descriptor we described earlier. This outline also assumes that the HcDoneHead register is zero, bit 1 in the HcInterruptStatus register is clear, all transfer descriptor DelayInterrupt fields will either be 111b or 000b, and that no errors will occur.

Figure 12-4: OHCI Outline



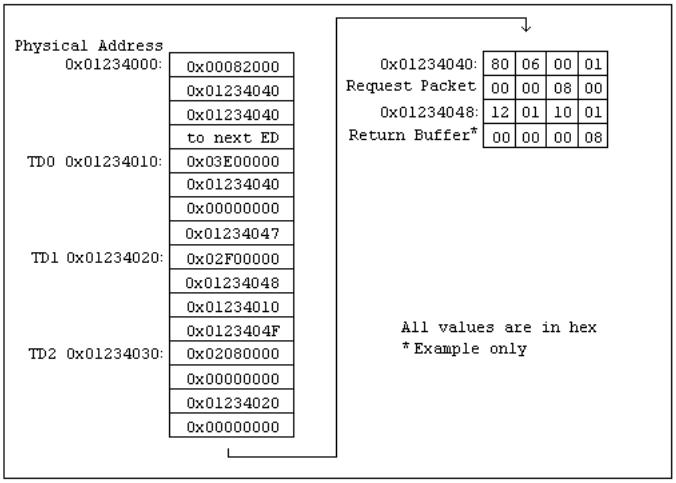
Each time the controller updates the HcDoneHead register, you have a linked list to all the retired transfer descriptors since the last HCCA.DoneHead update. Therefore, when we insert the endpoint descriptor in the earlier example, all we have to do, assuming no errors of course, is to wait for bit 1 in the HcInterruptStatus register to become set. Then read the value from HCCA.DoneHead and clear bit 1 in the HcInterruptStatus register for the next frame.

When you read the value from HCCA.DoneHead, remember to mask off bit 0 before you use it as a pointer to your retired transfer descriptors. Bit 0 is used to tell you that there might have been another reason for the interrupt. If you wait for an interrupt instead of polling bit 1 in the HcInterruptStatus register, your driver may expect the HCCA.DoneHead value to be valid. If HCCA.DoneHead is 0x00000001, that means there is another reason why the interrupt fired. If the value of HCCA.DoneHead is not zero and bit 0 is still set, the reason for the interrupt was because of the HCCA.DoneHead update and because of something else. Check the HcInterruptStatus register to found out what else also fired the interrupt.

Now that we have the value from HCCA.DoneHead, we now have a linked list to our transfer descriptors in reverse order. Using a conditional loop, we can check the error condition in each transfer descriptor until the NextTD field equals zero. If no errors were found, we have a successful transfer.

Here is the same diagram used before, the one before the insertion of the endpoint descriptor, now updated to show the status of each transfer descriptor after the HCCA.DoneHead update.

Figure 12-5: Request Return



At this point, you are probably wondering why we did not just request the whole 18 bytes of the descriptor at one time. There is no real reason other than preference and one more reason explained next. However, the first byte in the returned buffer tells us how long the device descriptor actually is. Also, the eighth byte is the max_packet_len byte, which tells us how many bytes we can read or write to endpoint zero, the control endpoint, at once. This tells us how many transfer descriptors we must have to retrieve the complete Device Descriptor. On a full-speed device, the max_packet_len could be 32 or even 64. If so, why request two 8-byte packets, and a single 2-byte packet, when you can request a single packet of 18 or more bytes?

The main reason is that some devices were only expected to work with the Windows Operating System. Win98SE for example, gets the first 8 bytes, does a reset of the port, sets the address of the device, and then finally gets all 18 bytes of the descriptor. Therefore, these few devices don't expect to send more than the first 8 bytes while in the default state, the state before it is given an address, and may not function properly until that sequence is received. It completely defies the given operation of the USB specification, but this is how some devices function. I have not personally seen or used any device that functions in this way, but have read documentation that states this might happen.

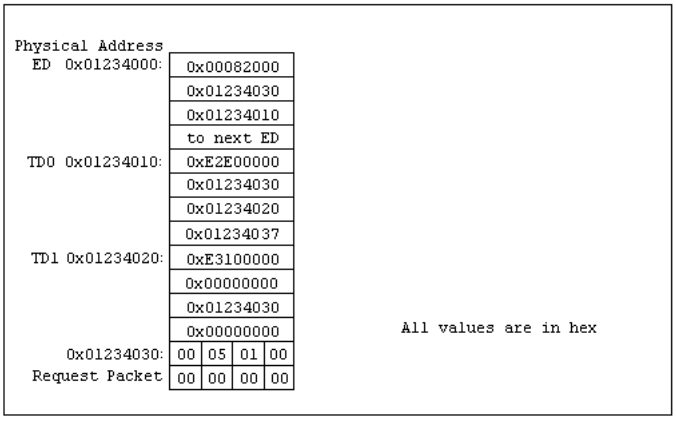
Usually, at this point you know if there is a valid device attached and that it is responding to your requests. Therefore, it is a good idea to reset the device, set its address to the next available address number, and retrieve the whole descriptor.

Setting the Address of the device

After you have successfully reset the device, you should now set the address. Up to this point the device has been in the default state. Most devices will handle most control requests while in this state, but you must set the address and move the device into the addressed state to configure the device for use.

To set the address, you send a SETUP packet similar to what we did before, except there is no data transfer. The address number to set the device to is sent in the SETUP packet, there is no data packet, and then there is a single STATUS packet. In this example, we will set the device's address number to 1, using the third byte in the request packet of the SETUP packet.

Figure 12-6: Set Address



Now you can get the whole descriptor using a similar queue as we used to get the first eight bytes. Using the descriptor size, the value in the first byte of the descriptor we got before, and adding or subtracting transfer descriptors depending on how many bytes you can receive in a single packet, and send the request again, being sure to update the max_packet_size field.

Getting the String Descriptor(s) of the Device

As mentioned before, a device may contain three or more string descriptors. A string descriptor is a descriptor of unknown size that contains 16-bit Unicode characters describing the device in human-readable form. I say unknown size, since each string may be of a different size. Then how do you get the size? Just as we did before to get the size of the device descriptor. Get the first eight bytes. Within the first eight bytes of the string descriptor is the descriptor's length. This tells you how many bytes to read.

Please note that in the previous paragraph, I state to get the first eight bytes. Now that you have retrieved the max_packet_size, you should actually get that amount of bytes instead, which could still be eight for low-speed devices.

There are two forms of string descriptors. The first is the language identification descriptor, index zero. It contains a list of 16-bit word values defining what language(s) this device supports. Once you have the list of LANGID's, as they are called, you can request the remaining string descriptors in any one of these languages, by sending the LANGID in the wIndex field of the Get String Descriptor request packet. To get the list of LANGID's supported, you call the Get String Descriptor request with an index of zero.

Table 12-3: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	Langid[0]	2	Value	Supported Language Code 0
4	Langid[1]	2	Value	Supported Language Code 1
6	Langid[n]	2	Value	Supported Language Code n

The table will hold as many language codes as it supports. To find out how many, simply take the total length, subtract two, then divide by two. With this in mind, a device can support up to 126 languages. Again, you need to get the first few bytes of the descriptor to find out how many more bytes to read.

Since the Length field in most descriptors is a byte-sized value, and can hold a maximum value of 255, instead of requesting eight bytes to get the length, then requesting the whole packet, you can request 255 bytes. Assuming 8-byte packets, simply set up an endpoint descriptor that has enough 8-bytes per packet transfer descriptors to hold 255 bytes, and set the bufferRounding bit in each transfer descriptor. The controller will stop executing the endpoint descriptor and start on the next endpoint descriptor at the first short packet it detects. However, you still need to send the status packet after detecting the short packet using an endpoint descriptor and a single transfer descriptor.

Once you have the LANGIDs, you can choose a language you would like to support. Then, each time you request a sting descriptor, place this LANGID code from above in the langid field in the request packet. A string will have a value of 1 or higher in the index field of the request packet and have the format listed here.

Table 12-4: String Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor

2	String	n	xx	Unicode Encoded String.

The string is in 16-bit Unicode encoding and must not be zero-terminated. You must get its size from the Length field in the descriptor, remembering that the Length field includes the first two bytes of the descriptor.

Some of these LANGID codes are:
0x0409 - English - United States
0x0C09 - English - Australian
0x0407 - German - Standard
0x1009 - English - Canadian
0x040C - French - Standard
0x080A - Spanish - Mexican

Notice that the English languages have the low byte as 0x09. This isn't a coincidence. The Primary Language is a 10-bit value in the lower ten bits, with English being 0x009. The sub-language identifier is in the high six bits of the high byte. For the complete list of LANGID codes, see the file USB_LANGIDs.pdf from the USB-IF web page (See Appendix A).

Getting/Setting the Configuration of the Device

Before most devices will function properly, you must request the configuration descriptor(s), decide which one you want to use, and set that configuration. Recall in an earlier chapter, the device descriptor shows how many configurations a device may have. Though most devices only have one, a device may have two or more configurations. For example, the first configuration may be used when the device is self-powered, and the second configuration may be used when the device is bus-powered.

The configuration descriptor also contains all the interface descriptors for the device along with each endpoint descriptor for each interface, not to be confused with the OHCI endpoint descriptor, they are two totally different descriptors. The Length field contains the size of the configuration header, while the TotalLength field contains the total length of all data returned, the header, each interface, and each endpoint.

You can get the different configurations returned by giving the desired index in the request packet.

Table 12-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x02	Config Descriptor
2	TotalLength	2	Value	Total length of all data.
4	NumInterfaces	1	Value	Number of Interfaces.
5	ConfigVal	1	Value	Value for this configuration
6	ConfigString	1	Value	Index for String Descriptor describing this config.
7	Attributes	1	Bitmap	D7 Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	mA	Maximum Power Consumption

To set the desired configuration, you send the SetConfiguration request with the desired ConfigVal in the request header.

Each configuration descriptor returns one or more interfaces. A device may have more than one interface. For example, a numpad with the usual number keys, enter, plus and minus keys, similar to the right hand part of a regular full sized keyboard, may also be used as a pointing device. The first interface may be the keypad's individual keys, while the second interface may be certain keys on the numpad used for direction and button press/release information. You cannot request an interface descriptor as you would a device descriptor using the GetDescriptor request. You retrieve the interface descriptor from the data returned after the configuration descriptor as described above.

Table 12-6: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x04	Interface Descriptor
2	Interface Num	1	Value	Number of Interface.
3	Alternate Set	1	Value	Used to select alt. setting.
4	Endpoints	1	Value	Number of Endpoints.

5	Class Code	1	Value	Class Code
6	Sub Class	1	Value	Sub Class Code
7	Protocol	1	Value	Protocol Code
8	Interface	1	Value	Index of String Descriptor describing this interface.

Depending on the interface, each interface may have zero or more endpoints. Usually, the first endpoint is the control endpoint and has an endpoint address of zero, which is assumed and usually not included in the configuration descriptor. The second endpoint may be an interrupt endpoint or a bulk endpoint. The Attributes field below specifies what type of endpoint it is, with bit 7 in the Address field specifying which direction, in or out.

Table 12-7: Endpoint Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x05	Endpoint Descriptor
2	Address	1	Value	Endpoint address: Bits 3:0 Endpoint Num 6:4 reserved (0) 7 0 = Out, 1 = In
3	Attributes	1	Bitmap	Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 7:2 Reserved unless it is an ISO endpoing, then: Bits 3:2 Sync type 00 = No Sync 01 = Ascynchronous 10 = Adaptive 11 = Synchronous Bits 5:4 Usage Type 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback 11 = Reserved Bits 7:6 reserved
4	MaxPacketSize	2	Value	Maximum packet size this endpoint is capable of.
6	Interval	1	Value	Interval for polling endpoint data transfers. Ignored by Bulk and Control, must be 1 for ISO, and can be 1 to 255 for Interrupt.

Using the technique explained in the first part of this chapter, you should be able to get all the descriptors explained to this point. In Part 3, I will explain the process to send and receive data from specific types of devices, such as keyboards, mice, and mass storage devices.

Chapter 13 – Device Enumeration with the EHCI

In this chapter, I will explain how to enumerate the devices on the EHCI root hub. Enumeration means to number or name one by one, the devices on the root hub. By doing this, we can find all devices attached, give them an address number, and allow them to start communicating with the host.

Here, in this chapter, I will use the EHCI controller for my examples. If you are going to use one of the other controller types, you may skip this chapter and go to the chapter specific for that controller. Those chapters have some of the same information, descriptions, and data tables as this chapter, but I thought it would be confusing to the reader to keep flipping pages, between controllers, to find a packet description, or other identically used items. Therefore, even though there are duplicate descriptions in these chapters, there is no need to flip back to the OHCI section to find a table, when using EHCI examples, for example.

If you have read one of the other chapters within Part 2, you will find some of the information here in this chapter repeated. This is intentional. These chapters were written independent of each other. One should not have to read one chapter to use the other. With this said, let's go on to the EHCI enumeration.

Let's assume that you already have set up the EHCI root hub hardware, have detected that there is a single device plugged into one of the ports and it has been reset and enabled correctly. We also assume that the device is a high-speed device. Any low- or full-speed device will be controlled with the OHCI or UHCI controllers.

With this in mind, we know that a newly attached device will respond to a device address of zero, so let's make our first packet transfer on the USB stack.

Driver Initialization and the BIOS

If your platform's BIOS supports the EHCI controller, you will need to indicate that you want it to release the controller so that your driver will have full access. If you do not do this, the BIOS may modify the controller's registers behind your back. If your BIOS does not support the EHCI controller, it is still good practice to start with the code listing below within your driver.

Listing 13-1: Release BIOS Ownership

```
/* Release BIOS ownership of controller
* On Entry:
*   list: pointer to the start of the Capability list
* On Return:
*   TRUE if ownership released
*
* Set bit 24 to indicate to the BIOS to release ownership
* The BIOS should clear bit 16 indicating it has successfully done so
* Ownership is released when bit 24 is set *and* bit 16 is clear.
* This will wait xHC_xECP_LEGACY_TIMEOUT ms for the BIOS to release
*  ownership. (It is unknown the exact time limit that the BIOS has
*  to release ownership.)
*/
bool xhci_stop_legacy(void *list) {
  struct xHC_xECP_LEGACY *const xECP = (struct xHC_xECP_LEGACY *) list;

  // set bit 24 asking the BIOS to release ownership
  xECP->owner_flags |= xHC_xECP_LEGACY_OS_OWNED;

  // Timeout if bit 24 is not set and bit 16 is not clear
  //  after xHC_xECP_LEGACY_TIMEOUT milliseconds
  int timeout = xHC_xECP_LEGACY_TIMEOUT;
  while (timeout--) {
    if ((xECP->owner_flags & xHC_xECP_LEGACY_OWNED_MASK)
                                == xHC_xECP_LEGACY_OS_OWNED)
      return TRUE;
    mdelay(1);
  }
  return FALSE;
}
```

The struct definition:

```
#define xHC_xECP_LEGACY_TIMEOUT     10    // in milliseconds
#define xHC_xECP_LEGACY_BIOS_OWNED  (1<<16)
#define xHC_xECP_LEGACY_OS_OWNED    (1<<24)
#define xHC_xECP_LEGACY_OWNED_MASK   \
              (xHC_xECP_LEGACY_BIOS_OWNED | xHC_xECP_LEGACY_OS_OWNED)
struct xHC_xECP_LEGACY {
  bit32u volatile owner_flags;
  bit32u volatile cntrl_status;
};
```

Please note the volatile keyword in the structure definition above. If you omit this keyword, your compiler may optimize the code and never actually check the values to see if they change.

Creating Queue Heads and Transfer Descriptors

The Operating System, or your driver, has no idea what kind of device has been attached. Therefore, each device has a standard device descriptor, along with other descriptors, that tells the operating system what kind of device it is. The device descriptor tells us a lot about the attached device. For example, we can see if the device complies to the version of the USB specification that our driver supports, it tells us the class code and sub-class code of the device, as well as what string descriptors it has, giving us a plain text description of the device. The class and subclass codes are probably the most important of these items. For example, if the device is a Mass Storage Device (MSD), it should have a class code of 0x08. It will also have a descriptor that gives us an interface type, which may be SCSI, RBC, MMC-2 for CD-ROMs, UFI for floppies, or some other form of communication. Without this information, we would not know how to communicate with the device and send and receive sector data. However, please note that if the class code is zero, the class and subclass codes should be retrieved from the interface descriptor, described later in this chapter.

The format of the Standard Device Descriptor is as follows:

Table 13-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	0x12	Size of this descriptor in bytes. For the Standard Device Descriptor, this should be 18 bytes.
1	Type	1	0x01	Device descriptor type. For the Standard Device Descriptor, should be 1.
2	Release Num	2	BCD	USB Specification Release Number in Binary-Coded Decimal. i.e., 2.10 = 0x0210. This field identifies the release of the UBS Specification revision with which the device and its descriptors are compliant with.
4	Device Class	1	Value	The Device Class code assigned by the USB-IF.[1]
5	Sub Class	1	Value	The Device Sub Class code.[1]
6	Protocol	1	Value	Protocol Code.[1]
7	MaxPacketSize	1	Value	Max packet size for endpoint zero (8, 16, 32, or 64)
8	Vendor ID	2	Value	Vendor ID
10	Product ID	2	Value	Product ID

12	Device Rel	2	BCD	Device release number in Binary-Coded Decimal
14	Manufacturer	1	Index	Index of string descriptor[2]
15	Product	1	Index	Index of string descriptor[2]
16	Serial Num	1	Index	Index of string descriptor[2]
17	Configurations	1	Value	Number of possible configurations.

[1]http://www.usb.org/developers/defined_class

[2]A value of zero means no string descriptor

Getting the Device Descriptor of the Device

To get this device descriptor from the device, we must send it a request packet called a Setup packet. The setup packet is an 8-byte data packet that consists of the following format and is the same for any type of device.

Table 13-2: Device Request Packet

Offset	Field	Size	Value	Description
0	Request type	1	Value	Bit 7 Transfer Direction 0 = Host to device 1 = Device to host Bit 6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	Value	The Request desired
2	Value	2	Value	Depends on the request
4	Index	2	Index	Depends on the request
6	Length	2	Count	If a data phase, number of bytes to transfer.

This is the data we will send to a device that has a device address value of zero. So how do we send this request to the device and receive the device descriptor data? We have to place a transfer descriptor on the USB stack, three to be exact, inserted into the asynchronous schedule. Please note that we request all 18 bytes of the device descriptor. Being a high-speed device, we don't have to get the first 8 bytes like the UHCI or OHCI techniques. See later for the values we place in this request packet.

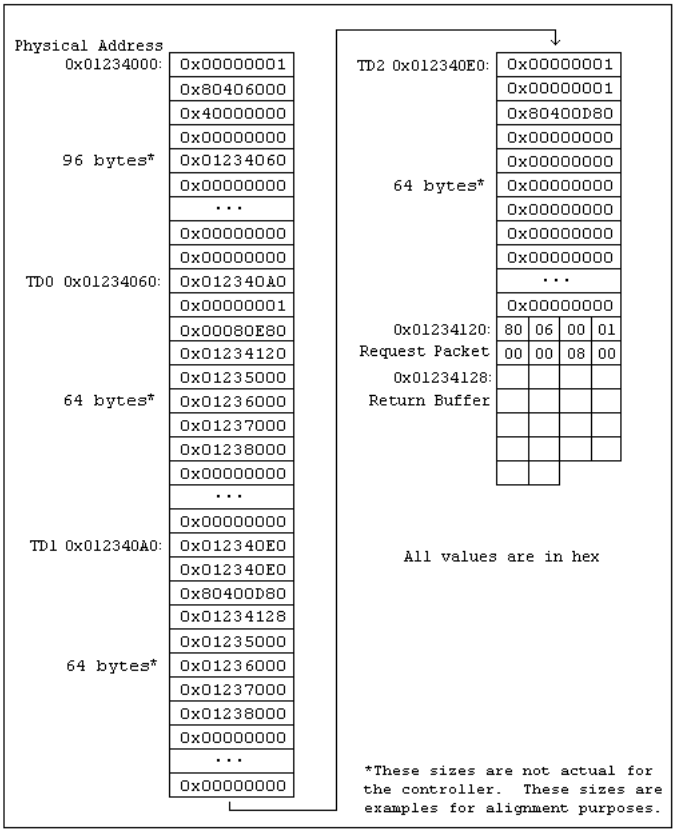
Please note on high-speed devices, you might have to request all 18 bytes or the device may not function correctly. Then, if it works, you don't have to request the descriptor again. However, note that low- and full-speed devices may not function at all unless you request only 8 bytes at first.

We need a buffer that is large enough to hold the queue head, the three transfer descriptors, the 8-byte request packet, and the 18-byte return packet. Since each queue head is 96 bytes, a transfer descriptor is 64 bytes, we need a minimum of 314 bytes and the queue head and transfer descriptor buffers must be 32-byte aligned. The FYSOS kernel has the amalloc() function which accepts an alignment value and returns an aligned allocated memory to that alignment. Therefore, I can simply call it to return a minimum of 314 bytes. For the purpose of this example, let us allocate 512 bytes, and say that it returns a memory block allocated at 0x01234000.

Please note that I used amalloc(). You can place these buffers on the stack or even have them be global buffers in your code. However, you must be able to get the physical address of the buffer, not the linear address. If your OS is using paging or another form of linear addressing, be sure to find a way to get the physical address of the buffer. I used amalloc() in the examples here assuming that the address returned is a physical address.

If we place the queue head, transfer descriptors, the request packet, and the return buffer all in the same memory buffer, we can do something like the illustration in Figure 13-1 below, in an EHCI stack frame.

Figure 13-1: EHCI Request



For the Queue Head, we ignore the Horizontal Link Pointer field for now, placing a 0x00000001 value in its place. Later, the Queue Insert routine will set this field for us.

In the next dword, we set the address and endpoint fields to zero and the max packet size field to 64 for now, assuming this is a high-speed device on the root hub. If it were a low-speed device through a high-speed external hub, you would set this field to 8. The EPS field is set to 10b for high-speed and the Data Toggle Control bit is set to tell the controller the data toggle is taken from the transfer descriptor instead of the queue head. The other fields are either zero or ignored at this point.

The next dword only has the Multiplier field set to 01b for one transaction per micro-frame. All other fields within this dword are set to zero since we are not dealing with an external hub.

For now, that is it for the Queue Head. The insert routine will modify a few of the fields above, which I will detail later. For now, let us move on to the three transfer descriptors.

For the first transfer descriptor which is located at physical address 0x01234060, we set the items for a SETUP packet. The Next Pointer points to the next transfer descriptor at 0x012340A0, the Alternate Next Pointer field is set to 0x00000001 since we shouldn't have a short packet detect, the Status field set to 0x80 for active, the PID Code set to 10b for SETUP, and the CERR field set to 11b for three tries.

The C_Page field is set to zero so that we use the first buffer pointer, the IOC field is cleared since we don't need an Interrupt on Complete at this point, the Total Bytes to Transfer field set to eight, the size of the Setup packet, and the Data Toggle bit clear since this is the first transfer descriptor.

The next dword is the physical pointer to the Setup packet. The controller will transfer all of the bytes, as long as the count is still more than zero, up to the next page boundary. Then in increments the C_Page field and moves to the next Buffer Pointer field. Since we are transferring only eight bytes, you may think that you only need the first Buffer Pointer field. However, what if your buffer is four bytes before the page boundary. If you do not set the next Buffer Pointer field, the controller will not receive the last four bytes of the packet. It is up to you whether you check the boundary and how many of the Buffer Pointer fields you initialize. It does not hurt to initialize all of them even though they will not be used. Refer to Figure 8-1 in Chapter 8 for more information on this.

There is nothing wrong with having your driver's queue head and transfer descriptor data structure larger than the physical amount the controller requires. The FYSOS driver uses a 96-byte structure for the queue head and a 64-byte structure for the transfer descriptor. The first dwords are exactly as the controller requires them, while the remaining dwords are used by the driver for transfer identification and other information. The controller doesn't require that the next transfer descriptor be 32 bytes away from the current one, so use as many bytes as you would like. However, make sure that the size of your data structure is a multiple of 32, so that it will be correctly aligned when calculate the address for the next one. The other fields will be detailed later in this chapter.

That was the Setup packet, which is responsible for telling the device that we are requesting its Device Descriptor. Now we need to create another transfer descriptor telling the device where to copy the requested eighteen bytes.

The second transfer descriptor starts at physical address 0x012340A0 and the fields in this transfer descriptor are very similar to the first transfer descriptor.

The only difference other than the pointers is that it now requests 64 bytes transferred, the Data Toggle bit is now set, and the PID Code is now set to 01b for an IN transfer. Please note that the Alternate Next Pointer is now used. It points to the Status transfer descriptor since we could and do expect a short packet detect, since we requested 64 bytes.

The USB specification states that we must send a zero length OUT packet to the device to tell the device we have received the data. With this in mind, the third transfer descriptor is almost exactly the same as the second except that we change the PID field to and OUT (00b). We also change the Data Toggle bit to 1 (which is actually the same since we only had one IN packet), and give it a length of zero. Make sure to also write a value of 0x00000001 to the Next TD Pointer so that the controller knows that this is that last transfer descriptor in this queue.

Right now we assume that the data will be transferred without error. Usually, your driver won't send the STATUS packet (the third transfer descriptor) until it actually receives a valid transfer of the device descriptor. However, for the purpose of this demonstration, we assume no errors.

As for the Data Toggle bit, for control transfers, you always start with zero, then toggle it for each consecutive transfer descriptor after that, within the current data transfer. Then, the last transfer descriptor, the Status packet, will always be set.

In the Setup packet, we send eight bytes of information. Using Figure 13-1, we can see that we set the Request Type field to 0x80. The requested data is coming from the device to the host with a standard request type for a device. The Request is the GET_DESCRIPTOR request, which has a value of 0x06, and the Value field contains two byte values. The high byte contains a value of 0x01 for DEVICE, while the low byte contains 0. These two bytes are

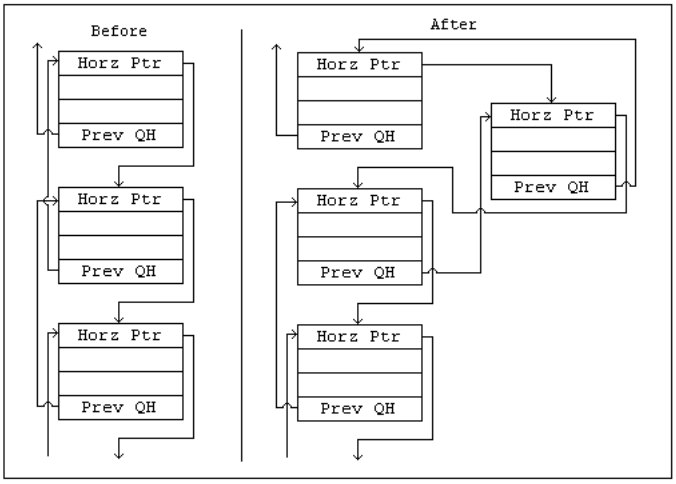
written as a 16-bit word in little-endian format. The Index field contains zero, and the Length field contains 64, the number of bytes we are requesting.

Inserting a Queue into the Stack

Now we are ready to insert our queue head and transfer descriptors into the EHCI's stack, which in this case will be the asynchronous schedule. If you will notice, in Figure 13-1, the Head Link Pointer field was set to 0x00000001 which until now was simply a safety value, incase this queue head was inserted prematurely. Also, if you recall, we have unused space after the queue head for our own use. This will come in handy in a few moments.

To insert our queue head into the current schedule, we simply need to choose where to insert it. Once we do, all we have to do is record the previous queue's head link pointer into our new queue's head link pointer and replace the previous one with the address of this one. That makes it very simple to insert a queue head. For now, the only other thing to do is save the physical address of the previous queue head in the unused area of our queue head structure. This will be explained in the queue head removal section later in this chapter.

Figure 13-2: Queue Head Insert



Please note that the controller does not use the previous address pointer. It is only used by our software to insert and remove queue heads.

Notice that each queue head simply points to the next one and also the previous one. This way we may insert or remove a queue head anywhere in the schedule. Once the controller is done with all of the transfer descriptors within one queue head, it will simply move to the next queue head. Here is a code sample from the FYSOS driver, where 'queue' points to the queue being inserted and 'cur_queue' points to the queue that we are inserting this queue head after.

```
queue->horz_ptr = cur_queue->horz_ptr;
cur_queue->horz_ptr = (bit32u) queue | type;  // type of ptr
queue->prev_ptr = (bit32u) cur_queue;
```

Please note that the first queue head usually has the 'H' bit set and that the last queue head always points to the first queue head. This makes a Round-Robin type schedule.

Unlike the OHCI scheduler, for the EHCI asynchronous schedule, you do not have to stop the schedule to insert a queue head. The controller expects the list to be active when you insert a queue head. Therefore, it contains a cached copy of the queue head's horizontal pointers and updates the cache only when it is safe to do so. Therefore, you may insert a queue head at any time, but note that it may take a few cycles of the schedule for the controller to reload its cache with the newly inserted queue.

Removing a queue from the Stack

Removing a queue from the schedule is not quite as simple as inserting one though very similar. To remove the queue from physical memory, you simply restore the horizontal pointer of the previous queue and the previous pointer of the next queue. Here is a pseudo code sample from the FYSOS driver, where 'queue' points to the queue being removed.

```
queue->prev_ptr->horz_ptr = queue->horz_ptr;
(queue->horz_ptr & ~0x1F)->prev_ptr = queue->prev_ptr;
```

As mentioned earlier, the controller holds a cache of the queue head pointers. Therefore, when we physically remove a queue, we should not modify the transfer descriptors in the queue until the controller is ready for us to do so. The controller has a mechanism to tell us when we can safely physically remove a queue from the current schedule. This is called the Asynchronous Schedule Advance Doorbell.

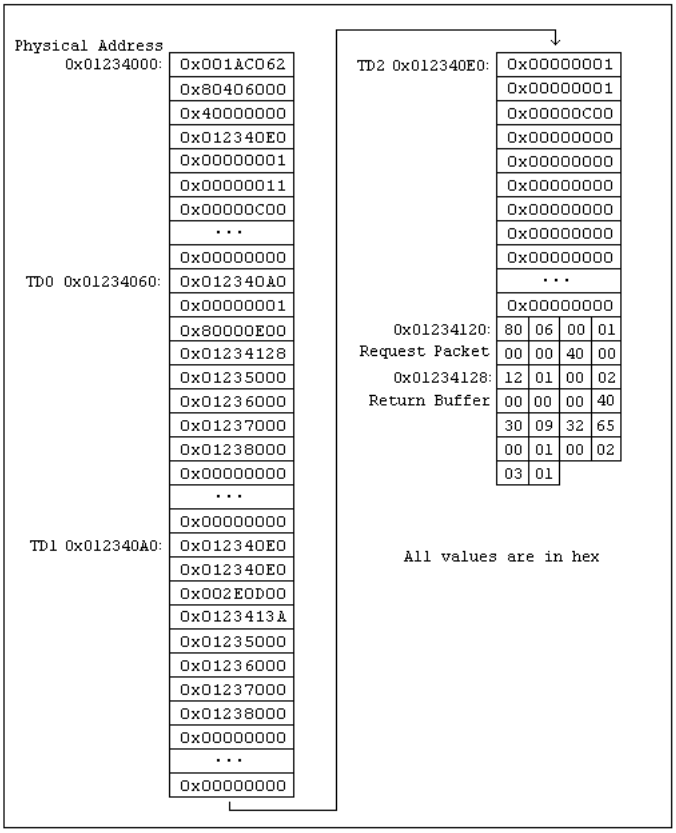
To receive this doorbell, you set bit 6 in the Command register and then wait for bit 5 to be set in the Status register. Once this happens, you can assume that the controller will not execute the removed queue's transfer descriptors and you may modify the memory they occupy as you wish. This is simply a safety mechanism to make sure that the queue you are removing is no longer in the controllers cache of queue pointers before you physically modify the contents of the queue.

Controller Has Processed the Transfer Descriptors

After the controller has processed the last data transfer descriptor, assuming no errors, you will have received an interrupt due to the IOC bit being set, and then you have already sent the status transfer descriptor. Therefore we should now have a successful transfer and our memory buffer should now contain the device's Device Descriptor.

Below is the same diagram used earlier, the one before the insertion of the queue head, now updated to show the status of the queue head and each transfer descriptor.

Figure 13-3: Request Return



Notice that the last transfer descriptor executed, the Status transfer descriptor, now overlays the overlay part of the queue head. Remember that the overlay is so that the controller knows where to continue execution if it stops mid-queue to go the periodic schedule.

The status of the transfer will be the seventh dword down with a value of 0x00000C00. The Bytes to Transfer field is now zero, the C_Page field is still zero since all bytes were within the first page used, the Cerr field is still 11b, since there were no errors, and the Status field is 0x00, meaning no errors were detected. If you will notice, the queue overlay has the NakCnt field where the transfer descriptor does not. This is the value we put in the RL field of the queue head.

The first transfer descriptor has a status dword of 0x80000E00. The Data Toggle bit is still set, Total Bytes to Transfer is zero as expected, C_Page is zero, since we only used the first Buffer Pointer, the Cerr field is still set to 11b, the PID is unchanged, and the status is zero, no errors.

The second transfer descriptor is the interesting one. The Data Toggle bit is clear as expected, but the Bytes to Transfer field is set to 0x2E (46d). This is expected. We requested 64 and received 18, therefore 64 minus 18 is 46. This is also where the controller senses a short packet detect and uses the Alternate Next TD Pointer field to skip the remaining data transfer requests and go directly to the Status transfer descriptor, though in this particular set there is no more and just goes to the next one anyway. Also note that the Current Offset field now points to the byte just after the 18th byte transferred and that our buffer is now filled with the device's Device Descriptor.

The last transfer descriptor was the last executed so it is the one that was copied to the overlay area of the queue head. However, notice that it was updated here first, then copied.

Since this is a high-speed device, we requested more than the actual length of the Device Descriptor. Therefore, we only need to do this once. The first byte in the returned buffer tells us how long the device descriptor actually is. Also, the eighth byte is the max_packet_len byte, which tells us how many bytes we can read or write to endpoint zero, the control endpoint, at a time. If it is different than the current value of 64, this is where you would save the new value for your code.

Please note that there may be a few devices that require a requested amount of 8 as the first device request, then a reset, then request the actual length, just as the sequence we use for low- and full-speed devices discussed in the UHCI and OHCI chapters. However, very few high-speed device will require this quirk.

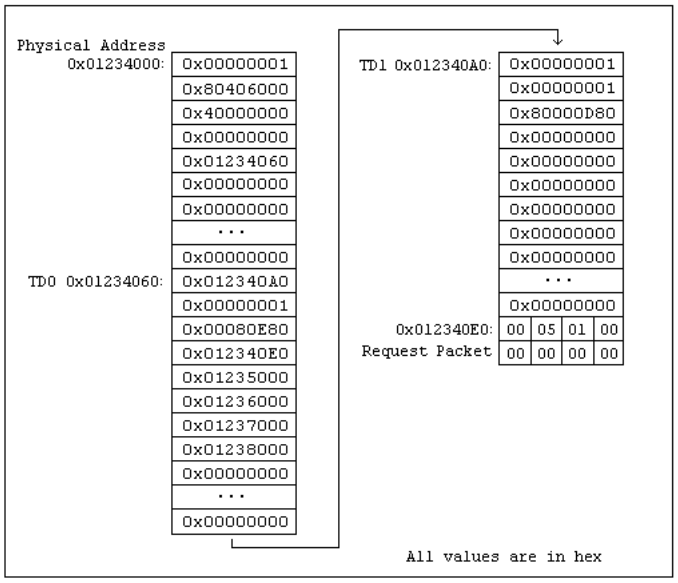
Usually, at this point you know if there is a valid device attached and that it is responding to your requests. Therefore, it is a good idea to reset the device, set its address to the next available address number, and retrieve the whole descriptor again.

Setting the Address of the device

After you have successfully reset the device, you should now set the address. Up to this point the device has been in the default state. Most devices will handle most control requests while in this state, but you must set the address and move the device into the addressed state to configure the device for use.

To set the address, you send a Setup packet similar to what we did earlier, except there is no data transfer. The address number to set the device to is sent in the Setup packet, there is no data packet, and then a single Status packet. In this example, we will set the device's address number to 1, using the third byte in the request packet of the Setup packet.

Figure 13-4: Set Address



Getting the String Descriptor(s) of the Device

As mentioned before, a device may contain three or more string descriptors. A string descriptor is a descriptor of unknown size that contains 16-bit Unicode characters describing the device in human-readable form. I say unknown size, since each string may be of a different size. Then how do you get the size? Just as we did before to get the size of the device descriptor. Get the first 64 bytes. Within the first eight bytes of the string descriptor is the descriptor's length. This tells you how many additional bytes to read.

There are two forms of string descriptors. The first is the language identification descriptor, index zero. It contains a list of 16-bit word values defining what language(s) this device supports. Once you have the list of LANGID's, as they are called, you can request the remaining string descriptors in any one of these languages, by sending the LANGID in the wIndex field of the Get String Descriptor request packet. To get the list of LANGID's supported, you call the Get String Descriptor request with an index of zero.

Table 13-3: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	Langid[0]	2	Value	Supported Language Code 0
4	Langid[1]	2	Value	Supported Language Code 1
6	Langid[n]	2	Value	Supported Language Code n

The table will hold as many language codes as it supports. To find out how many, simply take the length, subtract two, then divide by two. With this in mind, a device can support up to 126 languages. Again, you need to get the first 64 bytes of the descriptor to find out how many more bytes to read.

Since the Length field in most descriptors is a byte-sized value, and can hold a maximum value of 255, instead of requesting 64 bytes to get the length, then requesting the whole packet, you can request 255 bytes. Assuming 64-byte packets, simply set up a queue that has enough transfer descriptors to hold 255 bytes, making sure to have a valid Alternate Next Pointer in each descriptor. The controller will stop executing the queue and start on the next queue at the first short packet it detects. Remember to still send the status packet after detecting the short packet.

Once you have the LANGIDs, you can choose a language you would like to support. Then, each time you request a string descriptor, place this LANGID code from above in the langid field in the request packet. A string will have a value of 1 or higher in the index field of the request packet and have the format listed below.

Table 13-4: String Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	String	n	xx	Unicode Encoded String.

The string is in 16-bit Unicode encoding and must not be zero-terminated. You must get its size from the Length field in the descriptor, remembering that the Length field includes the first two bytes of the descriptor.

Some of these LANGID codes are:
0x0409 - English - United States
0x0C09 - English - Australian
0x0407 - German - Standard
0x1009 - English - Canadian
0x040C - French - Standard
0x080A - Spanish - Mexican

Notice that the English languages have the low byte as 0x09. This isn't a coincidence. The Primary Language is a 10-bit value in the lower ten bits, with English being 00_0000_1001b. The sub-language identifier is in the high six bits of the high byte. For the complete list of LANGID codes, see the file USB_LANGIDs.pdf from the USB-IF web page (See Appendix A).

Getting/Setting the Configuration of the Device

Before most devices will function properly, you must request the configuration descriptor(s), decide which one you want to use, and set that configuration. Recall earlier, the Device Descriptor shows how many configurations a device may have. Though most devices only have one, a device may have two or more configurations. For example, the first configuration may be used when the device is self-powered, and the second configuration may be used when the device is bus-powered.

The configuration descriptor also contains all the interface descriptors for the device along with each endpoint descriptor for each interface. The Length field contains the size of the configuration header, while the TotalLength field contains the total length of all data returned, the header, each interface, and each endpoint.

You can get the different configurations returned by giving the desired index in the request packet.

Table 13-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x02	Config Descriptor
2	TotalLength	2	Value	Total length of all data.
4	NumInterfaces	1	Value	Number of Interfaces.
5	ConfigVal	1	Value	Value for this configuration
6	ConfigString	1	Value	Index for String Descriptor describing this config.
7	Attributes	1	Bitmap	D7 Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	mA	Maximum Power Consumption

To set the desired configuration, you send the SetConfiguration request with the desired ConfigVal in the request header.

Each configuration descriptor returns one or more interfaces. A device may have more than one interface. For example, a high-speed cellular phone may have two interfaces. One interface to simply interface with the device as an external drive, to retrieve pictures and/or data. The other interface may be to connect to the Internet via the cell phone's cellular service. You cannot request an interface descriptor as you would a device descriptor using the GetDescriptor request. You retrieve the interface descriptor from the data returned after the configuration descriptor as described above.

Table 13-6: Interface Descriptor

Offset	Field	Size	Value	Description

0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x04	Interface Descriptor
2	Interface Num	1	Value	Number of Interface.
3	Alternate Set	1	Value	Used to select alt. setting.
4	Endpoints	1	Value	Number of Endpoints.
5	Class Code	1	Value	Class Code
6	Sub Class	1	Value	Sub Class Code
7	Protocol	1	Value	Protocol Code
8	Interface	1	Value	Index of String Descriptor describing this interface.

Depending on the interface, each interface may have zero or more endpoints. Usually, the first endpoint is the control endpoint and has an endpoint address of zero, and is assumed and usually not included in the configuration data. The second endpoint may be an interrupt endpoint or a bulk endpoint. The Attributes field below specifies what type of endpoint it is, with bit 7 in the Address field specifying which direction, in or out.

Table 13-7: Endpoint Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x05	Endpoint Descriptor
2	Address	1	Value	Endpoint address: Bits 3:0 Endpoint Num 6:4 reserved (0) 7 0 = Out, 1 = In
3	Attributes	1	Bitmap	Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 7:2 Reserved unless it is an ISO endpoing, then: Bits 3:2 Sync type 00 = No Sync 01 = Ascynchronous 10 = Adaptive 11 = Synchronous

				Bits 5:4 Usage Type 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback 11 = Reserved Bits 7:6 reserved
4	MaxPacketSize	2	Value	Maximum packet size this endpoint is capable of. (Bits 10:0 = mps)
6	Interval	1	Value	Interval for polling endpoint data transfers. Ignored by Bulk and Control, must be 1 for ISO, and can be 1 to 255 for Interrupt.

Using the technique explained in the first part of this chapter, you should be able to get all the descriptors explained to this point. In Part 3, I will explain the process to send and receive data from specific types of devices, such as a mass storage device.

Chapter 14 – Device Enumeration with the xHCI

Within this chapter, I will discuss the new USB 3.0 protocol the xHCI controller uses to control attached devices, the new cables, and other items associated with them.

If you are planning to write a driver, embedded application, or other system software to control the xHCI controller and attached devices, you will need to also read Chapters 9 and 10 in Part 1 of this book. These chapters explain the controller's registers, memory usage formats, and other similar items that you will need to know before using some of the information in this chapter. Once you have familiarized yourself with those chapters, you may start with this chapter where I will explain this new controller used to communicate with the new super-speed devices.

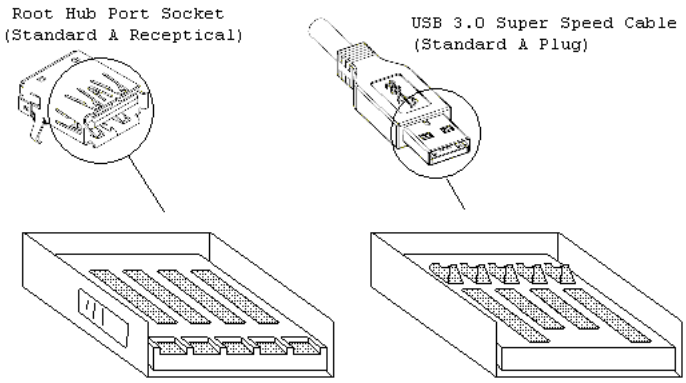
I will also explain how the controller is backward compatible with USB 2.0 devices, how the new cables and the Root Hub's ports allow USB 3.0 and the older USB 2.0 devices to be attached, along with new features of the USB 3.0 specification and its supported devices.

After I have discussed the new features of the new xHCI, I will then explain in detail how to write system software to control the xHCI controller, and how to communicate with attached devices. Shall we begin?

The New Cables and Backward Compatibility

So that the user may plug an existing USB 2.0 cable or the new USB 3.0 cable into the same USB 3.0 Root Hub the dimensions of the connectors have to remain the same size. However, the new connectors now contain five more pins placed just behind the existing four. See the figure below.

Figure 14-1: USB 3.0 Socket and Plug

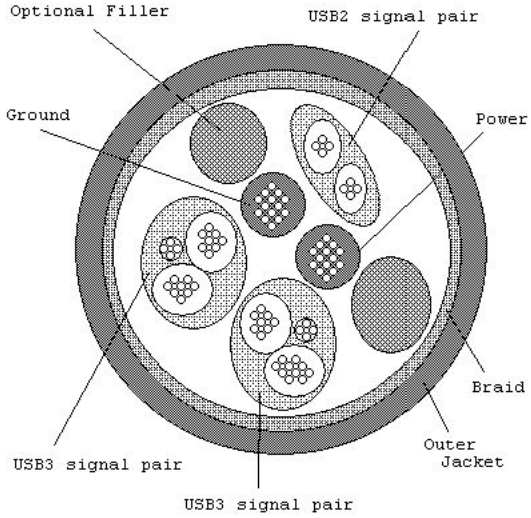


In the above figure, the image on the left is a socket of a xHCI root hub. It is almost identical to the USB 2.0 socket except that it now has 5 recessed connector plates at the front of the socket. Since these are recessed, a USB 2.0 Standard 'A' Plug will slide right in using the existing 4 connector plates. As for the mechanical hardware, there is nothing different, the USB 2.0 plug fits right in.

The image on the right is a Standard 'A' Plug for a USB 3.0 Super-Speed device. Notice that it has 5 spring-loaded pins in the back that will contact the 5 recessed connector plates on the socket. To imagine this, imagine flipping the image on the right, over and turning it 180 degrees. In a USB 2.0 socket, these pins tuck safely away.

These new pins then require more wires within the cable. The image below shows these new wires.

Figure 14-2: USB 3.0 Cable Cross-section



There is still a pair for the USB 2.0 communication, one power wire and one ground wire, but now includes two more pairs with each of these pairs sharing a drain wire. These additional two pairs with their drain wire, are both used for the super-speed communications. The color coding for these wires are listed below.

Table 14-1: Corresponding Cable Wiring Colors

Color	Corresponding wire
Red	Power
Black	Ground
White/Green	Unshielded twisted pair for USB 2.0 communications.
Blue/Yellow	Shielded differential pair #1 for USB 3.0 Super Speed communications.
Purple/Orange	Shielded differential pair #2 for USB 3.0 Super Speed communications.

The two drain-wires and the grounding braid are any color as long as it does not interfere with the color-coding already used.

Is it a USB 2.0 or USB 3.0 Port?

By now you are probably wondering how to tell which ports are USB 2.0 and which are USB 3.0. First, if you physically look at the socket and see that there are the five connecting slots on the front of the socket then it is a USB 3.0 port. However, how does software know which ports are USB 2.0 only or USB 3.0 with backward compatibility?

This is where the controller's Extended Capabilities registers are used. In Chapter 9, I described the format of these registers and how they are used. I explained that if a USB 2.0 compatible device is attached, the controller uses a different register set offset than if it were a USB 3.0 compatible device attached. Therefore, how do we find out which register set is USB 2.0 and which is USB 3.0?

By parsing the Supported Protocol Capability registers, outlined in Table 9-16 in Chapter 9, you will be able to find the register sets for the USB 3.0 ports and the USB 2.0 ports, remembering that a USB 3.0 socket can also be a USB 2.0 port. This parsing will return a list and a count. For example, if you have two physical USB 3.0 sockets, the list may return a count of two USB 3.0 register sets and a count of two USB 2.0 register sets. Therefore, the first two register sets may control the USB 3.0 devices, while the second two sets would control the USB 2.0 devices. The Number of Ports field in the HCSPARAMS1 register will be four.

However, a controller may have two physical USB 3.0 sockets and a single physical USB 2.0 socket. Therefore, the list returned will have two USB 3.0 register sets and three USB 2.0 register sets, with the Number of Ports field equal to five. The first two register sets may control the USB 3.0 devices and the last three register sets controlling the USB 2.0 devices.

Here's an example. One of the controllers I have has two physical USB 3.0 sockets (ports). The Number of Ports field in the HCSPARAMS1 register is set to four. Parsing the Supported Protocol Capabilities, it shows that the first two Port Register Sets are used for the USB 3.0 devices while the second two Port Register Sets are used for the USB 2.0 devices. Therefore, if I plug a USB 3.0 device into the first socket, the first Port Register Set will be used. If I plug a USB 2.0 device in to the first socket, the third Port Register Set will be used.

With this in mind, your driver will need to find the starting Port Register Set starting number for each protocol. Then depending on the physical port number used, you will know which register set to use. Once this is determined, you can find the correct Port Register Set using the following equation.

```
PortRegisterSet = ProtocolStart + PortNum;
```

The following code snippet shows how to parse the Supported Protocol Capability section of the controller's memory registers and returns the offset where it starts, and a count of sets it will use. See Listing 9-2 in Chapter 9 for code on how to pair a USB3 and USB2 register port.

Listing 14-1: Supported Protocol Capability

```
/* Returns offset and count of port register sets for a given version,
* On Entry:
*  list: pointer to the start or current position of the Capability list
*  version: 2 or 3.  Version of register set to find.
* On Return:
*  writes count of register sets found in *count.
*  if *count > 0, *offset is written with zero based offset
*  writes value of item->Protocol_defined in *flags
*  returns offset of next item in list or NULL if no more
* The following code assumes:
* - Little-Endian storage
* - there is a properly formatted list at this pointer
*/
void *xhci_get_proto_offset(void *list, const int version, int *offset,
                            int *count, bit16u *flags) {
  void *next;
```

```
 struct xHC_xECP_PROTO *item = (struct xHC_xECP_PROTO *) list;
 *count = 0;  // mark that there isn't any to begin with

 do {
  // calculate next item position
  next = (item->next) ? ((bit8u *) item + (item->next * 4)) : NULL;

  // is this a protocol item, if so is it the version we want?
  if ((item->id == xHC_xECP_ID_PROTO) && (item->major == version)) {
   *offset = item->offset - 1;  // make it zero based
   *count = item->count;
   *flags = item->flags & 0x0FFF;
   return next;
  }
  // point to next item
  item = (struct xHC_xECP_PROTO *) next;
 } while (item);

 // return no more
 return NULL;
}
```

The struct definition:

```
struct xHC_xECP_PROTO {
 bit8u  id;
 bit8u  next;
 bit8u  minor;
 bit8u  major;
 bit32u name;
 bit8u  offset;
 bit8u  count;
 bit16u flags;
};
```

Now all you have to do is call the above function to retrieve the offsets and lengths of each run of Port Register Sets.

Listing 14-2: Offsets and Lengths

```
 void *next = ADDRESS_TO_XHCI_EXTENDED_CAPABILITIES_LIST;
 int cnt, offset;
 bit16u flags;
 int ports_usb2 = 0;

 // find the USB 2.0 ports
 while (next) {
  next = xhci_get_proto_offset(next, 2, &offset, &cnt, &flags);
  if (cnt) {
   number_of_USB2_ports += cnt;
   for (i=0; i<cnt; i++) {
    our_saved_port_info_flags [offset + i].offset = ports_usb2++;
    our_saved_port_info_flags[offset + i] |= IS_A_USB2_PORT;
    if (flags & 2)
     our_saved_port_info_flags[offset + i] |= IS_HIGH_SPEED_ONLY;
   }
  }
 }
```

Do the same for the USB 3.0 ports. Simply call the function giving a value of ‘3’ for the second parameter, remembering to change ports_usb2 to ports_usb3 or something along these lines.

Most implementations of the Supported Protocol Capabilities registers will only have one entry for each protocol. However, the specification states that there may be multiple entries for the same protocol. The code example allows for multiple (fragmented) entries. However, since these values are written at manufacture time, why would you need to fragment the list?

The following memory dump shows the controller's Extended Capabilities registers for the controller mentioned earlier.

```
0x0500   01 04 00 00 00 00 00 C0-00 00 00 00 00 00 00 00
0x0510   02 04 00 03 55 53 42 20-01 02 00 00 00 00 00 00
0x0520   02 00 00 02 55 53 42 20-03 02 00 00 00 00 00 00
```

The first byte specifies that the current entry is the USB Legacy Support entry. Therefore, giving the second byte as 4 dwords to the next entry, we move to offset 0x0510. This entry is the first Supported Protocol entry, indicated by the value of 0x02 in the first byte.

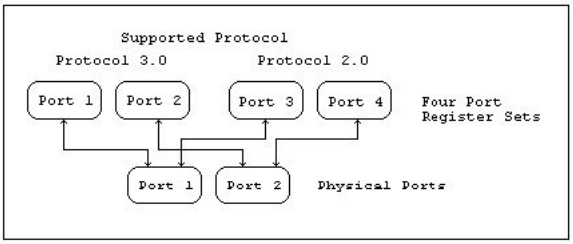
The third and forth bytes are the USB Specification version that will be defined by this entry. They are binary coded decimals with the forth byte being the major version and the third byte being the minor version. Therefore, with this example, we have version 03.00, or USB Specification version 3.0.

To get the offset of where these USB 3.0 ports start, retrieve the value from byte offset 8, which has a value of 0x01. Since valid values are 1 to MaxPorts, this is a 1 based offset. To get the count of port register sets that use this protocol starting at this offset, retrieve the value at byte offset 9. In this example, this would indicate two port register sets for this protocol.

Do the same for the next entry, starting at offset 0x0520 in this example, for the USB 2.0 register sets.

For this controller, the first two Register Sets are used for USB 3.0 devices and the second two Register Sets are used for USB 2.0 devices. The figure below, from Chapter 9, shows this set up.

Figure 14-3: Port Routing and Control



The advantage of the xHCI controller is that when a device is attached to one of the ports, it will use the port register set assigned to that port using that protocol.

For example, if you plug a USB 2.0 device into the first socket, the controller will show the attachment in the first or third register set. All your driver has to do is look up the protocol that each port register set is using so that you know what format of register set it is. Since we parsed and saved that information, we simply have to look at the our_saved_port_info_flags[] field to see if the IS_A_USB2_PORT bit is set. See the *Pairing up the USB2 and USB3 Port Registers* section in Chapter 9 to see which port register set to use.

Please see Chapter 21 on Super-Speed external hubs where both register sets are used at the same time for the same socket.

Driver Initialization

Now that you understand how the controller can use USB 3.0 devices and is backward compatible to use existing USB 2.0 devices, let's start by initializing and starting your driver.

If your platform's BIOS supports the xHCI controller, you will need to indicate that you want it to release the controller so that your driver will have full access. If you do not do this, the BIOS may modify the controller's registers behind your back. If your BIOS does not support the xHCI controller yet, it is still good practice to start with the listing below within your driver.

Listing 14-3: Release BIOS Ownership

```
/* Release BIOS ownership of controller
* On Entry:
*   list: pointer to the start of the Capability list
* On Return:
*   TRUE if ownership released
*
* Set bit 24 to indicate to the BIOS to release ownership
* The BIOS should clear bit 16 indicating it has successfully done so
* Ownership is released when bit 24 is set *and* bit 16 is clear.
* This will wait xHC_xECP_LEGACY_TIMEOUT ms for the BIOS to release
*  ownership. (It is unknown the exact time limit that the BIOS has
*  to release ownership.)
*/
bool xhci_stop_legacy(void *list) {
  struct xHC_xECP_LEGACY *const xECP = (struct xHC_xECP_LEGACY *) list;

  // set bit 24 asking the BIOS to release ownership
  xECP->owner_flags |= xHC_xECP_LEGACY_OS_OWNED;

  // Timeout if bit 24 is not set and bit 16 is not clear
  //  after xHC_xECP_LEGACY_TIMEOUT milliseconds
  int timeout = xHC_xECP_LEGACY_TIMEOUT;
  while (timeout--) {
    if ((xECP->owner_flags & xHC_xECP_LEGACY_OWNED_MASK)
                                  == xHC_xECP_LEGACY_OS_OWNED)
      return TRUE;
    mdelay(1);
  }
  return FALSE;
}
```

The struct definition:

```
#define xHC_xECP_LEGACY_TIMEOUT     10    // in milliseconds
#define xHC_xECP_LEGACY_BIOS_OWNED  (1<<16)
#define xHC_xECP_LEGACY_OS_OWNED    (1<<24)
#define xHC_xECP_LEGACY_OWNED_MASK   \
              (xHC_xECP_LEGACY_BIOS_OWNED | xHC_xECP_LEGACY_OS_OWNED)
struct xHC_xECP_LEGACY {
  bit32u volatile owner_flags;
  bit32u volatile cntrl_status;
};
```

Please note the volatile keyword in the structure definition above. If you omit this keyword, your compiler may optimize the code and never actually check the values to see if they change.

We also need to set up the Command, Status, and Configure Register, retrieve the page size from the Page Size Register, and set the Device Notification Control Register. This is done in Chapter 9, *Detecting and Resetting the Host Controller*.

Writing a non-zero value to the MaxSlotsEn field in the Configure Register indicates to the controller your driver is loaded.

Sending Commands to the Controller

The controller uses a command ring to allow your driver to send it commands. Your driver inserts a command TRB at the next available position and then rings the Command Ring Doorbell. The controller processes the command and then returns status with a TRB on the Event Ring. For more information on rings, please see Chapter 10.

To setup a Command Ring, you need to allocate a block of memory to hold the ring. This memory must be aligned on a 64-byte boundary and must not cross a 64k boundary. Since a command ring uses 16-byte TRB's, you need to allocate 16 bytes times the count of TRB's you wish to have in the ring. A Command Ring of 256 TRB's should be sufficient for the purpose of this book, therefore you need to allocate 4096 bytes.

To make sure that we do not cross a 64k boundary, we need to allocate 64k plus the size of our ring, in this case 68k, then use the following code to get a correctly aligned pointer to the ring.

```
void *mem_block = calloc(4096 + (65536 - 1));
void *aligned = (void *) (((bit32u) mem_block + 65535) & ~65535);
```

The pointer returned will now point to a memory block of at least 4096 bytes and be aligned on a 64k boundary.

Remember not to destroy the mem_block pointer member since you will need it to free the memory block. Sending the aligned pointer value to the mfree() function will certainly cause a fault in your driver. Also remember that the addresses must be physical addresses, not linear addresses.

With the number of different rings required, this example will occupy a minimum of 68k for each ring. If your platform is memory restricted, you may want to find another form of ring allocation, such as the aligned memory function of FYSOS defined in previous chapters.

Note in the code snippet on the previous page, I used the *calloc*() function call to clear the memory. The controller defines an empty ring when all fields in the ring are cleared to zero.

As with this type of ring, we also need to initialize the last TRB as a LINK TRB so the controller will return to the start of the ring. This can be done at ring allocation, or when your driver wants to insert a TRB into the last position in the ring.

Once you have a properly aligned pointer to the Command Ring, you will need to write it to the Operational Command Ring Control Register. Along with this pointer, you will need to set the initial Ring Cycle State bit to '1', and the Command Stop and Command Abort bits to zero.

```
xhci_write_oper_reg64(base, xHC_OPS_USBCrcr, (bit64u) aligned | (1<<0));
```

In order to receive status of a command you will also need to set up an Event Ring. Event rings also have an alignment requirement and use an Event Ring Address Table to indicate to the controller where the Event Ring resides in memory. Please see Chapter 9 for more information on this procedure.

The Command Ring uses the Primary Interrupter, the first interrupter. Therefore, before we send a command, we must first set up this interrupter's event ring. The controller uses a Table Segment Array to know where and what size the Event Ring is. The controller's HCSPARAMS2 register contains a field indicating the maximum segments an Event Ring may have. Therefore, if you use this maximum number, you must allocate 16 bytes times max-

allowed segments for the segment table. Then you must allocate max-allowed segments of no more than 64k each, each on a 64-byte boundary not to span a 64k boundary.

Please see Chapters 9 and 10 for more information on setting up a valid Event Ring.

Once you have an empty Command Ring and a correctly formatted Event Ring, you can start to send commands to the controller. For the following example, I will send the No Operation (No OP) command to the controller. This does nothing but advance the command ring to the next position ready for another command.

To send this command, insert the following TRB into the Command Ring and write a zero to the first doorbell.

Figure 14-4: No Op Command TRB

Reserved			
Reserved			
Reserved			
Reserved	Type=8	Reserved	C

The controller will place an Event TRB on the Event Ring and fire the interrupt. Your IRQ routine should find the following Event TRB on the Event Ring.

Figure 14-5: No Op Command Event TRB

Physical Address of No OP Command TRB				
Completion Code	Reserved			
Slot ID	VF ID	Type=33	Reserved	C

See the GDevDesc utility on the CDROM for information and an example on how to insert a command, wait for the interrupt, and extract the Event TRB. For now, let's move on to detecting a connection on the root hub.

Detecting a Connection on the Root Hub

When a device is connected to a Root Hub port, there will be a Port Change Event TRB on the Event Ring. When this happens, your driver needs to go through each used Port Register Set and check for changes. For now, we will assume a new connection was made.

Each device attached uses a Slot, a block of memory, by the controller. This Slot contains the information needed for the controller to communicate with the device. Assuming you have two physical ports on the Root Hub, and you do not attach an external hub, your driver will need at most two slots.

Depending on the MaxSlots field in the controller's HCSPARAMS1 register, you may have up to 255 slots. Then, depending on the Context Size bit in the HCCPARAMS1 register, each slot can occupy up to 2048 bytes. See Chapters 9 & 10 for more information.

Since the controller's Device Context Base Address Array Pointer Register, or DCBAAP Register, points to an array of pointers, you do not have to allocate each slot until that slot is actually needed. Simply place a zero in that particular slot's array entry until you need a slot. When you send the Enable Slot command, the controller will return an index into this array. You then use this index to allocate a memory block for this slot and place the physical address in this array index.

Now that we have detected a new connection, we must send the Enable Slot command. On a successful completion, the controller will return a slot index in an Event TRB. Using this index, we can then allocate and initialize a slot for this device.

Enabling a Slot for the Found Device

To send this command, insert the following TRB into the Command Ring and write a zero to the first doorbell.

Figure 14-6: Enable Slot Command TRB

Reserved				
Reserved				
Interrupter = 0	Reserved			
Reserved		Type=9	Reserved	C

The controller will place an Event TRB on the Event Ring and fire the interrupt. Your IRQ routine should find the following Event TRB on the Event Ring.

Figure 14-7: Slot Enabled Command Event TRB

Physical Address of Enable Slot Command TRB				
Completion Code	Reserved			
Slot ID	Reserved	Type=33	Reserved	C

If the command is successful, Completion Code equals 1, then the Slot ID field will contain a 1 based index into the Device Context Pointer Array of the slot to use. Your driver should then allocate enough memory to hold a Slot Context and any Endpoint Contexts it will use, then place this properly aligned address into this Array's index. If the Slot ID is zero, the command failed.

Remember that the first index in the array is the pointer to the Scratch Pad Buffers. Therefore, the 1 based Slot ID index returned above is actually a zero based index if you include the Scratch Pad Buffer index.

Now that we have a valid Slot ID and a buffer to hold the Slot Context and its Endpoint Contexts, we need to initialize the Slot Context and the first Endpoint Context.

To correctly initialize the Slot Context, we need to have the speed of the device attached and the Root Hub's port number it is attached to. Your driver should have already acquired this information, via the port register set, before this point.

Some of the items within a Slot Context are set differently depending on whether the device is an external hub. However, you cannot determine if it is an external hub until after you have received the device's Device Descriptor. Therefore, set these fields to zero until you determine that it is a hub.

Remember to calculate the size of the Slot Context, which also indicates the offset to the first Endpoint Context, using the Context Size field in the HCCPARAMS1 register.

Chapter 10, Figure 10-4 describes the format of the Slot Context data structure.

To initialize the physical memory the controller uses for the Slot Context, the memory you allocated before, you send the Set Address command. Therefore, you should initially set this memory to zero. Your driver will hold a copy of the Slot Context so that you can keep track of the slot's status. Then you send a copy of this memory to the controller when you send the Set Address command. Once you send this command, the memory that the controller uses for the Slot Context is considered read-only by system software until a Disable Slot command is

received by the controller. That is why you have a copy in your driver. Therefore, initialize this copy as follows.

The Slot Context should first be initialized to zero. The Context Entries field should be initialized to 1 since we will have and use the Default Endpoint to retrieve the device's descriptors. The Speed field should be set to the attached device's speed, the Root Hub Port Number field should be set to the 1-based port register index this device is attached to. The Interrupter Target field should be set to the interrupter number you will be using, zero for the purpose of this example. All other fields should have been set to zero at allocation time or just before initialization time.

The controller will set the Slot State and Device Address field when we send the Set Address command.

> The Reserved Opaque fields in each Context should be treated as opaque. Meaning that the controller may modify them for any reason and your driver should write the same value read when modifying the slot context.

Now that we have a valid Slot Context, we need to initialize the first Endpoint Context entry in this slot since all devices will have a Default Control Endpoint.

Chapter 10, Figure 10-6 describes the format of the Endpoint Context data structure.

> For the purpose of this example, we will only use Transfer Rings and not use any Streams.

The Endpoint Context should first be initialized to zero, and then the initial values for the fields are as follows. If a field is not defined below, it is assumed to be zero.

The EP State field should be set to zero to ensure the slot is in the Disabled state, the LSA field should be set to 1 to disable Streams, and the Interval field to be set to zero. The CErr field set to 3 and the EP Type field set to 4. The Max Packet Size field should be set to 8, 64, or 512 depending on the speed of the device.

> I have found that for the Set Address command to work, the Max Packet Size field in the Control Endpoint Context must be set to 8 for a low-speed device, 64 for a full- or high-speed device, and 512 for a super-speed device. Once you have set the address and then retrieved the MPS from the device's descriptor, you can then set the Max Packet Size field accordingly, remembering to send the Evaluate Context command afterward, if the new value is different than the 8, 64, or 512 above.

The 64-bit TR Dequeue Pointer field should be set to the physical address for the Transfer Ring for this Endpoint. Therefore, you must allocate and create a valid initial Transfer Ring. See Chapter 10 for more on Transfer Rings. Don't forget to set the DCS, or Dequeue Cycle State field to match your Ring's current state. If the Cycle State bit in your newly created Ring is 0, the Endpoints DCS should be set to 1.

The Average TRB Length field should be set to 8 for all Control Endpoints, while the Max ESIT Payload field should be zero.

Setting the Address of a Device

To validate and start an Endpoint, you must first send the Set Address command to the controller with a pointer to a valid Input Context structure, detailed later in this section. The controller will verify the passed information, then if valid, will copy that information to the Slot's Context memory block pointed to by this Slot's Device Context Base Address Array pointer, updating the appropriate fields.

To send this command, insert the following TRB into the Command Ring and write a zero to the first doorbell.

Figure 14-8: Set Address Command TRB

Physical Address of Input Context Buffer				
Interrupter = 0	Reserved			
Reserved	Type=11	B	Reserved	C

The controller will place an Event TRB on the Event Ring and fire the interrupt. Your IRQ routine should find the following Event TRB on the Event Ring.

Figure 14-9: Set Address Command Event TRB

Physical Address of Set Address Command TRB				
Completion Code	Reserved			
Slot ID	Reserved	Type=33	Reserved	C

If the command is successful the Completion Code will equal 1.

The controller uses an Input Context buffer, see Figure 14-10 below, which is the same size as a slot context entry, 32 or 64 bytes, and is followed by the Slot Context data structure and a specified amount of Endpoint Context data structures.

You create an Input Context Buffer containing the correctly initialized bit fields for the Input Control Context and the already initialized Slot Context and Endpoint Contexts, in this case only one endpoint for now, and send the correctly aligned physical address to the controller using the Set Address command we just discussed.

The figure below shows the format of the Input Context Buffer used.

Figure 14-10: Input Context Buffer

Input Control Context
Slot Context
EP0 (Control) Context
EP1 Out Context
EP1 In Context
...

The Input Control Context data structure is the same size as a slot context and has the format described here.

Figure 14-11: Input Control Context

D31	D30	D29	D28	D27	D26	D25	D24	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D
A31	A30	A29	A28	A27	A26	A25	A24	A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A
Reserved and Zero'd																					

	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	...
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd
	Reserved and Zero'd

The first dword is used to indicate to drop a context while the second dword is used to add a context. Each bit represents an index into the slot's context structure. For example, bit 4 is used to indicate whether to Drop or Add the fifth context, in this case Endpoint 2's Out Endpoint context structure.

If a Drop bit is set, the controller will disable that context. Notice that the first two bits are reserved. You cannot disable the Slot Context or the Control Endpoint's Context. If an Add bit is set, the controller will evaluate that context, then if valid, enable that context.

The controller will read and execute the Drop bit first, then read and execute the Add bit. Therefore, if both bits are set for a specified context, it will first be disabled, then evaluated and if valid, enabled again.

For the purpose of the Set Address Command, create a buffer aligned on a PAGE_SIZE boundary, containing the Input Control Context structure with the A0 and A1 bits set, then the copy of the Slot Context and Control Endpoint structures initialized earlier, and set the Physical Address field in the Set Address Command to this buffer.

Before sending the command, you will need to either set or clear the B field in the Set Address Command TRB. The B field, or Block Set Address Request field, when set, indicates to the controller to not send the Set Address request to the device.

All of the description to this point may be quite confusing the first time or two. Just note that you need to allocate physical memory for a slot and point the controller to this memory using the DCBAAP register. From this point on, you won't write to this memory until you have disabled the slot. To initialize this memory, you use the Set Address or Evaluate Context commands, which will copy a temporary slot context that you have initialized already, to this allocated memory.

You must send the Set Address command to enable the slot and before you can use the Control Endpoint to request the first 8 bytes of the device's Device Descriptor. However, you will need to set the Block bit to 1 so that for older devices that must see the request for the first eight bytes of the device descriptor before it sees the Set Address request, these devices still work correctly. After you request the first eight bytes of the device descriptor, you can simply send the Set Address command again, with this bit zero. The controller, seeing that the Slot is enabled and already has an address, will simply send the Set Address Request to the device.

The controller will block any standard Set Address Request sent to the device via the USB 2 protocol. You must use the Set Address command to set the address.

After a successful initial Set Address Command with the Block bit set, the controller will set the Slot State field to Default (1), and the Device Address field to a new address. The Slot is now ready to be used. After the second Set Address Command, with the Block bit cleared, the controller will set the Slot State field to Addressed (2).

The controller may or may not set the Device Address field to a unique value throughout all used Slots. However, the value it does use and the combination of the Root Hub Port Number field will be unique to any slot. For example, if there are two Root Hub ports and each has a device attached, the Device Address field may contain the same value in each Slot, though the Root Hub Port number is different. If an External Hub is attached, the Root Hub Port Number will be the same for the Hub and any devices attached to that Hub, so the Device Address for these slots will all be unique.

Now that we have enabled the Slot with a valid Slot Context and Control Endpoint Context using the Set Address Command, we can now get the first eight bytes of the device's Device Descriptor, reset the device, send the Set Address Request, using the Set Address command again, and then continue to request the remaining device's descriptors.

Again, for the first time, this may be quite confusing. However, if you have a look at the GDevDesc utility on the CDROM, it shows all the steps to get to this point in your driver. The best way to understand this might be to print the source to the GDevDesc utility for the xHCI controller, then follow along the source listing while you read this chapter.

Getting the Status of a Transfer

The xHCI controller allows you to place an Event Data TRB after a Data Stage or Status Stage transfer descriptor. When the controller finds an Event Data TRB, it will put the amount of bytes not transferred into the Status field of the Event TRB.

For example, if the transfer was an OUT transfer and you requested to write 64 bytes but only 24 bytes were written, then the Status field will contain a value of 40 bytes not written. A successful OUT transfer will have zero in the Status field of the Event TRB.

If this is an IN transfer, the Status field will hold the amount of bytes not received. For example, if you requested 64 bytes and only 18 bytes were received, the Status field will contain a value of 46.

So, why is this beneficial to us? The Parameter field of the Event TRB will hold the 16-byte aligned 64-bit pointer you placed in the Parameter field of the Event Data TRB, therefore allowing you to pass a pointer to the controller so that you can write status bytes to a specified location for each transfer.

With this in mind, place an Event Data TRB after your last Data Stage TRB, with a 16-byte aligned pointer to any sized memory block, in the Event Data TRB's Parameter field. For this example, we will use a 32-bit dword. Also place another Event Data TRB after the Status Stage TRB, using the same technique.

Remember that the pointer in the Parameter field of the Event Data TRB must be paragraph aligned.

Now, in your IRQ handler, when you receive an Event TRB of type 32, you can now write any data you wish to the memory pointed to by the Parameter field. For this example, we write the Event TRB's Status field. We also set bit 31 in that field as a flag indicating to our caller, that the controller has executed this TRB.

The Event TRB for the Event Data TRB will contain the amount of data not transferred in the low 24 bits of the TRB's Status field. See later in this chapter on how we accomplish this.

Getting the Device Descriptor of the Device

To get the device descriptor from the device, we must send it a request packet called a Setup packet. The setup packet is an 8-byte data packet that consists of the format listed below and is the same for any type of device.

Table 14-2: Device Request Packet

Offset	Field	Size	Value	Description
0	Request type	1	Value	Bit 7 Transfer Direction 0 = Host to device 1 = Device to host Bit 6:5 Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	Value	The Request desired
2	Value	2	Value	Depends on the request
4	Index	2	Index	Depends on the request
6	Length	2	Count	If a data phase, number of bytes to transfer.

This is the data we will send to a device that has a device address value of zero. So how do we send this request to the device and receive the device descriptor data? We have to insert a transfer descriptor into the Control Endpoint's transfer ring.

Please note, for this example, we will request the first 8 bytes, reset the device, set the address again, then request all 18 bytes. You could simply request all 18 bytes once, though for this example, to show how to do it, I will use the technique shown for the UHCI and OHCI controllers.

This is where the xHCI controller shines. No longer do you have to allocate a bunch of memory for each transfer, set up queues, or any of the techniques that the other controllers must do to simply send a control transfer. To send a control transfer, you simply have to insert TRBs into the Control Endpoint's transfer ring. For this example, we insert five TRBs, a Setup, a Data, a Status, and two Event Data TRBs.

Let's first create a Setup TRB. Since the request packet is only 8 bytes, using the Immediate bit in the TRB, we can send the request packet in the TRB itself, rather than setting up a memory offset to hold it.

In the Setup packet, we send eight bytes of information. Using Figure 14-12, shown below, we can see that we set the Request Type field to 0x80. The requested data is coming from the device to the host with a standard request type for a device. The Request is the GET_DESCRIPTOR request, which has a value of 6, and the Value field contains two byte values. The high byte contains a value of 0x01 for DEVICE, while the low byte contains 0. These two bytes are written as a 16-bit word in little-endian format. The Index field contains zero, and the Length field contains 8, the number of bytes we are requesting. See Appendix G for more on the 0x80, DEVICE, and GET_DESCRIPTOR request values.

Figure 14-12: Setup Stage TRB

Value = 0x0100				Request=6	RequestType = 0x80	
Length = 8				Index = 0		
Interrupter = 0	Reserved		TRB Transfer Length = 8			
Resv	TRT=3		Type=2	1	0	Resv	C

The Parameter field of the TRB should now be 0x0008_0000_0100_0680.

Set the Interrupter Target field to 0, the primary interrupter, the TRB Transfer Length to 8, the amount of bytes we are requesting, the TRT field to 3, and the TRB Type to 2, a Setup TRB. Set the IDT bit since we send the request packet in this TRB, and clear the IOC bit.

Now for the Data Stage TRB.

Figure 14-13: Data Stage TRB

Physical Address of data buffer					
Interrupter = 0	TD Size=0	Transfer Length = 8			
Reserved		1	Type=3	0x09	C

Set the Parameter field to the 64-bit physical address of the buffer to hold the first 8 bytes of the device descriptor, set the Interrupter Target field to 0, the TD Size to 0, and the TRB Transfer Length to 8 bytes. Set the DIR field to 1, the TRB Type to 3, the Chain bit to 1, and the ENT bit to 1.

Now for the first Event Data TRB.

Figure 14-14: Event Data TRB #1

Physical Address of data buffer (paragraph aligned)			
Interrupter = 0	Reserved		
Reserved	Type=7	0x10	C

Set the Parameter field to the 16-byte aligned 64-bit physical address of the buffer to hold the status of the transfer and set the Interrupter Target field to 0. Set the TRB Type to 7, the Chain bit to 0, and the IOC bit to 1.

Now insert those three TRBs and then ring the Control Endpoint's Doorbell. On a successful transfer you should receive an Event TRB on the Primary Event Ring with the upper 8 bits of the Status field set to 1 for a successful transfer and the lower 24 bits containing the amount of bytes not transferred, which should be zero for this example.

This is where you would write the status of the transfer to the address pointed to by the Parameter field. For this example, copy the Status field to the memory pointed to by the Parameter field, remembering to set bit 31 to indicate to our insert routine that we have completed this TRB.

Bit 31 is used as a flag by our calling routine to indicate that we have completed this TRB. This bit is used by software and has nothing to do with the controller.

If we had a successful transfer, we must send the Status TRB along with its Event Data TRB.

Figure 14-15: Status Stage TRB

Zero	

Interrupter = 0	Reserved			
Reserved		0	Type=4	0x08	C

Set the Parameter field to 0, the Interrupter Target to 0, the Direction field to 0, and the TRB Type to 4. Clear the IOC bit, set the chain bit, and clear the ENT bit.

Now for the last Event Data TRB.

Figure 14-16: Event Data TRB #2

Physical Address of data buffer (paragraph aligned)				
Interrupter = 0	Reserved			
Reserved		Type=7	0x10	C

Set the Parameter field to the 16-byte aligned 64-bit physical address of the buffer to hold the status of the transfer and set the Interrupter Target field to 0. Set the TRB Type to 7, the Chain bit to 0, and the IOC bit to 1.

Now insert those two TRBs and then ring the Control Endpoint's Doorbell. On a successful transfer you should receive an Event TRB on the Primary Event Ring with the upper 8 bits of the Status field set to 1 for a successful transfer.

Now reset the device again, send the Set Address command discussed earlier, remembering to clear the Block bit, and then request all of the Device Descriptor. You will now know how long this descriptor is since you found it at offset 00h in the first 8 bytes you read earlier. Remember to get the MaxPacketSize value too, setting it accordingly.

Getting the String Descriptor(s) of the Device

As mentioned before, a device may contain three or more string descriptors. A string descriptor is a descriptor of unknown size that contains 16-bit Unicode characters describing the device in human-readable form. I say unknown size, since each string may be of a different size. Then how do you get the size? Just as we did above to get the size of the device descriptor. Get the first 64 bytes. Within the first eight bytes of the string descriptor is the descriptor's length. This tells you how many additional bytes to read.

There are two forms of string descriptors. The first is the language identification descriptor, index zero. It contains a list of 16-bit word values defining what language(s) this device supports. Once you have the list of LANGID's, as they are called, you can request the remaining string descriptors in any one of these languages, by sending the LANGID in the wIndex field of the Get String Descriptor request packet. To get the list of LANGID's supported, you call the Get String Descriptor request with an index of zero.

Table 14-3: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	Langid[0]	2	Value	Supported Language Code 0
4	Langid[1]	2	Value	Supported Language Code 1
6	Langid[n]	2	Value	Supported Language Code n

The table will hold as many language codes as it supports. To find out how many, simply take the length, subtract two, then divide by two. With this in mind, a device can support up to 126 languages. Again, you need to get the first few bytes of the descriptor to find out how many more bytes to read.

Since the Length field in most descriptors is a byte-sized value, and can hold a maximum value of 255, instead of requesting 8 bytes to get the length, then requesting the whole packet, you can request 255 bytes. Simply set up a TD that has enough TRBs to hold 255 bytes. The controller will "skip" all transfer TRBs on the first short packet it detects (See the section below on Short Packet Detection). If this is a Super-Speed device, you will only need the one Data TRB anyway, since it allows a max packet size of 512 bytes.

Once you have the LANGIDs, you can choose a language you would like to support. Then, each time you request a sting descriptor, place this LANGID code from above in the langid field in the request packet. A string will have a value of 1 or higher in the index field of the request packet and have the format listed below.

Table 14-4: String Descriptor

Offset	Field	Size	Value	Description
0	Length	1	Value	Size of descriptor in bytes.
1	Type	1	0x03	String Descriptor
2	String	n	xx	Unicode Encoded String.

The string is in 16-bit Unicode encoding and must not be zero-terminated. You must get its size from the Length field in the descriptor, remembering that the Length field includes the first two

bytes of the descriptor.

Some of these LANGID codes are:
0x0409 - English - United States
0x0C09 - English - Australian
0x0407 - German - Standard
0x1009 - English - Canadian
0x040C - French - Standard
0x080A - Spanish - Mexican

Notice that the English languages have the low byte as 0x09. This isn't a coincidence. The Primary Language is a 10-bit value in the lower ten bits, with English being 00_0000_1001b. The sub-language identifier is in the high six bits of the high byte. For the complete list of LANGID codes, see the file USB_LANGIDs.pdf from USB-IF web page (See Appendix A).

Configuring an Attached Device

Using the same technique to get the Device Descriptor, now you can request the Configuration Descriptor to receive the endpoint descriptors, so that we can configure the device.

To configure the device, we must write a valid endpoint context to each endpoint context entry in the slot. Since we now have the endpoint descriptors, we can create endpoint context entries. To do so, start by clearing out all entries, specifically the 'A' and 'D' array dwords in the input context buffer.

Initialize the Slot Context entry and the Control Endpoint entry with the values you retrieved from the device's descriptors, set the first two bits in the 'A' array dword, and initially set the num_entries field in the Slot Context to 1.

Now it is time to initialize all of the found endpoints. There is an IN and OUT endpoint entry in the Input Context Buffer for every endpoint with a value of one through fifteen. If the device's bulk IN endpoint descriptor, for example, has a value of 82h, you would use the 2nd entry's IN endpoint context entry. See Figure 14-19 on page 14-25 for an example.

Once you have initialized all of the endpoint entries in the slot, be sure to set the corresponding bit in the 'A' array dword and adjust the num_entries field in the slot entry to match that last initialized entry number.

Now you are ready to send the Configure command. To send this command, insert the following TRB into the Command Ring and write a zero to the first doorbell.

Figure 14-17: Configure EP Command TRB

Physical Address of Input Context Buffer					
Interrupter = 0	Reserved				
Slot ID	Reserved	Type=12	0	Reserved	C

The controller will place an Event TRB on the Event Ring and fire the interrupt. Your IRQ routine should find the following Event TRB on the Event Ring.

Figure 14-18: Configure EP Command Event TRB

Physical Address of Configure EP Command TRB				
Completion Code	Reserved			
Slot ID	Reserved	Type=33	Reserved	C

If the command is successful the Completion Code will equal 1. You then can read from the physical memory where the slot is located to get the state of the slot and each endpoint along with the max packet size of each end point.

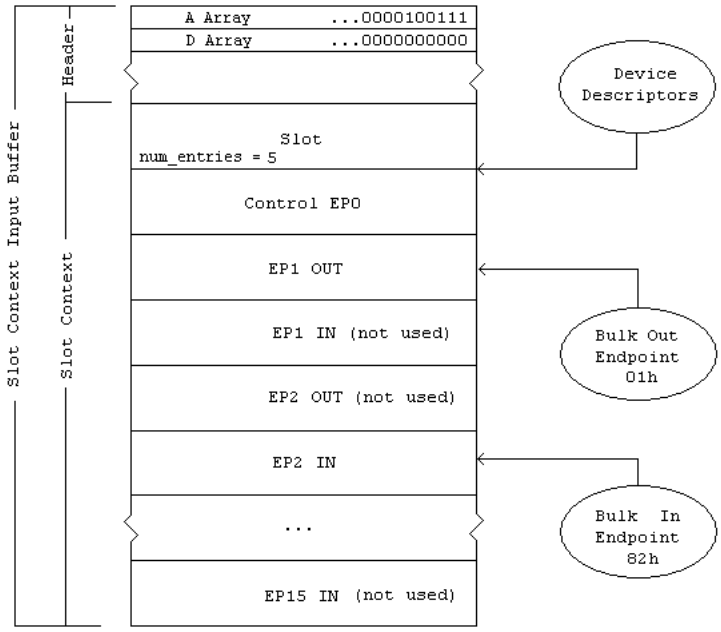
Short Packet Detection

If a short packet is detected during a Transfer Descriptor, the xHCI controller will only execute Link and Event TRBs until it finds the end of the Transfer Descriptor. This means that all IN TRBs will be skipped until it finds the next Transfer Descriptor.

This technique makes it so that your driver doesn't have to calculate what to do on short packets. The controller does it for you. As far as your driver is concerned, each TRB was executed as expected and you are now at the end of the Transfer Descriptor where you check the length of the data that was transferred.

The controller will execute the first Event Data TRB, but skip all remaining Event Data TRB's until the end of the Transfer Descriptor. However, in version 1.10 of the xHCI specification, you now have a bit in HCCPARAMS1 register, the Parse All Event Data bit, indicating if it will skip all but the first, or execute all Event Data TRB's. Please keep this in mind with your short packet detection implementation.

Figure 14-19: Example Input Context Buffer



Part 3

Within the next few chapters, I will explain how to communicate with certain common USB devices, which include mice, keyboards, mass storage devices, external floppy drives, and external hubs.

These chapters will discuss the USB protocol the device expects, and will be controller-type independent.

To use these chapters, you will need to have sufficient knowledge of the controller type you are using, and how to send and receive packets via that controller type.

The information within each chapter is specific to that device and how to send data to and receive data from that specific device. For example, how to receive mouse data packets from a three button mouse, or how to read or write a sector of data to a mass storage device.

The devices described here are low-speed devices, such as a mouse or keyboard; full-speed devices, such as some mass storage devices; high-speed devices, such as newer mass storage devices; and the new super-speed devices.

Each of the chapters may have identical information in them. I wrote each chapter as if it was completely independent of another chapter. Therefore, you don't have to read all chapters in this part if you only needed information about a particular device.

Chapter 15 – Using the USB HID Mouse

In this chapter, I will explain how to initialize and communicate with a three-button USB HID scroll mouse, how it is enumerated and how to capture movement and button presses.

This chapter assumes that you know what a Human Interface Device (HID) is, how to parse the HID report descriptor, and find the format of the returned data received from the mouse. If you are not familiar with this technique, please see Chapter 24. Within Chapter 24, I discuss how to parse the HID report that you will retrieve within this chapter. The HID report tells you what bits/bytes within the device's interrupt report data is the movement and button press information.

In this chapter, for the examples, I use the USB HID mouse described below. I have already parsed the HID report descriptor and know the format of the returned data, further called an interrupt report. I will show you how to retrieve the HID report descriptor, and then direct you to Chapter 24 on how to parse the HID report descriptor. Another brand and/or model USB HID mouse may have a similar interrupt report. Other than this interrupt report, everything else should be very similar no matter the brand or model of your mouse. Therefore you may use any USB HID mouse with the examples within this chapter.

In the examples shown here, I use the Micro Innovations Mobile Optical Scroll Mouse, item number PS250P. It is a small travel optical mouse that works well with my laptop on the arm of a chair. It has a left button, a right button, a scroll wheel, and a third button when you press the scroll wheel. Any USB HID mouse similar to this will work with the following examples.

Since most if not all mice are low-speed devices, you need to remember that you can only receive 8-byte packets, though the actual max packet size amount is stored in the device's descriptor.

The HID Mouse Device Descriptors

During the enumeration process, discussed in Part 2 of this book, you should have been able to retrieve the mouse's first descriptor, the Device Descriptor, and it should be similar to the memory dump listed here.

```
0x00000000  12 01 10 01 00 00 00 08-FC 04 03 00 01 01 00 02
0x00000010  00 01
```

and explained below.

Table 15-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0110**	USB Specification Release Number in Binary-Coded Decimal. 0x0110 = v1.10
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x08**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x04FC**	Vendor ID[1]: Sunplus Technology Co., Ltd.
10	Product ID	2	**0x0003**	Product ID: CM1092 Optical Scroller Mouse
12	Device Rel	2	**0x0101**	Release version 1.01
14	Manufacturer	1	**0x00**	No Manufacturer descriptor. (see note below)
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	No Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump

```
0x00000000  04 03 09 04
```

and explained here.

Table 15-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.

2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

If you will notice above, the Device Descriptor has a zero for the Manufacturer String Index. The Manufacturer index is usually 0x01, and very seldom ever zero. Notice that the Product string index is 0x02. If the manufacturer really didn't want to have a Manufacturer's String Index, wouldn't they have the value of 0x01 instead of 0x02? I decided to try to retrieve the Manufacturers String Index as if it were index 0x01. The device returned a NAK, though it did also return the first 4 bytes of string index 2. The device didn't do anything wrong, the NAK was correct. The 4 bytes returned is considered undefined behavior as it did correctly NAK. The string that is indexed as 1, could be in one of the other descriptors, but doubtful.

This mouse returns a string index for the Product ID with the following values.

```
0x00000000  18 03 57 00 68 00 65 00-65 00 6C 00 20 00 4D 00
0x00000010  6F 00 75 00 73 00 65 00
```

Table 15-3: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x18**	Size of descriptor in bytes
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x16	**0x0057** **0x0068** **0x0065** **0x0065** **0x006C** **0x0020** **0x004D** **0x006F** **0x0075** **0x0073** **0x0065**	W h e e l <space> m o u s e

To this point, you might know what the device might be and have a human readable string to associate it with. However, as noted above, the Class Code in the Device Descriptor was zero, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor:

```
0x00000000  09 02 22 00 01 01 00 A0-32 09 04 00 00 01 03 01
0x00000010  02 00 09 21 00 01 00 01-22 4A 00 07 05 81 03 04
0x00000020  00 0A
```

Notice that the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration. Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 22 00 01 01 00 A0-32
```

Table 15-4: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x0022**	Total length of all data within this configuration descriptor (34 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.
7	Attributes	1	**0xA0**	**D7** Reserved (See Appendix D) D6 Self Powered **D5** Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x32**	Maximum Power Consumption. 0x32 = 100mA

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows those 9 bytes.

```
0x00000000                  09 04 00 00 01 03 01
0x00000010  02 00
```

Table 15-5: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x03**	Class Code: 0x03 = HID
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Boot interface (see note below).
7	Protocol	1	**0x02**	Protocol Code: Mouse.
8	Interface Str	1	**0x00**	No string descriptor.

This descriptor now tells us what kind of device it is. Using the Class Code, we know that it is a Human Interface Device (HID).

The HID differs greatly in the subclass code with other devices. In the early development of the HID specification, it was determined that a single subclass code would be too restrictive, so they included a HID report descriptor that would not have this restriction. Therefore, the subclass code is used to indicate whether this device has a boot protocol interface.

A boot protocol interface was included so that the BIOS would not have to have the extensive code to parse the HID report descriptor to find the format of the interrupt report. Therefore, if the subclass code was equal to 1, then the Protocol code indicated which protocol to use, 1 for a keyboard and 2 for a mouse.

If the subclass code was 1, then the BIOS knew to use default interrupt report format for either the keyboard or the mouse. It didn't have to parse the report descriptor. All HID devices with a subclass code of 1 must return the same first 3 bytes for all mice. Therefore, the BIOS could ignore the HID report descriptor and didn't have to have the extensive code to parse it.

For mice with subclass 1, the report below may be assumed.

Table 15-6: Default Mouse Input Report

Byte	Description
0	Bit 7:3 device specific 2 Button 3 1 Button 2 0 Button 1
1	X Displacement
2	Y Displacement
3 to *n*	Device specific (optional)

The interrupt report returned may be longer than the 3 byte mouse report listed above, but the first 3 bytes must have an identical format for all HID devices that have a subclass code of 1 and a protocol code of 2. A HID aware operating system, if desired, can read in the HID report descriptor, parse it, and use that report format, which may or may not be different than the above, by using the Get_Protocol and Set_Protocol requests.

For example, since my Micro Innovations mouse is a scroll mouse, if we used the default boot protocol above, we may not get the scroll displacement field, therefore not being able to use the scroll part of the scroll mouse.

If a device has a subclass code of zero, then your driver must read in the HID report descriptor and parse it to find the format of the interrupt report. The Protocol field is then ignored.

Next in the Configuration descriptor is a HID descriptor. All HID devices will have a HID descriptor. This descriptor indicates that there may be a Report descriptor and a Physical Report descriptor. The Report descriptor is where you get the data to parse to find out the format of the interrupt report on input from an interrupt endpoint. This descriptor is described later in this chapter. The Physical Report descriptor is used to indicate what physical part of the human body is used for input. The specification uses the example of the right hand thumb to activate button 5. The Micro mouse returns the following HID descriptor.

```
0x00000010      09 21 00 01 00 01-22 4A 00
```

Table 15-7: Interface's HID Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x21**	HID Descriptor
2	Release	2	**0x0100**	Release version = 1.00
4	CountryCode	1	**0x00**	Country Code = none
5	NumDescrptors	1	**0x01**	There is one descriptor.
6	DescType[]	1	**0x22**	Report type
7	DescLength[]	2	**0x004A**	There are 74 bytes in the

				report descriptor.

The first 6 bytes are always included in this descriptor, with offset 5 indicating how many report descriptors there are, with at least 1 set of DescType and DescLength fields starting at offset 6. For example, if the mouse included a Physical Report Descriptor, it would have returned the HID descriptor shown below instead of the above descriptor.

```
0x00000010      0C 21 00 01 00 02-22 4A 00 23 20 00
```

Table 15-8: Interface's HID Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x0C**	Size of descriptor in bytes.
1	Type	1	**0x21**	HID Descriptor
2	Release	2	**0x0100**	Release version = 1.00
4	CountryCode	1	**0x00**	Country Code = none
5	NumDescrptors	1	**0x02**	There are two descriptors.
6	DescType[0]	1	**0x22**	Report type
7	DescLength[0]	2	**0x004A**	There are 74 bytes in the report descriptor.
8	DescType[1]	1	**0x23**	Physical Report type
9	DescLength[1]	2	**0x0020**	There are 32 bytes in the physical report descriptor.

Our HID descriptor, the first one shown, indicated that we have one Report Descriptor, and it is 74 bytes long. I will return to this shortly.

The Configuration descriptor also indicates that there is one Endpoint Descriptor.

```
0x00000010                      07 05 81 03 04
0x00000020  00 0A
```

Table 15-9: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0004**	Max packet size: 4 bytes
6	Interval	1	**0x0A**	Poll every 10mS

Configure the HID Mouse

Now that we have retrieved most of the device descriptors, we now know that it is a HID class device using the HID protocol, having a single interrupt endpoint with an endpoint value of 1.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the Set_Configuration request with a configuration number of 1, retrieved from the ConfigVal field above.

When manufacturers design and manufacture their devices, they try to follow what the major operating systems do to enumerate and configure that device. Then after manufacturing, they test on that major or most popular operating system. If the tests pass on that operating system, they feel that their device passes all the tests it needs to, and sends to mass marketing.

Unfortunately, some devices expect certain requests before that device will function properly. For example, if that major operating system sent a request that the USB specification says was optional, some devices would expect this optional request and would not function correctly without it.

A HID mouse can be configured to not report a specific report until a change has taken place. Depending on the way a manufacturer decides to set this configuration's default, depends on what the device does. For example, if the manufacturer decides to not report any reports until it sees the Set_Idle request, if you never send the Set_Idle request, you will never get any reports.

The Set_Idle request silences a particular report on the interrupt-in endpoint until a new event occurs or the specified amount of time passes. If you send a duration of zero and a report ID of zero, this tells the device to inhibit all reports until a report changes, then reporting this report once, until it changes again.

Since the Set_Idle request is considered an optional request by the USB HID specification, technically we should not have to send this request to get a mouse to report changes. However, as mentioned before, some devices will not function until it receives this optional request.

Either way, we send the Set_Idle request to the mouse.

Table 15-10: Set Idle Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0A**	Set Idle
2	Value	2	**0x0000**	High byte: 0 = indefinitely Low byte: 0 = All reports

4	Index	2	**0x00**	Interface number
6	Length	2	**0x0000**	Nothing sent

The HID Report

Here is where we need to make a decision. If the Interface subclass code was set to 1, indicating a Boot Interface, we can simply use the default Mouse report format specified and continue on to receive mouse movement. However, as noted before, we know we have a scroll mouse and the Boot Interface's default report does not have a Scroll Displacement field.

Therefore, if we want to be able to use the scroll function of the mouse, we need to retrieve the Report Descriptor, parse it, and send the Set_Protocol request. Until we send the Set_Protocol request, the mouse will continue to send the default Boot Interface report, as it was designed to do.

First, let's retrieve the Report Descriptor.

Table 15-11: Get Report Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x81**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x2200**	High byte: 0x22 = REPORT Low byte: 0x00 = index 0
4	Index	2	**0x00**	Interface number
6	Length	2	**0x004A** (74d)	Descriptor length retrieve from HID Descriptor above.

This will retrieve the HID Report Descriptor. This descriptor isn't a set format as with other descriptors. This descriptor is a list of tokens that describe the format of the interrupt report you get when you read from the interrupt-in endpoint. My Micro Innovations mouse returns the 74 bytes listed here.

```
0x00000000  05 01 09 02 A1 01 09 01-A1 00 05 09 19 01 29 03
0x00000010  15 00 25 01 75 01 95 03-81 02 75 05 95 01 81 01
0x00000020  05 01 09 30 09 31 09 38-15 81 25 7F 75 08 95 03
0x00000030  81 06 C0 09 3C 05 FF 09-01 15 00 25 01 75 01 95
0x00000040  02 B1 22 75 06 95 01 B1-01 C0
```

This block of 74 bytes, when parsed correctly, will describe the format of the interrupt-in endpoint. See Chapter 24 for information on how to parse this report.

After parsing the report above, the returned report looks something like this.

Table 15-12: Mouse Input Report

Byte	Description
0	Bit 7:3 reserved 2 Button 3 (0 or 1) 1 Button 2 (0 or 1) 0 Button 1 (0 or 1)
1	X Displacement (-127 -> 127)
2	Y Displacement (-127 -> 127)
3	Z Displacement (-127 -> 127)

The first byte has three button fields each either set or clear. Set means that the button is currently pressed. The high 5 bits are reserved. The next two bytes are the X and Y displacement, with a value that is greater than or equal to -127 and less than or equal to 127. This is the difference in movement from the last position. The last byte is the scroll wheel and is used just like the X and Y displacement values.

If we had not parsed the HID Report descriptor, we may not have found the scroll displacement position.

The mouse probably already returned this exact data since the first three bytes are identical to the BIOS Interface default. This is okay for a device as long as the first three bytes match the Boot Protocol. However, don't assume that the device already returns the Z Displacement field, without sending the Set_Protocol request first.

Now that we have the format of the whole interrupt-in packet, we can send the Set_Protocol request to ensure that the device is returning that report.

Table 15-13: Set Protocol Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0B**	Set Protocol
2	Value	2	**0x0001**	0 = Use Boot Protocol **1 = Use Report Protocol**
4	Index	2	**0x00**	Interface number
6	Length	2	**0x0000**	No data sent or received.

Retrieving mouse movement

We now have a configured, ready to use USB HID mouse. Since the Endpoint Descriptor indicated a 10mS interval, you will need to place an Interrupt-in transfer descriptor in your periodic schedule to be executed every 10mS.

If at the time your scheduler executes that transfer descriptor, the report for the mouse has changed, you will receive an interrupt report, in this case four bytes as described before.

Remember to update your transfer descriptor each time. The scheduler will mark it as inactive and will get skipped on the next 10mS interval if you do not.

Note that in the examples in this book, we don't have an interval in our schedule of 10mS. However, we do have one at 8mS. It is okay to request it a bit sooner, but you might miss something if you request it a bit later.

To retrieve the mouse's interrupt report, i.e. the information in Table 15-12, you simply place an IN packet on the USB stack requesting, in this case 4 bytes of data, every 10mS.

Chapter 16 – Using the USB HID Keyboard

In this chapter, I will explain how to initialize and communicate with a standard QWERTY USB HID keyboard, how it is to be enumerated and how to capture key presses.

This chapter assumes that you know what a Human Interface Device (HID) is, how to parse the HID report descriptor, and find the format of the returned data received from the keyboard. If you are not familiar with this technique, please see Chapter 24. Within Chapter 24, I discuss how to parse the HID report that you will retrieve within this chapter. The HID report returns the information you will need to know to use the bytes within the device's interrupt report.

In this chapter, for the examples, I use the USB HID keyboard described below. I have already parsed the HID report descriptor and know the format of the returned data, further called an interrupt report. I will show you how to retrieve the HID report descriptor, and then direct you to Chapter 24 on how to parse the HID report descriptor. Another brand and/or model USB HID keyboard may have a similar interrupt report. Other than this interrupt report, everything else should be very similar no matter the brand or model of your keyboard. Therefore you may use any USB HID keyboard with the examples below.

In the examples below, I use the Dell™ USB Keyboard, model number L100. It is a standard sized QWERTY keyboard with function keys and a keypad to the right. Pretty much a standard, everyday keyboard. It doesn't have any special keys for the Internet or media devices. Any USB HID keyboard similar to this will work with the following examples.

Since most if not all keyboards are low-speed devices, you need to remember that you can only receive 8-byte packets, though the actual max packet size amount is stored in the device's descriptor.

The HID Keyboard Device Descriptors

During the enumeration process, described in Part 2 of this book, you should have been able to retrieve the keyboard's first descriptor, the Device Descriptor, and it should be similar to the memory dump listed here.

```
0x00000000  12 01 10 01 00 00 00 08-3C 41 05 21 50 03 01 02
0x00000010  00 01
```

And explained here.

Table 16-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0110**	USB Specification Release Number in Binary-Coded Decimal. 0x0110 = v1.10
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x08**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x413C**	Vendor ID[1]: Dell Computer Corp.
10	Product ID	2	**0x2105**	Product ID: Model L100 Keyboard.
12	Device Rel	2	**0x0350**	Release version 3.50
14	Manufacturer	1	**0x01**	Index of string descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	No Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump:

```
0x00000000  04 03 09 04
```

and explained here.

Table 16-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.

2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This keyboard returns a string index for the Manufacture ID and Product ID with the following values.

```
0x00000000  0A 03 44 00 65 00 6C 00-6C 00
```

Table 16-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x0A**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x08	**0x0044** **0x0065** **0x006C** **0x006C**	D e l l

and

```
0x00000000  24 03 44 00 65 00 6C 00-6C 00 20 00 55 00 53 00
0x00000010  42 00 20 00 4B 00 65 00-79 00 62 00 6F 00 61 00
0x00000020  72 00 64 00
```

Table 16-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x24**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x22	**0x0044** **0x0065** **0x006C** **0x006C** **0x0020** **0x0055** **0x0053** **0x0042** **0x0020** **0x004B** **0x0065** **0x0079** **0x0062** **0x006F** **0x0061** **0x0072** **0x0064**	D e l l <space> U S B <space> K e y b o a r d

To this point, we should have an idea what the device is and have a human readable string to associate it with. However, as noted above, the Class Code in the Device Descriptor was zero,

so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor for the before mentioned keyboard:

```
0x00000000  09 02 22 00 01 01 04 A0-23 09 04 00 00 01 03 01
0x00000010  01 05 09 21 10 01 00 01-22 41 00 07 05 81 03 08
0x00000020  00 18
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 22 00 01 01 04 A0-23
```

Table 16-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x0022**	Total length of all data within this configuration descriptor (34 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x04**	Index for String Descriptor describing this config.
7	Attributes	1	**0xA0**	**D7** Reserved (See Appendix D) D6 Self Powered **D5** Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x23**	Maximum Power Consumption. 0x23 = 70mA

This particular Configuration Descriptor has a string index to a string describing this configuration. It happens to be the same string returned for the string index number 2 shown in Table 16-4, so I will not list it here.

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows those 9 bytes.

```
0x00000000                  09 04 00 00 01 03 01
0x00000010  01 05
```

Table 16-6: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface	1	**0x00**	Number of Interface.

	Num			
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x03**	Class Code: 0x03 = HID
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Boot interface (see note below).
7	Protocol	1	**0x01**	Protocol Code: Keyboard.
8	Interface Str	1	**0x05**	Index for String Descriptor describing this config.

This Interface Descriptor has a string index describing this interface.

```
0x00000000  1C 03 45 00 50 00 31 00-20 00 49 00 6E 00 74 00
0x00000010  65 00 72 00 72 00 75 00-70 00 74 00
```

Table 16-7: String Descriptor 5

Offset	Field	Size	Value	Description
0	Length	1	**0x1C**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x1A	**0x0045** **0x0050** **0x0031** **0x0020** **0x0049** **0x006E** **0x0074** **0x0065** **0x0072** **0x0072** **0x0075** **0x0070** **0x0074**	E P 1 <space> I n t e r r u p t

The Configuration Descriptor now tells us what kind of device it is. Using the Class Code, we know that it is a Human Interface Device (HID).

The HID differs greatly in the subclass code with other devices. In the early development of the HID specification, it was determined that a single subclass code would be too restrictive, so they included a HID report descriptor that would not have this restriction. Therefore, the subclass code is used to indicate whether this device has a boot interface.

A boot interface was included so that the BIOS would not have to have the extensive code to parse the HID report descriptor to find the format of the interrupt report. Therefore, if the subclass code was equal to 1, then the Protocol code indicated which protocol to use, 1 for a keyboard and 2 for a mouse.

If the subclass code was 1, then the BIOS knew a default interrupt report format for either the keyboard or the mouse. It didn't have to parse the report descriptor. All HID devices with a subclass code of 1, must use a fixed known 8-byte IN report and a single 1 byte OUT report for keyboards. Therefore, the BIOS could ignore the HID report descriptor and didn't have to have the extensive code to parse it.

For keyboards with subclass 1, the reports listed below may be assumed.

Table 16-8: Default Keyboard Input Report

Byte	Description
0	Modifier Keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

Table 16-9: Default Keyboard Output Report (1 byte)

Bit	Description
0	Num Lock
1	Caps Lock
2	Scroll Lock
3	Compose
4	KANA
7:5	Constant

The first report must be returned as is for all HID keyboards with a subclass code of 1. A HID aware operating system, if desired, can read in the HID report descriptor, parse it, and use that report format, which may or may not be different than the above, by using the Get_Protocol and Set_Protocol requests. The second single-byte report above is used to set the state of the Num Lock, Caps Lock, and Scroll Lock states.

If a device has a subclass code of zero, then your driver must read in the HID report descriptor and parse it to find the format of the interrupt report. The Protocol field is then ignored.

Next in the Configuration descriptor is a HID descriptor. All HID devices will have a HID descriptor. This descriptor indicates that there may be a Report descriptor and a Physical Report descriptor. The Report descriptor is where you get the data to parse to find out the format of the interrupt report on input from an interrupt endpoint. This descriptor is described later in this chapter. The Physical Report descriptor is used to indicate what physical part of the human body is used for input. The specifications use the example of the right hand thumb to activate button 5 of a HID mouse. The Dell keyboard I have returns the HID descriptor listed here.

```
0x00000010        09 21 10 01 00 01-22 41 00
```

Table 16-10: Interface's HID Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x21**	HID Descriptor
2	Release	2	**0x0110**	Release version = 1.10
4	CountryCode	1	**0x00**	Country Code = none
5	NumDescrptors	1	**0x01**	There is one descriptor.
6	DescType[]	1	**0x22**	Report type
7	DescLength[]	2	**0x0041** (65d)	There are 65 bytes in the report descriptor.

The first 6 bytes are always included in this descriptor, with the 6th byte (offset 5) indicating how many report descriptors there are. Then there is at least 1 set of DescType and DescLength fields starting at offset 6. For example, if the keyboard I have included a Physical Report Descriptor of 43 bytes, it would have returned the following HID descriptor instead of the above descriptor.

```
0x00000010      0C 21 10 01 00 02-22 41 00 23 2B 00
```

Table 16-11: Interface's HID Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x0C**	Size of descriptor in bytes.
1	Type	1	**0x21**	HID Descriptor
2	Release	2	**0x0110**	Release version = 1.10
4	CountryCode	1	**0x00**	Country Code = none
5	NumDescrptors	1	**0x02**	There are two descriptors.
6	DescType[0]	1	**0x22**	Report type
7	DescLength[0]	2	**0x0041**	There are 65 bytes in the report descriptor.
8	DescType[1]	1	**0x23**	Physical Report type
9	DescLength[1]	2	**0x002B** (43d)	There are 43 bytes in the physical report descriptor.

Our HID descriptor, Table 16-10, indicated that we have one Report Descriptor, 65 bytes long. I will return to this shortly.

The Interface Descriptor also indicates that there is one Endpoint Descriptor.

```
0x00000010                      07 05 81 03 08
0x00000020  00 18
```

Table 16-12: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0008**	Max packet size: 8 bytes
6	Interval	1	**0x18**	Poll every 24mS

Configure the HID Keyboard

Now that we have retrieved most of the device descriptors, we now know that it is a HID class device using the HID protocol, having a single interrupt endpoint with an endpoint value of 1.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the Set_Configuration request with a configuration number of 1, retrieved from the ConfigVal field.

When manufacturers design and manufacture their devices, they try to follow what the major operating systems do to enumerate and configure that device. Then after manufacturing, they test on that major or most popular operating system. If the tests pass on that operating system, they feel that their device passes all the tests it needs to, and sends to mass marketing.

Unfortunately, some devices expect certain requests before that device will function properly. For example, if that major operating system sent a request that the USB specification says was optional, some devices would expect this optional request and would not function correctly without it.

A HID keyboard can be configured to not report a specific report until a change has taken place. Depending on the way a manufacturer decides to set this configuration's default, depends on what the device does. For example, if the manufacturer decides to not report any reports until it sees the Set_Idle request, if you never send the Set_Idle request, you will never get any reports.

The Set_Idle request silences a particular report on the interrupt-in endpoint until a new event occurs or the specified amount of time passes. If you send a duration of zero and a report ID of zero, this tells the request to inhibit all reports until a report changes, then reporting this report once, until it changes again.

Since the Set_Idle request is considered an optional request by the USB HID specification, technically we should not have to send this request to get a keyboard to report changes. However, as mentioned before, some devices will not function until it receives this optional request. Either way, we send the Set_Idle request to the keyboard.

Table 16-13: Set Idle Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0A**	Set Idle
2	Value	2	**0x0000**	High byte: 0 = indefinitely Low byte: 0 = All reports
4	Index	2	**0x00**	Interface number
6	Length	2	**0x0000**	Nothing sent

The HID Report

Here is where we need to make a decision. If the Interface subclass code was set to 1, indicating a Boot Interface, we can simply use the default keyboard report format and continue on to receive key press data. However, what if the keyboard has further functionality?

If we want to be able to use the extended functionality of the keyboard, we need to retrieve the Report Descriptor, parse it, and send the Set_Protocol request. Until we send the Set_Protocol request, the keyboard will continue to send the default Boot Interface report. First, let's retrieve the Report Descriptor by sending the following request.

Table 16-14: Get Report Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x81**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x2200**	High byte: 0x22 = REPORT Low byte: 0x00 = index 0
4	Index	2	**0x00**	Interface number
6	Length	2	**0x0041** (65d)	Descriptor length retrieve from HID Descriptor above.

This will retrieve the HID Report Descriptor. This descriptor isn't a set format as with other descriptors. This descriptor is a list of tokens that describe the format of the interrupt report you get when you read from the interrupt-in endpoint. My Dell Keyboard returns the 65 bytes listed here.

```
0x00000000  05 01 09 06 A1 01 05 07-19 E0 29 E7 15 00 25 01
0x00000010  75 01 95 08 81 02 95 01-75 08 81 01 95 03 75 01
0x00000020  05 08 19 01 29 03 91 02-95 01 75 05 91 01 95 06
0x00000030  75 08 15 00 26 FF 00 05-07 19 00 2A FF 00 81 00
0x00000040  C0
```

This block of 65 bytes, when parsed correctly, will describe the format of the interrupt-in endpoint. See Chapter 24 for information on how to parse this report.

After parsing the report, the interrupt-in format returned looks something like this.

Table 16-15: Default Keyboard Input Report

Byte	Description
0	Modifier Keys

1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

While the interrupt/control-out format looks something like this.

Table 16-16: Default Keyboard Output Report (1 byte)

Bit	Description
0	Num Lock
1	Caps Lock
2	Scroll Lock
7:3	Constant (Reserved)

Now that we have the format of the whole interrupt-in packet, we can send the Set_Protocol request to ensure that the device is returning that report.

Table 16-17: Set Protocol Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0B**	Set Protocol
2	Value	2	**0x0001**	0 = Use Boot Protocol **1 = Use Report Protocol**
4	Index	2	**0x00**	Interface number
6	Length	2	**0x0000**	No data sent or received.

Retrieving Keyboard Key Press Data

We now have a configured, ready to use USB HID keyboard. Since the Endpoint Descriptor indicated a 24mS interval, you will need to place an Interrupt-in transfer descriptor in your periodic schedule to be executed every 24mS.

If at the time your scheduler executes that transfer descriptor, the report for the keyboard has changed, you will receive an interrupt report, in this case eight bytes as described earlier.

You will receive a report when a key is pressed and another when that key is released. For example, the following 24 reports were sent when I type the following key sequence: “ben was here”. Twelve key presses, and twelve releases.

```
00 00 05 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 08 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 11 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 2C 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 1A 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 04 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 16 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 2C 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 0B 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 08 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 15 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 08 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Remember to update your transfer descriptor each time you receive a report. The scheduler will mark it as inactive and will get skipped on the next 24mS interval, unless you update it with a new one.

Note that in the examples in this book, we don’t have an interval in our schedule of 24mS. However, we do have one at 16mS. It is okay to request it a bit sooner, but you might miss something if you request it a bit later. Also, since 24 is a multiple of 8, you could place an IN packet in the schedule every 8mS, ignoring two of the three packets. Besides, two of the three will probably return zero bytes anyway.

To retrieve the keyboards’s interrupt report, i.e. the information in Table 16-15, you simply place an IN packet on the USB stack requesting, in this case 8 bytes of data, every 24mS.

Setting the Keyboards LED's

To send a report to the keyboard, you either send it to the interrupt-out endpoint, if declared in the Configuration Descriptor, or if not, send it to the default control endpoint.

To set the LED's on the keyboard, you simply send a Set_Report request to the endpoint. If you know what the report format is, as discussed earlier, you simply send that report to the keyboard.

For example, at keyboard connection, or boot up, your Operating System's keyboard state may be Num Lock and Scroll Lock on, with the Caps Lock off. Since the keyboard was just connected, or the computer was just booted, you will need to set the HID keyboards' LED's accordingly.

With the format described earlier, we would need to send a single byte report with a value of 0x05. To do so, first send the following request to the control endpoint.

Table 16-18: Set Report Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x09**	Set Report
2	Value	2	**0x0200**	Type = 0x02 = Output ID = 0x00 = not used
4	Index	2	**0x00**	Interface number
6	Length	2	**0x0001**	Sending 1 byte

Then send one byte, a value of 0x05, to the endpoint, then receiving a zero byte status packet. A value of 0x05 will turn on the Num Lock and Scroll Lock LED's, and turn off the Caps Lock LED.

Note that sending this command to turn on and/or off an LED does not change the state of that key. For example, if the Caps Lock is off and you send this report to turn on the Caps Lock LED, the Caps Lock state will still be off. You are only modifying the LED state, not the keyboard's state.

Other Comments

A few other things to think about when writing your USB HID keyboard driver. It is perfectly legal to attach a USB HID keyboard to an already booted computer with an existing PS2 type keyboard. Each keyboard will function independently. However, when the Caps Lock, for example, is changed on one keyboard, you need to update the LED's on both keyboards.

Remember, the state of the keyboard, the shift state, Num Lock state, etc., is maintained by the Operating System, not the keyboard. If you press Caps Lock on one keyboard, the other keyboard should match.

Therefore, when your Operating System sees that the Caps Lock key has been pressed, you should send a command to all keyboard drivers to set or clear the Caps Lock status for each keyboard depending on the current state of the key.

Chapter 17 – Using the USB Mass Storage Device

About 15 years ago, it was standard for a computer to have a floppy drive installed and if you are like me, you had hundreds of 1.44meg floppy disks stored in Rolodex type cases with anything from data to games stored on them. Later, when the 1.44meg limit was not enough, Zip drives became available. These drives could hold more, but you couldn't just take a zip disk to another machine like you could a floppy disk.

Then recently, USB thumb drives became available. Any computer with at least one available USB port and OS support for thumb drives, could read and write these devices. At first, 256meg was a lot of space, much more than the 1.44meg floppies. As these thumb drives became more popular, larger drives became available. It is quite common to have a 16-gigabyte thumb drive. A quick and dirty calculation gives about 10,500 floppies to one 16gig thumb drive.

> As of this writing, a 16-, 32-, or 64-Gig thumb drive is widely available, with 128-Gig drives on the way.

In this chapter, I will explain how to configure and program a USB Mass Storage Device (MSD) in the form of a thumb drive, jump drive, flash drive, and many other names of this type of device including external USB hard drives.

The USB-IF has created a form of protocol to use for the USB MSD drive. It is called the USB Mass Storage Class, Bulk-Only Transport Specification. This specification is from the USB-IF web page (See Appendix A) as usbmassbulk_10.pdf.

From this point on, within this chapter, I refer to these types of devices as simply drives. The drive I have used for the research of this chapter is a Kingston™ 256meg drive, model KUSBDTI256, rev 1.04. All of the example descriptors and information in this chapter are for this unit. However, the information and code within this chapter should work on any similar drive unit.

> Kingston™ states that the KUSBDTI256 discussed above is full-speed only[7]. However, when plugged into a high-speed controller, the EHCI controller reset procedure sets the enable bit. This means that it successfully reset a high-speed device. Throughout my tests, I found that it is a high-speed device and enumerates and sends and receives packets as a high-speed device. It also returns a valid Device Qualifier and Other Speed Config Descriptor even though the USB 2.0 specifications say that it should return a 'request error', if it was a full-speed device.

As mentioned before, you should be familiar with how to retrieve the device's descriptors via the enumeration process and send and receive data packets.

Please note that in this chapter I assume the MSD device is either a full- or high-speed device. If you have a super-speed MSD device, see Chapter 22.

The MSD Drive's Device Descriptors

During the enumeration process you should have been able to retrieve the drive's first descriptor, the Device Descriptor, and it should be similar to the following memory dump.

```
0x00000000  12 01 00 02 00 00 00 40-30 09 32 65 00 01 00 02
0x00000010  03 01
```

and explained here.

Table 17-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0200**	USB Specification Release Number in Binary-Coded Decimal. 0x0200 = v2.00
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x40**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x0930**	Vendor ID[1]: Toshiba Corp.
10	Product ID	2	**0x6532**	Product ID: 256M Stick
12	Device Rel	2	**0x0100**	Release version 1.00
14	Manufacturer	1	**0x00**	No Manufacturer descriptor. (see note below)
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x03**	Index of Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the memory dump listed below.

```
0x00000000  04 03 09 04
```

and explained here.

Table 17-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First	2	**0x0409**	0000_01|00_0000_1001b

	Language ID word			Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

If you will notice before, the Device Descriptor has a zero for the Manufacturer String Index. The Manufacturer index is usually 0x01, and very seldom every zero. Notice that the other two string index's are 0x02 and 0x03 respectively. If the manufacturer really didn't want to have a Manufacturers String Index, wouldn't they have values 0x01 and 0x02 instead? I decided to try to retrieve the Manufacturers String Index as if it were index 0x01. What do you know, it returned a string descriptor. I have included it below as if the Device Descriptor indicated it as index 0x01.

This drive returns a string index for the Manufacturer ID, listed below, then the Product ID, listed next, and Serial Number, listed last.

```
0x00000000  12 03 4B 00 69 00 6E 00-67 00 73 00 74 00 6F 00
0x00000010  6E 00
```

Table 17-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x10	**0x004B** **0x0069** **0x006E** **0x0067** **0x0073** **0x0074** **0x006F** **0x006E**	K i n g s t o n

Product ID:

```
0x00000000  22 03 44 00 61 00 74 00-61 00 54 00 72 00 61 00
0x00000010  76 00 65 00 6C 00 65 00-72 00 20 00 32 00 2E 00
0x00000020  30 00
```

Table 17-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x22**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0044** **0x0061** **0x0074** **0x0061** **0x0054** **0x0072**	D a t a T r

			0x0061	a
			0x0076	v
			0x0065	e
			0x006C	l
			0x0065	e
			0x0072	r
			0x0020	<space>
			0x0032	2
			0x002E	.
			0x0030	0

Serial Number:

```
0x00000000  22 03 30 00 43 00 38 00-30 00 32 00 36 00 35 00
0x00000010  31 00 37 00 31 00 34 00-31 00 30 00 41 00 45 00
0x00000020  30 00
```

Table 17-5: String Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x22**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0030**	0
			0x0043	C
			0x0038	8
			0x0030	0
			0x0032	2
			0x0036	6
			0x0035	5
			0x0031	1
			0x0037	7
			0x0031	1
			0x0034	4
			0x0031	1
			0x0030	0
			0x0041	A
			0x0045	E
			0x0030	0

To this point, we might now know what the device is and have a human readable string to associate it with. However, as noted above, the Class Code in the Device Descriptor above was zero, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a dump of the whole descriptor:

```
0x00000000  09 02 27 00 01 01 00 80-64 09 04 00 00 03 08 06
0x00000010  50 00 07 05 81 02 40 00-00 07 05 02 02 40 00 00
0x00000020  07 05 83 03 40 00 01
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration. Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 27 00 01 01 00 80-64
```

Table 17-6: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x0027**	Total length of all data within this configuration descriptor (39 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.
7	Attributes	1	**0x80**	**D7** Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x64**	Maximum Power Consumption. 0x64 = 200mA

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows with these 9 bytes.

```
0x00000000                  09 04 00 00 03 08 06
0x00000010  50 00
```

Table 17-7: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x03**	Number of Endpoints: 3
5	Class Code	1	**0x08**	Class Code: 0x08 = Mass Storage
6	Sub Class	1	**0x06**	Sub Class Code: 0x06 = SCSI transparent command set[1].
7	Protocol	1	**0x50**	Protocol Code: Bulk-Only[1].
8	Interface Str	1	**0x00**	No string descriptor.

[1]See page 6 and 7 of the USB Mass Storage Class Specification Overview.

This descriptor now tells us what kind of device it is. Using the class code, we know that it is a Mass Storage Device (MSD) and the subclass code indicates that it uses the SCSI transparent command set, while the protocol tells us that it uses the Bulk-Only Command Specification for transferring data. The Protocol field is 0x50, which it is listed as Bulk-Only transport. See the USB Mass Storage Class Specification Overview file usb_msc_overview_1.2.pdf, from the USB-IF web page (See Appendix A) for more information.

This Interface Descriptor also indicates that there are three Endpoint Descriptors that follow.

0x00000010 07 05 81 02 40 00-00

Table 17-8: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

0x00000010 07 05 02 02 40 00 00

Table 17-9: Endpoint Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x02**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **2**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

0x00000020 07 05 83 03 40 00 01

Table 17-10: Endpoint Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x83**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = 3
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control

				01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x01**	Poll every 1mS.

The first endpoint is the IN endpoint and has an endpoint value of 1. This is the endpoint that you read data at a maximum of 64 bytes at a time (MaxPacketSize).

The second endpoint is the OUT endpoint and is identical to the first except that you write data using this endpoint instead of reading data. This endpoint descriptor has an endpoint value of 2.

The USB MSD Bulk-only Transport specification states that it “does not require the use of an Interrupt endpoint”, though we found one in the configuration descriptor. Through my research, I have found that this may simply be an inheritance from the CBI (Control/Bulk/Interrupt) protocol used by some MSD devices. The device includes the hardware for it, though the firmware doesn’t use it. Therefore, if you find an interrupt endpoint on a device that uses the Bulk-only Transport protocol, you can simply ignore it.

We need to retrieve one more device descriptor before we configure the device. This descriptor is the Device Qualifier descriptor. It is used with full- and high-speed capable devices. This descriptor is what would be returned as the Device Descriptor if the device were operating in the other speed. For example, if the device were currently operating in full-speed, this descriptor would be the Device Descriptor for the device when operating in high-speed, and visa-versa.

The reason why this descriptor is missing some of the Device Descriptor fields is in either speed the Vendor ID, Product ID, Device Release code, and string descriptors are the same no matter which speed the device is currently operating in. Therefore, there is no need to return this information again.

If the USB MSD drive you are using is a full-speed only device and has a Release Num field of 0x0200, the USB 2.0 specification states, on page 264, that this device should return a ‘request error’ for the following request. High-speed devices will return correctly.

Table 17-11: Device Qualifier Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x80**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x0600**	High byte is type 0x06, Device Qualifier, low byte

				is index, 0x00.
4	Index	2	**0x0000**	Zero
6	Length	2	**0x000A**	We want 10 bytes returned.

The Kinston I have returns:

```
0x00000010  0A 06 00 02 00 00 00 40-01 00
```

As stated earlier, the KUSBDTI/256 I have returns a valid descriptor here when it should have returned a 'request error', if it truly was a full-speed only device.

Table 17-12: Device Qualifier Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x0A**	Size of this descriptor in bytes.
1	Type	1	**0x06**	Device Qualifier Descriptor
2	Release Num	2	**0x0200**	USB Specification Release Number in Binary-Coded Decimal. 0x0200 = v2.00
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x40**	Max packet size for the control endpoint.
8	Configurations	1	**0x01**	Only one configuration.
9	reserved	1	**0x00**	Reserved for future use.

Starting with USB 3.0, the Device Qualifier and Other Speed Configuration descriptors are no longer supported. See USB 3.0 section 9.4.3. To replace them, the USB 3.0 specification has now added the Binary Device Object Store (BOS) Descriptor, USB 3.0 section 9.6.2.

If you felt inclined to do so, this is where you would request the Other Speed Configuration descriptor. The host is not to request this descriptor unless a valid Device Qualifier descriptor has been received.

At this point is probably where a major operating system pops up a balloon to state that "this device would run faster if plugged into a high speed port". Then it lets you click on the balloon to see a list of high-speed controllers. My question is, by seeing a list of high-speed controllers, how are you going to know which port, whether it is in the front or the back of the computer, is a high-speed port?

Configure the MSD Drive

Now that we have retrieved all of the device descriptors, we now know that it is a MSD using the Bulk-Only protocol, having an IN endpoint with an endpoint value of 1, an OUT endpoint with an endpoint value of 2, and an INTERRUPT endpoint with an endpoint value of 3.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the “Set Configuration” request with a configuration number of 1, retrieved from the ConfigVal field.

USB Mass Storage Device LUN's

When we request data from the drive, we send and receive this data within a Command Block Wrapper (CBW) and a Command Status Wrapper (CSW). The CBW contains a Logical Unit Number field (bCBWLUN). Therefore, we need to see how many logical units this device supports by requesting the maximum Logical Unit Number, or LUN. The following is the format of the Request Packet to send to receive a single byte indicating the maximum LUN. See a little later for more information on LUN's.

Table 17-13: Get Max LUN's Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA1**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0xFE**	Get Max LUN
2	Value	2	**0x0000**	Zero
4	Index	2	**0x00**	Interface number, retrieved from the interface descriptor.
6	Length	2	**0x0001**	We want one byte in return.

This request will return a number from zero to 15. Remember that this is the zero-based maximum Logical Unit Number, not the count of logical units. A device may return a STALL if this request is not supported, and your driver must assume a LUN of zero and a total count of 1.

Before we get too much further, you may be wondering what a Logical Unit Number or a LUN is used for. In the SCSI world, a device may have more than one unit. A physical device, like an external storage device, may have more than one physical storage media device. For example, an external storage device may have one SCSI connection but multiple volumes. Each of these volumes would be considered a Logical Unit. Since there is only one SCSI port to send commands to, you must be able to tell the device which volume to read to or write from. This is where the LUN comes in to use.

If a USB MSD has multiple volumes, it will return a non-zero value for the GetMaxLUN request above. The device is still a single device, and your USB driver should treat it as one device. However, your file system driver should now see multiple volumes on the single device. Therefore, if the above request returns a non-zero value, you would need to mount each volume storing the LUN number for each, with the first being, zero, the second, one, and so on.

Then finally when your file system driver requested a data transfer from one of the volumes on the device, it would request the appropriate volume using the LUN field.

Please don't get volumes mixed up here with file system volumes. When I use the term volume above, I mean a physical separate volume from another physical volume. For example, a single SCSI device may have two different physical drives in it, with the first being LUN 0 and the second being LUN 1.

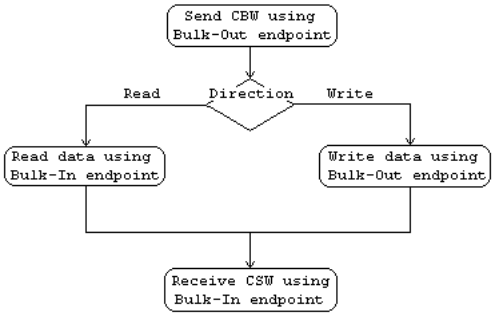
Some devices may return 0xFF as the max LUN. You must treat this as if it returned 0x00.

Introduction to the Bulk-Only Protocol

The Bulk-Only protocol uses the Bulk-Out endpoint to send commands and the corresponding data to the device. This protocol uses the Bulk-In endpoint to read data from the device and read the status of the transfer.

The figure below shows an example.

Figure 17-1: Bulk Only Protocol



The Interface subclass field specified code 0x06, the SCSI transparent command set. However, there isn't any mention of a "SCSI transparent command set" at www.t10.org. Therefore, it is recommended that you use the Inquiry command to retrieve the command set code. In the returned data, bits 4:0 of the first byte indicate a peripheral device type. See further in this chapter for the Inquiry command.

> "*I invented the plain hex that is bInterfaceSubClass = 0x06 first to represent the reality that only op 0x12 'INQUIRY', not the bInterfaceSubClass, tells you what kind of SCSI you get from the device, and then I left the work of naming that hex in English to the committee. I remember my suggestion for an English translation was 'Mac, Linux, Windows SCSI', trying to convey the reality of a de facto industry standard, as opposed to the actually unworkable theory found in dispute between ANSI t10.org, SFF, etc. That English later grew to 'Mac, Linux, Solaris, Windows SCSI' per participation of an engineer at Sun. Later the name quietly and suddenly changed to 'Transparent SCSI' while I was not attending some meeting held only offline.*", Pat LaVarre.[8]

The SCSI Primary Commands (SPC) specification defines various types and the specifications they should comply with. This specification may be acquired from www.t10.org. Within this specification, the peripheral device type is specified on page 254. The code that is usually returned by a USB MSD thumb drive is 0x00 and is described as the SBC-3 command set. The thumb drive I use for the purpose of this chapter, returns 0x00.

> A USB MSD CDROM would return code 0x05, using the MMC-5 command set. See table 133 on page 254 of the SPC-4 specification listed before.

The SBC-3, or SCSI Block Commands version 3 is available at www.t10.org. Not all of the commands in the specification are supported by all devices. Therefore, the Mass Storage Class Compliance Test Specification, from the USB-IF web page (See Appendix A) MSC-compliance-0_9a.pdf, includes the required and optional commands that a USB compliant device should support.

The following is a list containing the commands from this specification. These commands are detailed in the SBC-3 specification mentioned earlier. If the command name has a number in parentheses after it, this means that it uses that many bytes in the CDB, and is an updated version of the command. For example, the Read(10) command uses a 10-byte command block

and a 16-bit transfer length, while Read(12) uses a 12-byte command block and a 32-bit transfer length.

Table 17-14: Partial SBC-3 Command Code List

Command	Description
0x00	Test Unit Ready
0x03	Request Sense
0x12	Inquiry
0x15	Mode Select(6)
0x1A	Mode Sense(6)
0x1B	Start/Stop Unit
0x1E	Prevent/Allow Removal of Media
0x23	Read Format Capacities[1]
0x25	Read Capacity(10)
0x28	Read(10)
0x2A	Write(10)
0x35	Synchronize Cache
0x55	Mode Select(10)
0x5A	Mode Sense(10)
0x88	Read(16)
0x8A	Write(16)
0x9E/10	Read Capacity(16)
0xA8	Read(12)
0xAA	Write(12)

[1]Detailed in the Multimedia Commands (MMC-6) specification, section 6.23.

Being that these commands are based on SCSI command interfaces, which have many more command codes, some devices may and could support more commands than listed above. However, unless you know the specific device attached, don't assume any other commands are supported.

Sending Commands Using the CBW/CSW Interface

As stated before, these commands are sent using a Command Block Wrapper. This is a 31-byte buffer containing transfer information and the actual command being sent.

I don't know why the designers chose 31 bytes for the CBW and 13 bytes for the CSW. For actual data transfer via the disk drive controller, it may have been to save time since it really didn't matter how many bytes you sent or received. As the USB side of it, 32 bytes and 16 bytes would have been more efficient. However, SCSI was developed well before USB was.

To send a command to the MSD drive, after filling this CBW with valid data, you send it as a bulk-out packet, then depending on the command, you send a bulk-out packet or receive a bulk-in packet, and finally receive another bulk-in packet. This bulk-in packet contains the status of the command transfer, called a Command Status Wrapper.

The Command Block Wrapper's format is listed below.

Table 17-15: Command Block Wrapper

Offset	Field	Size	Description
0x00	Signature	4	This dword is a signature, or magic number, to indicate that this is a CBW. It should contain the little-endian value of 0x43425355.
0x04	Tag	4	A signature, or magic number, given by the host software to identify this CBW with the ending CSW. The device will return a CSW's Tag field with this value for this command sequence.
0x08	Transfer Length	4	The number of bytes that will be transferred in this transfer not counting this CBW or the CSW.
0x0C	Flags	1	Bits 7 0 = Out, 1 = In 6:0 reserved
0x0D	LUN	1	Bits 7:4 reserved 3:0 The LUN for this transfer
0x0E	Command Len	1	The length of the command within the next field. Only values 1 through 16 are allowed.
0x0F	Command	16	The command data block (CDB).

The Signature is simply a value of 0x43425355 ('USBC' when stored to memory). This is so the device knows that this 31-byte block is a CBW.

The Tag is a value of your choice. The device will return a valid CSW with its Tag value equal to this value. This is so you can match this CBW with the returning CSW.

The Tag value can be any value you wish. A few suggestions might be to start with a random number, then increment that number for each consecutive transfer; or use a 32-bit time-stamp that is incremented every consecutive transfer. This is simply a number to identify the corresponding CSW that you might receive.

The Transfer Length is the amount of bytes you will transfer in-between the CBW and CSW. If you were going to read a 512-byte sector, you would place 512 in this field.

The Flags field is simply a direction indicator. A value of 0x00 indicates that you are writing data, using the bulk-out endpoint, and a value of 0x80 indicates that you expect data from a read, using the bulk-in endpoint.

The LUN field is the logical unit number you wish to use for transferring data. As stated earlier, if you want to read from the first volume on the device, you would write a value of zero to this field.

All values in the CBW to this point (and CSW below), use little-endian format when storing values larger than 8 bits. All values in the command data block described below use big-endian format.

The last two fields contain the command to issue. The first field being the length of the command byte sequence, up to 16 bytes, while the second field is the command data block. Each command contains one or more bytes, the first being the command, while the remaining, the command's parameters. Remember that any parameter larger than 8 bits is stored in big-endian format.

Once you have written this to the device using the bulk-out endpoint, depending on the command, whether it requires data sent or received, you would send enough bulk-in or bulk-out packets to satisfy the transfer.

Then finally do a read from the bulk-in endpoint for the 13-byte status packet, the CSW.

Table 17-16: Command Status Wrapper

Offset	Field	Size	Description
0x00	Signature	4	This dword is a signature, or magic number, to indicate that this is a CSW. It should contain the little-endian value of 0x53425355.
0x04	Tag	4	The signature, or magic number, given by the host software to identify this CSW with the starting CBW in this command sequence.
0x08	Data Residue	4	This is the value of the difference, if any, of the length of data actually read or written.
0x0C	Status	1	This is a status byte indicating whether the command succeeded or failed.

The Signature is simply a value of 0x53425355 ('USBS' when stored to memory). This is so the host knows that this 13-byte block is a CSW.

The Tag field should be the same value you wrote in the Tag field of the CBW. This is so that you can make sure that this CSW corresponds to the sent CBW.

The Data Residue field indicates a short packet type indication. If the direction of data is to the device, using bulk-out endpoint packets, this value is the difference in amount expected and amount processed. For example, if the amount expected was 512 and the device only processed 448 bytes, the Data Residue field would contain a value of 64.

For more information on this, read section 5.2 of the USB MSD Bulk-Only Transport specifications.

When sending the CBW, if you receive a stall, your driver should process a Reset Recovery process. In this process you would send the Bulk-Only MSD Reset request, then a Clear Feature HALT to each of the two bulk endpoints. In other words, if you receive a stall after sending the CBW, you need to reset the device using the reset request, and clear any possible stall conditions on the two bulk endpoints, which will reset the data PID bits. You do not need to read a CSW for this stalled CBW if you do a reset.

If you have a successful CBW, but then any one of the bulk-in or bulk-out packets are not successful, you still need to receive a CSW. For every CBW, you must either read in a matching CSW, or do a Bulk-Only MSD Reset request.

Command Definition

Here I will explain some of the commands indicated earlier, specifically the ones that you will use to initialize the device and read and write sectors. I will define them in the order that I use them to accomplish this.

Inquiry

We first send the INQUIRY command. This provides two purposes, the first to see if the device is ready and capable of receiving commands and to retrieve some more information about the device. We will request that the device send up to 36 bytes of further information.

To send this command, use the following 31-byte Command Block Wrapper.

Table 17-17: Inquiry Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A80**	Any value you wish
0x08	Transfer Length	4	**0x00000024**	We want 36 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**6**	This command is 6 bytes
0x0F	Command	16	**0x12**	INQUIRY
			0x00	No vital product data
			0x00	Page code = 0
			0x00	Length in big-endian format.
			0x24	
			0x00	Control = 0
			Zeros	Remaining bytes to 0's

As discussed before, set the LUN to the correct unit, in this case zero, the Page Code field should be zero, and the Allocation Length field is set to the maximum number of bytes to return. In this case we want the first 36 bytes.

The device should return the data in the following format.

Table 17-18: Inquiry Command Return Data

Bit Byte	7	6	5	4	3	2	1	0
0	Peripheral Qualifier			Peripheral Device Type				
1	RMB	Reserved						
2	Version							
3	Reserved		Norm ACA	HiSup	Response Data Format			
4	Additional Length							
5	SCCS	ACC	TPGS		3PC	Reserved		Prot
6	resv	ENC Serv	VS	MultiP	Reserved			Addr 16
7	Reserved		Wbus 16	SYNC	Reserved		Cmnd Que	VS
8 ... 15	Vendor Information							
16 ... 31	Product Identification							
32 ... 	Product Revision Level							

35	

The drive I have returns the following information.

```
00 80 00 01 1F 00 00 00-4B 69 6E 67 73 74 6F 6E
44 61 74 61 54 72 61 76-65 6C 65 72 20 32 2E 30
31 2E 30 34
```

The Peripheral Device Type will be either 0x00, Direct Access Block Device, or 0x05, a CD-ROM/DVD device. The Peripheral Qualifier is 0x00 to specify that there is a device connected to this LUN of type Peripheral Device Type.

The RMB field, when set, indicates that the device is capable of removable media. Since thumb drives are all removable, this should always be set.

Which I think is strange. I would think that this bit means that the device itself has removable media. Not the device being removable.

The third byte contains the version number and is zero.

The Response Data Format field should be 0x01 or 0x02.

The SBC-3 specification states that it uses the SPC-4 specification for the INQUIRY return data. The SPC-4 specification states that the Response Data Format should return a value of 0x02 while a value less than 0x02 is obsolete. The SPC-2 version, dated 18 July 2001 also states that anything below 0x02 is obsolete. The thumb drive I have returns a value of 0x01.

The Additional Length field indicates how many more bytes, starting here, is in this data block. This field doesn't indicate how many bytes returned. This field indicates how many more bytes are available for return. The total bytes available for return is this field's value plus 4.

The Vendor Information and Product Identification fields each contain an ASCII text string. These can be whatever the vendor desires, though it must be ASCII text and must be left aligned in the field.

The Product Revision Level contains an ASCII string of the version of the firmware contained in the device. For more information on the remaining fields, see the SPC-4 specifications.

Don't forget to read in the 13-byte Command Status Wrapper.

Table 17-19: Command Status Wrapper

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x53425355**	'USBS'
0x04	Tag	4	**0x04A67A80**	Same TAG from the CBW
0x08	Data Residue	4	**0**	No residue
0x0C	Status	1	**0x00**	Good status return

Once we have received a successful Inquiry transfer, we can assume that the drive is functional and ready for use.

Read Format Capacities

Next, we must find out what size of media the thumb drive is. We do this by sending the Read Format Capacities command. This command will return the current or maximum format capacity of the disk or drive. If the drive is empty, it will return the maximum format capacity. If the drive has a disk inserted, it will return the current format capacity of the disk, plus any other format capacities the disk will allow. These are called Formattable Capacity Descriptors.

Table 17-20: Read Format Capacities Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A81**	Any value you wish
0x08	Transfer Length	4	**0x000000FC**	We want at most 252 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x0A**	This command is 10 bytes
0x0F	Command	16	**0x23**	READ FORMAT CAPACITIES
			0x00	Reserved
			0x00	Reserved
			0x00	Reserved
			0x00	Reserved
			0x00	Reserved
			0x00	Reserved
			0x00	Allocation length is 252 in Big-endian
			0xFC	
			0x00	Control = 0
			Zero	Remaining bytes to 0's

The Allocation Length field is in big-endian format and indicates the maximum length in bytes the command can return. We set this field to a value of 252.

Now do enough bulk-in packets to receive all 252 bytes, stopping after the first short packet detected. In the case of this example, only one bulk-in packet is requested since we only receive 12 bytes of a 64- or 512-byte packet. However, don't assume that you will only receive 12 bytes. Always request 252 bytes to make sure that you always get all of the capacities.

The first 4 bytes will contain a Capacity List Header and is outlined here.

Table 17-21: Capacity List Header

Bit Byte	7	6	5	4	3	2	1	0
0	Reserved							
1	Reserved							
2	Reserved							
3	Capacity List Length							

The Capacity List Length field will contain the length of the remaining data in the buffer if the entire buffer was received. For example, if you only requested the first few bytes, the Capacity List Length field would still contain the length of all data it can return. This length field should always be a multiple of 8 since it returns 8-byte descriptors.

In my opinion, I think that they should have used this byte to return how many 8-byte descriptors follow, not how many total bytes follow. The current form only allows 31 descriptors to follow. However, I haven't seen any drive that can have 32 or more format capacities.

The reason we set the Allocation Length field above to 252 is that the Capacity List Length field returned contains the length of the remaining data. Since this field is a byte, the maximum value it can contain is 255, minus 4 for the first 4 bytes. However, I am guessing the reason why the Allocation Length field in the command block is 16-bits is to be consistent with the SCSI-2 specifications.

After the Capacity List Header above, there are a number of 8-byte Capacity Descriptors that follow. If the drive is empty, there will only be one and it will be the Maximum Capacity Descriptor. This will indicate the maximum capacity disk the drive will allow.

Please note that when I say the drive is empty, I mean that there is a physical USB device attached with the media not inserted. A USB thumb drive will not be empty. If it was empty, the device would not be attached and you would not be sending USB commands to it.

If the drive has a disk inserted, the first descriptor will be the Current Capacity Descriptor. This will indicate the current capacity of the currently inserted disk. There may be one or more Formattable Capacity Descriptors following this descriptor, with each indicating a possible formattable capacity the inserted disk is capable of.

Table 17-22: Capacity Descriptor

Bit Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1				Number of Blocks				
2								
3								(LSB)
4	Reserved							Descriptor Code
5	(MSB)							
6				Block Length				
7								(LSB)

The Descriptor Code will contain one of four values.

Table 17-23: Capacities Descriptor Code

Value	Description
00h	Reserved
01b	Unformatted disk inserted. Is a Maximum Capacity Descriptor for current disk inserted.
10b	Formatted disk inserted. Is a Current Capacity Descriptor.
11b	No disk in drive. Is a Maximum Capacity Descriptor for any disk inserted.

Since a thumb drive is always inserted, and can only be formatted to one size, we should only receive a single 8-byte Current Capacity Descriptor.

The thumb I have drive returns the 12 bytes listed below.

```
00 00 00 08
00 07 A0 00 02 00 02 00
```

This indicates that there are 8 bytes remaining, one more 8-byte descriptor. The number of blocks is 0x0007A000 (499,712d), the Descriptor Code is 10b, and the block size is 0x000200 (512d). These values indicate that it is a 256-megabyte drive.

Disk manufactures thought it would be a great advertising idea to divide the amount of bytes by 1,000,000 instead of 1,048,576. This makes it look like the drive is actually larger than it is. For example, the above, when divided by 2^{20} (1 megabyte), it is a 244-megabyte drive (((499,712 * 512) / 1,048,576) = 244.00). When divided by 10^6 (1 million), it is a 256-megabyte drive (((499,712 * 512) / 1,000,000) = 255.85)

Don't forget to read in the 13-byte Command Status Wrapper afterward.

Table 17-24: Command Status Wrapper

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x53425355**	'USBS'
0x04	Tag	4	**0x04A67A81**	Same TAG from the CBW
0x08	Data Residue	4	**0**	No residue
0x0C	Status	1	**0x01**	Command Failed

If you have made it this far, you have probably noticed that your drive did not return any valid data, returned a stall, and you are probably wondering what you did wrong. In my research, I have found that most drives require a Request Sense command after the first stalled Read Format Capacities command, and then the next Read Format Capacities command is successful. I have not found any information telling why this is, though I have found a lot of information saying that this does happen.

I have found, once, that the Read Format Capacities command stalled again after the above sequence. However, simply doing the above sequence again, it worked correctly. Place the Read Format Capacities command and the following Request Sense command in a loop of at least three iterations to make sure it works. If after the third time it didn't work, your drive may not be functional.

The Read Capacity command explained on page 17-26 has a few more comments about determining the number of blocks on a device.

Request Sense

Since the Read Format Capacities command probably didn't work the first time, it returning a stall, you will need to send a Request Sense command. This command returns information about the last command received.

Table 17-25: Request Sense Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A82**	Any value you wish
0x08	Transfer Length	4	**0x00000012**	We want at most 18 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x06**	This command is 6 bytes
0x0F	Command	16	**0x03**	REQUEST SENSE
			0x00	Reserved/DESC (Bit 0 = 0)
			0x00	Reserved
			0x00	Reserved
			0x12	Allocated Length
			0x00	Control = 0
			Zero	Remaining bytes to 0's

This command should return at least 18 bytes. The total amount it will return is indicated in the Additional Sense Length field below. To retrieve all of the data, you could either read the first 18 bytes, then calculate the total length and send the command again, or you can request 255 bytes and expect a possible short packet return. For most cases, 18 bytes is plenty, since the Sense Key, ASC, and ASCQ fields are within these first 18 bytes.

On a successful return, the following table explains the format returned.

Table 17-26: Request Sense Return Data

Bit Byte	7	6	5	4	3	2	1	0
0	Valid	Error Code (0x70)						
1	Reserved							
2	File Mark	EOM	ILI	Resv	Sense Key			
3	(MSB) Information							
4								
5								
6	(LSB)							
7	Additional Sense Length							
8	Reserved							
9								
10								
11								
12	Additional Sense Code (ASC)							
13	Additional Sense Code Qualifier (ASCQ)							
14	Field Replaceable Unit Code							
15	SKSV							

16	Sense Key Specific
17	

My thumb drive returns the following 18 bytes after the stall of the Read Format Capacity command above.

```
70 00 06 00 00 00 00 0A 00 00 00 00 28 00 00 00 00 00
```

The Valid field is set when the Information field is valid. If the Valid field is not set, the value in the Information field should be ignored.

The Error Code field will always be 0x70 or 0x71. The Additional Sense Length will be at least 10 and at most 244.

The Sense Key, Additional Sense Code, and Additional Sense Code Qualifier are the three fields you will need to use. See Table 17-27 below, for a list of a few common codes. See Appendix E for a full list of these codes.

For example, if the Sense Key is zero, there was no error to return. If the Sense Key is 5, then the last command was an illegal request. An illegal request can mean that a command had an invalid parameter passed within its CBW.

The ASC and ASCQ bytes are for more detailed error reporting. For example, if the Sense Key is 5, as in above, the ASC value is 0x21, and the ASCQ value is 0x00, then the Logical Block Address of the CBW was out of range.

Most of the time, you can just check the ASC and ASCQ values. Check the ASC value, and then if needed, check the ASCQ value.

Table 17-27: Sense Key, ASC, and ASCQ

Sense Key	ASC value	ASCQ value	Description
00	00	00	No Sense
01	17	01	Recovered data with retries
01	18	00	Recovered data with ECC
02	04	01	Logical drive not ready - Becoming ready
		02	Logical drive not ready - Initialization required
		04	Logical Unit not ready - Format in progress
		FF	Logical Unit not ready - Device is busy
	08	00	Logical unit communication error
		01	Logical unit communication time-out
		80	Logical unit communication overrun
	3A	00	Medium not present
	54	00	USB to host system interface failure
	80	00	Insufficient resources
	FF	FF	Unknown error
03	02	00	No seek complete
	03	00	Write fault
	10	00	ID CRC error
	11	00	Unrecovered read error
	14	00	Recorded Entity not found
	30	01	Cannot read medium - unknown format

05	1A	00	Parameter list length error
	20	00	Invalid command operation code
	21	00	Logical block address out of range
	24	00	Invalid field in command packet
	25	00	Logical unit not supported
	26	00	Invalid field in parameter list
	26	01	Parameter not supported
	26	02	Parameter value invalid
06	28	00	Not ready to ready transition - Media changed
07	27	00	Write protected media

You may want to take notice of the last two listed. The Media Changed code may happen for a floppy drive or a CDROM drive, but usually not on a thumb drive.

Don’t forget to read in the 13-byte Command Status Wrapper.

Table 17-28: Command Status Wrapper

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x53425355**	‘USBS’
0x04	Tag	4	**0x04A67A82**	Same TAG from the CBW
0x08	Data Residue	4	**0**	No residue
0x0C	Status	1	**0x00**	Good Status

Read Capacity(10)/ Read Capacity(12)

The Read Capacity command returns the amount of readable blocks a device may have.

Table 17-29: Read Capacity(10) Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A83**	Any value you wish
0x08	Transfer Length	4	**0x00000008**	We want at most 8 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x0A**	This command is 10 bytes
0x0F	Command	16	**0x25**	READ CAPACITY(10)
			0x00	Reserved
			0x00	LBA = 0x0000
			0x00	
			0x00	
			0x00	
			0x00	Reserved
			0x00	Reserved
			0x00	Reserved
			0x00	Control = 0
			Zero	Remaining bytes to 0's

The Logical Block Address field is in big-endian format and should be set to zero.

Now do a bulk-in packet to receive 8 bytes. The first 4 bytes will contain the Logical Block Address of the last readable block on the media. In other words, LBA + 1 = total sectors. The second 4 bytes contain the block size in bytes.

Table 17-30: Read Capacity (32-bit)

Bit Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1				Logical Block Address				
2								
3								(LSB)
4	(MSB)							
5				Block Size in bytes				
6								
7								(LSB)

The thumb drive I have returns 8 bytes.

```
00 07 9F FF 00 00 02 00
```

The LBA of the last block is 0x00079FFF (499,711) and the block size is 0x000200 (512).

If the Logical Block Address value above is 0xFFFFFFFF, this means that the device could not store the actual capacity in the 32-bits allotted above. Therefore, you need to issue the Read Capacity(16) command to retrieve the 64-bit Logical Block Address of the last accessible block.

Table 17-31: Read Capacity(16) Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A83**	Any value you wish
0x08	Transfer Length	4	**0x00000020**	We want at most 32 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x10**	This command is 16 bytes
0x0F	Command	16	**0x9E**	READ CAPACITY(16)
			0x10	Service Action = 0x10
			0x00	LBA = 0x00000000
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	Allocation Length
			0x00	
			0x00	
			0x00	
			0x00	Reserved
			0x00	Control = 0

The Logical Block Address field is in big-endian format and should be set to zero.

Now do a bulk-in packet to receive 32 bytes.

The first 8 bytes will contain the Logical Block Address of the last readable block on the media. In other words, LBA + 1 = total sectors. The second 4 bytes contain the block size in bytes.

Table 17-32: Read Capacity (64-bit)

Bit Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1								
3								
4				Logical Block Address				
5								
6								
7								(LSB)
8	(MSB)							
9								
10				Block Size in bytes				
11								(LSB)
12	Reserved					P_Type		Prot EN

13	P_I_Exponent	LB's/physical block exp
14	TPE | TPRZ | (MSB)	
15	Lowest Aligned Logical Block Address (LSB)	
16	Reserved	
...		
31		

This command allows for the last LBA to be 0xFFFF_FFFF_FFFF_FFFE, an extremely large thumb drive.

You shouldn't need to worry about the fields from offset 12 on, though see the SBC-3 specification, section 5.14.2, for more information.

Don't forget to read in the 13-byte Command Status Wrapper on either command above.

Table 17-33: Command Status Wrapper

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x53425355**	'USBS'
0x04	Tag	4	**0x04A67A83**	Same TAG from the CBW
0x08	Data Residue	4	**0**	No residue
0x0C	Status	1	**0x00**	Good Status

If we used the 16-byte version above, we know that from now on, we can use the 16-byte version of the Read and Write commands. Your driver should set a flag indicating so. However, you should try the 12-byte version first. If you try the 16-byte version and it doesn't work, you will have to try the 12-byte version anyway.

Through out the years, many manufactures disagreed on what the above Logical Block Address field was supposed to return. Some thought it was supposed to return the logical address of the last accessible block, while others thought it was supposed to contain the count of accessible blocks. Some devices return an "off-by-one" amount that is one too many. Some devices will simply crash if you try to read a non-existent sector, such as the last sector that really isn't there. Therefore, it is safe practice to subtract 1 from the LBA address returned. The only thing this will hurt, you will no longer have access to the last sector of the drive, such as GUID partition data. There are numerous other ways to fix this problem. You can research and find all devices that are "off-by-one" and compare their descriptor ID's with a list you have collected. This takes a lot of work, and you won't be able to get them all.

Read(10)/Read(12)/Read(16)

Now that you have retrieved the disk's format capacity, you will want to read in a few sectors to detect the file system used.

The normal file system for a USB MSD Thumb Drive is FAT32. However, this isn't always the case. You may want to read in more than one sector to detect the file system. The examples used here read in 64 sectors.

You should have set a flag in your driver to indicate whether you can use the 10-, 12-, or 16-byte version of the read command as described in the Read Capacity command. The 10-byte version may not allow you to access all of the drive, while the 16-byte version may return a stall or other error if it is not available to the device. Always check which version you can use and set a flag so that each consecutive call uses the same version.

Table 17-34: Read(10) Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A84**	Any value you wish
0x08	Transfer Length	4	**0x00008000**	We want at most 32768 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x0A**	This command is 10 bytes
0x0F	Command	16	**0x28**	READ (10)
			0x00	Reserved
			0x00	LBA = 0x00000000
			0x00	
			0x00	
			0x00	
			0x00	Reserved
			0x00	64 blocks
			0x40	
			0x00	Control = 0
			Zero	Remaining bytes to 0's

The only difference in the Read(10) and Read(12) commands is that the latter uses a 32-bit Transfer Length field and a 12-byte CDB.

Table 17-35: Read(12) Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A84**	Any value you wish
0x08	Transfer Length	4	**0x00008000**	We want at most 32768 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x0C**	This command is 12 bytes
0x0F	Command	16	**0xA8**	READ (12)
			0x00	Reserved
			0x00	

	0x00	LBA = 0x00000000
	0x00	
	0x00	
	0x00	64 blocks
	0x00	
	0x00	
	0x40	
	0x00	Reserved
	0x00	Control = 0
	Zero	Remaining bytes to 0's

The only difference in the Read(12) and Read(16) commands is that the latter uses a 64-bit LBA field and a 16-byte CDB.

Table 17-36: Read(16) Command

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x43425355**	'USBC'
0x04	Tag	4	**0x04A67A84**	Any value you wish
0x08	Transfer Length	4	**0x00008000**	We want at most 32768 bytes returned.
0x0C	Flags	1	**0x80**	Receive an in packet
0x0D	LUN	1	**0x00**	First volume
0x0E	Command Len	1	**0x10**	This command is 16 bytes
0x0F	Command	16	**0x88**	READ (16)
			0x00	Reserved
			0x00	LBA =0x00000000
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	
			0x00	64 Blocks
			0x00	
			0x00	
			0x40	
			0x00	Reserved
			0x00	Control = 0

The Logical Block Address (LBA) is the starting sector to start reading from. This is a zero-based field and is stored as a Big-Endian value. Zero based LBA's start with sector 0, where as the second sector of the volume would be LBA 1. The Transfer Length is the number of contiguous blocks to transfer.

Once you send the above CBW, you would read enough bulk-in endpoint packets to complete the 64-sector read. Don't forget to read in the 13-byte Command Status Wrapper on either command above.

Table 17-37: Command Status Wrapper

Offset	Field	Size	Value	Description
0x00	Signature	4	**0x53425355**	‘USBS’
0x04	Tag	4	**0x04A67A84**	Same TAG from the CBW
0x08	Data Residue	4	**0**	No residue
0x0C	Status	1	**0x00**	Good Status

Write(10)/Write(12) /Write(16)

The three write commands are identical to the three read commands except for the command code and the use of the bulk-out endpoint instead of the bulk-in endpoint, and of course the command codes of 0x2A, 0xAA, and 0x8A respectively.

Reset

To reset the MSD device, you send a control packet to the control endpoint. To do so, you send an interface request of 0xFF.

Table 17-38: Reset Bulk-only Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0xFF**	Reset Bulk-only device
2	Value	2	**0x0000**	Zero
4	Index	2	**0x00**	Interface number, retrieved from the interface descriptor.
6	Length	2	**0x0000**	No return bytes.

This will reset the mass storage part of the MSD device. After the reset, your driver should send the Clear_Feature(ENDPOINT_HALT) request to each of the two bulk endpoints to clear any stall status and to reset the data toggle states.

Other Commands and Comments

With these commands, you should be able to successfully use a USB MSD thumb drive using the Bulk-only transfer protocol. There are a few other commands that you might want to use, such as the Test Unit Ready command. This and the other commands are detailed in the SBC-3 specification from www.t10.org. You must signup at www.t10.org to download this specification. However, you may signup as a guest for free and download most of the documents they offer.

Please note that even though I found a few faults with the Kingston drive used in this chapter, this in no way implies that Kingston doesn't make great products. I used this specific device for the specific purpose to show what you need to expect in your research. Not all devices will work perfectly. I do, however, have numerous Kingston products and they have all exceeded my expectations.

Chapter 18 – Using the USB Floppy Storage Device

About 17 years ago, it was standard for a computer to have a floppy drive installed and if you are like me, you had hundreds of 1.44meg floppy disks stored in Rolodex type cases with anything from data to games stored on them.

I actually still have numerous 5 1/4" floppies and two working drives along with a few 8" floppies and an 8" drive. I don't know if the 8" drive works, since it has a cable interface just different than the standard FDC interface.

A few years after that, the computer manufactures were starting to ask if you would like a floppy drive installed. Most of the time the answer was yes, but they still asked, especially with laptops. The space the floppy drive occupied was valuable real estate.

Then a few years after that, and more recently, you have to specifically ask for a floppy drive or you don't get one. Some computers don't even offer the floppy drive anymore.

So what do you do with all those 1.44meg floppies you have stored away? Well if you are like me, you find a computer that has a floppy drive and a writable CD drive. Then you copy all of those floppies to a CD. If you are using a laptop, which is more likely now-a-days, you may not even have the room for a floppy drive. In comes the USB external floppy drive.

If you are an operating system developer and need to test your boot code on other machines that may not have a floppy drive, as long as the machines BIOS supports USB floppy booting, you can use a USB floppy to boot your floppy disk. The BIOS will emulate the INT 13h calls.

In this chapter, I will explain how to configure and program the USB external floppy drive, further known as floppy drive or just drive, to send and receive data sectors.

The USB-IF has created a form of protocol to use for the USB floppy drive. It is called the USB Floppy Interface (UFI) Command Specification and is part of the USB Mass Storage Class. This specification is available from the USB-IF web page as usbmass-ufi10.pdf (See Appendix A).

The floppy drive I have used for the research of this book is a TEAC™ External Floppy Disk Drive Unit model FD-05PUW. All of the example descriptors and information in this chapter are for this unit. However, the information and code within this chapter should work on any similar floppy drive unit.

As mentioned before, you should be familiar with how to retrieve the device's descriptors via the enumeration process and send and receive data packets.

The Floppy Drive's Device Descriptors

During the enumeration process you should have been able to retrieve the floppy drive's first descriptor, the Device Descriptor, and it should be similar to the following dump.

```
0x00000000  12 01 00 02 00 00 00 08-44 06 00 00 00 00 01 02
0x00000010  00 01
```

Explained here.

Table 18-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0200**	USB Specification Release Number in Binary-Coded Decimal. 0x0200 = v2.00
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x08**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x0644**	Vendor ID[1]: TEAC Corp.
10	Product ID	2	**0x0000**	Product ID: 0x0000 = Floppy
12	Device Rel	2	**0x0000**	Release version 0.00
14	Manufacturer	1	**0x01**	Index of string descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	No Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump:

```
0x00000000  04 03 09 04
```

and explained here.

Table 18-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID	2	**0x0409**	0000_01|00_0000_1001b

	word			Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This drive returns a string index for the Manufacturer ID and Product ID with the following values.

```
0x00000000  12 03 54 00 45 00 41 00-43 00 20 00 20 00 20 00
0x00000010  20 00
```

Table 18-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x10	**0x0054** **0x0045** **0x0041** **0x0043** **0x0020** **0x0020** **0x0020** **0x0020**	T E A C <space> <space> <space> <space>

and

```
0x00000000  22 03 54 00 45 00 41 00-43 00 20 00 46 00 44 00
0x00000010  2D 00 30 00 35 00 50 00-55 00 57 00 20 00 20 00
0x00000020  20 00
```

Table 18-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x22**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0054** **0x0045** **0x0041** **0x0043** **0x0020** **0x0046** **0x0044** **0x002D** **0x0030** **0x0035** **0x0050** **0x0055** **0x0057** **0x0020** **0x0020** **0x0020**	T E A C <space> F D - 0 5 P U W <space> <space> <space>

To this point, we might now know what the device is and have a human readable string to associate it with. However, as noted above, the Class Code in the Device Descriptor was zero, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor:

```
0x00000000  09 02 27 00 01 01 00 80-FA 09 04 00 00 03 08 04
0x00000010  00 00 07 05 81 02 40 00-00 07 05 02 02 40 00 00
0x00000020  07 05 83 03 02 00 20
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 27 00 01 01 00 80-FA
```

Table 18-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x0027**	Total length of all data within this configuration descriptor (39 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.
7	Attributes	1	**0x80**	**D7** Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0xFA**	Maximum Power Consumption. 0xFA = 500mA

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows these 9 bytes.

```
0x00000000                  09 04 00 00 03 08 04
0x00000010  00 00
```

Table 18-6: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.

4	Endpoints	1	**0x03**	Number of Endpoints: 3
5	Class Code	1	**0x08**	Class Code: 0x08 = Mass Storage
6	Sub Class	1	**0x04**	Sub Class Code: 0x04 = UFI Command Spec.
7	Protocol	1	**0x00**	Protocol Code: 00h, 01h, 50h
8	Interface Str	1	**0x00**	No string descriptor.

This descriptor now tells us what kind of device it is. Using the class code, we know that it is a Mass Storage Device (MSD) and the subclass code indicates that it uses the UFI Command Specification for transferring data. The UFI is mostly used for floppy drives. The Protocol field can be either 0x00 or 0x01. If zero, it uses Control/Bulk/Interrupt (CBI) protocol with command completion interrupt, if 0x01, it uses CBI without the command completion interrupt, and if it is 0x50, it is Bulk-Only transport. A protocol of 0x50 here defines that this floppy drive uses the same protocol as the MSD devices in Chapter 17.

This descriptor also indicates that there are three Endpoint Descriptors that follow.

0x00000010 07 05 81 02 40 00-00

Table 18-7: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

0x00000010 07 05 02 02 40 00 00

Table 18-8: Endpoint Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x02**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **2**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk

				11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

0x00000020 07 05 83 03 02 00 20

Table 18-9: Endpoint Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x83**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = 3
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0002**	Max packet size: 2 bytes
6	Interval	1	**0x20**	Poll every 32mS.

The first endpoint is the IN endpoint and has an endpoint value of 1. This is the endpoint that you read data at a maximum of 64 bytes at a time, assuming a 64-byte max packet size.

The second endpoint is the OUT endpoint and is identical to the first except that you write using this endpoint instead of reading data. This endpoint has an endpoint value of 2.

The last endpoint is the INTERRUPT endpoint. If the interface protocol is a value of 0x00, then this endpoint is used to return the status of the command at end of command.

If your driver doesn't poll for this interrupt, the device will clear the interrupt when it successfully receives the next valid command. Therefore, if your driver doesn't support this interrupt, protocol 0x00 and 0x01 are identical.

We need to retrieve one more device descriptor before we configure the device. This descriptor is the Device Qualifier descriptor. It is used with full- and high-speed capable devices. This descriptor is what would be returned as the Device Descriptor if the device were operating in the other speed. For example, if the device were currently operating in full-speed, this descriptor would be the Device Descriptor for the device when operating in high-speed, and visa-versa.

The reason why this descriptor is missing some of the Device Descriptor fields is in either speed the Vendor ID, Product ID, Device Release code, and string descriptors are the same no matter which speed the device is currently operating in. Therefore, there is no need to return this information again.

The USB floppy drive used in this chapter is a full-speed only device and has a Release Num field of 0x0200, therefore the USB 2.0 specifications state, on page 264, that this device should return a 'request error'.

I have included the table below to show what the values might be in case your test floppy drive does return a valid descriptor.

Table 18-10: Device Qualifier Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x0A**	Size of this descriptor in bytes.
1	Type	1	**0x06**	Device Qualifier Descriptor
2	Release Num	2	**0x0210**	USB Specification Release Number in Binary-Coded Decimal. 0x0210 = v2.10
4	Device Class	1	xx	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	xx	See Device Class above.
6	Protocol	1	xx	See Device Class above.
7	MaxPacketSize	1	xx	Max packet size for the control endpoint.
8	Configurations	1	xx	Number of configurations.
9	reserved	1	**0x00**	Reserved for future use.

Starting with USB 3.0, the Device Qualifier and Other Speed Configuration descriptors are no longer supported. USB 3.0 section 9.4.3. To replace them, the USB 3.0 specifications have now added the Binary Device Object Store (BOS) Descriptor, USB 3.0 section 9.6.2.

Configure the Floppy Drive

Now that we have retrieved all of the device descriptors, we know that it is a MSD using the UFI protocol, having an IN endpoint with an endpoint value of 1, an OUT endpoint with an endpoint value of 2, and an interrupt endpoint with an endpoint value of 3.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the “Set Configuration” request with a configuration number of 1, retrieved from the ConfigVal field.

When we request data from the drive, we send and receive this data within command blocks. These command blocks contain a field called the Logical Unit Number field. Therefore, we need to see how many logical units this device supports by requesting the maximum Logical Unit Number, or LUN. The following is the format of the Request Packet to send to get a single byte indicating the maximum LUN.

Table 18-11: Get Max LUN’s Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA1**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0xFE**	Get Max LUNs
2	Value	2	**0x0000**	Zero
4	Index	2	**0x00**	Interface number, retrieved from the interface descriptor.
6	Length	2	**0x0001**	We want one byte in return.

This request will return a number from zero to 15. Remember that this is the zero-based maximum Logical Unit Number, not the count of logical units. A device may return a STALL if this request is not supported, and your driver must assume a LUN of zero.

The drive I have returns a STALL, so my driver assumes a zero.

See Chapter 17, Using the Mass Storage Device, for more information on LUNs and other values it may return.

Introduction to the Floppy Protocol

From this point on we will use the UFI command specification to send to and receive data from the drive. This specification can use three different protocols: Control/Bulk/Interrupt (CBI) with completion interrupt, with out completion interrupt (called CB), and Bulk-Only transport. This chapter will discuss and use the first two protocols, explained in the usb_msc_cbi_1.1.pdf file available from the web (See Appendix A), while the last is discussed in the previous chapter. Also see the file usbmass-ufi10.pdf.

The first two protocols are identical except that the first one has a status interrupt endpoint that you may poll for command completion status. In other words, as soon as you send the command, you may poll the interrupt endpoint until you receive the status word. The second protocol does not have this endpoint.

As the name implies, your driver will use the Control endpoint, endpoint 0, the Bulk endpoints, endpoints 1 and 2, and the Interrupt endpoint, endpoint 3.

The control endpoint is used for non-data and data commands. Non-Data commands are used to send a command and receive a status, but no other data is transferred. Data commands have data transferred via one of the two bulk endpoints between the command and status stages.

The two bulk endpoints transfer data between the host and the device.

If the interrupt protocol is used, the interrupt endpoint is polled for the status of the command.

For example, to send a command to the device, you will send a 12 byte packet to the Control Endpoint (the default endpoint), read data from the IN Endpoint, or write data to the OUT Endpoint, then read two bytes from the Interrupt Endpoint. If this command is a non-data command, the read or write portion is not sent.

The following is a list of commands that the UFI specifications support.

Table 18-12: UFI Command Codes

Command	Description
0x00	Test Unit Ready.
0x01	ReZero. Send the read heads to cylinder zero.
0x03	Request Sense.
0x04	Format Unit.
0x12	Inquiry.
0x1B	Start/Stop Unit.
0x1D	Send Diagnostic / Reset Unit.
0x1E	Prevent/Allow Removal of Media.
0x23	Read Format Capacities.
0x25	Read Capacity.
0x28	Read(10)
0x2A	Write(10)
0x2B	Seek(10)
0x2E	Write and Verify
0x2F	Verify
0x55	Mode Select
0x5A	Mode Sense
0xA8	Read(12)
0xAA	Write(12)

Being that the UFI specification is based on the SCSI-2 and SFF-8070I command interfaces, which have many more command codes, some devices may and could support more commands than listed above. However, unless you know the specific device attached, don't assume any other commands are supported.

These commands are sent using a 12 byte Command Descriptor Block (CDB) with the command in the first byte, the LUN in the high three bits of the second byte, and the remaining block used as the command requires. See Page 12 of the UFI specification.

To indicate whether the command was successful depends on the protocol being used. If you are using CBI, then you can poll the interrupt endpoint for the two-byte return value. If you are using CB, then you must use the Request Sense command to retrieve the error codes. See later in this chapter for more on this command.

All of the commands listed require a valid Logical Unit Number (LUN) field in the CDB except for the Inquiry command (0x12). If an invalid LUN value is sent, the device is to return an error as the response.

The LUN number is a SCSI term that allows the command to select a specific unit (drive) on the SCSI bus. Therefore, you would send a command to the SCSI bus, with a LUN of zero to select the first device. The writers of the UFI specification brought this technique from the SCSI specification to allow its same use with a UFI device.

However, within my research, I have not found a device or any reference to a UFI device that uses more than one LUN. This doesn't mean that there isn't one, this just means that it is rare that a UFI device uses more than one LUN.

Therefore, most UFI devices will respond to the Get Max LUN's request described before with either a stall or return a zero. If this is the case with your device, you can assume there is only one LUN and always place a zero in the LUN field of the CDB. This chapter assumes only one LUN and uses a value of zero.

If you remember, with Table 18-11 we got the count of LUN's this device will use, and it should return a value from zero to 15. However, this takes 4-bits to store a value of 15, where as the UFI's Command Block Descriptor only has 3 bits for the LUN value. Interesting. I guess there probably won't ever be a LUN value higher than 7, but you should probably abort the driver initialization if you ever find one with a LUN value of 8 to 15.

Sending Commands Using the CBI interface

As stated earlier, this chapter assumes the CBI interface. If you were using the CB interface, instead of polling for the 2-byte interrupt packet, you would send a Request Sense command to retrieve the status of the command.

To send a command to the floppy drive, you send a control out packet, then depending on the command, you send a bulk out packet or receive a bulk in packet, and then poll for the 2-byte interrupt status packet.

To send the control out packet, you send a standard request packet, a 12-byte control out packet, and receive a zero byte status packet. You send the SETUP packet as listed below.

Table 18-13: CB/CBI Command Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x00**	Zero
2	Value	2	**0x0000**	Zero
4	Index	2	**0x0000**	Zero
6	Length	2	**12**	We are sending a 12-byte Command Data Block

Since the Request Type is Class and the recipient is the Interface, the device does not need a Request Type value. Therefore place 0x21 in the first byte and the length of the CDB, 12, in the last byte, with the remaining bytes as zero.

Then send the 12-byte Command Data Packet as a Control Out packet, finally receiving the zero byte status packet.

Remember that even though the UFI device may have a max packet size of 64 for bulk transfers, the control endpoint may and usually does still only have a max packet size of 8. This value is retrieved from the Device Descriptor we requested earlier. Therefore, you will have to send the 12 byte Control Out packet above as two packets, the first as 8 bytes, and the second as 4 bytes, assuming 8 byte max packets.

Then depending on the command, whether it requires data sent or received, you would send enough bulk-in or bulk-out packets to satisfy the request.

Then finally poll the Interrupt endpoint for the 2-byte status packet.

This interrupt status packet contains two bytes with the first being the Additional Sense Code value and the second being the Additional Sense Code Qualifier value of the status of the

command. See a later section on the Request Sense command for more information on these two values.

Again for more information on the CBI protocol, see the usb_msc_cbi_1.1.pdf file available from the web (see Appendix A). Please also see the note later in this chapter and Page 14 of the UFI specification, on the Send Diagnostic/Reset command.

UFI Command Definition

Here I will explain some of the UFI commands, specifically the ones that you will use to initialize the device, read and write sectors, and get return status of a command. I will define them in the order that I use them to accomplish this.

Inquiry

We first send the INQUIRY command. This provides two purposes, the first to see if the device is ready and capable of receiving commands and to retrieve some more information about the device. The device will send up to 36 bytes of further information, depending on the Allocation Length field below.

To send this command, use the 12-byte Command Descriptor Block listed below.

Table 18-14: Inquiry Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0x12**)							
1	LUN (**0**)			Reserved (**0**)				
2	Page Code (**0x00**)							
3	Reserved (**0x00**)							
4	Allocation Length (**0x24**)							
5	Reserved (**0x00**)							
6	Reserved (**0x00**)							
7	Reserved (**0x00**)							
8	Reserved (**0x00**)							
9	Reserved (**0x00**)							
10	Reserved (**0x00**)							
11	Reserved (**0x00**)							

As discussed before, set the LUN to zero. The UFI specification only allows one set of information data, so the Page Code field should be zero. The Allocation Length field is set to the maximum number of bytes to return. In this case we want all 36 bytes.

The device will return the data and format listed below.

Table 18-15: Inquiry Command Return Data

Bit Byte	7	6	5	4	3	2	1	0
0	Reserved			Peripheral Device Type				
1	RMB	Reserved						
2	ISO Version		ECMA Version			ANSI Version		
3	Reserved				Response Data Format			
4	Additional Length							
5	Reserved							
6								
7								
8	Vendor Information							
...								
15								
16	Product Identification							
...								
31								
32	Product Revision Level							
...								
35								

The drive I have returns the following information.

```
00 80 00 01 1F 00 00 00-54 45 41 43 20 20 20 20
46 44 2D 30 35 50 55 57-20 20 20 20 20 20 20 20
33 30 30 30
```

The Peripheral Device Type will be either 0x00 (floppy) or 0x1F for no FDD connected to the requested Logical Unit.

? This could be another way to find how many LUN's the device has. Simply send this command, incrementing the LUN field in the CDB until you receive a value of 0x1F in this field. The drive I have returns a stall, though the UFI specification states that other than the Inquiry command, it should not stall, but return Logical Unit Not Supported. It doesn't state, however, what it should do for the Inquiry command.

The RMB field, when set, indicates that the device is capable of removable media. Since floppies are all removable, this should always be set.

The third byte contains SCSI version numbers of three different standards. For the UFI specifications, these must all be zero.

The Response Data Format field should be 0x01. The only format used with the UFI specification.

The Additional Length field should always be 0x1F (31) no matter the Allocation Length in the CDB sent to the device.

The Vendor Information and Product Identification fields each contain an ASCII text string. These can be whatever the vendor desires though it must be ASCII text and must be left aligned in the field.

The Product Revision Level contains an ASCII string of the version of the firmware contained in the device.

? The specification has "n.nn" in Table 10 - Inquiry Data Format, but the drive I have returns "3000" as shown before. It doesn't really specify that you must have it in the form of "n.nn", it just says four ASCII characters.

Once we have received a successful Inquiry transfer, we can assume that the drive is functional and ready for use.

Read Format Capacities

Next, we must find out what size of disk is in the drive, if any, or find out the maximum size of disk that can be formatted in the drive. We do this by sending the Read Format Capacities command. This command will return the current or maximum format capacity of the disk or drive. If the drive is empty, it will return the maximum format capacity. If the drive has a disk inserted, it will return the current format capacity of the disk, plus any other format capacities the disk will allow. These are called Formattable Capacity Descriptors.

The UFI specification uses Big-Endian format when using numbers larger than 8 bits.

Table 18-16: Read Format Capacities Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0x23**)							
1	LUN (**0**)			Reserved (**0**)				
2	Reserved (**0x00**)							
3	Reserved (**0x00**)							
4	Reserved (**0x00**)							
5	Reserved (**0x00**)							
6	Reserved (**0x00**)							
7	Allocation Len (**0xFC00**) (0x00, 0xFC)							
8								
9	Reserved (**0x00**)							
10	Reserved (**0x00**)							
11	Reserved (**0x00**)							

The Allocation Length field is in Big-Endian format and indicates the maximum length in bytes the command can return. We set this field to a value of 252. Whether the drive has a disk inserted or not, the first 4 bytes will contain a Capacity List Header.

Table 18-17: Capacity List Header

Bit Byte	7	6	5	4	3	2	1	0
0	Reserved							
1	Reserved							
2	Reserved							
3	Capacity List Length							

The Capacity List Length field will contain the length of the remaining data in the buffer if the entire buffer was received. For example, if you only requested the first few bytes, the Capacity List Length field would still contain the length of all data it can return.

The reason we set the Allocation Length field above to 252 is that the Capacity List Length field returned contains the length of the remaining data. Since this field is a byte, the maximum value it can contain is 255. However, I am guessing the reason why the Allocation Length field above is 16-bits is to be consistent with the SCSI-2 specifications.

After the Capacity List Header above, there are a number of 8-byte Capacity Descriptors that follow. If the drive is empty, there will only be one and it will be the Maximum Capacity Descriptor. This will indicate the maximum capacity disk the drive will allow.

If the drive has a disk inserted, the first will be the Current Capacity Descriptor. This will indicate the current capacity of the currently inserted disk. There may be one or more Formattable Capacity Descriptors following this descriptor, which each indicate a possible formattable capacity the inserted disk is capable of.

Table 18-18: Capacity Descriptor

Bit Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1	Number of Blocks							
2								
3								(LSB)
4	Reserved							Descriptor Code
5	(MSB)							
6	Block Length							
7								(LSB)

The Descriptor Code will contain one of four values.

Table 18-19: Capacities Descriptor Code

Value	Description
00b	Reserved.
01b	Unformatted disk inserted. Is a Maximum Capacity Descriptor for current disk inserted.
10b	Formatted disk inserted. Is a Current Capacity Descriptor.
11b	No disk in drive. Is a Maximum Capacity Descriptor for any disk inserted.

If the drive I use here is empty, it will return the following data.

```
00 00 00 08
00 00 0B 40 03 00 02 00
```

This indicates that there are 8 bytes remaining, one more 8-byte descriptor. The number of blocks is 0x00000B40 (2880), the Descriptor Code is 11b, and the block size is 0x000200 (512). These values indicate that it can format and/or use a disk of up to 1.44meg in size.

If this drive has a 1.44meg formatted disk inserted, it will return the following data.

```
00 00 00 20
00 00 0B 40 02 00 02 00
00 00 0B 40 00 00 02 00
00 00 04 D0 00 00 04 00
00 00 09 60 00 00 02 00
```

This indicates that there are 32 bytes remaining, four more 8-byte descriptors. The first descriptor is the Current Capacity Descriptor indicating a 1.44meg with 2,880 512-byte blocks. The three remaining descriptors are Formattable Capacity Descriptors. Each descriptor is described below.

Table 18-20: Formattable Capacity Descriptors returned

Descriptor	Size	Tracks	Sectors/	Total Blocks	Block Size

			Track		
Current	1.44m	80	18	**0x0B40** (2880)	**0x200** (512)
Formatted 0	1.44m	80	18	**0x0B40** (2880)	**0x200** (512)
Formatted 1	1.25m	77	8	**0x04D0** (1232)	**0x400** (1024)
Formatted 2	?	?	?	**0x0960** (2400)	**0x200** (512)

Notice that the first Formattable Capacity Descriptor is identical to the Current Capacity Descriptor, excluding the Descriptor Code. The device will include all capacities it is capable of even though the current disk is using one of the capacities.

I put question marks in the last descriptor's fields because the UFI specification has a list of three formats it allows, and that entry is not one of them. Therefore, this drive returns a format that the UFI specification doesn't list as valid. The UFI specification supports the following formats.

Table 18-21: Supported Capacity Descriptors

Medium type	Size	Tracks	Sectors/ Track	Total Blocks	Block Size
DD	720k	80	9	0x05A0 (1440)	0x200 (512)
HD	1.25m	77	8	0x04D0 (1232)	0x400 (1024)
HD	1.44m	80	18	0x0B40 (2880)	0x200 (512)

If you have made it this far, you have probably noticed that your drive did not return any valid data, and returned a stall, and you are probably wondering what you did wrong. In my research, I have found that most drives require a Request Sense command after the first stalled Read Format Capacities command, and then the next Read Format Capacities command is successful. I have not found any information telling why this is, though I have found a lot of information saying that this does happen.

I have found, once, that the Read Format Capacities command stalled again after the above sequence. However, simply doing the above sequence again, it worked correctly. Place the Read Format Capacities command and the Request Sense command, shown below, in a loop of at least three iterations to make sure it works. If after the third time it didn't work, your drive may not be functional.

Request Sense

Since the Read Format Capacities command above probably didn't work the first time, it returning a stall, you will need to send a Request Sense command. This command returns information about the last command received.

Table 18-22: Request Sense Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0x03**)							
1	LUN (**0**)			Reserved (**0**)				
2	Reserved (**0x00**)							
3	Reserved (**0x00**)							
4	Allocation Length (**0x12**)							
5	Reserved (**0x00**)							
6	Reserved (**0x00**)							
7	Reserved (**0x00**)							
8	Reserved (**0x00**)							
9	Reserved (**0x00**)							
10	Reserved (**0x00**)							
11	Reserved (**0x00**)							

The Allocation Length is set to 18, the maximum allowed number of bytes to return.

On a successful return, the following data format is returned.

Table 18-23: Request Sense Return Data

Bit Byte	7	6	5	4	3	2	1	0
0	Valid	Error Code (0x70)						
1	Reserved							
2	Reserved				Sense Key			
3	(MSB) Information							
4								
5								
6								(LSB)
7	Additional Sense Length (0x0A)							
8	Reserved							
9								
10								
11								
12	Additional Sense Code (ASC)							
13	Additional Sense Code Qualifier (ASCQ)							
14	Reserved							
15	Reserved							
16								
17								

My drive returns the following 18 bytes.

70 00 06 00 00 00 00 0A-00 00 00 00 28 00 00 00 00 00

The Valid field is set when the Information field is valid. If the Valid field is not set, the value in the Information field should be ignored.

The Error Code field will always be 0x70. The Additional Sense Length will always be 10.

The Sense Key, Additional Sense Code, and Additional Sense Code Qualifier are the three fields you will need to use. Section 5 of the UFI specification details what these three bytes will return. See Appendix E for a full list of all these codes.

For example, if the Sense Key is zero, there was no error to return. If the Sense Key is 5, then the last command was an illegal request. An illegal request can mean that a command had an invalid parameter passed within its CDB.

The ASC and ASCQ bytes are for more detailed error reporting. For example, if the Sense Key is 5, as in above, the ASC valid is 0x21, and the ASCQ value is 0x00, then the Logical Block Address of the CDB was out of range.

Most of the time, you can just check the ASC and ASCQ values. This is what the Interrupt endpoint returns in the CBI protocol anyway. Check the ASC value, and then if needed, check the ASCQ value.

See Appendix E for a complete list of Sense Key, ASC, and ASCQ codes. Of this list, two of the codes that you will want to take notice of are listed below.

Table 18-24: Sense Key, ASC, ASCQ

06	28	00	Not ready to ready transition - Media changed
07	27	00	Write protected media

The first is just like the Floppy Disk Controller's disk change line bit, while the second is to indicate if the disk is write protected.

Read(10)/Read(12)

Now that you have retrieved the disk's format capacity, you will want to read in a few sectors to detect the file system used. Most floppy disks are formatted with the FAT file system, so the first sector is enough.

Table 18-25: Read(10) Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0x28**)							
1	LUN (**0**)			DPO	FUA	Reserved (**0**)		RelAdr
2	(MSB)							
3	Logical Block Address							
4								
5								(LSB)
6	Reserved (**0x00**)							
7	(MSB)							
8	Transfer Length							(LSB)
9	Reserved (**0x00**)							
10	Reserved (**0x00**)							
11	Reserved (**0x00**)							

The only difference in the Read(10) and Read(12) commands is that the latter uses a 32-bit Transfer Length field. Since a floppy drive most likely will not have more than a 16-bit value of sectors (or blocks), you will probably never need the 12-byte version of this command. However, it is always a good idea to know that it exists and how to use it.

Table 18-26: Read(12) Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0xA8**)							
1	LUN (**0**)			DPO	FUA	Reserved (**0**)		RelAdr
2	(MSB)							
3	Logical Block Address							
4								
5								(LSB)
6	(MSB)							
7	Transfer Length							
8								
9								(LSB)
10	Reserved (**0x00**)							
11	Reserved (**0x00**)							

The DPO, FUA, and RelAdr fields should all be zero.

The Logical Block Address (LBA) is the starting sector to start reading from. This is a zero-based field and is stored as a Big-Endian value. Zero based LBA's start with sector 0, where as the second sector of the volume would be LBA 1. The Transfer Length is the number of contiguous blocks to transfer.

For example, the following 12-byte Read(10) CDB would read one sector starting at LBA 0, the first sector of the disk.

28 00 00 00 00 00 00 00-01 00 00 00

Then, if the block size is 512, you would read 512 bytes using the Bulk In endpoint.

Remember that the LBA and Transfer Length fields are Big-Endian and that you must send the above 12-byte packet as one 8-byte packet and one 4-byte packet, and that the Bulk In endpoint might not be a 512 byte endpoint. You may need to read eight 64-byte packets, as with my drive. Also remember to poll the Interrupt Endpoint for the two-byte status return.

Write(10)/Write(12)

The two write commands are identical to the two read commands except for the command code and the use of the Bulk Out endpoint instead of the Bulk In endpoint.

Whether is it a read or a write, the floppy device will start and stop the motor as needed, and move the heads to the appropriate cylinder as needed. There is no need for your driver to worry about the drive motor or seeking to the cylinder before reading or writing.

Send Diagnostic/Reset

One other command that you will need to be familiar with is the Send Diagnostic/Reset command. In the specification, this is described as the Send Diagnostic command. However, it also states that the UFI device may interpret this command as the CBI Command Block Reset command. Therefore, if you turn to the CBI specification, page 13, it states that this command sequence, when sent with the reserved fields as 0xFF, the device can distinguish the reset from the Send Diagnostic command.

Therefore, to reset the UFI device you send the Send Diagnostic command with trailing 0xFF values as the reserved fields.

Table 18-27: Send Diagnostic/Reset Command

Bit Byte	7	6	5	4	3	2	1	0
0	Command Code (**0x1D**)							
1	LUN (**0**)			FP	Resved	Self Test	DefOfl	UnitOfl
2	Reserved (**0xFF**)							
3	Reserved (**0xFF**)							
4	Reserved (**0xFF**)							
5	Reserved (**0xFF**)							
6	Reserved (**0xFF**)							
7	Reserved (**0xFF**)							
8	Reserved (**0xFF**)							
9	Reserved (**0xFF**)							
10	Reserved (**0xFF**)							
11	Reserved (**0xFF**)							

An example of this 12-byte CDB would be.

```
1D 04 FF FF FF FF FF FF-FF FF FF FF
```

This will reset the mass storage part of the UFI device. After you send the reset, your driver should send the Clear_Feature(ENDPOINT_HALT) request to each of the two bulk endpoints to clear any stall status and to reset the data toggle states.

Other Commands and Comments

With these commands, you should be able to successfully use a USB UFI floppy disk drive using the CB or CBI transfer protocol. There are a few other commands that you might want to use, such as the Format Unit command. These commands are detailed in the USB UFI specification starting with page 15. This specification is available on the web. See Appendix A for more information.

Chapter 19 – Using the USB External Hub

In this chapter, I will explain how to initialize and communicate with a four-port USB full-speed external hub, how it is to be enumerated and how to capture port status changes.

In the examples, I use the Micro Innovations Hide-a-Hub, item number USB104N. It has one upstream port in the form of a USB 2.0 standard-A plug connector, and four downstream ports in the form of USB 2.0 standard-A socket connectors. It is an external non-powered hub. Any USB external hub similar to this will work with the following examples.

External hubs attached to the xHCI controller are discussed in Chapter 21 of this book. However, you may want to be familiar with this chapter since a non super-speed external hub may be attached to a xHCI controller.

In this chapter, I will use this hub to show how to determine the amount of downstream ports, reset each port, and determine if another device is attached to one of these downstream ports.

An Introduction to External Hubs

Let's start with the simple fact that all USB devices answer only to their respective address numbers, ignoring any packet with an address number different than its assigned address. With this in mind, let's think of an external hub as simply tying all of the power wires together, all the data wires together, and the ground wires together. This way, when a packet comes from the upstream port, the hub sends the packet to all of the downstream ports.

In theory, this would work, except for the fact that we would not know when a device was connected or disconnected to one of the downstream ports. Therefore, an external hub must be able to detect a connection and a disconnection, and be able to reset and enable the port on a new connection. It must also report other events, such as an over-current event. An external hub also has other features like port indicators that can signal use and errors with the port.

When a hub is first attached, it responds to address number zero just like any other device. Not until it is in the addressed state can we reset and enable any of the ports on the hub. Therefore, any device already attached to the hub will not respond to the default packet address of zero, until after the hub is in the address state. Then just like a root hub, you reset and enable each port individually so that you don't have more than one device responding to the default address at a time.

When you write your USB driver, it will be much easier if you treat the root hub as an external hub. This way, all you have to keep track of is which hub a device is on, and send a packet to that hub. Then have a single flag in your hub data block that indicates whether the hub is the root hub or an external hub.

The Hub's Device Descriptors

During the enumeration process you should have been able to retrieve the hub's first descriptor, the Device Descriptor, and it should be similar to the following memory dump.

```
0x00000000  12 01 10 01 09 00 00 08-EB 03 01 33 00 03 00 02
0x00000010  00 01
```

and explained here.

Table 19-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0110**	USB Specification Release Number in Binary-Coded Decimal. 0x0110 = v1.10
4	Device Class	1	**0x09**	Class code = 0x09 = Hub
5	Sub Class	1	**0x00**	Not used.
6	Protocol	1	**0x00**	Not used.
7	MaxPacketSize	1	**0x08**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x03EB**	Vendor ID[1]: Atmel Corp.
10	Product ID	2	**0x3301**	Product ID: at43301 4-Port Hub
12	Device Rel	2	**0x0300**	Release version 3.00
14	Manufacturer	1	**0x00**	No Manufacturer descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	No Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump:

```
0x00000000  04 03 09 04
```

and explained here.

Table 19-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

If you will notice, the Device Descriptor has a zero for the Manufacturer String Index. The Manufacturer index is usually 0x01, and very seldom every zero. Notice that the Product string index is 0x02. If the manufacturer really didn't want to have a Manufacturer's String Index, wouldn't they have the value of 0x01 instead of 0x02? I decided to try to retrieve the Manufacturers String Index as if it were index 0x01. The device correctly returned a NAK.

This hub returns a string index for the Product ID with the following values.

```
0x00000000  22 03 53 00 74 00 61 00-6E 00 64 00 61 00 72 00
0x00000010  64 00 20 00 55 00 53 00-42 00 20 00 48 00 75 00
0x00000020  62 00
```

Table 19-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x22**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0053** **0x0074** **0x0061** **0x006E** **0x0064** **0x0061** **0x0072** **0x0064** **0x0020** **0x0055** **0x0053** **0x0042** **0x0020** **0x0048** **0x0075** **0x0062**	S t a n d a r d <space> U S B <space> H u b

To this point, we should now know what the device is and have a human readable string to associate it with. Unlike the HID class devices or MSD devices in the previous chapters, the class code in the Device Descriptor does tell us what class of device this is. However, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor.

```
0x00000000  09 02 19 00 01 01 04 E0-20 09 04 00 00 01 09 00
0x00000010  00 05 07 05 81 03 01 00-FF
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 19 00 01 01 04 E0-20
```

Table 19-4: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x0019**	Total length of all data within this configuration descriptor (25 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x04**	Index for String Descriptor describing this config.
7	Attributes	1	**0xE0**	**D7** Reserved (See Appendix D) **D6** Self Powered **D5** Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x20**	Maximum Power Consumption. 0x20 = 64mA

Notice that the Self-powered bit is set. This particular hub does not have an external power source. To be sure if the device is self-powered, send the Get_Status request, checking bit 0 of the returned data.

The Configuration Descriptor has a string descriptor index of 0x04. For this hub, the same string is returned as index 0x02 above, so I won't display it here.

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows those 9 bytes.

```
0x00000000                    09 04 00 00 01 09 00
0x00000010  00 05
```

Table 19-5: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x09**	Class Code: 0x09 = HUB
6	Sub Class	1	**0x00**	Sub Class Code: 0x00
7	Protocol	1	**0x00**	Protocol Code: 0x00
8	Interface Str	1	**0x05**	String descriptor index.

This descriptor has a string descriptor index of 0x05. For this hub, the same string is returned as index 0x02 above, so I won't display it here. This descriptor, like the configuration descriptor, also tells us what kind of device it is.

The Configuration descriptor also indicates that there is one Endpoint Descriptor.

```
0x00000010      07 05 81 03 01 00-FF
```

Table 19-6: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0001**	Max packet size: 1 byte
6	Interval	1	**0xFF**	Poll every 255mS

This indicates that we need to poll this interrupt endpoint every 255mS for a 1-byte status value.

The interrupt endpoint is used to detect a change in the hub and/or an individual port. The max packet size of this endpoint will depend on how many ports there are. The format of the report returned is the same no matter the number of ports, just that there will be a larger report returned. For example, our 4 port hub will need 5 bits to report the status change bitmap. Therefore, the endpoint's max packet size is 1 byte. The format of the report returned is as follows. Each bit, when set indicates that there has been a change to that item.

Table 19-7: Interrupt Endpoint Return Data

Bit	Description
7:5	Reserved
4	Port 4 change detected.
3	Port 3 change detected.
2	Port 2 change detected.
1	Port 1 change detected.
0	Hub status change detected.

If there are 8 or more ports, then at least another byte is needed. The report will be wide enough to have a bit for each port in existence plus one more for bit 0.

It is up to you to decide how you want to monitor changes in each port on the hub. You can continue to send Get_Status requests to each port, which takes time and schedule bandwidth, or you can simply create an Interrupt Endpoint packet in your schedules periodic list and monitor the report.

Configure the External Hub

Now that we have retrieved most of the device descriptors, we now know that it is an external hub class device, having a single interrupt endpoint with an endpoint value of 1.

Please note that the USB 2.0 specification states that you must first configure the hub before you can request the Hub Descriptor. All USB 2.0 devices I have seen allow this descriptor request without the hub being configured. However, the USB 3.0 hub I have does not. Please see Chapter 21 on USB 3.0 hubs for a Catch-22 situation about retrieving the Hub Descriptor before the device is configured.

Now is when we can retrieve the hub descriptor.

Table 19-8: Get Hub Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bits 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bits 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x2900**	Type and Index.
4	Index	2	**0x0000**	Zero or Language ID.
6	Length	2	**0x0047**	At most 71 bytes.

We request at most 71 bytes. After describing the format of the returned descriptor, I will explain why we request this amount in a later section of this chapter.

Please note that if you are reading this chapter and requesting this descriptor from a USB 3.0 device, the Type field will be 0x2A and the Length field will be 12. This is because the USB 3.0 hub descriptor is different than the USB 2.0 descriptor. See Chapter 21 on USB 3.0 hubs for more information.

The Micro Innovations hub I have returns the following memory dump.

```
0x00000000  09 29 04 00 00 32 40 00-1E
```

and explained here.

Table 19-9: Hub Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x29**	USB 2.0 Hub Descriptor
2	NumPorts	1	**0x04**	Number of downstream ports.
3	HubChars	2	**0x0000**	Bits 15:8 Reserved

				Bit 7 Port indicators 0 = not supported 1 = supported Bits 6:5 TT Think Time 00: 8 FS bit times 01: 16 FS bit times 10: 24 FS bit times 11: 32 FS bit times Bits 4:3 Over-current PM 00: Global 01: Individual 1x: none Bit 2 Compound device? 0: No 1: Yes Bits 1:0 Power Switching 00: Global 01: Individual 1x: none
5	POTPGT	1	**0x32**	Power on to power good time. 0x32 = 100mS
6	MaxHubCurrent	1	**0x40**	Maximum current used by the hubs controller. 0x60 = 64mA
7	Removable	*n*	**0x00**	Bit 0 = reserved Bit 1 = 1st port = removable device attached? Bit 2 = 2nd port = removable device attached? etc.
7+*x*	PwrCntrlMask	*n*	**0x1E**	To be backward compatible with version 1.0 of the USB specification. Each bit should be set for each port present, with bit 0 reserved

Detecting Number of Downstream Ports

With the hub descriptor above, we know how many downstream ports there are on this hub, in this case, 4.

Byte 3, the Hub's Characteristic field, has some important information and not so important information. Bit 7 indicates whether this hub supports Port Indicators. Port Indicators are the green and orange lights you sometimes see on hubs. If bit 7 is set, you can use the Port_Indicator request to control these indicators.

Bits 4:3 specify if an over-current event is reported as, any port having an over-current event, with no port specified, or if the hub will report on a per-port event.

Bit 2 specifies if the hub is part of a compound device.

Bits 1:0 specify whether all ports are powered at once, or if each individual port can be powered.

The POTPGT field specifies how long, in 2mS intervals, your driver must wait after powering the port, before it can access that port. If the Global Power Switching bits above are 00b, then just make sure you don't access any of the ports POTPGT * 2 milliseconds after the hub is powered.

Byte 7, the Device Removable field, indicates whether the attached device is removable. Bit 0 is reserved. The first port starts with bit 1, the second port with bit 2, etc. If the bit is set, the attached device is non-removable. For example, I have a 7 port external hub. It is actually two 4-port hubs with the last port on the first hub physically attached to the second hub. Therefore, the last port on the first hub is non-removable. This field's size is the amount of bits used, rounded up to a byte. For example, we have 4 ports on this hub. With the first bit reserved, we use 4 more bits. Rounded up, this would take 1 byte.

The next field, the PortPwrCtrlMask field, is right after the Device Removable field. Its offset is 7 + the amount of bytes used for the Device Removable field. This field was used in the USB version 1.0 specification and now is set for backward compatibility. With bit 0 being reserved, each bit from then on should be set for an existing port. This field's length is also found by counting the available ports, adding one, and rounding to the next byte.

With this in mind, we now know why we can read a maximum of 71 bytes for this descriptor. If the specification says that we can have up to 255 ports, if bit 0 is reserved, that is 32 bytes per field above. Thirty two times 2, plus 7 other bytes is 71.

Now let's get the status of the hub to verify that it is a self-powered hub.

Table 19-10: Get (Hub) Status Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x80**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface

				2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x00**	GET STATUS
2	Value	2	**0x0000**	Zero
4	Index	2	**0x0000**	Zero
6	Length	2	**0x0002**	Request two bytes

On a control-in packet of two bytes, you should receive the following format.

Table 19-11: Status Request Return Data

Offset	Description
0	Bit 0 Self Powered 0 = Bus powered 1 = Self powered Bit 1 Remote Wakeup 0 = Disabled 1 = Enabled Bits 7:2 Reserved
1	Reserved

My Micro Innovations hub still reports that it is self-powered, even though it is obviously not.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the Set_Configuration request with a configuration number of 1, retrieved from the ConfigVal field above.

Table 19-12: Set Configuration Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x00**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x09**	SET CONFIGURATION
2	Value	2	**0x0001**	Config value from Configuration Descriptor.
4	Index	2	**0x0000**	Zero
6	Length	2	**0x0000**	No bytes transferred.

An Interface Number, retrieved from the Interface Descriptor, may have a value of zero, whereas the Config Number in the Configuration Descriptor will always be at

least 1, since setting the Configuration with a Config value of zero will place the device in the address state instead of the configured state.

Resetting and Enabling the Hub

We now have a configured, ready to use USB external hub. You reset and enable a hub attached to the root hub just like any other device attached to the root hub.

The reset and enable of a device on the external hub, is a little different than a device on the root hub. The next section shows how to reset and enable a port on the external hub.

Powering, Resetting, and Enabling a Port on the Hub

Now that we have a ready to use hub, we need to power each port. We could check the Global Power Switching mode in the hub's descriptor, but it doesn't hurt anything to try to power each port individually.

When sending a port number to the hub, all ports are 1 based. Meaning that the first port is port 1, not port zero.

To get information and set or clear a feature on the port, you use the Set_Feature and Get_Feature requests. A list of Request values for the hub list listed here.

Table 19-13: Set_Feature/Clear_Feature

Request	Value
GET_STATUS	0 (0x00)
CLEAR_FEATURE	1 (0x01)
Reserved	2 (0x02)
SET_FEATURE	3 (0x03)
Reserved	4 (0x04)
Reserved	5 (0x05)
GET_DESCRIPTOR	6 (0x06)
SET_DESCRIPTOR	7 (0x07)
CLEAR_TT_BUFFER	8 (0x08)
RESET_TT	9 (0x09)
GET_TT_STATE	10 (0x0A)
STOP_TT	11 (0x0B)

Set_Feature and Clear_Feature each send a Feature Selector, which could be considered a task to do, in the Value field of the standard request packet.

Table 19-14: Feature Selector

Request	Value
PORT_CONNECTION	0 (0x00)
PORT_ENABLE	1 (0x01)
PORT_SUSPEND	2 (0x02)
PORT_OVER_CURRENT	3 (0x03)
PORT_RESET	4 (0x04)
PORT_POWER	8 (0x08)
PORT_LOW_SPEED	9 (0x09)
C_PORT_CONNECTION	16 (0x10)
C_PORT_ENABLE	17 (0x11)
C_PORT_SUSPEND	18 (0x12)
C_PORT_OVER_CURRENT	19 (0x13)
C_PORT_RESET	20 (0x14)
PORT_TEST	21 (0x15)
PORT_INDICATOR	22 (0x16)

The Feature Selectors with a C_ prefix indicate that they effect the Change Status bits.

It might be worth noting that the C_ prefix values are the same as the non C_ prefix value of the same name, except that bit 4 is set.

To set the power for a hub's port, you send the Set_Feature(PORT_POWER) request.

Table 19-15: Set Feature Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x23**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint **3 = Other** 4..31 = Reserved
1	Request	1	**0x03**	SET FEATURE
2	Value	2	**0x0008**	PORT_POWER
4	Index	2	**0x0001**	Low byte = port (1 based)
6	Length	2	**0x0000**	No bytes transferred.

So that your driver does not get a false Connect Status Change message, once you have set the power to each port, waiting the indicated time (POTPGT*2), you then need to clear the Connection Status bit in the Port Change status word.

To do this, send the Clear_Feature(C_PORT_CONNECTION) request.

Table 19-16: Clear Feature Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x23**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint **3 = Other** 4..31 = Reserved
1	Request	1	**0x01**	CLEAR FEATURE
2	Value	2	**0x0010**	C_PORT_CONNECTION
4	Index	2	**0x0001**	Low byte = port (1 based)
6	Length	2	**0x0000**	No bytes transferred.

Next we can get the status of the port by sending the Get_Status request.

Table 19-17: Get (Port) Status Request

Offset	Field	Size	Value	Description

0	Request type	1	**0xA3**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint **3 = Other** 4..31 = Reserved
1	Request	1	**0x00**	GET STATUS
2	Value	2	**0x0000**	Zero
4	Index	2	**0x0001**	Low byte = port (1 based)
6	Length	2	**0x0004**	Request four bytes

The four bytes returned are described in the next two tables.

The first 16-bit word is the Port Status bits.

Table 19-18: Port Status Field

Bit	Description
15:13	Reserved
12	**Port Indicator Control**: (PORT_INDICATOR) 0 = Port indicator uses default colors. 1 = Port indicator uses software controlled color.
11	**Port Test Mode**: (PORT_TEST) 0 = Port is not in the Port Test Mode. 1 = Port is in the Port Test Mode.
10	**High-speed Device Attached**: (PORT_HIGH_SPEED) 0 = Full-speed device attached. 1 = High-speed device attached.
9	**Low-speed Device Attached**: (PORT_LOW_SPEED) 0 = Full- or High-speed device attached. 1 = Low-speed device attached.
8	**Port Power**: (PORT_POWER) 0 = Port is in powered off state. 1 = Port is powered.
7:5	Reserved
4	**Reset**: (PORT_RESET) 0 = Port is not in reset. 1 = Port is currently in reset.
3	**Over-Current**: (PORT_OVER_CURRENT) 0 = No over-current condition. 1 = An over-current condition exists on this port.
2	**Suspend**: (PORT_SUSPEND) 0 = not suspended. 1 = suspended or currently resuming.
1	**Port Enabled/Disabled**: (PORT_ENABLE) 0 = Port is disabled.

	1 = Port is enabled.
0	**Current Connect Status**: (PORT_CONNECTION) 0 = No device attached. 1 = A device is attached.

The second 16-bit word is the Port Change Status bits.

Table 19-19: Port Change Status Field

Bit	Description
15:5	Reserved
4	**Reset Change**: (C_PORT_RESET) 0 = No Change. 1 = Reset complete.
3	**Over-Current Indicator Change**: (C_PORT_OVER_CURRENT) 0 = No change. 1 = An over-current indicator has changed.
2	**Suspend Change**: (C_PORT_SUSPEND) 0 = No change. 1 = Resume completed.
1	**Port Enable/Disable Change**: (C_PORT_ENABLE) 0 = No change. 1 = The port has been disabled due to an error condition.
0	**Current Connect Status Change**: (C_PORT_CONNECTION) 0 = No change. 1 = Change in current connect status.

The Micro Innovations hub I have, with a low speed mouse attached, returns the following 4 bytes.

```
01 03
00 00
```

Once you have the status of the port, you can check the PORT_CONNECTION bit to see if a device is attached. If so, you need to send a Set_Feature(PORT_RESET) request.

Table 19-20: Set Feature Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x23**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint **3 = Other** 4..31 = Reserved

1	Request	1	**0x03**	SET FEATURE
2	Value	2	**0x0004**	PORT_RESET
4	Index	2	**0x0001**	Low byte = port (1 based)
6	Length	2	**0x0000**	No bytes transferred.

When the port is finished with the reset process, it will automatically stop the reset. It will then immediately enable the port, if a device is attached.

If you had your interrupt endpoint scheduled, this is where you would get a report returned indicating that this port has changed.

Get the status again, using Table 19-17 Get (Port) Status Request. Now the status returned should be similar to the following.

```
03 03
10 00
```

This indicates that a device is attached, the port is enabled and powered, and is a low-speed device. The C_PORT_RESET bit is also set in the Port Change field. Therefore, you need to send a Clear_Feature(C_PORT_RESET) request to clear the bit.

Table 19-21: Clear Feature Request

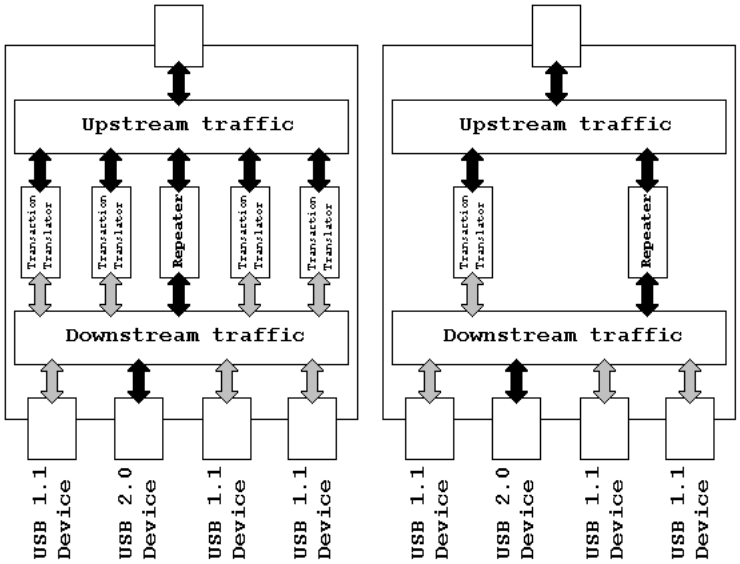
Offset	Field	Size	Value	Description
0	Request type	1	**0x23**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device 1 = Interface 2 = Endpoint **3 = Other** 4..31 = Reserved
1	Request	1	**0x01**	CLEAR FEATURE
2	Value	2	**0x0014**	C_PORT_RESET
4	Index	2	**0x0001**	Low byte = port (1 based)
6	Length	2	**0x0000**	No bytes transferred.

USB 2.0 Hubs and Transaction Translators

There are a few other things you may need to know about hubs. For example, high-speed hubs must have at least one Transaction Translator (TT) to translate any low- or full-speed transactions to high-speed transactions. A high-speed hub can have a single TT or can be a multi-TT. If a hub has a single TT, each USB 1.1 device plugged into the hub must share the single TT and therefore will operate slower than if that same USB 2.0 hub had multiple TT's. For example, a high end USB 2.0 hub will have a TT for every down stream port allowing for faster transactions.

If a high-speed device is plugged into one of the ports, the hub's repeater skips the TT and allows the transaction to continue to the parent hub. The transaction does not need to be translated. See the figure below for an example of a four port hub with a single TT and then a multi-TT setup.

Figure 19-1: Single- and Multi-TT



The hub on the left is a multi-TT hub and has a Transaction Translator for each port on the hub while the hub on the right only has a single TT which must translate all traffic on all ports. The hub on the left will transfer data much faster than the one on the right when multiple USB 1.1 devices are attached. USB 2.0 devices don't need a TT and therefore all USB 2.0 traffic passes through the repeater.

Some of the fields in the xHCI host controller's slot context are Single- and Multi-TT aware.

Other Comments

That is all for the hub configuration. Since you have reset the port that a device is attached to, that device will be in the default state and will answer to the default address of zero. The Hub will now do all of the work for us. You can send the enumeration process out on the bus as if the hub was not there. In other words, enumerate the newly attached device as if it was attached to the root hub.

Be sure that you do not reset and enable more than one port on the hub at a time without first setting the address of each newly attached device. If you don't set the address of the newly attached device first, each device will be in the default state and answer to the default address of zero.

It will be up to your driver to monitor the status of the hub, to find when another device is attached or an existing is removed. Your driver will also have to keep track of the hierarchy of the hub tree. If the user unplugs a hub, your driver will need to remove all references to any devices that may have been plugged in to that hub, in turn, causing a disconnect for all of those devices.

Technically, you may have as many hubs plugged in to each other as you want as long as you don't violate the cable length before a powered hub specification and have 127 or fewer devices attached to the root hub.

Chapter 20 – Using the USB Printer Device

Do you remember the Dot-Matrix printers, the printers with numerous pins that shot out at a ribbon to leave an impression on the paper? You sent special codes to the printer via a serial or parallel line, then the small processor on the printer would change those codes to magnetic pulses to push the pins in and out.

Later, inkjets and laser printers that can handle much more detail and many more dots per inch started to show on the market. Most of those printers were connected via USB or a network card. Those printers were controlled via a printing language sent to the printer's processor, which would transform it to an image and print it to the paper media. Hewlett Packard (HP) was, and still is, a major printer manufacturer, and created some of the languages that are used. For example, the HP PCL language (HP Page Description Language) was used on later dot-matrix printers around 1980 and is now currently used on a majority of printers.

With the PCL protocol, any generic driver could use the general functions of the printer. However, this required a processor on the printer to take the PCL data and transform it to a pixel image to print to the media. These processors made the printers a little more expensive. To save money and to be able to sell the printers for less, HP decided to create Host Based printers. These printers would rely on the host's driver to change the image to be printed to a pixel image and send only the image to the printer, therefore eliminating the need for the printer's processor. However, this drastically slowed the host since it now had to be used to transform the image. This does away with the PCL protocol and those generic printer drivers could no longer print to the printer.

> When you go printer shopping, read the fine print on the box, or better yet, research the printer on-line before you make your decision. I strongly suggest that you do not purchase a "Host-based" printer. Each time you send a print job to the printer, or if you have other computers on the network that use your printer, your computer will slow considerably while the print job is being performed. "Host-based" printers don't print .pdf files very well either. On some printers, HP will actually place on the box that this printer is not intended to print .pdf files.

Fortunately, the USB Printer Class specifies a request that all USB Printer Class printers must support. This request returns information about the printer and what languages it supports.

Since each printer may be different, may be a host-based printer, may have a different protocol than PCL, or may be completely different, I will not go into much detail about the actual printing of an image. I will show how to get the IEEE 1284 Device ID String which specifies what type of printer it is, and attempt to print a single line of text to the printer.

Be warned that unless your printer supports the same version of the PCL language I use in the example below, you may not have successful results. My intent is to show how to communicate with the printer on the USB side, not to show how to create an image and print images.

The printer I used for the research of this book is a HP™ PSC 750xi. All of the example descriptors and information in this chapter are for this unit. However, the information and code within this chapter should work on any similar unit.

> The printer has the model number printed as a HP PSC 750. However, the string descriptors and Device ID String all say that it is a HP PSC 750xi.

As mentioned before, you should be familiar with how to retrieve the device's descriptors via the enumeration process and send and receive data packets.

The Printer's Descriptors

During the enumeration process you should have been able to retrieve the printer's first descriptor, the Device Descriptor, and it should be similar to the following memory dump.

```
0x00000000  12 01 10 01 00 00 00 08-F0 03 11 15 00 01 01 02
0x00000010  03 01
```

and explained here.

Table 20-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0110**	USB Specification Release Number in Binary-Coded Decimal. 0x0110 = v1.10
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x08**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x03F0**	Vendor ID[1]: Hewlett-Packard
10	Product ID	2	**0x1511**	Product ID: PSC 750xi
12	Device Rel	2	**0x0100**	Release version 1.00
14	Manufacturer	1	**0x01**	Index of string descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x03**	Index of Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump.

```
0x00000000  04 03 09 04
```

and explained here.

Table 20-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID	2	**0x0409**	0000_01|00_0000_1001b

	word			Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This printer returns a string index for the Manufacturer ID, Product ID, and Serial Number with the following values.

```
0x00000000  20 03 48 00 65 00 77 00-6C 00 65 00 74 00 74 00
0x00000010  2D 00 50 00 61 00 63 00-6B 00 61 00 72 00 64 00
```

Table 20-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x20**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x1E	**0x0048** **0x0065** **0x0077** **0x006C** **0x0065** **0x0074** **0x0074** **0x002D** **0x0050** **0x0061** **0x0063** **0x006B** **0x0061** **0x0072** **0x0064**	H e w l e t t - P a c k a r d

```
0x00000000  14 03 50 00 53 00 43 00-20 00 37 00 35 00 30 00
0x00000010  78 00 69 00
```

Table 20-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x14**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x12	**0x0050** **0x0053** **0x0043** **0x0020** **0x0037** **0x0035** **0x0030** **0x0078** **0x0069**	P S C <space> 7 5 0 x i

```
0x00000000  1A 03 4D 00 59 00 32 00-39 00 51 00 44 00 34 00
```

```
0x00000010  31 00 44 00 35 00 57 00-42 00
```

Table 20-5: String Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x1A**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x18	**0x004D** **0x0059** **0x0032** **0x0039** **0x0051** **0x0044** **0x0034** **0x0031** **0x0044** **0x0035** **0x0057** **0x0042**	M Y 2 9 Q D 4 1 D 5 W B

To this point, we should have an idea what the device is and have a human readable string to associate it with. However, as noted before, the class code in the Device Descriptor above was zero, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor.

```
0x00000000  09 02 4E 00 01 01 00 C0-01 09 04 00 00 03 07 01
0x00000010  03 00 07 05 01 02 40 00-00 07 05 81 02 40 00 00
0x00000020  07 05 82 03 08 00 0A 09-04 00 01 02 07 01 02 00
0x00000030  07 05 01 02 40 00 00 07-05 81 02 40 00 00 09 04
0x00000040  00 02 01 07 01 01 00 07-05 01 02 40 00 00
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 4E 00 01 01 00 C0-01
```

Table 20-6: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x004E**	Total length of all data within this configuration descriptor (78 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.

7	Attributes	1	**0xC0**	**D7** Reserved (See Appendix D) **D6** Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x01**	Maximum Power Consumption. 0x01 = 2mA

This Configuration Descriptor indicates that we have only one Interface Descriptor which follows those 9 bytes. Be sure to see the note later in this chapter on how many interfaces this device actually has. The interface descriptor is as shown next.

```
0x00000000                    09 04 00 00 03 07 01
0x00000010  03 00
```

Table 20-7: Interface Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Alternate Interface 0
4	Endpoints	1	**0x03**	Number of Endpoints: 3
5	Class Code	1	**0x07**	Class Code: 0x07 = Printer
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Printers.
7	Protocol	1	**0x03**	Protocol Code[1].
8	Interface Str	1	**0x00**	No string descriptor.

[1]See page 4 of the USB Printer Class specification, usbprint.pdf.

This descriptor now tells us what kind of device it is. Using the class code, we know that it is a Printer Class and the subclass code indicates that it is a Printer, while the protocol tells us that it uses the IEEE 1284.4 compatible bi-directional interface for transferring data. Please see notes below on the two alternate interfaces and their protocols.

This descriptor also indicates that there are three Endpoint Descriptors that follow.

```
0x00000010      07 05 01 02 40 00-00
```

Table 20-8: Endpoint Descriptor 1 of Interface 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x01**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes

6	Interval	1	**0x00**	No Polling needed.

0x00000010 07 05 81 02 40 00 00

Table 20-9: Endpoint Descriptor 2 of Interface 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

0x00000020 07 05 82 03 08 00 0A

Table 20-10: Endpoint Descriptor 3 of Interface 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x82**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **2**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0008**	Max packet size: 8 bytes
6	Interval	1	**0x0A**	Poll every 10mS.

The first endpoint is the OUT endpoint and has an endpoint value of 1. This is the endpoint that you write data to at a maximum of 64 bytes at a time, as specified in the max packet field.

The second endpoint is the IN endpoint and is identical to the first except that you read data using this endpoint instead of writing data. This endpoint has an endpoint value of 1 also. You read and write to the same endpoint.

The third endpoint is the interrupt endpoint. This endpoint is used with the IEEE 1284.4 protocol.

If you will note, the configuration descriptor only indicated one interface descriptor. However, we have not reached the end of the configuration descriptor yet. The USB Printer Class specification says that this class device may have up to three interface types, each with a

protocol of 0x01, 0x02, or 0x03, explained later in this chapter. For now, lets view the remaining interface descriptors.

0x00000020 09-04 00 01 02 07 01 02 00

Table 20-11: Interface Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x01**	Alternate Interface 1
4	Endpoints	1	**0x02**	Number of Endpoints: 2
5	Class Code	1	**0x07**	Class Code: 0x07 = Printer Class
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Printers.
7	Protocol	1	**0x02**	Protocol Code. (Bi-dir)
8	Interface Str	1	**0x00**	No string descriptor.

This alternate interface uses a protocol of 0x02 for bi-directional interface using the OUT endpoint to send data to the printer and the IN endpoint for status.

This descriptor also indicates that there are two Endpoint Descriptors that follow.

0x00000040 07 05 01 02 40 00 00

Table 20-12: Endpoint Descriptor 1 of Interface 2

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x01**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No polling needed.

0x00000040 07-05 81 02 40 00 00

Table 20-13: Endpoint Descriptor 2 of Interface 2

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0)

				3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No Polling needed.

The first endpoint is the OUT endpoint and has an endpoint value of 1. This is the endpoint that you write data to at a maximum of 64 bytes at a time, as stated in the max packet size field.

The second endpoint is the IN endpoint and is the endpoint you read status from. It also has an endpoint value of 1, meaning you read from and write to the same endpoint.

Here is the final alternate interface descriptor.

```
0x00000040                              09 04
0x00000050  00 02 01 07 01 01 00
```

Table 20-14: Interface Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x02**	Alternate Interface 2
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x07**	Class Code: 0x07 = Printer
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Printers.
7	Protocol	1	**0x01**	Protocol Code. (Uni-dir)
8	Interface Str	1	**0x00**	No string descriptor.

This alternate interface uses a protocol of 0x01 for Uni-directional interface using the OUT endpoint to send data to the printer and the control endpoint for status.

This descriptor indicates that there is one Endpoint Descriptor that follows.

```
0x00000050              07-05 01 02 40 00 00
```

Table 20-15: Endpoint Descriptor 1 of Interface 3

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x01**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type

				00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0040**	Max packet size: 64 bytes
6	Interval	1	**0x00**	No polling needed.

This endpoint is the OUT endpoint and has an endpoint value of 1. This is the endpoint that you write data to at a maximum of 64 bytes at a time, again, as indicated in the max packed size field. To receive status, you read from the control endpoint with a GET_PORT_STATUS request.

The Three Available Protocols

As stated before, there are three available protocols specified in the Printer Class specification. The printer I have supports all three by giving a default interface and two more alternate interfaces each supporting one of the different protocols.

The default interface, the first interface it lists, uses a protocol value of 0x03. The Printer Class specification states:

> *IEEE 1284.4 compatible Bi-directional Interface. The 1284.4 interface supports sending data to the printer via a Bulk OUT endpoint, and receiving status and other information from the printer via a Bulk IN endpoint, just as the bi-directional interface does. The 1284.4 interface additionally specifies that data will be transmitted to and from the device using the 1284.4 protocol. The 1284.4 protocol is defined by the "IEEE P1284.4 Standard for Data delivery and logical channels for IEEE std. 1284 interfaces, version 1.0." This specification is pending final approval.*

Since my printer has a scanner, it must use this interface to receive data from the scanner.

The first alternate interface used above has a protocol value of 0x02, the Bi-directional protocol. This protocol is used to send data to the printer using the Bulk-Out endpoint and receiving status and other information from the Bulk-in endpoint.

The second alternate interface has a protocol value of 0x01, the Uni-directional protocol and is identical to the Bi-directional protocol for sending data. However, it uses the control endpoint to receive the status of the printer, discussed below.

The last two protocols above, protocols 0x01 and 0x02, have a feature to return the status compatible to the old parallel port type status. To do so, send the following request on the control endpoint.

Table 20-16: Get Port Status Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA1**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x01**	Get Port Status
2	Value	2	**0x0000**	Zero
4	Index	2	xx	Interface (zero based)
6	Length	2	**0x0001**	We want 1 byte returned.

This will return a byte with the following format.

Table 20-17: Get Port Status Return

Bit	7	6	5	4	3	2	1	0

Byte								
	Reserved		Empty	Select	No Error	Reserved		

If bit 5 is set, the paper tray is empty, if bit 4 is set, the printer is selected, and it bit 3 is set, there is no error.

Configure the Printer

Now that we have retrieved all of the device descriptors, we now know that it is a Printer capable of using one of the three protocols. We can set the configuration and select and set one of the available interfaces.

As mentioned before, this chapter is used to teach you how to use the USB interface, not the printer interface. Therefore, I use the Uni-directional protocol for the remainder of this chapter. However, using the Bi-directional protocol and ignoring the status bulk-in endpoint, you will have the same results.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the "Set Configuration" request with a configuration number of 1, retrieved from the ConfigVal field above.

Table 20-18: Set Configuration Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x00**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x09**	SET CONFIGURATION
2	Value	2	**0x0001**	Config value from Configuration Descriptor.
4	Index	2	**0x0000**	Zero
6	Length	2	**0x0000**	No bytes transferred.

Then send the Set Interface request. This is where you would pick the alternate interface you wish to use. In this example, we use a value of 02h, which is the Uni-directional interface of this printer device.

Table 20-19: Set Interface Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x01**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint

				3 = Other 4..31 = Reserved
1	Request	1	**0x0B**	SET INTERFACE
2	Value	2	**0x0002**	Alternate Interface Number
4	Index	2	**0x0000**	Interface Number
6	Length	2	**0x0000**	No bytes transferred.

Now we can do a Soft Reset of the printer. This will clear all printer buffers, reset the Bulk endpoints, clear all stall conditions, but does not reset the interface or configuration already set above.

Table 20-20: Soft Reset Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other
1	Request	1	**0x02**	SOFT RESET
2	Value	2	**0x0000**	Zero
4	Index	2	**0x0000**	Interface Number
6	Length	2	**0x0000**	No bytes transferred.

A well known, not so recent operating system I have, sends the Request Type above as 0x23 instead of 0x21. Version 1.0 of the Printer Class specification incorrectly stated that the request type was 0x23. Version 1.1 has it as 0x21 and states that any printer device should accept and successful handle a request type of 0x23.

Get IEEE 1284 Device ID String

Now that we have configured and reset the printer, we can get the printer's identification string. A USB Printer Class printer must accept the IEEE 1284 Get Device ID String request.

Table 20-21: Get Device ID Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA1**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x00**	GET DEVICE ID
2	Value	2	**0x0000**	Config Index (zero based)
4	Index	2	**0x0000**	High byte = Interface. Low byte = Alt Interface.
6	Length	2	**0x03FF**	Request at most 1023 bytes.

I don't have a copy of the IEEE 1284 specifications to find out why, but I am uncertain the reason for the request of 0x03FF bytes. Two of the modern operating systems I checked also request this amount. One of them requested 0x03F1 bytes in one instance, so I don't think it is a constant amount.

The printer will then return at most the requested 1023 bytes of ID in the form used by the IEEE 1284 specification, with the first two-byte word being the size returned in big-endian format.

Another operating system checks this length value and if it is larger than expected, it reads it in as a little-endian word and checks the length again. The comment included states that it does this to check for devices that "forgot" to big-endian the length.

The remaining bytes are ASCII characters as keys (or tokens) with parameters with the format outlined here.

Key: parameter0 **{**, parameter1**}**;

There must be a key and at least one parameter separated by commas with the end marked with a semi-colon.

My printer returns 229 bytes.

```
0x00000000  00 E5 4D 46 47 3A 48 65-77 6C 65 74 74 2D 50 61
0x00000010  63 6B 61 72 64 3B 4D 44-4C 3A 50 53 43 20 37 35
0x00000020  30 78 69 3B 43 4D 44 3A-4D 4C 43 2C 50 43 4C 2C
0x00000030  50 4D 4C 3B 43 4C 41 53-53 3A 50 52 49 4E 54 45
0x00000040  52 3B 44 45 53 43 52 49-50 54 49 4F 4E 3A 48 65
```

```
0x00000050  77 6C 65 74 74 2D 50 61-63 6B 61 72 64 20 50 53
0x00000060  43 20 37 30 30 20 53 65-72 69 65 73 3B 31 32 38
0x00000070  34 2E 33 4D 3A 66 37 66-2C 66 37 66 3B 31 32 38
0x00000080  34 2E 34 44 4C 3A 34 64-2C 34 65 2C 31 3B 53 45
0x00000090  52 4E 3A 30 31 32 33 34-35 36 37 38 39 41 42 3B
0x000000A0  56 53 54 41 54 55 53 3A-24 48 42 30 24 4E 43 30
0x000000B0  2C 66 66 2C 44 4E 2C 49-44 4C 45 2C 43 55 54 2C
0x000000C0  4B 30 2C 43 33 2C 53 4D-2C 4E 52 2C 4B 50 30 31
0x000000D0  31 2C 43 50 30 30 30 3B-41 69 4F 3A 30 3B 44 57
0x000000E0  2D 50 43 4C 3B
```

The first two bytes, in big-endian format, are 0x00E5, or 229d, and the remaining in token form is listed below. The CRLF pairs in the listing, is not included in the data returned above. I included them below to be more pleasing to the eye.

```
MFG:Hewlett-Packard;
MDL:PSC 750xi;
CMD:MLC,PCL,PML;
CLASS:PRINTER;
DESCRIPTION:Hewlett-Packard PSC 700 Series;
1284.3M:f7f,f7f;
1284.4DL:4d,4e,1;
SERN:0123456789AB;

VSTATUS:$HB0$NC0,ff,DN,IDLE,CUT,K0,C3,SM,NR,KP011,CP000;
AiO:0;
DW-PCL;
```

I don't know why the last entry has only a key value and no parameters. Again, I don't have a copy of the specification, and my intent for this chapter is not to explain the printer specification other than the USB protocol used to get there. If you need more information on the above format, you can purchase the IEEE 1284 specification from www.ieee.org for a minimal fee.

Just for interest sake, Christoph Lindemann has an interesting page at:

www.undocprint.org/formats/communication_protocols/ieee_1284

with some information about some of the keys above. For example, the VSTATUS:K0 key tells me that the black ink cartridge is present. For the color cartridge, the VSTATUS:C3 should be C0 to show that it is present. However, since the VSTATUS:CP000 shows that the percentage left of ink in that cartridge is 0, maybe the C3 means it is present but empty. Either way, you will need to acquire the IEEE 1284 specifications to be sure.

Printing Text Using the PCL Language

As mentioned before, your printer may or may not support the PCL printer language. This chapter uses this language for the examples below. If your printer does not support this language, you will need to modify the examples to accommodate the language you choose.

Now your printer should be ready to receive PCL codes. You simply send the PCL code as bulk-out packets to the bulk out endpoint. If the printer is a full-speed printer using 64-byte packets, you can send 64 bytes at a time.

I have included a file in the cdrom\misc folder on the included disc called pcl_demo0.bin. When sent to the printer, it will print the text shown in the figure below.

Figure 20-1: Print Demo 1

Untitled

This is a test.

Page 1

This is the usual Windows™ Notepad® style printout of a test file.

I have also included pcl_demo1.bin. It is a monochrome bitmap image of the following, which should be printed in the middle of the page.

Figure 20-2: Print Demo 2

Other Comments

Remember that all printers are different, especially if they are “host-based” as explained at the beginning of this chapter. That is why you see so many different printer drivers.

This chapter is only intended to show how to communicate with the printer, not to show how to print.

The HP PCL specification is available online. First try the web page at http://www.undocprint.org. There are other resources too.

Chapter 21 – Using a USB HUB on the xHCI Root Hub

Using a USB hub on the xHCI root hub is very similar to a USB 2.0 root hub. For example, if you plug a USB 2.0 external hub in to the xHCI root hub, it will enumerate as a USB 2.0 hub as expected. If you plug a super-speed external hub into a USB 2.0 root hub, it will also enumerate as a USB 2.0 external hub with little or no indication that it is a super-speed device. For an example, see below.

This chapter explains a few differences when you plug a super-speed external hub in to a xHCI root hub, and the additional descriptors and details you will need to know.

First, here is an example of a descriptor set when you plug a super-speed external hub into an OHCI root hub. Note that it enumerates just as if it were a full-speed USB 2.0 hub.

```
Device Descriptor
0x00000000  12 01 00 02 09 00 00 40-09 21 31 34 75 02 00 01
0x00000010  00 01
String Lang Descriptor
0x00000000  04 03 09 04
String Descriptor
0x00000000  16 03 55 00 53 00 42 00-32 00 2E 00 30 00 20 00
0x00000010  48 00 75 00 62 00
                1 [USB2.0 Hub]
Config Descriptor
0x00000000  09 02 19 00 01 01 00 E0-32 09 04 00 00 01 09 00
0x00000000  00 00 07 05 81 03 01 00-FF
Interface Descriptor
0x00000000  09 04 00 00 01 09 00 00-00
Hub Descriptor
0x00000000  09 29 04 E9 00 32 64 00-FF
Endpoint Descriptor
0x00000000  07 05 81 03 01 00 FF
```

Notice that the max packet size field is 64 and that it returns a string descriptor that states that it is a USB 2.0 hub. The hub will act and function just as a USB 2.0 full-speed hub until it is plugged into a super-speed hub or root hub.

In the remainder of this chapter, I will explain how to initialize and communicate with a four-port USB super-speed external hub plugged into a xHCI root hub, how it is to be enumerated, and how to capture port status changes.

In the examples, I use the J-Tech Digital™, model number JTDUSB3.0-4P four port super-speed external hub. It has one upstream port in the form of a USB 3.0 Micro-B plug connector, and four downstream ports in the form of USB 3.0 standard-A socket connectors. It is an externally powered Hub. Any super-speed USB 3.0 external hub similar to this will work with the following examples.

In this chapter, I will use this hub to show how to determine the amount of downstream ports, reset each port, and determine if another device is attached to one of these downstream ports. You might want to read the chapter on USB 2.0 external hubs before this chapter. It has a little more explanation on the function of external hubs.

Please note that a Super-Speed capable external hub plugged into a xHCI host socket will actually enumerate as two separate devices, the Super-Speed capable

hub and the USB 2.0 capable hub. Also note, that both the USB3 register set and the USB2 register set within the xHCI port register sets will be active and used as if they were separate devices. The external hub is actually a Super-Speed Hub and a USB 2.0 hub sharing only the two Vcc (+) and Vdd (-) wires. In theory, there are two separate devices within the housing, the USB 3.0 Hub and the USB 2.0 hub. They also use the same cable, because if you remember from Chapter 14, the Super-Speed cable contains both USB 2 and USB 3 communication wires. The hub will send all USB 2.0 communications to the USB 2 port register set and all USB 3.0 communications to the USB 3 port register set.

Since they are as separate devices, this chapter will discuss only the Super-Speed device. The USB 2.0 Hub device is discussed in an earlier chapter.

The Hub's Device Descriptors

During the enumeration process you should have been able to retrieve the hub's first descriptor, the Device Descriptor, and it should be similar to the following memory dump

```
0x00000000  12 01 00 03 09 00 03 09-09 21 10 08 75 03 01 02
0x00000010  00 01
```

and explained here.

Table 21-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0300**	USB Specification Release Number in Binary-Coded Decimal. 0x0300 = v3.00
4	Device Class	1	**0x09**	Class code = 0x09 = Hub
5	Sub Class	1	**0x00**	Not used.
6	Protocol	1	**0x03**	USB 3.0
7	MaxPacketSize	1	**0x09**	Max packet size for the control endpoint (see comment below).
8	Vendor ID	2	**0x2109**	Vendor ID[1].
10	Product ID	2	**0x0810**	Product ID.
12	Device Rel	2	**0x0375**	Release version 3.75
14	Manufacturer	1	**0x01**	Manufacturer descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	No Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

Now that USB 3.0 allows max packet sizes of 512, this value no longer fits in the byte field above. Therefore, any USB 3.0+ device descriptor, indicated by the Release Num field, will have a value used as the exponent of two. For example, the value above is 9, giving a max packet size of 512 for endpoint zero.

$$2^{\text{MaxPacketSize}} = 2^9 = 512$$

Currently, only the value of 9, a max packet size of 512, is defined. Any other value is considered illegal.

The Release Num field is in JJ.M.N format. JJ = Major, M = Minor, N = Sub-minor. Currently, version 3.1.0 is available.

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump

```
0x00000000  04 03 09 04
```

and explained here.

Table 21-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This hub returns a string index for the Manufacture ID with the following values.

```
0x00000000  1E 03 56 00 49 00 41 00-20 00 4C 00 61 00 62 00
0x00000010  73 00 2C 00 20 00 49 00-6E 00 63 00 2E 00
```

Table 21-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x1E**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x1C	**0x0056** **0x0049** **0x0041** **0x0020** **0x004C** **0x0061** **0x0062** **0x0073** **0x002C** **0x0020** **0x0049** **0x006E** **0x0063** **0x002E**	V I A <space> L A b s , <space> I n c .

This hub returns a string index for the Product ID with the following values.

```
0x00000000  26 03 34 00 2D 00 50 00-6F 00 72 00 74 00 20 00
0x00000010  55 00 53 00 42 00 20 00-33 00 2E 00 30 00 20 00
0x00000020  48 00 75 00 62 00
```

Table 21-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x26**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.

2	String in Unicode format	0x24	**0x0034** **0x002D** **0x0050** **0x006F** **0x0072** **0x0074** **0x0020** **0x0055** **0x0053** **0x0042** **0x0020** **0x0033** **0x002E** **0x0030** **0x0020** **0x0048** **0x0075** **0x0062**	4 - P o r t <space> U S B <space> 3 . 0 <space> H u b

To this point, we should now have an idea what the device is and have a human readable string to associate it with. However, unlike the HID class devices in the previous chapters, the class code in the Device Descriptor above does tell us what class of device this is. However, let's continue.

The BOS Descriptor

Here is where the USB 3.0 super-speed device differs. We now need to get the Binary Device Object Store (BOS) Descriptor. This descriptor is used to add device level capabilities to the standard descriptors. This descriptor is accessed very similar to the configuration descriptor. It has a stub that contains the total length of all of the included descriptors within the BOS descriptor. You get the descriptor just like the device descriptor, however sending 0x0F instead of 0x01.

Here is a memory dump of the whole descriptor for this particular device.

```
0x00000000  05 0F 2A 00 03 07 10 02-02 00 00 00 0A 10 03 00
0x00000010  0F 00 03 04 E7 00 14 10-04 00 00 00 00 00 00 00
0x00000020  00 00 00 00 00 00 00 00-00 00
```

Please note that each super-speed hub may have a different BOS descriptor with the "sub-descriptors" in different order as the one used here.

The stub of this descriptor has the following data and format.

```
0x00000000  05 0F 2A 00 03
```

Table 21-5: BOS Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x05**	Size of stub in bytes.
1	Type	1	**0x0F**	BOS Descriptor
2	TotalLength	2	**0x002A**	Total length of all data within this BOS descriptor (42 bytes)
4	NumCapEntries	1	**0x03**	Number of Capability entries following this stub.

A BOS descriptor may contain one of the following Capability entries and each will start with the header listed here.

Table 21-6: BOS Capability Stub

Offset	Field	Size	Value	Description
0	Length	1	varies	Size of this entry in bytes.
1	Type	1	**0x10**	Device Capability (Constant)
2	Capability Type	1	varies	Capability type (listed below)
3	Remaining entry	Len - 3	varies	Remaining data in this entry.

The first entry type is the Wireless USB entry and has a Capability type of 0x01. The device I use here, does not have wireless capability, therefore it does not have this entry in the BOS descriptor.

To see the two capability descriptors defined for wireless devices, see section 7.4.1 in the USB Wireless Specification.

The second entry type is the USB 2.0 Extension entry and has a Capability type of 0x02. The example I use has the following memory dump for this capability.

```
0x00000000          07 10 02-02 00 00 00
```

Table 21-7: BOS Descriptor: USB 2.0 Extension

Offset	Field	Size	Value	Description	
0	Length	1	**0x07**	Size of stub in bytes.	
1	Type	1	**0x10**	Device Capability	
2	Cap Type	1	**0x02**	USB 2.0 Extension	
3	Attributes	4	**0x0002**	Bit	Description
				31:2	**Reserved**. Should be zeros.
				1	**LPM**. If set, this device supports Link Power Management.
				0	**Reserved**. Should be zeros.

The third entry type is the Super-speed USB Device entry and has a Capability type of 0x03.

```
0x00000000                          0A 10 03 00
0x00000010  0F 00 03 04 E7 00
```

Table 21-8: BOS Descriptor: Super Speed USB Device

Offset	Field	Size	Value	Description	
0	Length	1	**0x0A**	Size of stub in bytes.	
1	Type	1	**0x10**	Device Capability	
2	Cap Type	1	**0x03**	Super-speed Device	
3	Attributes	1	**0x00**	Bit	Description
				7:2	**Reserved**. Should be zeros.
				1	**LTM**. If set, this device supports Latency Tolerance Messages.
				0	**Reserved**. Should be zeros.
4	Supported Speeds	2	**0x000F**	Bit	Description
				15:4	**Reserved**. Should be zeros.
				3	**Super Speed**. If set, device supports speeds to 5 GBps.
				2	**High Speed**. If set, device supports High-speed transfers.
				1	**Full Speed**. If set, device supports Full-speed transfers.
				0	**Low Speed**. If set, device supports Low-speed transfers.
6	Supported Functionality	1	**0x03**	The lowest speed at which this device functions with full	

				functionality. 3 = all functionality at super-speed
7	U1DevExitLat	1	**0x04**	U1 Device Exit Latency. Valid values are 0 to 10uS.
8	U2DevExitLat	2	**0x00E7**	U2 Device Exit Latency. Valid values: 0 to 2047uS.

The fourth and last entry type is the Container-ID entry and has a Capability type of 0x04.

```
0x00000010                14 10-04 00 00 00 00 00 00
0x00000020  00 00 00 00 00 00 00 00-00 00
```

Table 21-9: BOS Descriptor: Container-ID

Offset	Field	Size	Value	Description
0	Length	1	**0x14**	Size of stub in bytes.
1	Type	1	**0x10**	Device Capability
2	Cap Type	1	**0x04**	Container ID
3	Reserved	1	**0x00**	Reserved and zero.
4	Container ID	16	varies	UUID.

The UUID field is used as a Universal Unique Identification number and is explained in the EITF RFC 4122 document found on the web. This number is used to uniquely identify different devices on the bus, especially so that system software can identify the device when it is operating in different modes/speeds.

Please remember that these values and the order of the capability entries used here are for the example device I used. The values can and may be different for each device, whether it is a hub or not.

Please see in Chapter 22, the section about this BOS descriptor, for information you will need to send to the device pertaining to the information that you retrieved here.

The Configuration Descriptor

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor.

```
0x00000000  09 02 1F 00 01 01 00 C0-01 09 04 00 00 01 09 00
0x00000010  00 00 07 05 81 13 02 00-10 06 30 00 00 02 00
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 1F 00 01 01 00 C0-01
```

Table 21-10: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x001F**	Total length of all data within this configuration descriptor (31 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config. (0 = none)
7	Attributes	1	**0xC0**	**D7** Reserved (See Appendix D) **D6** Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x01**	Maximum Power Consumption. 0x01 = 2mA at high-speed. 0x01 = 8mA at super-speed.

Notice that the Self Powered bit is set. This particular hub actually does have an external power source. Though, to be sure if the device is Self-Powered, send the Get_Status request checking bit 0 of the returned data.

This Configuration Descriptor has a string descriptor index of 0x00 indicating that there is no string associated with this configuration.

This Configuration Descriptor indicates that we have only one Interface Descriptor.

```
0x00000000                  09 04 00 00 01 09 00
0x00000010  00 00
```

Table 21-11: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface	1	**0x00**	Number of Interface.

	Num			
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x09**	Class Code: 0x09 = HUB
6	Sub Class	1	**0x00**	Sub Class Code: 0x00
7	Protocol	1	**0x00**	Protocol Code: 0x00
8	Interface Str	1	**0x00**	String descriptor index.

This descriptor, like the configuration descriptor, also tells us what kind of device it is. Using the class code, we know that it is an external hub. The Interface Descriptor also indicates that there is one Endpoint Descriptor.

```
0x00000012        07 05 81 03 01 00-FF
```

Table 21-12: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0001**	Max packet size: 1 byte
6	Interval	1	**0xFF**	Poll every 255mS

This indicates that we need to poll this endpoint every 255mS for a 1-byte status.

Since the largest value to fit in a byte is 255mS, as in byte 6 above, it will not hurt to poll the endpoint every 256mS, since 256 is a multiple of 2.

The interrupt endpoint is used to detect a change in the hub and/or an individual port. The max packet size will depend on how many ports there are. The format of the report returned is the same no matter the number of ports, just that there will be a larger report returned. For example, our 4 port hub will need 5 bits to report the status change bitmap. Therefore, the endpoint's max packet size is 1 byte. The format of the report returned is as follows. Each bit, when set indicates that there has been a change to that item.

Table 21-13: Interrupt Endpoint Return Data

Bit	Description
7:5	Reserved
4	Port 4 change detected.
3	Port 3 change detected.
2	Port 2 change detected.
1	Port 1 change detected.
0	Hub status change detected.

If there are 8 or more ports, then at least another byte is needed. The report will be wide enough to have a bit for each port in existence, plus bit 0.

It is up to you to decide how you want to monitor changes in each port on the hub. You can continue to send Get_Status requests to each port, which takes time and schedule bandwidth, or you can simply create an Interrupt Endpoint packet in your schedule's periodic list and monitor the report.

The Super Speed Endpoint Companion

The USB 3.0 specification includes a small descriptor appended to each super-speed Endpoint Descriptor. This descriptor is called the Endpoint Companion Descriptor. The Default Control Endpoint does not include this companion descriptor but all endpoints defined as a super-speed endpoint will.

```
0x00000019                    06 30 00 00 02 00
```

Table 21-14: Endpoint Companion Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x06**	Size of descriptor in bytes.
1	Type	1	**0x30**	Endpoint Companion
2	Max Burst	1	**0x00**	The max amount of packets per burst. 0 = 1, 1 = 2.
3	Attributes	1	**0x00**	**Control Endpoints & Interrupt Endpoints:** Bits 7:0 Reserved and set to zero. **Bulk Endpoints**: Bits 7:5 Reserved and set to zero. Bits 4:0 Max Streams. **Isochronous Endpoints**: Bits 7:2 Reserved and set To zero. Bits 1:0 Mult.
4	Bytes Per Interval	2	**0x0002**	Max number of bytes transferred per interrupt service.

The Max Burst field is used in super-speed transactions and is given here as a zero-based number with zero indicating 1 burst, a 1 indicating 2 bursts, etc. The Max Stream field is also used in super-speed transactions, with a value of zero indicating that streams are not supported. Values 1 through 16 are supported and is shown below.

$$2^{MaxStreams} = 2^5 = 32$$

The Mult field specifies the total amount of packets allowed for a service interval for an Isochronous Endpoint. Valid values are zero through 2 and is calculated as:

$$Max\ Packets = Max_Burst * (Mult + 1)$$

Configuring the External Hub and a Catch-22

Now that we have retrieved most of the device descriptors, we now know that it is an external hub class device, having a single interrupt endpoint with an endpoint value of 1. If you have read the chapter on high-speed hubs, you might think at this point we would get the Hub Descriptor.

The USB 2.0 and USB 3.0 specifications state that the hub should be configured before you attempt to retrieve the Hub Descriptor. All USB 2.0 hubs that I have tried will successfully send the Hub Descriptor before the device is configured. However, the USB 3.0 hub I show here will not.

If you read further in the USB 3.0 specification, you will notice that to configure the hub you must be able to know how many downstream ports the hub has. However, to get the number of downstream ports, you must get the Hub Descriptor first, which you can not get until the hub is configured.

To overcome this issue, you can send a default value of 4 downstream ports with the Configure request. Then after you have the correct amount of downstream ports, you can send the Configure request again.

Set Depth Request

After you have configured the hub the first time, and before you get the Hub Descriptor, you must send the Set Depth request telling the hub the depth this device is in the topology. See Chapter 10 and Figure 10-5 for more on how to figure the hub depth. For now, we will assume that this hub is plugged directly into the root hub so we will send a value of zero.

Table 21-15: Set Hub Depth Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x20**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bits 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bits 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0C**	Set Hub Depth
2	Value	2	**0x0000**	Hub Depth
4	Index	2	**0x0000**	Zero or Language ID.
6	Length	2	**0x0000**	No data.

Now is when we can retrieve the hub descriptor. The USB 3.0 specification states that the hub descriptor is as follows, which is different from the USB 2.0 descriptor.

Table 21-16: Get USB 3.0 Hub Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bits 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bits 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x2A00**	Type and Index.
4	Index	2	**0x0000**	Zero or Language ID.
6	Length	2	**0x000C**	At most 12 bytes.

Please note that if you are reading this chapter and requesting this descriptor from a USB 2.0 device, the Type field will be 0x29 and the Length field will be 71.

This is because the USB 2.0 hub descriptor is different than the USB 3.0 descriptor. See the chapter on USB 2.0 hubs for more information.

The hub I have returns the following memory dump

```
0x00000000  0C 2A 04 09 00 00 02 00-00 00 00 00
```

and explained here.

Table 21-17: USB 3.0 Hub Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x2A**	USB 3.0 Hub Descriptor
2	NumPorts	1	**0x04**	Number of downstream ports.
3	HubChars	2	**0x0009**	Bits 15:5 Reserved Bits 4:3 Over-current PM 00: Global **01: Individual** 1x: none Bit 2 Compound device? **0: No** 1: Yes Bits 1:0 Power Switching 00: Global **01: Individual** 1x: none
5	POTPGT	1	**0x00**	Power on to power good time. 0x32 = 100mS. 0x00 = Power switching is not supported.
6	MaxHubCurrent	1	**0x02**	Maximum current used by the hubs controller.
7	Hub Packet Header Decode Latency	1	**0x00**	Values 0x00 - 0x04 are allowed.
8	Hub Delay	2	**0x00**	Hub Delay in nanoseconds in flowing header packets.
10	Removable	2	**0x00**	Bit 0 = reserved Bit 1 = 1st port = removable device attached? Bit 2 = 2nd port = removable device attached? etc.

If you notice, the POTPGT field is actually zero meaning that power switching is not supported, though bit 1 in the Hub Characteristics field is set. Not everyone is perfect.

Unlike the USB 2.0 Hub Descriptor, the USB 3.0 Hub Descriptor assumes that there will never be more than 15 downstream ports on a hub even though the specification allows up to 255. Therefore, the size of this descriptor will always be 12 unlike the 71 bytes that the USB 2.0 descriptor could be.

Detecting Number of Downstream Ports

With the hub descriptor above, we know how many downstream ports there are on this hub, in this case 4. The remaining fields are very similar to the USB 2.0 specifications. See Chapter 19 for more on these fields.

Configuring the External Hub Again

Now that we have all the information we need to configure the hub correctly, including the correct count of downstream ports, we will need to send the Set Configuration Request again.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the Set_Configuration request with a configuration number of 1, retrieved from the ConfigVal field shown on earlier in this chapter.

Table 21-18: Set Configuration Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x00**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x09**	SET CONFIGURATION
2	Value	2	**0x0001**	Config value from Configuration Descriptor.
4	Index	2	**0x0000**	Zero
6	Length	2	**0x0000**	No bytes transferred.

Resetting and Enabling the Hub

We now have a configured, ready to use USB external hub. You reset and enable a hub attached to the root hub just like any other device attached to the root hub.

The reset and enable of a hub, or any other device, attached to another external hub, is a little different. This information is already discussed in Chapter 19, so I will not discuss it here.

Other Comments

That is all for the hub configuration. Since you have reset the external hub's port that a device is attached to, that device will be in the default state and will answer to the default address of zero. The hub will now do all of the work for us. You can send the enumeration process out on the bus as if the hub was not there. In other words, enumerate the newly attached device as if it was attached to the root hub.

Be sure that you do not reset and enable more than one port on the hub with out first setting the address of each newly attached device. If you don't set the address of the newly attached devices first, each device will be in the default state and answer to the default address of zero.

It will be up to your driver to monitor the status of the hub, to find when another device is attached or an existing is removed. Your driver will also have to keep track of the hierarchy of the hub tree. If the user unplugs a hub, your driver will need to remove all references to any devices that may have been plugged in to that hub, in turn, causing a disconnect for all of those devices.

Technically, you may have as many hubs plugged in to each other as you want as long as you don't violate the cable length before a powered hub, and have 127 or fewer devices attached to the root hub.

See Chapter 19 for information on Transaction Translators, which USB 3.0 devices may use.

Chapter 22 – Using a USB 3.0 Mass Storage device

In this chapter, I will explain how to configure and program a USB 3.0 Super Speed Mass Storage Device (MSD) in the form of a thumb drive, jump drive, flash drive, and many other names of this type of device including external USB hard drives.

This chapter will be very similar to Chapter 17. However, in this chapter, I am using a USB 3.0 super-speed MSD device that will have different values for the descriptors and will have new descriptors that the information in Chapter 17 doesn't account for. If you are using a full- or high-speed MSD device, you will need to study Chapter 17 instead.

The USB-IF has created a form of protocol to use for the USB MSD drive. It is called the USB Mass Storage Class, Bulk-Only Transport Specification. This specification is found on the web as usbmassbulk_10.pdf. See Appendix A for more information.

From this point on, within this chapter, I refer to these types of devices as simply drives. The drive I have used for the research of this chapter is a Corsair™ 8Gig drive, model CMFVY35-8GB. All of the example descriptors and information in this chapter are for this unit. However, the information and code within this chapter should work on any similar drive unit.

As mentioned before, you should be familiar with how to retrieve the device's descriptors via the enumeration process and send and receive data packets.

Please note that in this chapter I assume the MSD device is a super-speed device. If you have a full- or high-speed MSD device, please see Chapter 17.

The MSD Drive's Device Descriptors

During the enumeration process you should have been able to retrieve the drive's first descriptor, the Device Descriptor, and it should be similar to the following memory dump

```
0x00000000  12 01 00 03 00 00 00 09-1C 1B 03 1A 00 0A 01 02
0x00000010  03 01
```

and explained next.

Table 22-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0300**	USB Specification Release Number in Binary-Coded Decimal. 0x0300 = v3.00
4	Device Class	1	**0x00**	A class code of 0x00 indicates to use the Class Code in the Interface Descriptor(s).
5	Sub Class	1	**0x00**	See Device Class above.
6	Protocol	1	**0x00**	See Device Class above.
7	MaxPacketSize	1	**0x09**	Max packet size for the control endpoint[1].
8	Vendor ID	2	**0x1B1C**	Vendor ID[2]: Corsair.
10	Product ID	2	**0x1A03**	Product ID: Voyager GT
12	Device Rel	2	**0x0A00**	Release version 1.10
14	Manufacturer	1	**0x01**	Index to string descriptor
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x03**	Index of Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] See below why this is 0x09 instead of 512

[2] http://www.linux-usb.org/usb-ids.html

Now that USB 3.0 allows max packet sizes of 512, this value no longer fits in the byte field above. Therefore, any USB 3.0+ device descriptor, indicated by the Release Num field, will have a value used as the exponent of two. For example, the value above is 9, giving a max packet size of 512 for endpoint zero.

$$2^{\text{MaxPacketSize}} = 2^9 = 512$$

Currently, only the value of 9, a max packet size of 512, is defined for control endpoints. Any other value is considered illegal.

The String Descriptors

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump

```
0x00000000  04 03 09 04
```

and explained here.

Table 22-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This drive returns a string index for the Manufacturer ID, listed here, then the Product ID, listed next, and Serial Number, listed last, with the following values.

```
0x00000000  10 03 43 00 6F 00 72 00-73 00 61 00 69 00 72 00
```

Table 22-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x10**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x10	**0x0043** **0x006F** **0x0072** **0x0073** **0x0061** **0x0069** **0x0072**	C o r s a i r

Product ID:

```
0x00000000  18 03 56 00 6F 00 79 00-61 00 67 00 65 00 72 00
0x00000010  20 00 33 00 2E 00 30 00
```

Table 22-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x18**	Size of this descriptor.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x1C	**0x0056** **0x006F** **0x0079** **0x0061** **0x0067** **0x0065**	V o y a g e

			0x0072 **0x0020** **0x0033** **0x002E** **0x0030**	r <space> 3 . 0

Serial Number:

```
0x00000000  36 03 30 00 39 00 30 00-32 00 30 00 30 00 30 00
0x00000010  30 00 30 00 30 00 31 00-31 00 38 00 31 00 37 00
0x00000020  30 00 30 00 30 00 30 00-30 00 30 00 30 00 30 00
0x00000030  33 00 37 00 37 00
```

Table 22-5: String Descriptor 3

Offset	Field	Size	Value	Description
0	Length	1	**0x36**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0030** **0x0039** **0x0030** **0x0032** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0031** **0x0031** **0x0038** **0x0031** **0x0037** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0030** **0x0033** **0x0037** **0x0037**	0 9 0 2 0 0 0 0 0 0 1 1 8 1 7 0 0 0 0 0 0 0 0 3 7 7

The BOS Descriptor

Here is where the USB 3.0 super-speed device differs. We now need to get the Binary Device Object Store (BOS) Descriptor. This descriptor is used to add device level capabilities to the standard descriptors. This descriptor is accessed very similar to the configuration descriptor. It has a stub that contains the total length of all of the included descriptors within the BOS descriptor. You get the descriptor just like the device descriptor, however sending 0x0F instead of 0x01.

Here is a memory dump of the whole descriptor for this particular device.

```
0x00000000  05 0F 16 00 02 07 10 02-02 00 00 00 0A 10 03 00
0x00000010  0E 00 01 0A 20 00
```

Please note that each super-speed device may have a different BOS descriptor with the "sub-descriptors" in different order as the one used here.

The stub of this descriptor has the following data and format.

```
0x00000000  05 0F 16 00 02
```

Table 22-6: BOS Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x05**	Size of stub in bytes.
1	Type	1	**0x0F**	BOS Descriptor
2	TotalLength	2	**0x0016**	Total length of all data within this BOS descriptor
4	NumCapEntries	1	**0x02**	Number of Capability entries following this stub.

The BOS descriptor may have up to four Capability entries. See Chapter 21 for more on these entries.

The first BOS entry this device returned is the USB 2.0 Extension entry and has a Capability type of 0x02.

```
0x00000000              07 10 02-02 00 00 00
```

Table 22-7: BOS Descriptor: USB 2.0 Extension

Offset	Field	Size	Value	Description	
0	Length	1	**0x07**	Size of stub in bytes.	
1	Type	1	**0x10**	Device Capability	
2	Cap Type	1	**0x02**	USB 2.0 Extension	
3	Attributes	4	**0x0002**	Bit	Description
				31:2	**Reserved**. Should be zeros.
				1	**LPM**. If set, this device supports Link Power Management.
				0	**Reserved**. Should be zeros.

The second entry is the super-speed USB Device entry.

```
0x00000000                    0A 10 03 00
```

0x00000010 0E 00 01 0A 20 00

Table 22-8: BOS Descriptor: Super Speed USB Device

Offset	Field	Size	Value	Description	
0	Length	1	**0x0A**	Size of stub in bytes.	
1	Type	1	**0x10**	Device Capability	
2	Cap Type	1	**0x03**	Super-speed Device	
3	Attributes	1	**0x00**	Bit	Description
				7:2	**Reserved**. Should be zeros.
				1	**LTM**. If set, this device supports Latency Tolerance Messages.
				0	**Reserved**. Should be zeros.
4	Supported Speeds	2	**0x000E**	Bit	Description
				15:4	**Reserved**. Should be zeros.
				3	**Super Speed**. If set, device supports speeds to 5 GBps.
				2	**High Speed**. If set, device supports High-speed transfers.
				1	**Full Speed**. If set, device supports Full-speed transfers.
				0	**Low Speed**. If set, device supports Low-speed transfers.
6	Supported Functionality	1	**0x01**	The lowest speed at which this device functions with full functionality. A value of 0x01 = all functionality at full-speed and up.	
7	U1DevExitLat	1	**0x0A**	U1 Device Exit Latency. Valid values are 0 to 10uS.	
8	U2DevExitLat	2	**0x0020**	U2 Device Exit Latency. Valid values are 0 to 2047uS.	

The last two fields in Table 22-8 are used for power management. You need to send the Set_Sel request to the device, setting both the U1 and U2 System Exit Latency, and the U1 or U2 exit latency for all the links between this device and a root port on the host. Therefore, once you have retrieved the information in Table 22-8, send the following request.

Table 22-9: Set Select Request Packet

Offset	Field	Size	Value	Description
0	Request type	1	**0x00**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type

				0 = Standard 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x30**	Set Select Request
2	Value	2	**0x0000**	Not used
4	Index	2	**0x0000**	Not used
6	Length	2	**0x0006**	Will send 6 bytes next

Then send the following six bytes as a Control Out packet, followed by the required Control Status packet.

Table 22-10: Set Select Data Out

Offset	Field	Size	Value	Description
0	U1Sel	1	**0x0A**	Time in uS for U1 System Exit Latency
1	U1Pel	1	**0x0A**	Time in uS for U1 Device to Host Exit Latency
2	U2Sel	2	**0x0020**	Time in uS for U2 System Exit Latency
4	U2Pel	2	**0x0020**	Time in uS for U2 Device to Host Exit Latency

If you will notice, the values you sent in Table 22-9, are the values you retrieved from the BOS Descriptor from Table 22-8.

The values you send in Table 22-9, assume no external hubs are used.

The Configuration Descriptor

To this point, we might now know what the device is and have a human readable string to associate it with. However, as noted before, the class code in the Device Descriptor was zero, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor.

```
0x00000000  09 02 2C 00 01 01 00 80-3F 09 04 00 00 02 08 06
0x00000010  50 00 07 05 02 02 00 04-00 06 30 0F 00 00 00 07
0x00000020  05 81 02 00 04 00 06 30-0F 00 00 00
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration. Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 2C 00 01 01 00 80-3F
```

Table 22-11: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x002C**	Total length of all data within this configuration descriptor (44 bytes)
4	NumInterfaces	1	**0x01**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.
7	Attributes	1	**0x80**	**D7** Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x3F**	Maximum Power Consumption. 0x70 = 126mA

This Configuration Descriptor indicates that we have only one Interface Descriptor, which follows with these 9 bytes.

```
0x00000000                  09 04 00 00 02 08 06
0x00000010  50 00
```

Table 22-12: Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x02**	Number of Endpoints: 2

5	Class Code	1	**0x08**	Class Code: 0x08 = Mass Storage
6	Sub Class	1	**0x06**	Sub Class Code: 0x06 = SCSI transparent command set[1].
7	Protocol	1	**0x50**	Protocol Code: Bulk-Only[1].
8	Interface Str	1	**0x00**	No string descriptor.

[1]See page 6 and 7 of the USB Mass Storage Class Specification Overview.

This descriptor now tells us what kind of device it is. Using the class code, we know that it is a Mass Storage Device (MSD) and the subclass code indicates that it uses the SCSI transparent command set, while the protocol tells us that it uses the Bulk-Only Command Specification for transferring data. The Protocol field is 0x50, which it is Bulk-Only transport. The USB Mass Storage Class Specification Overview file can be found on the web as usb_msc_overview_1.2.pdf. See Appendix A more information.

This descriptor also indicates that there are two Endpoint Descriptors that follow.

```
0x00000010        07 05 02 02 00 04-00
```

Table 22-13: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x02**	Endpoint address: Bits 7 0 = **Out**, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = **2**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0400**	Max packet size: 1024 bytes
6	Interval	1	**0x00**	No Polling needed.

The USB 3.0 specification includes a small descriptor appended to each super-speed Endpoint Descriptor. This descriptor is called the Endpoint Companion Descriptor. The Default Control Endpoint does not include this companion descriptor but all endpoints defined as super-speed endpoints will.

```
0x00000010                  06 30 0F 00 00 00
```

Table 22-14: Endpoint Companion Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x06**	Size of descriptor in bytes.
1	Type	1	**0x30**	Endpoint Companion
2	Max Burst	1	**0x0F**	The max amount of packets per burst. 0x0F = 16.
3	Attributes	1	**0x00**	**Bulk Endpoints**: Bits 7:5 Reserved.

				Bits 4:0 Max Streams.
4	Bytes Per Interval	2	**0x0000**	Reserved.

The Max Burst field is used in super-speed transactions and is given here as a zero-based number with zero indicating 1 burst, a 1 indicating 2 bursts, etc. The Max Stream field is also used in super-speed transactions, with a value of zero indicating that streams are not supported. Values 1 through 16 are supported and calculated as:

$$2^{MaxStreams} = 2^5 = 32$$

The Mult field specifies the total amount of packets allowed for a service interval for an Isochronous Endpoint. Valid values are zero through 2 and is calculated as:

$$Max\ Packets = Max_Burst * (Mult + 1)$$

The second endpoint has the following data.

0x00000010 07 05 81 02 00 04 00

Table 22-15: Endpoint Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x02**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous **10** = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x0400**	Max packet size: 1024 bytes
6	Interval	1	**0x00**	No Polling needed.

The second endpoint's Companion Descriptor has the following data.

0x00000010 06 30 0F 00 00 00

And outlined here.

Table 22-16: Endpoint Companion Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x06**	Size of descriptor in bytes.
1	Type	1	**0x30**	Endpoint Companion
2	Max Burst	1	**0x0F**	The max amount of packets per burst. 0x0F = 16.
3	Attributes	1	**0x00**	**Bulk Endpoints**: Bits 7:5 Reserved and set to zero.

				Bits 4:0 Max Streams.
4	Bytes Per Interval	2	**0x0000**	Reserved.

The first endpoint is the OUT endpoint and has an endpoint value of 2. This is the endpoint that you write data to the device at a maximum of 1,024 bytes at a time.

The second endpoint is the IN endpoint and is identical to the first except that you read data from the device using this endpoint instead of writing data. This endpoint descriptor has an endpoint value of 1.

Configure the MSD Drive

Now that we have retrieved all of the device descriptors, we now know that it is a MSD using the Bulk-Only protocol, having an OUT endpoint with an endpoint value of 2, an IN endpoint with an endpoint value of 1.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the “Set Configuration” request with a configuration number of 1, retrieved from the ConfigVal field before.

When we request data from the drive, we send and receive this data within a Command Block Wrapper (CBW) and a Command Status Wrapper (CSW). For more information and a description on how to use this type of transport, see the sections in Chapter 17. Reading and writing sectors is identical to that of a high-speed MSD device, explained in Chapter 17, except that we can now read and write more bytes with a single packet.

Other Comments

A few things to remember:

- A super-speed device will not like the Device Qualifier or Other Speed requests. Your code should check to make sure that the device is at least version 2.10 and below version 3.00 before sending these two requests.

- Only super-speed devices will have an Endpoint Companion Descriptor following each regular Endpoint Descriptor.

- Only devices with a version at least 2.10 will support the BOS Descriptor.

- Since USB Specification revision 3.10 has now been released, be sure to check only the high part of the version number when checking for USB 3.0 compatibility. If you check for 0x0300, and are using a 3.10 device, your code will return FALSE.

Part 4

In Part 4, I will show how to use a USB hardware analyzer, how to parse the HID (Human Interface Device) report descriptor, along with other USB related items that aren't specific to any single item.

Chapter 23 – Using a USB Protocol Analyzer

In this chapter, I will explain what is and how to use a USB protocol analyzer, where to get one, and the advantage it has when used for hardware and software development of USB devices and controllers.

A USB protocol analyzer is an in-line device that captures the activity of the device attached without interrupting the bus activity. To use the analyzer, you will need a test machine with a USB controller with an available port, and the desired test device. You will also need a host machine, with a USB controller with an available port. The host machine's controller and speed is independent of the test machine and device.

The Beagle from TotalPhase

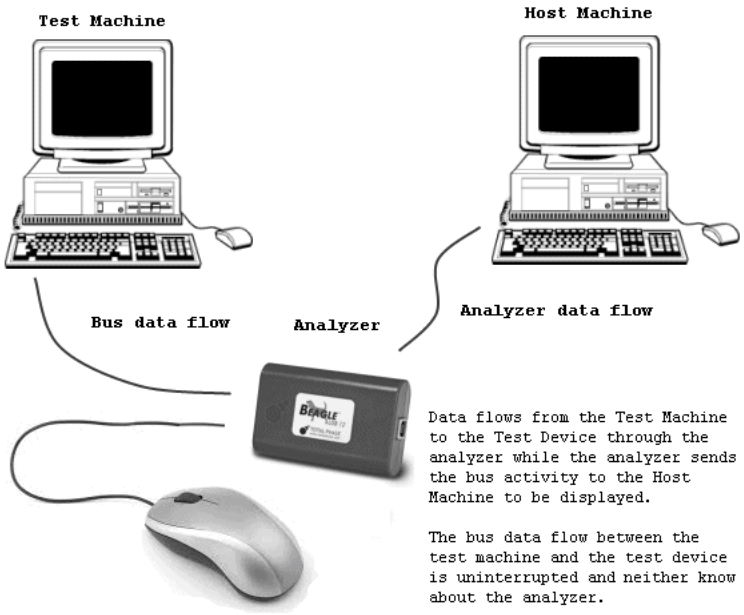
I use the Beagle USB protocol Analyzer from *TotalPhase* available on the web at http://www.totalphase.com/products/beagle_usb12/. It is a very good analyzer and comes with documentation, drivers, and software to analyze the bus traffic. This particular one is for low- and full-speed traffic, though they have one for high- and super-speed traffic as well. See later in this chapter for a few examples it will produce.

Setting up the hardware

Once you have installed the analyzer's driver(s) and software on the host machine, plug the desired USB device into the device port on the analyzer, then using a standard USB cable, connect the analyzer's test side port into an available port on the controller of the test machine. At this point, the attached test device will work as if the analyzer was a coupler device connecting two USB cables together. The test device and computer will not know the analyzer is attached. Plug another USB cable into the protocol side of the analyzer and into an available port on the host machine and start the analyzer's monitor software.

Below is a diagram showing a test machine, a mouse as a test device, the analyzer, and the host machine showing the bus traffic.

Figure 23-1: Analyzer Hook-up



The test machine and test device, have no knowledge of the analyzer. As far as they are concerned, it is just a coupler connecting two USB cables together. The analyzer sends all bus activity to the host machine, while the analyzer software displays the data.

Receiving bus traffic

Once you have plugged everything in as described by the analyzer's documentation, loaded the analyzer's software, and started the software to collect bus activity, then turn on the test machine. All bus activity between the test machine's controller and the test device will be logged on the host machine analyzer software.

This technique will log all activity including any activity the BIOS sends before the operating system receives control. This amount of data may be too much for you to analyze. There are other things that you can do.

For example, clear the log on the host machine. Unplug the test device from the analyzer, start the host machine's analyzer software to collect bus activity, then plug the test device into the analyzer. This will only capture the reset and device enumeration. Once this is complete, you can stop the analyzer software from capturing bus traffic.

There are many ways you can use this analyzer to capture bus traffic. Place the analyzer between the test machine and an external hub to see all downstream traffic. This would include any traffic to the hub itself, along with any devices attached to the hub.

This allows you to see what traffic your test software is sending to the device, whether it is getting there, and if it is correctly acknowledged or lost.

Reading and Understanding the Report

Once you have captured bus traffic, you will need to know what to look for. For the following example, I have used the *Beagle* from *TotalPhase* described before with a USB three-button mouse attached as the test device. The analyzer is plugged directly into one of the root hub ports of the test machine, and the mouse was plugged in after both the test machine and host machine was booted and ready for device attachment.

The image below closely resembles what is displayed on the host machine as the mouse is connected to the analyzer, which in turn connects to the test machine. From here on out, unless otherwise stated, the term host means the test machine's operating system and USB drivers.

Figure 23-2: Analyzer Listing 1

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
3	0:05.344.158					Bus event	<Reset> / <Target disconnected>
4	0:05.370.410					Bus event	<Unreset> / <Target connected>
5	0:05.414.816	8 B		00	00	Get Device Descriptor	Index=0 Length=64
9	0:05.415.815	8 B		00	00	IN txn	12 01 10 01 00 00 00 08
13	0:05.416.814	8 B		00	00	IN txn	FC 04 03 00 01 01 00 02
17	0:05.417.815	2 B		00	00	IN txn	00 01
21	0:05.419.814	0 B		00	00	OUT txn	
25	0:05.373.814	40.0 ms		01	01	[6 ORPHANED]	
27	0:05.421.051					Bus event	<Reset> / <Target disconnected>
28	0:05.446.592					Bus event	<Unreset> / <Target connected>
29	0:05.473.813	8 B		00	00	Set Address	Address=02
33	0:05.474.811	0 B		00	00	IN txn	
37	0:05.523.810	8 B		02	00	Get Device Descriptor	Index=0 Length=18
41	0:05.524.809	8 B		02	00	IN txn	12 01 10 01 00 00 00 08
45	0:05.525.852	8 B		02	00	IN txn	FC 04 03 00 01 01 00 02
49	0:05.526.808	2 B		02	00	IN txn	00 01
53	0:05.527.809	0 B		02	00	OUT txn	
57	0:05.529.810	8 B		02	00	Get Configuration Descriptor	Index=0 Length=9
61	0:05.530.808	8 B		02	00	IN txn	09 02 22 00 01 01 00 A0
65	0:05.531.809	1 B		02	00	IN txn	32
69	0:05.532.809	0 B		02	00	OUT txn	
73	0:05.534.810	8 B		02	00	Get Configuration Descriptor	Index=0 Length=255
77	0:05.535.809	8 B		02	00	IN txn	09 02 22 00 01 01 00 A0
81	0:05.536.808	8 B		02	00	IN txn	32 09 04 00 00 01 03 01
85	0:05.537.809	8 B		02	00	IN txn	02 00 09 21 00 01 00 01
89	0:05.538.808	8 B		02	00	IN txn	22 4A 00 07 05 81 03 04
93	0:05.539.808	2 B		02	00	IN txn	00 0A
97	0:05.541.851	0 B		02	00	OUT txn	
101	0:05.543.809	8 B		02	00	Get String Descriptor	Index=0 Length=255
105	0:05.544.808	4 B		02	00	IN txn	04 03 09 04

Line (index) 3 and 4 is where the host recognizes the connection and does the reset. Starting with line 5 through line 21 is the Get_Device_Descriptor request. Notice that the host requested 64 bytes, and since the mouse is a low-speed device, received 8-byte packets.

Line 5 is the setup packet. In the analyzer software, if you click on the plus sign (point A above), the software will display the setup packet's contents.

80 06 00 01 00 00 40 00

Line 9 shows the first 8 bytes of the device's descriptor, Line 13 is the second 8 bytes, and line 17 is the last 2 bytes of the 18-byte descriptor. Line 21 is the status packet.

Then the host operating system does a reset at line 27 and 28. You may ignore the ORPHANED line 25. This shows in each capture just before the reset.

At line 29, the host sends the Set_Address request with an address of 2, then receives the status packet. If you will notice, the address has always been zero until this point. From here on, the address is 2, as indicated in the DEV column.

Now the host retrieves the Device Descriptor again, all 18 bytes. Notice that the length requested is now 18 instead of 64 as before. Since the host retrieved the device descriptor the first time, it now knows that it is 18 bytes.

Next the host retrieves the first 9 bytes of the Configuration Descriptor. The first 9 bytes, since the first part of the Configuration Descriptor is always 9 bytes. With these 9 bytes, it now can figure how long the total descriptor is and at line 73 through 97, it requests and receives the whole 34-byte descriptor. However, notice that the requested length is 255 (Line 73: Length = 255). This is the advantage of using a protocol analyzer. You can check the actual output that your driver creates. Without it, would you ever know that you forgot to change the request length field from 255 to 34?

The report continues with the figure on the next page.

At line 101, the host retrieves the zero indexed string descriptor. At line 149, it requests the string descriptor with an index of 2.

At line 173, the host repeats some of the previous requests, then at line 237, it sends the Set_Configuration request.

At line 245 we can see where the analyzer shows its benefits. The Host sends the Set_Idle request. Since the Set_Idle request is an optional HID request, the device is not required to accept it. Therefore, at line 249, at the IN token, the device returns a STALL. The analyzer shows the STALL.

Next the host requests the HID report descriptor and the analyzer shows that the device returned 74 bytes.

The benefits of having a protocol analyzer as the *Beagle* from *TotalPhase* are fantastic. This analyzer has helped in my USB development more than any other resource. To be able to plug in devices and see what actually happens across the wire is amazing and helpful. If you are serious about USB development, the moderate price of the analyzer is well worth it.

When you inquire about the *Beagle*, mention that you read about it in this book. They will be pleased to hear from you.

Figure 23-3: Analyzer Listing 2

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
93	0:05.539.808	2 B		02	00	+ IN txn	00 0A
97	0:05.541.851	0 B		02	00	+ OUT txn	
101	0:05.543.809	8 B		02	00	+ Get String Descriptor	Index=0 Length=255
105	0:05.544.808	4 B		02	00	+ IN txn	04 03 09 04
109	0:05.546.808	0 B		02	00	+ OUT txn	
113	0:05.548.809	8 B		02	00	+ Get String Descriptor	Index=2 Length=255
117	0:05.549.850	8 B		02	00	+ IN txn	18 03 57 00 68 00 65 00
121	0:05.550.807	8 B		02	00	+ IN txn	65 00 6C 00 20 00 4D 00
125	0:05.551.806	8 B		02	00	+ IN txn	6F 00 75 00 73 00 65 00
129	0:05.552.808	0 B		02	00	+ IN txn	
133	0:05.554.809	0 B		02	00	+ OUT txn	
137	0:05.556.808	8 B		02	00	+ Get String Descriptor	Index=0 Length=255
141	0:05.557.850	4 B		02	00	+ IN txn	04 03 09 04
145	0:05.559.807	0 B		02	00	+ OUT txn	
149	0:05.561.807	8 B		02	00	+ Get String Descriptor	Index=2 Length=255
153	0:05.562.807	8 B		02	00	+ IN txn	18 03 57 00 68 00 65 00
157	0:05.563.808	8 B		02	00	+ IN txn	65 00 6C 00 20 00 4D 00
161	0:05.564.808	8 B		02	00	+ IN txn	6F 00 75 00 73 00 65 00
165	0:05.565.849	0 B		02	00	+ IN txn	
169	0:05.567.807	0 B		02	00	+ OUT txn	
233	0:05.613.847	0 B		02	00	+ OUT txn	
237	0:05.615.804	8 B		02	00	+ Set Configuration	Configuration=1
241	0:05.616.805	0 B		02	00	+ IN txn	
245	0:05.635.802	8 B		02	00	+ SETUP txn	
249	0:05.636.804			02	00	+ IN txn (STALL)	
252	0:05.638.804	8 B		02	00	+ SETUP txn	
256	0:05.639.804	8 B		02	00	+ IN txn	05 01 09 02 A1 01 09 01
260	0:05.640.803	8 B		02	00	+ IN txn	A1 00 05 09 19 01 29 03
264	0:05.641.803	8 B		02	00	+ IN txn	15 00 25 01 75 01 95 03
268	0:05.642.804	8 B		02	00	+ IN txn	81 02 75 05 95 01 81 01
272	0:05.643.803	8 B		02	00	+ IN txn	05 01 09 30 09 31 09 38
276	0:05.644.802	8 B		02	00	+ IN txn	15 81 25 7F 75 08 95 03
280	0:05.645.846	8 B		02	00	+ IN txn	81 06 C0 09 3C 05 FF 09
284	0:05.646.803	8 B		02	00	+ IN txn	01 15 00 25 01 75 01 95
288	0:05.647.802	8 B		02	00	+ IN txn	02 B1 22 75 06 95 01 B1
292	0:05.648.803	2 B		02	00	+ IN txn	01 C0
296	0:05.650.802	0 B		02	00	+ OUT txn	

Chapter 24 – The Human Interface Device

In this chapter, I will explain how to parse the HID Report Descriptor to find the format of a returned report from a Human Interface Device. This chapter assumes that you already know how to retrieve this descriptor and then know what the resulting format is used for.

A Brief Introduction to Human Interface Devices

The Human Interface Device class, or HID class, specifies a device as a device that is used to convey information to the host via some form of human interaction. For example, the two most used HID devices are keyboards and mice. For the keyboard, you use your fingers to press keys, and for the mouse you use your hand for motion, and fingers for button presses.

If these two devices were the only HID class type devices, there would not be a need for a HID specification. The USB-IF would have just created two smaller specifications, one for a keyboard and one for a mouse.

However, there are many more devices that are used for human input. How about the gaming industry? If you have a data glove, how do you tell the host that your index finger moved toward your palm? How about a steering wheel?

The USB-IF didn't want to make a specification for every device known, then another specification when a new device became available. Also, not every steering wheel would be the same. One game might need a different input parameter, say a horn for example.

This is where the HID class specification comes into play. Your device describes to the computer what the format of the incoming data packet will be. How many buttons and how many directional axis and their limits, and many other items.

The HID class specification also has information on how to send data to the HID devices. For example, a hand held game controller may need to have movement, such as vibration, to interact with the user. The HID class specification allows you to find out the format of the data to send to the device to turn on the vibration and the duration of that vibration.

In this chapter, I will briefly explain how to parse the HID class device's report descriptor for a three-button scroll mouse. The HID class specification is available at the USB-IF website as HID1_11.pdf. Also available is the Hut1_12.pdf file. It explains the Usage Tables used in this chapter.

I will not go into much more detail than needed to parse the three-button scroll mouse used in Chapter 15 of this book.

The HID Descriptor

The HID descriptor is returned via the Get_Descriptor request.

Table 24-1: Get Report Descriptor Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x81**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x2200**	High byte: 0x22 = REPORT Low byte: 0x00 = index 0
4	Index	2	xx	Interface number (0x00)
6	Length	2	xx	Descriptor length retrieve from HID Descriptor.

This will return the report, but not as you would think. Any other Get_Descriptor request returns a set format of information. This descriptor request returns tokens that you must parse to extract the information needed. Only the tokens have a set format. The information returned does not.

The fields in the report describe the size and format of the resulting report. The example below is given by the HID specification.

```
Report Size (3)
Report Count (2)
Input
Report Size (8)
Input
Output
```

Once an item has been initialized, such as a Report Count, it remains that value until changed. Let's start with the above example. It first initializes the Size field to 3 bits. From now on, until another Report Size field is found, any item is three bits in length.

Next is the Report Count field and it specifies a count of 2. As soon as the Input fields are reached, we now know that there will be 2 fields of 3 bits each for the input report buffer.

Now another Report Size field is reached, specifying a size of 8 bits. Next another Input field is found. Since the Count is still 2, the input report buffer gets 2 fields of 8 bits added to it.

Finally, the Output field is reached. Since the Count is still 2 and the size is still 8, the output report buffer gets 2 counts of 8 bits each.

We now know the size of each field in the input and output reports.

Figure 24-1: Input/Output Reports

Input			
3 bits	3 bits	8 bits	8 bits

Output	
8 bits	8 bits

There are other tokens in the HID Report Descriptor that indicates what type of item the above fields are used for.

How to parse the HID descriptor

The Report Descriptor contains two different types of item formats, short items and long items. If an item can be represented in 4 bytes or less, it is considered a short item, else it is a long item. Each item, short or long, contains a Tag, Type, and Size field in the first byte.

Table 24-2: Items 1-byte Prefix

7	6	5	4	3	2	1	0
Tag				Type		Size	

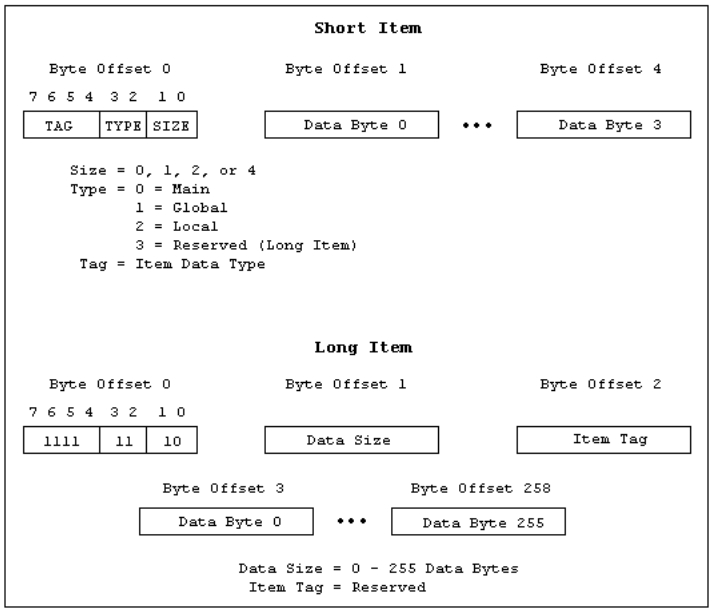
The size field indicates how many more bytes make up this short item. The type field indicates if it is a Main (0), Global (1), Local (2), or Reserved (3) item, while the Tag field indicates the function of the item. Then a count of Size bytes follow the item.

If the Type field is Reserved, this indicates that it is a long item. A long item will have the first byte's value as 0xFE, with a Size field of two more bytes to declare the long item Type and long item Size fields. Then the next byte is the actual Size field, while the third byte is the actual Type field. See the figure on the next page for the format of the short and long item layout.

As of this writing, the long format does not support a value in the LongItemTag field, though a value of 0xF0 to 0xFF is defined as vendor defined. Your driver may ignore this field if it has a value less than 0xF0.

A Short item may have 0, 1, 2, or 4 bytes after the first byte, while a long item will have 2 more for the long item declaration and then up to 255 data bytes following that. Please note that for a Short item, the size field may or may not be correct. The actual data, defined by the Tag field, determines the size of the item. Do not rely on the size field to be accurate for a Short item.

Figure 24-2: Short/Long Items


Short Item
Byte Offset 0
Byte Offset 1
Byte Offset 4
7 6 5 4 3 2 1 0
TAG
TYPE
SIZE
Data Byte 0
•••
Data Byte 3
Size = 0, 1, 2, or 4
Type = 0 = Main
1 = Global
2 = Local
3 = Reserved (Long Item)
Tag = Item Data Type
Long Item
Byte Offset 0
Byte Offset 1
Byte Offset 2
7 6 5 4 3 2 1 0
1111
11
10
Data Size
Item Tag
Byte Offset 3
Byte Offset 258
Data Byte 0
•••
Data Byte 255
Data Size = 0 - 255 Data Bytes
Item Tag = Reserved

The Included HID Parser Code

Again, I don't intend to go into a lot of detail about how to parse the HID Report Descriptor. There is enough information needed to learn how to do this for another book. This chapter is simply an overview of what you will need to know, and then includes the all-important code to parse the descriptor. Therefore, let's get right to the code and its output.

The mouse used in Chapter 15 returns the following report.

```
0x00000000  05 01 09 02 A1 01 09 01-A1 00 05 09 19 01 29 03
0x00000010  15 00 25 01 75 01 95 03-81 02 75 05 95 01 81 01
0x00000020  05 01 09 30 09 31 09 38-15 81 25 7F 75 08 95 03
0x00000030  81 06 C0 09 3C 05 FF 09-01 15 00 25 01 75 01 95
0x00000040  02 B1 22 75 06 95 01 B1-01 C0
```

Using the utility included on the disc, HIDParse, we get the following output.

```
HIDParser -- Parse a HID Report Descriptor  v1.00.00
Forever Young Software  Copyright 1984-2014

05 01    Usage Page (Generic Desktop)
09 02    Usage (Mouse)
A1 01    Collection (Application)
09 01      Usage (Pointer)
A1 00      Collection (Physical)
05 09        Usage Page (Button)
19 01        Usage min (1)
29 03        Usage max (3)
15 00        Logical Min (0)
25 01        Logical Max (1)
75 01        report size (1)
95 03        report count (3)
81 02        Input (Data,Var,Abs,NoW,Lin,PState,NoNull,Bit)
75 05        report size (5)
95 01        report count (1)
81 01        Input (Cons,Array,Abs,NoW,Lin,PState,NoNull,Bit)
05 01        Usage Page (Generic Desktop)
09 30        Usage (X)
09 31        Usage (Y)
09 38        Usage (Wheel)
15 81        Logical Min (-127)
25 7F        Logical Max (127)
75 08        report size (8)
95 03        report count (3)
81 06        Input (Data,Var,Rel,NoW,Lin,PState,NoNull,Bit)
C0         End Collection
09 3C      Usage (Motion Wakeup)
05 FF      Usage Page (Reserved)
09 01      Usage (Unsupported Usage Page)
15 00      Logical Min (0)
25 01      Logical Max (1)
75 01      report size (1)
95 02      report count (2)
B1 22      Feature (Data,Var,Abs,NoW,Lin,NoPre,NoNull,Bit)
75 06      report size (6)
95 01      report count (1)
```

```
B1 01      Feature (Cons,Array,Abs,NoW,Lin,PState,NoNull,Bit)
C0         End Collection
```

Let's look at each individual line.

```
05 01      Usage Page (Generic Desktop)
```
A Usage Item needs 32-bits to indicate which Usage to use. To save space within the Report Descriptor, if a Usage Page is specified, each Usage afterwards can now be 16-bits with the Usage Page as the high 16-bits.
```
09 02      Usage (Mouse)
```
These two combined indicate that the Usage is a Mouse. Usage Page = 0x01, Usage = 0x02. The USB-IF Usage Table specification says that this Usage is defined as:

> *A hand-held, button-activated input device that when rolled along a flat surface, directs an indicator to move correspondingly about a computer screen, allowing the operator to move the indicator freely in select operations or to manipulate text or graphics. A mouse typically consists of two axes (X and Y) and one, two, or three buttons.*

```
A1 01      Collection (Application)
```
Your application collection starts here.
```
09 01      Usage (Pointer)
```
Telling you that the following will be describing a pointing device.
```
A1 00      Collection (Physical)
```
The physical collection starts here. i.e.: this is where you start building the buffer format. Start with an empty, zero-sized buffer and build from there.
```
05 09      Usage Page (Button)
```
Telling you that the next item(s) will be buttons.
```
19 01      Usage min (1)
```
The first button field will be declared as index 1 in Table 14: "Button Usage Page", page 69, HUT1_12.pdf. i.e.: It will be Button 1, or the primary Button.
```
29 03      Usage max (3)
```
The last button field will be declared as index 3 in Table 14, "Button 3". All fields in-between will be consecutive. For example, the second field will be Button 2.
```
15 00      Logical Min (0)
```
This indicates that the minimum value this field(s) may contain is zero.
```
25 01      Logical Max (1)
```
This indicates that the maximum value this field(s) may contain is one. We haven't found out yet, but below it tells us that the size of this field is 1 bit. Since a single bit can only be 0 or 1, the Logical Min and Max would then be 0 and 1.
```
75 01      report size (1)
```
Indicates that the size of each field is a single bit.
```
95 03      report count (3)
```
Indicates that we will declare three fields.
```
81 02      Input (Data,Var,Abs,NoW,Lin,PState,NoNull,Bit)
```
This is where you add to the input buffer format. You will add three fields of 1 bit each. The parameters are:

```
Data    - The fields can be modified.
Var     - Each field represents one item.
Abs     - Is either 0 or 1, not a change from last's value.
NoW     - No Wrap.
Lin     - Linear. The values are not modified.
Pstate  - The button has a Preferred State when not in use.
NoNull  - No Null State Position
Bit     - Bit field
```

To this point, your input buffer should be three bits in size.
```
75 05      report size (5)
```
Indicates that the next field(s) is/are 5 bits in size.

95 01 report count (1)

Indicates that we will declare 1 field.

81 01 Input (Cons,Array,Abs,NoW,Lin,PState,NoNull,Bit)

Add one field of 5 bits each. The parameters are:

Cons - The value remains fixed. Read only. (reserved)
- When the Cons parameter is used, some of the other parameters have no effect.

Bit - Bit field

To this point, your input buffer should be eight bits in size.

05 01 Usage Page (Generic Desktop)

Move back to the Generic Desktop Usage Page.

09 30 Usage (X)

The next field will be an X-coordinate field.

09 31 Usage (Y)

The next field after that will be a Y-coordinate field.

09 38 Usage (Wheel)

The next field after that will be a Wheel (Z-coordinate) field.

15 81 Logical Min (-127)

This indicates that the minimum value these fields may contain is -127.

25 7F Logical Max (127)

This indicates that the maximum value these fields may contain is 127.

75 08 report size (8)

Indicates that these fields are 8 bits in size.

95 03 report count (3)

Indicates that we will declare 3 fields.

81 06 Input (Data,Var,Rel,NoW,Lin,PState,NoNull,Bit)

Add three fields of 8 bits each. The parameters are:

Data - The fields can be modified.
Var - Each field represents one item.
Rel - The returned value indicates a relative position from the last value returned.
NoW - No Wrap.
Lin - Linear. The values are not modified.
Pstate - The value has a Preferred State when not in use.
NoNull - No Null State Position
Bit - Bit field

To this point, your input buffer should be 32 bits in size.

C0 End Collection

This indicates that this finishes describing this set of buffers.

The remainder of the report will need to be ignored since an unsupported usage was found.

With what we have learned above, we should now have a 4-byte input report and a zero-byte output report. The two buffers should have been built like the following.

Table 24-3: Mouse Input Report

Input								
Bits	7	6	5	4	3	2	1	0
	Reserved					Button	Button	Button
	X Coordinates							
	Y Coordinates							
	Z Coordinates (Wheel)							

Output
0 bits

If you will notice, if a field is less than 8 bytes, it starts with the next lowest significant bit. For example, when the first three fields were added, it started with bit 0, not bit 7. Then it added 5 bits so that the next field would be on a byte boundary.

Table 24-3 now describes the data format of the data returned by an interrupt data packet, i.e.: this is what is sent by the mouse. Your driver will need to keep track of what each field represents.

Other Comments

For much more information, see the two USB documents named HID1_11.pdf and Hut1_12.pdf available at the USB-IF website. See Appendix A for more information.

Frank Zhao has written an online (Java) HID Report Descriptor Parser at http://eleccelerator.com/usbdescreqparser/. You paste in the hex values and it will parse the descriptor and display it all from that page.

Part 5

In Part 5, I will show how to enumerate and use a USB video camera, how to find the resolutions it is capable of, and how to use isochronous transfers to display real-time video to the screen.

Within the next few chapters, I will show the enumeration process of the video camera, the descriptors, and how to find the correct resolution and other functions of the video. Then I will show how to negotiate the resolution wanted, start the streaming, and show the format of the streamed data. Following that, I will show how to create Isochronous Transfer Descriptors using the EHCI controller.

Since the Low-Speed controllers don't allow ISO transfers, and Full-Speed controllers may not be fast enough for streaming data, especially for hi-resolution video, I will assume you have at least an EHCI controller and will use it for the remaining of Part 5.

Chapter 25 – Using a USB Video Camera

In the next chapters, I will explain how to configure and program a USB Video Camera, find and set a desired resolution, then retrieve video images from the camera as live stream video.

The USB-IF has created a form of protocol to use for the USB Video. This specification is from the USB-IF web page (See Appendix A) as USB_Video_Class_1_1.zip.

Please note that when I refer to a camera within this chapter, I am referring to a live stream video camera. A camera for taking pictures that stores images to self contained media is not explained within this chapter since it most likely will use the MSD type protocol explained in Chapter 17.

The camera I have used for the research of this chapter is a V7™ Vantage Webcam 300, model CS0300-1N. All of the example descriptors and information in this chapter are for this unit. However, the information and code within this chapter should work on any similar unit.

At this point, you should already be familiar with how to retrieve the device's descriptors via the enumeration process and send and receive data packets.

The Camera's Device Descriptors

During the enumeration process you should have been able to retrieve the camera's first descriptor, the Device Descriptor, and it should be similar to the following memory dump

```
0x00000000  12 01 00 02 EF 02 01 40-EC 18 99 33 00 01 01 02
0x00000010  00 01
```

and explained here.

Table 25-1: Device Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x01**	Standard Device Descriptor.
2	Release Num	2	**0x0200**	USB Specification Release Number in Binary-Coded Decimal. 0x0200 = v2.00
4	Device Class	1	**0xEF**	A class code of 0xEF, a sub class code of 0x02, and a protocol of 0x01 indicates an Interface Association Descriptor.
5	Sub Class	1	**0x02**	See Device Class above.
6	Protocol	1	**0x01**	See Device Class above.
7	MaxPacketSize	1	**0x40**	Max packet size for the control endpoint.
8	Vendor ID	2	**0x18EC**	Vendor ID[1]: Arkmicro Technologies Inc.
10	Product ID	2	**0x3399**	Product ID: Not Listed
12	Device Rel	2	**0x0100**	Release version 1.00
14	Manufacturer	1	**0x01**	No Manufacturer descriptor. (see note below)
15	Product	1	**0x02**	Index of string descriptor
16	Serial Num	1	**0x00**	Index of Serial Number descriptor
17	Configurations	1	**0x01**	Only one configuration.

[1] http://www.linux-usb.org/usb-ids.html

If you will notice, the Device Class field above is not the Video Class code as you might have expected. Since the camera can capture video at different resolutions and can capture sound at the same time, there needs to be a slightly different descriptor format used. The Class code of 0xEF, a Sub Class code of 0x02, and a Protocol code of 0x01 above is the "Interface Association Descriptor" class. This will tell us that we need to watch for multiple Interface descriptors within multiple Interface Association descriptors. These combinations allow for interfaces to be grouped to a function. I will discuss this later in this chapter. See Appendix A for the two specification files used. For now, let's go on with the familiar descriptors.

The String Language Descriptor, or the first string indexed descriptor, returns the following memory dump

```
0x00000000  04 03 09 04
```

and explained here.

Table 25-2: String Language Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x04**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	First Language ID word	2	**0x0409**	0000_01|00_0000_1001b Low 10 bits is 0x09 meaning English, while the upper 6 bits is 0x01 meaning the United States.

This camera returns a string index for the Manufacturer ID, listed here, and then the Product ID, listed next, with the values listed on the next few pages.

```
0x00000000  12 03 41 00 42 00 43 00-4D 00 49 00 43 00 52 00
0x00000010  4F 00
```

Table 25-3: String Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x10	**0x0041** **0x0042** **0x0043** **0x004D** **0x0049** **0x0043** **0x0052** **0x004F**	A B C M I C R O

Product ID:

```
0x00000000  22 03 55 00 53 00 42 00-32 00 2E 00 30 00 20 00
0x00000010  50 00 43 00 20 00 43 00-41 00 4D 00 45 00 52 00
0x00000020  41 00
```

Table 25-4: String Descriptor 2

Offset	Field	Size	Value	Description
0	Length	1	**0x22**	Size of this descriptor in bytes.
1	Type	1	**0x03**	Standard String Descriptor.
2	String in Unicode format	0x20	**0x0055** **0x0053** **0x0042** **0x0032** **0x002E** **0x0030**	U S B 2 . 0

			0x0020	<space>
			0x0050	P
			0x0043	C
			0x0020	<space>
			0x0043	C
			0x0041	A
			0x004D	M
			0x0045	E
			0x0052	R
			0x0041	A

To this point, we might now know what the device is and have a human readable string to associate it with. However, as noted above, the Class Code in the Device Descriptor above was not the Video Class Code we were expecting, so our driver doesn't know what type of device it is yet. Therefore, let's continue to see what the Configuration Descriptor tells us.

As noted in a previous chapter, the Configuration Descriptor contains all Interface and Endpoint Descriptors for this configuration. Here is a memory dump of the whole descriptor:

Listing 25-1: The Configuration Descriptor Dump

```
0x00000000  09 02 9B 02 04 01 00 80-64 08 0B 00 02 0E 03 00
0x00000010  00 09 04 00 00 01 0E 01-00 00 0D 24 01 00 01 33
0x00000020  00 00 6C DC 02 01 01 12-24 02 01 01 02 00 00 00
0x00000030  00 00 00 00 00 03 00 00-00 0B 24 05 02 01 00 00
0x00000040  02 3F 06 00 09 24 03 03-01 01 00 02 00 07 05 81
0x00000050  03 10 00 0A 05 25 03 10-00 09 04 01 00 00 0E 02
0x00000060  00 00 0E 24 01 01 1C 01-83 00 03 02 01 00 01 00
0x00000070  1B 24 04 01 05 59 55 59-32 00 00 10 00 80 00 00
0x00000080  AA 00 38 9B 71 10 01 00-00 00 00 2A 24 05 01 00
0x00000090  80 02 E0 01 00 00 77 01-00 00 CA 08 00 60 09 00
0x000000A0  15 16 05 00 04 15 16 05-00 2A 2C 0A 00 40 42 0F
0x000000B0  00 80 84 1E 00 2A 24 05-02 00 A0 00 78 00 00 00
0x000000C0  77 01 00 00 CA 08 00 96-00 00 15 16 05 00 04 15
0x000000D0  16 05 00 2A 2C 0A 00 40-42 0F 00 80 84 1E 00 2A
0x000000E0  24 05 03 00 B0 00 90 00-00 00 77 01 00 00 CA 08
0x000000F0  00 C6 00 00 15 16 05 00-04 15 16 05 00 2A 2C 0A
0x00000100  00 40 42 0F 00 80 84 1E-00 2A 24 05 04 00 40 01
0x00000110  F0 00 00 00 77 01 00 00-CA 08 00 58 02 00 15 16
0x00000120  05 00 04 15 16 05 00 2A-2C 0A 00 40 42 0F 00 80
0x00000130  84 1E 00 2A 24 05 05 00-60 01 20 01 00 00 77 01
0x00000140  00 00 CA 08 00 18 03 00-15 16 05 00 04 15 16 05
0x00000150  00 2A 2C 0A 00 40 42 0F-00 80 84 1E 00 1B 24 03
0x00000160  00 05 80 02 E0 01 A0 00-78 00 B0 00 90 00 40 01
0x00000170  F0 00 60 01 20 01 01 01-06 24 0D 00 00 00 09 04
0x00000180  01 01 01 0E 02 00 00 07-05 83 05 C0 00 01 09 04
0x00000190  01 02 01 0E 02 00 00 07-05 83 05 80 01 01 09 04
0x000001A0  01 03 01 0E 02 00 00 07-05 83 05 00 02 01 09 04
0x000001B0  01 04 01 0E 02 00 00 07-05 83 05 80 02 01 09 04
0x000001C0  01 05 01 0E 02 00 00 07-05 83 05 20 03 01 09 04
0x000001D0  01 06 01 0E 02 00 00 07-05 83 05 C0 03 01 09 04
0x000001E0  01 07 01 0E 02 00 00 07-05 83 05 80 0A 01 09 04
0x000001F0  01 08 01 0E 02 00 00 07-05 83 05 20 0B 01 09 04
0x00000200  01 09 01 0E 02 00 00 07-05 83 05 E0 0B 01 09 04
0x00000210  01 0A 01 0E 02 00 00 07-05 83 05 80 13 01 09 04
0x00000220  01 0B 01 0E 02 00 00 07-05 83 05 E8 13 01 08 0B
0x00000230  02 02 01 01 00 04 09 04-02 00 00 01 01 00 04 09
```

```
0x00000240  24 01 00 01 28 00 01 03-0C 24 02 01 01 02 00 02
0x00000250  03 00 00 00 09 24 03 03-01 01 00 02 00 0A 24 06
0x00000260  02 01 01 00 03 03 00 09-04 03 00 00 01 02 00 04
0x00000270  09 04 03 01 01 01 02 00-04 07 24 01 03 01 01 00
0x00000280  0B 24 02 01 01 02 10 01-80 BB 00 09 05 82 05 60
0x00000290  00 04 00 00 07 25 01 01-00 00 00
```

If you will notice, the descriptor length is only 9, yet there is considerably more length to the descriptor, 667 bytes to be exact. The Configuration Descriptor has a 16-bit field to indicate the total length of the descriptor, which includes all Interface and Endpoint Descriptors for this configuration.

When I first created my code for enumerating USB devices, I didn't think there would be a configuration descriptor larger than 512 bytes, so I set the limit to 512 and forgot about it. Later, when I started using this device, I could not figure out why I was short some interface descriptors. It was only when I came back to the code to find that I was only getting the first 512 bytes of any descriptor instead of all 667 bytes. This has since been fixed. :-)

Let's start with the first part of the Configuration Descriptor.

```
0x00000000  09 02 27 00 01 01 00 80-64
```

Table 25-5: Configuration Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x02**	Config Descriptor
2	TotalLength	2	**0x029B**	Total length of all data within this configuration descriptor (667 bytes)
4	NumInterfaces	1	**0x04**	Number of Interfaces.
5	ConfigVal	1	**0x01**	Value for this configuration
6	ConfigString	1	**0x00**	Index for String Descriptor describing this config.
7	Attributes	1	**0x80**	**D7** Reserved (See Appendix D) D6 Self Powered D5 Remote Wakeup D4..0 reserved (0)
8	MaxPower	1	**0x64**	Maximum Power Consumption. 0x64 = 200mA

This Configuration Descriptor indicates that we have four Interface Descriptors. Remember also that the Interface Association Descriptor class code in the Device Descriptor told us to watch for other Interface and Function Descriptors.

Interface Association and Function Descriptors

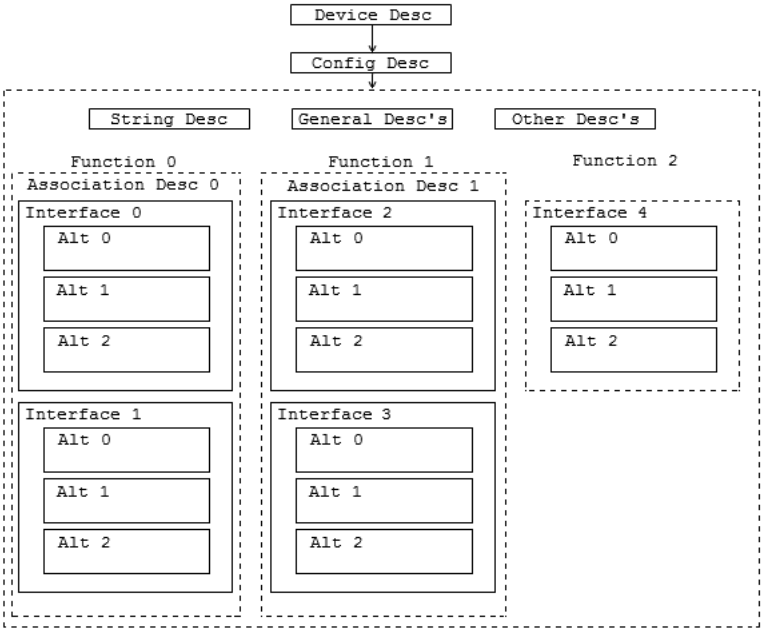
As with the camera described in this chapter, some devices may have more than one function. For example, this camera captures video as well as sound, as if it was two different devices, a camera and a microphone. Each of these functions may have multiple alternate interfaces, and we need to know how to keep the two functions separated, knowing which interface(s) go to which function.

This is where the Interface Association Function comes in. Since the Device Descriptor only has room for one type of device, it now has the values to indicate that this device is a multi-function device and will have at least two Interface Association Descriptors, with these Interface Association Descriptors containing the class codes for each function of the device.

In the figure below, you will see an example device that contains three different functions, two with multiple interfaces, and a separate interface for the third function. The first function might be the video-capture, the second function the sound, while the third function, the interface by itself, could be touch key pad.

As far as the operating system is concerned, there are three separate devices plugged into this port, each requiring its own driver set.

Figure 25-1: Interface Association Descriptor



However, since all three functions will have the same general configuration, such as string descriptors, product ids, etc., they all use the same Device Descriptor, Config Descriptor, and any general descriptors.

In Figure 25-1 above, your operating system will see the first function as “Device 0”, having a Device Descriptor, a Configuration Descriptor using the part of that descriptor described by the Interface Association Descriptor, zero or more String Descriptors, and possibly other descriptors such as a Device Qualifier Descriptor.

Then your operating system will see the second function as “Device 1”, having the exact same descriptors except that the Configuration Descriptor now only has the parts within the second Interface Association Descriptor. It uses the same general descriptors, such as the String Descriptors. Finally, your operating system will see a third function as “Device 2”, again containing all the same descriptors, except the Configuration Descriptor now only has Interface 4 within it.

Please remember that the device’s Configuration Descriptor will contain all three functions. It is up to your driver code to split it up into the three different devices.

Please note that each Interface may have zero or more alternate interfaces, but an Interface Association Descriptor must have at least two Interfaces. If a device only has one interface, whether it has multiple alternates or not, it should not be “surrounded” with an Interface Association Descriptor.

With the camera described within Part 5 of this book, we only discuss the first function, or all of the parts of the first Interface Association Descriptor. The second, this device only having two, is the microphone portion, and will by left to the reader to implement that function if you so desire.

Note that the Configuration Descriptor shown in Listing 25-1, lists both Association Descriptors.

The Video Control Interface

The first Interface Association Descriptor is the video interface, containing two interfaces, the first for the Video Control, containing no alternate interfaces, and the second for the Video Streaming, containing multiple alternate interfaces.

```
0x00000000                    08 0B 00 02 0E 03 00
0x00000010  00
```

Table 25-6: Interface Association Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x08**	Size of descriptor in bytes.
1	Type	1	**0x0B**	Interface Association Desc.
2	First Inter-face number	1	**0x00**	Number of first interface associated with this function.
3	Interface Count	1	**0x02**	Count of contiguous interfaces associated with this function.
4	Class Code	1	**0x0E**	Class Code: **0x0E** = Video
5	Sub Class	1	**0x03**	Sub Class Code: **0x03** = Interface Collection.
6	Protocol	1	**0x00**	Protocol Code: not used.
7	Function Str	1	**0x00**	No string descriptor.

The descriptor gave us a count of interfaces within this function and from which interface to start counting from. Therefore, Interface 0 and Interface 1 are within the scope of this function.

Next is the first Interface Descriptor within this function.

```
0x00000010    09 04 00 00 01 0E 01-00 00
```

Table 25-7: Interface Descriptor: 0

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x00**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: 1
5	Class Code	1	**0x0E**	Class Code: 0x0E = Video
6	Sub Class	1	**0x01**	Sub Class Code: **0x01** = Video Control.
7	Protocol	1	**0x00**	Protocol Code: Not Used.
8	Interface Str	1	**0x00**	No string descriptor.

```
0x00000010                    0D 24 01 00 01 33
0x00000020  00 00 6C DC 02 01 01
```

This Interface Descriptor gives us a Video Control interface. A Video Control interface is used to give us what kind of Video it is, in this case a camera, what controls we can use to manipulate the camera, what kind of terminal it is, streaming in this case, as well as other information to control the camera.

The first Interface Function descriptor gives us the version of the specification, the size in bytes of all interface function descriptors for this interface, the clock frequency and collection count.

Table 25-8: Function Descriptor: Video Control

Offset	Field	Size	Value	Description
0	Length	1	**0x0D**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface Function
2	Sub Type	1	**0x01**	Video Control Header
3	BCD version	2	**0x0100**	BCD version **0x0100** = 1.00
5	Total Length	2	**0x0033**	Total length of all Function Descriptors in this interface (**51**)
7	Clock Freq.	4	**0x02DC6C00**	Clock Frequency in Hz. (**0x02DC6C00** = 48000000Hz)
11	Collection Count	1	**0x01**	Number of Streaming Interfaces
12	Interface Num	1	**0x01**	Streaming Interface Number.

Note that the BCD Version is 1.00 (for 1.0a). It can also be 1.10 (for 1.1), or 1.50 (for 1.5), with each using a slightly different formatted descriptor. Be sure to watch for this.

This must be an old camera. The USB-IF doesn't even have version 1.0a listed on their site anymore.

```
0x00000020                12-24 02 01 01 02 00 00 00
0x00000030  00 00 00 00 00 03 00 00-00
```

Table 25-9: Function Descriptor: VC Input

Offset	Field	Size	Value	Description
0	Length	1	**0x12**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface Function
2	Sub Type	1	**0x02**	Video Input Terminal
3	Terminal ID	1	**0x01**	Terminal Address
4	Terminal Type	2	**0x0201**	Camera
6	Output Terminal ID	1	**0x00**	Associated Output Terminal or zero if none.
7	Terminal Str	1	**0x00**	No string descriptor.
8	Focal Len Min	2	**0x0000**	Minimum Focal length
10	Focal Len Max	2	**0x0000**	Maximum Focal length
12	Ocular focal	2	**0x0000**	Ocular focal length
14	Next Field Sz	1	**0x03**	Next field size in bytes (n)
15+n	Control Bitmap	n	**000000**	Bit: 0 - Scanning mode

				1 - Auto-Exposure Mode 2 - Auto-Exposure Priority 3 - Exposure Time (Absolute 4 - Exposure Time (Relative 5 - Focus (Absolute) 6 - Focus (Relative) 7 - Iris (Absolute) 8 - Iris (Relative) 9 - Zoom (Absolute) 10 - Zoom (Relative) 11 - Pan Tilt (Absolute) 12 - Pan Tilt (Relative) 13 - Roll (Absolute) 14 - Roll (Relative) 15 - Reserved 16 - Reserved 17 - Focus, Auto 18 - Privacy 19 - Remaining are Reserved

0x00000030 0B 24 05 02 01 00 00
0x00000040 02 3F 06 00

Table 25-10: Function Descriptor: VC Processing Unit

Offset	Field	Size	Value	Description
0	Length	1	**0x0B**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface Function
2	Sub Type	1	**0x05**	Video Processing Unit
3	Unit ID	1	**0x02**	Unit Address (The Entity)
4	Source ID	1	**0x01**	Unit/Function this Unit is connected to
5	Max Multiplier	2	**0x0000**	Max (Zoom) Multiplier * 100. 0x0000 = not supported.
7	Control Size	1	**0x02**	Size of next field in bytes (n)
8	Controls	n	**0x063F**	Bit: **0 - Brightness** **1 - Contrast** **2 - Hue** **3 - Saturation** **4 - Sharpness** **5 - Gamma** 6 - White Balance Temp 7 - White Balance Component 8 - Backlight Compensation **9 - Gain** **10 - Power Line Frequency** 11 - Hue, Auto

				12 - White Bal. Temp, Auto 13 - White Bal. Comp, Auto 14 - Digital Multiplier 15 - Digital Mult. Limit 16 - not used
8+n	Unit Str	1	**0x00**	No string descriptor.

0x00000040 09 24 03 03-01 01 00 02 00

Table 25-11 Function Descriptor: VC Output

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface Function
2	Sub Type	1	**0x03**	Video Output Terminal
3	Terminal ID	1	**0x03**	Terminal Address
4	Terminal Type	2	**0x0101**	Streaming
6	Output Terminal ID	1	**0x00**	Associated Input Terminal or zero if none.
7	Source ID	1	**0x02**	Unit/Function this Unit is connected to
8	Terminal Str	1	**0x00**	No string descriptor.

0x00000040 07 05 81
0x00000050 03 10 00 0A

Table 25-12: Endpoint Descriptor 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x81**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **1**
3	Attributes	1	**0x03**	Bits 7:2 Reserved Bits 1:0 Transfer type 00 = Control 01 = Isochronous 10 = Bulk **11** = Interrupt
4	MaxPacketSize	2	**0x0010**	Max packet size: 16 bytes
6	Interval	1	**0x0A**	Intervals of 10

0x00000050 05 25 03 10-00

Table 25-13: Function Descriptor: VC Processing Unit

Offset	Field	Size	Value	Description
0	Length	1	**0x05**	Size of descriptor in bytes.
1	Type	1	**0x25**	Endpoint Function
2	Sub Type	1	**0x03**	Interrupt
3	Max Transfer Size	2	**0x0010**	16 bytes

```
0x00000050                    09 04 01 00 00 0E 02
0x00000060  00 00
```

At this point we have received all of the Video Control descriptors, have determined that the video feed is of type Streaming, has an at most 16 byte Interrupt In packet set to an interval of 10 (micro)frames, and have the control items we can manipulate using Unit 2, also called Entity 2.

This is where, after sending the Set Configuration and Set Interface requests, we would get the information about each Control Selector, using GET_INFO, GET_MIN, GET_MAX, GET_DEF, etc. explained in the specification file USB_Video_Class_1_X.pdf, depending on which version you are using. Since we are going to use the defaults, let's move on to the Streaming Descriptors, choose a capture resolution, and start receiving streaming data.

The Video Streaming Interface

The remaining descriptors describe the capable resolutions and compression used for the streaming data. The first few are used to indicate how many payload formats there are and what endpoint number and type to use. In this example, there is only one payload format, Uncompressed, and offering five resolutions. This also allows single image capture, or Still Capture, using Method 2 and a button on the camera for the trigger. Below is the Interface Descriptor, being Interface number 1. Once you chose a resolution, you use that alternate interface descriptor.

Table 25-14: Interface Descriptor: 1

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x01**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x00**	Number of Endpoints: **0**
5	Class Code	1	**0x0E**	Class Code: **0x0E** = Video
6	Sub Class	1	**0x02**	Sub Class Code: **0x02** = Video Streaming.
7	Protocol	1	**0x00**	Protocol Code: Not Used.
8	Interface Str	1	**0x00**	No string descriptor.

Please note that the Function Descriptors will have the same Sub Type value as some of the ones for the control interface. It is up to the driver to know which interface you are currently parsing to know which type of function descriptor set to use. You may use the Sub Class code in the interface descriptor. All Interface Function Descriptors will follow the Interface they are describing.

0x00000060 0E 24 01 01 1C 01-83 00 03 02 01 00 01 00

Table 25-15: Function Descriptor: Video Streaming

Offset	Field	Size	Value	Description
0	Length	1	**0x0E**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x01**	Video Streaming Input Header
3	Num Formats	1	**0x01**	Payload Formats (p)
4	Total Length	2	**0x011C**	Total length of all Function Descriptors in this interface (284)
6	Endpoint Address	1	**0x83**	Endpoint address: Bits 7 0 = Out, **1 = In** 6:4 reserved (0) 3:0 Endpoint Num = **3**
7	Caps Bitmap	1	**0x00**	Bits 7:1 reserved (0) 0 Dynamic format change supported **0 = No** 1 = Yes
8	Terminal ID	1	**0x03**	Term ID of output Term this

				Endpoint is connected to.
9	Still Capture	1	**0x02**	Still Image Capture Method: 0 = None 1 = Method 1 **2 = Method 2** 3 = Method 3
10	Trigger Support	1	**0x01**	Trigger Support 0 = Not Supported **1 = Supported**
11	Trigger Usage	1	**0x00**	Host Trigger Usage **0 = Initiate image capture** 1 = General purpose event
12	Control Size	1	**0x01**	Bitmap Control Field Size in bytes (n)
13	Control Bitmap[p]	n	**0x00**	Bits (n*8-1):6 reserved (0) 5 Update Frame Seg 4 Generate Key Frm 3 Compression Win 2 Compression Qnty 1 P Frame Rate 0 Key Frame Rate

0x00000070 1B 24 04 01 05 59 55 59-32 00 00 10 00 80 00 00
0x00000080 AA 00 38 9B 71 10 01 00-00 00 00

Table 25-16: Function Descriptor: VS Format Uncompressed

Offset	Field	Size	Value	Description
0	Length	1	**0x1B**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x04**	Format Uncompressed
3	Index	1	**0x01**	Index of this Format Desc.
4	Count	1	**0x05**	Count of descriptors associated with this Format.
5	GUID	16	**Varies**	GUID for Identification
21	Bits / Pixel	1	**0x10**	Bits per pixel (**16**)
22	Default Frame Index	1	**0x01**	Optimal Frame Index for this stream.
23	X Ratio	1	**0x00**	Aspect Ratio. X Dimension (**0 = non interlaced**)
24	Y Ratio	1	**0x00**	Aspect Ratio. Y Dimension (**0 = non interlaced**)
25	Interlace Flags	1	**0x00**	Interlace information: Bits 7:6 **00 = Bob only** 01 = Weave only 10 = Bob or Weave 5:4 **00 = Field 1 only** 01 = Field 2 only 10 = Regular 11 = Random 3 reserved

				2 1 = Field 1 first 1 **0 = 2 Fields/frm** 1 = 1 Field/frame 0 1 = Interlace or variable
26	Copy Protect	1	**0x00**	**0 = No Restrictions** 1 = Restrict Duplication

```
0x00000080                              2A 24 05 01 00
0x00000090  80 02 E0 01 00 00 77 01-00 00 CA 08 00 60 09 00
0x000000A0  15 16 05 00 04 15 16 05-00 2A 2C 0A 00 40 42 0F
0x000000B0  00 80 84 1E 00
```

Table 25-17: Function Descriptor: VS Frame Uncompressed (1)

Offset	Field	Size	Value	Description
0	Length	1	**0x2A**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x05**	Frame Uncompressed
3	Index	1	**0x01**	Index of this Format Desc.
4	Capabilities	1	**0x00**	Bits 7:1 reserved 0 Still Images
5	Width	2	**0x0280**	Width (**640**)
7	Height	2	**0x01E0**	Height (**480**)
9	Min Bit Rate	4	**0x01770000**	Minimum bit rate at the longest frame interval (**24576000** = 5 Frames per second (640x480x16bits)
13	Max Bit Rate	4	**0x08CA0000**	Maximum bit rate at the longest frame interval (**147456000** = 30 Frames per second (640x480x16bits)
17	Max Buffer Size	4	**0x00096000**	Max Frame Buffer Size (**614400** bytes) (640x480x16bits)
21	Default Frame Interval	4	**0x00051615**	Default Frame Interval
25	Frame Interval Type (see next fields)	1	**0x04**	Type: 0 = Continuous x = number of discrete intervals supported(**4**)
If Frame Interval Type == 0 (Continuous)				
26	Min Frame Interval	4	x	Shortest Frame Interval supported, in 100nS units
30	Max Frame Interval	4	x	Longest Frame Interval supported, in 100nS units
34	Frame Step	4	x	Granularity frame interval range, in 100nS units
If Frame Interval Type > 0 (Discrete Intervals) (**4**)				
26	Interval[0]	4	**0x00051615**	Shortest Frame Interval supported, in 100nS units
30	Interval[1]	4	**0x000A2C2A**	Next Frame Interval

					supported, in 100nS units
	34	Interval[2]	4	**0x000F4240**	Next Frame Interval supported, in 100nS units
	38	Interval[3]	4	**0x001E8480**	Longest Frame Interval supported, in 100nS units

```
0x000000B0              2A 24 05-02 00 A0 00 78 00 00 00
0x000000C0  77 01 00 00 CA 08 00 96-00 00 15 16 05 00 04 15
0x000000D0  16 05 00 2A 2C 0A 00 40-42 0F 00 80 84 1E 00
```

Table 25-18: Function Descriptor: VS Frame Uncompressed (2)

Offset	Field	Size	Value	Description
0	Length	1	**0x2A**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x05**	Frame Uncompressed
3	Index	1	**0x02**	Index of this Format Desc.
4	Capabilities	1	**0x00**	Bits 7:1 reserved 0 Still Images
5	Width	2	**0x00A0**	Width (**160**)
7	Height	2	**0x0078**	Height (**120**)
... Remaining items are as previous descriptor				

```
0x000000D0                                  2A
0x000000E0  24 05 03 00 B0 00 90 00-00 00 77 01 00 00 CA 08
0x000000F0  00 C6 00 00 15 16 05 00-04 15 16 05 00 2A 2C 0A
0x00000100  00 40 42 0F 00 80 84 1E-00
```

Table 25-19: Function Descriptor: VS Frame Uncompressed (3)

Offset	Field	Size	Value	Description
0	Length	1	**0x2A**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x05**	Frame Uncompressed
3	Index	1	**0x03**	Index of this Format Desc.
4	Capabilities	1	**0x00**	Bits 7:1 reserved 0 Still Images
5	Width	2	**0x00B0**	Width (**176**)
7	Height	2	**0x0090**	Height (**144**)
... Remaining items are as previous descriptor				

```
0x00000100                      2A 24 05 04 00 40 01
0x00000110  F0 00 00 00 77 01 00 00-CA 08 00 58 02 00 15 16
0x00000120  05 00 04 15 16 05 00 2A-2C 0A 00 40 42 0F 00 80
0x00000130  84 1E 00
```

Table 25-20: Function Descriptor: VS Frame Uncompressed (4)

Offset	Field	Size	Value	Description
0	Length	1	**0x2A**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x05**	Frame Uncompressed
3	Index	1	**0x04**	Index of this Format Desc.
4	Capabilities	1	**0x00**	Bits 7:1 reserved 0 Still Images

5	Width	2	**0x0140**	Width (**320**)
7	Height	2	**0x00F0**	Height (**240**)
... Remaining items are as previous descriptor				

```
0x00000130          2A 24 05 05 00-60 01 20 01 00 00 77 01
0x00000140  00 00 CA 08 00 18 03 00-15 16 05 00 04 15 16 05
0x00000150  00 2A 2C 0A 00 40 42 0F-00 80 84 1E 00
```

Table 25-21: Function Descriptor: VS Frame Uncompressed (5)

Offset	Field	Size	Value	Description
0	Length	1	**0x2A**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x05**	Frame Uncompressed
3	Index	1	**0x05**	Index of this Format Desc.
4	Capabilities	1	**0x00**	Bits 7:1 reserved 0 Still Images
5	Width	2	**0x0160**	Width (**352**)
7	Height	2	**0x0120**	Height (**288**)
... Remaining items are as previous descriptor				

Next is the descriptor describing the Still Capture format.

```
0x00000150                          1B 24 03
0x00000160  00 05 80 02 E0 01 A0 00-78 00 B0 00 90 00 40 01
0x00000170  F0 00 60 01 20 01 01 01
```

Table 25-22: Function Descriptor: VS Still Frame

Offset	Field	Size	Value	Description
0	Length	1	**0x1B**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x03**	Still Frame
3	Endpoint Address	1	**0x00**	Endpoint address: Method 3 used: Bits 7 0 = Out, 1 = In 6:4 reserved (0) 3:0 Endpoint Num = 3 **Method 2** used: Bits 7:0 reserved (**0x00**)
4	Num Patterns	1	**0x05**	Number of Patterns (Width and Height pairs below)
5	Width[0]	2	**0x0280**	Width (**640**)
7	Height[0]	2	**0x01E0**	Height (**480**)
9	Width[1]	2	**0x00A0**	Width (**160**)
11	Height[1]	2	**0x0078**	Height (**120**)
13	Width[2]	2	**0x00B0**	Width (**176**)
15	Height[2]	2	**0x0090**	Height (**144**)
17	Width[3]	2	**0x0140**	Width (**320**)
19	Height[3]	2	**0x00F0**	Height (**240**)
21	Width[4]	2	**0x0160**	Width (**352**)
23	Height[4]	2	**0x0120**	Height (**288**)
25	Num Comp Pats	1	**0x01**	Num Compression Patterns (**1**)

26	CompPat[0]	1	**0x01**	Compression pattern[0] (**1**)

The 16-byte GUID field in Table 25-16 has a four-digit sample format identifier. The two allowed are “YUY2” and “NV12”. This indicates the format used for the samples received. For example, YUY2 might be equivalent to the UYVY format, which is a 4:2:2 sample, and used by most codec’s. For more information, you might visit http://www.fourcc.org/yuv.php.

The Color Matching Descriptor

The Color Matching Descriptor is optional and usually only included if the image does not imply its own color characteristics, and then sometimes it doesn't even show up then. In this example, it is included, but all of the values are unspecified, zeros. Since color matching is not part of the scope of this book, I will simple show the descriptor here and then ignore any further reference.

The specification for the Uncompressed Format says that the Color Matching Descriptor is mandatory for this format. Since all of the values listed below are Unspecified, I would assume you use the defaults, or simply ignore this descriptor if Unspecified is indicated.

0x00000170 06 24 0D 00 00 00

Table 25-23: Function Descriptor: VS Color Matching

Offset	Field	Size	Value	Description
0	Length	1	**0x06**	Size of descriptor in bytes.
1	Type	1	**0x24**	Interface
2	Sub Type	1	**0x0D**	Video Color Matching
3	Primaries	1	**0x00**	Color Primaries: 0 = **Unspecified** 1 = BT.709 sRGB (Default) 2 = BT.470-2 (M) 3 = BT.470-2 (B,G) 4 = SMPTE 170M 5 = SMPTE 240M 6:255 = reserved
4	Transfer Chars	1	**0x00**	Transfer Characteristics: 0 = **Unspecified** 1 = BT.709 (Default) 2 = BT.470-2 (M) 3 = BT.470-2 (B,G) 4 = SMPTE 170M 5 = SMPTE 240M 6 = Linear (V = Lc) 7 = sRGB (similar to 1) 8:255 = reserved
5	MatrixCo	1	**0x00**	Matrix Coefficients: 0 = **Unspecified** 1 = BT.709 2 = FCC 3 = BT.470-2 (B,G) 4 = SMPTE 170M (BT.601) (Default) 5 = SMPTE 240M 6:255 = reserved

Alternate Interface Descriptors

Here is where the Alternate Interface Descriptors are. Once you choose a resolution above, say the first one, 640 x 480, you would then choose the Alt Interface Descriptor below. More on this later in this chapter.

```
0x00000170                                09 04
0x00000180  01 01 01 0E 02 00 00 07-05 83 05 C0 00 01
```

Table 25-24: Alt Interface Descriptor: 1

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**0x01**	Number of Interface.
3	Alternate Set	1	**0x01**	Used to select alt. setting.
4	Endpoints	1	**0x01**	Number of Endpoints: **1**
5	Class Code	1	**0x0E**	Class Code: **0x0E** = Video
6	Sub Class	1	**0x02**	Sub Class Code: **0x02** = Video Streaming.
7	Protocol	1	**0x00**	Protocol Code: Not Used.
8	Interface Str	1	**0x00**	No string descriptor.

Table 25-25: Alt Interface 1: Endpoint 1

Offset	Field	Size	Value	Description
0	Length	1	**0x07**	Size of descriptor in bytes.
1	Type	1	**0x05**	Endpoint Descriptor
2	Address	1	**0x83**	Endpoint address: Bits 7 0 = Out, 1 = **In** 6:4 reserved (0) 3:0 Endpoint Num = **3**
3	Attributes	1	**0x05**	Bits 7:6 Reserved 5:4 Usage Type **00** = Data Endpoint 01 = Feedback Endpoint 10 = Implicit Feedback 11 = Reserved 3:2 Sync Type 00 = No Synchronization **01** = Asynchronous 10 = Adaptive 11 = Synchronous 1:0 Transfer type 00 = Control **01** = Isochronous 10 = Bulk 11 = Interrupt
4	MaxPacketSize	2	**0x00C0**	Bits 15:13 reserved (**0**) 12:11 transactions (**0**) 10:0 Max packet size (**192** bytes)
6	Interval	1	**0x01**	1mS polling (2 ^(**1**-1))

The remaining 10 Alternate Interface/Endpoint descriptor pairs are identical to the two listed above except for the Alternate Setting value in the Interface Descriptor and then the Max Packet Size field in the Endpoint Descriptor. Max Packet Size fields are 0:192, 0:384, 0:512, 0:640, 0:800, 0:960, 1:640, 1:800, 1:992, 2:896, and 2:1000 respectively. I have listed the values below.

```
0x00000180                                    09 04
0x00000190  01 02 01 0E 02 00 00 07-05 83 05 80 01 01 09 04
0x000001A0  01 03 01 0E 02 00 00 07-05 83 05 00 02 01 09 04
0x000001B0  01 04 01 0E 02 00 00 07-05 83 05 80 02 01 09 04
0x000001C0  01 05 01 0E 02 00 00 07-05 83 05 20 03 01 09 04
0x000001D0  01 06 01 0E 02 00 00 07-05 83 05 C0 03 01 09 04
0x000001E0  01 07 01 0E 02 00 00 07-05 83 05 80 0A 01 09 04
0x000001F0  01 08 01 0E 02 00 00 07-05 83 05 20 0B 01 09 04
0x00000200  01 09 01 0E 02 00 00 07-05 83 05 E0 0B 01 09 04
0x00000210  01 0A 01 0E 02 00 00 07-05 83 05 80 13 01 09 04
0x00000220  01 0B 01 0E 02 00 00 07-05 83 05 E8 13 01
```

The remaining descriptors in the Configuration Descriptor are the descriptors for the microphone portion of the device. They have similar characteristics of the descriptors already shown, and should function very similarly. Once you have learned how to receive the video from the device, receiving the sound should be an easy task. For the sake of the data, I will display the remaining bytes here anyway.

```
0x00000220                                    08 0B
0x00000230  02 02 01 01 00 04 09 04-02 00 00 01 01 00 04 09
0x00000240  24 01 00 01 28 00 01 03-0C 24 02 01 01 02 00 02
0x00000250  03 00 00 00 09 24 03 03-01 01 00 02 00 0A 24 06
0x00000260  02 01 01 00 03 03 00 09-04 03 00 00 01 02 00 04
0x00000270  09 04 03 01 01 01 02 00-04 07 24 01 03 01 01 00
0x00000280  0B 24 02 01 01 02 10 01-80 BB 00 09 05 82 05 60
0x00000290  00 04 00 00 07 25 01 01-00 00 00
```

A Known Video Device

We now know that the attached device is a video camera, uses end-point three for video, as an isochronous type input, and know the various formats and resolutions that the camera can return. We can now choose a format and resolution to use.

Since there is only one format specified for this device, we must use the Uncompressed format given. As for the resolution, we will choose the 640 x 480 resolution, the first Interface Function descriptor, the descriptor with an index of 1, shown in Table 25-17.

Shortly, we will select this alternative interface and end point to retrieve data from the camera. For now, we will get a little more information from the camera.

A Little More Information

We need to retrieve one more device descriptor before we configure the device. This descriptor is the Device Qualifier descriptor. It is used with full- and high-speed capable devices. This descriptor is what would be returned as the Device Descriptor if the device were operating in the other speed. For example, if the device were currently operating in full-speed, this descriptor would be the Device Descriptor for the device when operating in high-speed, and visa-versa.

The reason why this descriptor is missing some of the Device Descriptor fields, is that in either speed the Vendor ID, Product ID, Device Release code, and string descriptors are the same no matter which speed the device is currently operating in. Therefore, there is no need to return this information again.
If the USB camera you are using is a full-speed only device and has a Release Num field of 0x200 in the Device Descriptor, the USB 2.0 specification states, on page 264, that this device should return a 'request error'.

Table 25-26: Device Qualifier Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x80**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x06**	Get Descriptor
2	Value	2	**0x0600**	High byte is type 0x06, Device Qualifier, low byte is index, 0x00.
4	Index	2	**0x0000**	Zero
6	Length	2	**0x000A**	We want 10 bytes returned.

The camera I have returns

```
0x00000010  0A 06 00 02 EF 02 01 40-01 00
```

Table 25-27: Device Qualifier Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x0A**	Size of descriptor in bytes
1	Type	1	**0x06**	Device Qualifier Descriptor
2	Release Num	2	**0x0200**	USB Specification Release Number in Binary-Coded Decimal. 0x0200 = v2.00
4	Device Class	1	**0xEF**	Again, a class code of

				0xEF, sub class of 0x02, and a protocol of 0x01.
5	Sub Class	1	**0x02**	See Device Class above.
6	Protocol	1	**0x01**	See Device Class above.
7	MaxPacketSize	1	**0x40**	Max packet size for the control endpoint.
8	Configurations	1	**0x01**	Only one configuration.
9	reserved	1	**0x00**	Reserved for future use.

Starting with USB 3.0, the Device Qualifier and Other Speed Configuration descriptors are no longer supported. See USB 3.0 section 9.4.3. To replace them, the USB 3.0 specification has now added the Binary Device Object Store (BOS) Descriptor, USB 3.0 section 9.6.2.

If you felt inclined to do so, this is where you would request the Other Speed Configuration descriptor. The host is not to request this descriptor unless a valid Device Qualifier descriptor has been received.

At this point is probably where a major operating system pops up a balloon to state that “this device would run faster if plugged into a high speed port”. Then it lets you click on the balloon to see a list of high-speed controllers. My question is, by seeing a list of high-speed controllers, how are you going to know which port, whether it is in the front or the back of the computer, is a high-speed port?

Configure the Camera for Video

Now that we have retrieved all of the device descriptors, we now know that it is a video camera using isochronous transfers and having an IN endpoint with an endpoint value of three.

Since there was only one Configuration Descriptor, there is only one configuration to use, therefore, send the “Set Configuration” request with a configuration number of 1, retrieved from the ConfigVal field.

Other Commands and Comments

You have now received all of the information you need from the camera to successfully choose and set a resolution and start the streaming of the data. The next chapter explains how to negotiate a resolution with the camera, start the streaming, and the format of the streamed data.

Chapter 26 – Setting Up the USB Video Camera

In the previous chapter, we retrieved all of the descriptors of the camera, finding the resolutions supported, the format supported, and a few other items.

Starting with this chapter, we will now use that information, choose resolution, and start the streaming data.

At the end of the previous chapter, you sent the Set Configuration request to set the configuration. Since we will be working with the Streaming Interface found earlier, we need to set that interface by sending a Set Interface Request with the index number of that interface, the interface with a class code of Video (0x0E), and a sub class code of Video Streaming (0x02). In this case, it was the second interface in the first Interface Association Block, and has an Interface Number value of 1.

Choosing/Setting the Camera's Resolution

To set the resolution of the camera, you must negotiate with the camera using a 26-byte packet and the Probe and Commit Controls. You first send a Control Out packet containing the SET_CUR request with the Probe Packet containing the Format Index and Frame Index values. You then request the packet back using a Control In packet containing the GET_CUR request. If the values are satisfactory to your needs, you then send the packet back telling the camera that these values are usable. More on this process in a little while.

First, as an example, to show how to retrieve this 26-byte packet, let's get the current settings using the Probe Control, by sending a Control In request with the following Request Packet.

Table 26-1: VS_Probe_Control GET_CUR Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xA1**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x81**	GET_CUR
2	Value	2	**0x0100**	High byte: 0x01 = VS_PROBE Low byte: 0x00 = unused
4	Index	2	**0x0001**	High byte: 0x00 = unused Low byte: 0x01 = Interface
6	Length	2	**0x001A**	Descriptor length. (26d)

My camera returns the following values.

```
0x00000000  00 00 01 01 15 16 05 00-00 00 00 00 00 00 00 00
0x00000010  00 00 00 60 09 00 B8 0B-00 00
```

Table 26-2: VS_Probe_Control Return

Offset	Field	Size	Value	Description
0	Hint[1]	2	**0x0000**	Hint to the camera for with parameters to keep contant: D15..5 reserved (**0**) D4 CompWindowSize D3 CompQuality D2 PFrameRate D1 KeyFrameRate D0 FrameInterval
2	FormatIndex	1	**0x01**	The Index of the Format Descriptor being used
3	FrameIndex	1	**0x01**	The Index of the Frame

				Descriptor within the Format Index above
4	FrameInterval	4	**0x00051615**	Frame interval in 100nS units. (~30 fps)
8	KeyFrameRate	2	**0x0000**	Key Frame Rate
10	PFrameRate	2	**0x0000**	PFrame Rate (**0x0000** = first frame is the key frame)
12	CompQuality	2	**0x0000**	Compression Quality
14	CompWindowSize	2	**0x0000**	Window size for average bit rate control
16	Delay (Host R/O)	2	**0x0000**	Internal video streaming interface latency in mS
18	MaxVideoFrame Size[2]	4	**0x00096000**	Max Video Frame Size (**614400** = 640x480x16-bits)
22	MaxPayload TransferSize	4	**0x00000BB8**	Max Packet Size for a single transfer (**3,000**)
[1]Device Read Only [2]Host Read Only				

Please remember that the table on the previous page is for version 1.0a of the Video Class Specification. Newer versions have more fields. You must check the Video Control descriptor, Table 25-8 in Chapter 25, for the length of this packet the device will return.

The Camera is already set to use the first Format Index, the only one we have with this camera, and the first Frame Index, one of the five we found earlier. It also returned a Frame Interval of about 30 frames a second and a Max Video Frame Size of 614,400, the size of the resolution and bits per pixel, along with the Max Payload Size of 3,000 bytes.

This means that the camera is using the highest resolution we found, 640 x 480, which is the 1st frame in the only format we have, and will transfer at most 3,000 bytes per payload transfer.

Resolution Negotiation

Once you have chosen one of the resolutions, you will need the Format Index value from the Format descriptor, in this case a value of 1, the Frame Index value from one of the Frame descriptors, 1 to 5, and the Frame Interval value from the selected Frame descriptor.

With these values, you create the following 26-byte Probe packet. For this example, I have selected format index 1, frame index 1, and set the interval field. All other fields should be zero.

```
0x00000000  00 00 01 01 15 16 05 00-00 00 00 00 00 00 00 00
0x00000010  00 00 00 00 00 00 00 00-00 00
```

Then you send it as a Control Out Probe request.

Table 26-3: VS_Probe_Control SET_CUR Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x01**	SET_CUR
2	Value	2	**0x0100**	High byte: 0x01 = VS_PROBE Low byte: 0x00 = unused
4	Index	2	**0x0001**	High byte: 0x00 = unused Low byte: 0x01 = Interface
6	Length	2	**0x001A**	Descriptor length. (26d)

When that is successful, request the packet back as you did with a GET_CUR request from Table 26-1 before. You should now receive a 26-byte packet similar to the following.

```
0x00000000  00 00 01 01 15 16 05 00-00 00 00 00 00 00 00 00
0x00000010  00 00 00 60 09 00 B8 0B-00 00
```

Check to see that the Format Index, Frame Index, and Frame Interval are unchanged. You should now also have the MaxVideoFrameSize and MaxPayloeTransferSize fields filled with new values. If the first three mentioned fields are unchanged and the two latter fields are now non-zero, you can then send the packet back to the device using the Commit request.

Table 26-4: VS_Commit_Control GET_CUR Request

Offset	Field	Size	Value	Description
0	Request type	1	**0x21**	Bit 7 Transfer Direction **0 = Host to device** 1 = Device to host Bit 6:5 Type

				0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x01**	SET_CUR
2	Value	2	**0x0200**	High byte: 0x02 = VS_COMMIT Low byte: 0x00 = unused
4	Index	2	**0x0001**	High byte: 0x00 = unused Low byte: 0x01 = Interface
6	Length	2	**0x001A**	Descriptor length. (26d)

If the control pipe does not stall, you have successfully negotiated a format and resolution. You now need to choose one of the alternate interfaces and its corresponding endpoint to receive payload data.

Choosing the Alternate Interface

The camera's Streaming Interface should have given a list of alternate interfaces and corresponding endpoints. Depending on how much bandwidth you want to use, how many devices are already plugged in, you will need to choose one of these endpoints that has a Max Packet Size equal to or greater than a value you wish. For the sake of this book, we assume no other device is plugged in and we have the whole bandwidth. Therefore, we choose the largest Max Packet Size found, that only has 1 transaction per microframe, a value of 00b.

We could choose alternate interface 1:11 since it will transfer a total of 3,000 bytes, with a max packet size of 1,000 having 2 additional transactions per microframe (1,000 * 3 = 3,000, hence the 3,000 value in the MaxPayload TransferSize field in Table 26-1). However, to keep it simple, I am choosing an alternate interface with only one transaction per microframe.

Most Video Streaming interfaces will contain Isochronous transfers, will include the initial interface (alternate setting zero), no corresponding endpoint, and one or more alternate interfaces with corresponding endpoints.

However, a Video Streaming interface may use a bulk transfer method. In this case, there won't be any alternate interfaces, and you use the single interface, which should be the only interface descriptor, and now contains a corresponding endpoint descriptor.

If your camera uses a bulk transfer method, you will need to read a previous part of this book on bulk transfers, since the remaining chapters in Part 5 are for Isochronous transfers.

The example described within this chapter returns alternate interface number 1:6 with an endpoint with a Max Packet Size of 960. Therefore, choose that interface.

Table 26-5: Set Alternate Interface 1.6

Offset	Field	Size	Value	Description
0	Request type	1	**0x01**	Bit 7 Transfer Direction **0 = Host to device** 1 **=** Device to host Bit 6:5 Type **0 = Standard** 1 = Class 2 = Vendor 3 = Reserved Bit 4:0 Recipient 0 = Device **1 = Interface** 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x0B**	SET_INTERFACE
2	Value	2	**0x0006**	Alternate Interface Number
4	Index	2	**0x0001**	Interface Number
6	Length	2	**0x0000**	Sending no data

You should not assume that the alternate interfaces have endpoints listed with sizes in ascending order, even though this example device and most other devices

may have them this way.

Receiving Payload Data

We should now be ready to receive payload data. Once you have set up your Periodical Frame for Isochronous transfers (see the next chapter for EHCI ISO's), make a request for a 960-byte packet, 960 being the Max Packet Size we are using.

The packet received should have a 12-byte header at the beginning, something like the following dump.

```
0x00000000  0C 8C 00 00 00 00 21 2D-9B 4A 55 01
```

Since we are using the Uncompressed format, this 12-byte packet is called the Stream Header for Uncompressed Streams.

Table 26-6: Stream Header

Offset	Field	Size	Value	Description
0	Header Len	1	**0x0C**	Length of this Header
1	Bit Field Header	1	**0x8C**	Bit 7 - End of Header 0 = FALSE **1 = TRUE** Bit 6 - Error Bit **0 = FALSE** 1 = TRUE Bit 5 - Still Image **0 = FALSE** 1 = TRUE Bit 4 - Reserved bit **0 = Reserved** Bit 3 - Source Clock Ref 0 = FALSE **1 = TRUE** Bit 2 - Presentation Time 0 = FALSE **1 = TRUE** Bit 1 - End of Frame **0 = FALSE** 1 = TRUE Bit 0 - Frame Identifier **0** or 1 (toggle bit)
2	Presentation Time Stamp	4	**0x00000000**	Presentation Time Stamp value (if bit 2 above is set)
6	Source Clock Reference	6	**0x01554A9B 2D21**	Source Clock Reference value (if bit 3 above is set)

Bit 0 will toggle at the beginning of the next frame. For example, the first video frame will have this bit zero on every header until the last packet for this frame is received and the next packet for start of the next frame. Then, this bit will be one until this frame is complete. This is so you can tell when the current frame ends and a new one starts.

The Source Clock Reference is used to make sure you keep the timing correct. i.e.: Make sure you display the frames in the correct timing.

Since these fields are used for video display, and it is not my intent to teach video display, I will leave the rest of this to the reader. I will, however, give an example of what I would do.

Stream Buffer

If I were to write a display driver, I would do something similar to the following.

1. Create a set of ten ISO In packets, consecutive in the schedule.
2. Have each one of these point to one of ten buffers each, either consecutively aligned or linked listed.
3. Initiate a pointer to address the first ISO and its corresponding buffer.
4. Start the schedule.
    5. Wait for the current ISO to be processed.
    6. Place a new ISO In packet at the next available location in the schedule.
    7. Copy the contents of the current buffer to your display driver.
    8. Update the addressee to point to the next ISO In packet and buffer.
    9. Go to 5

It is simple, but effective. Please remember that if there is an error with one of the packets, you must ignore it and continue to the next. The packets are not reliant on the completion of the previous packet. If you find that a number of packets are in error, you should stop the stream and report the error(s).

Comments

The intent of this chapter was to enumerate a video camera device, showing the packets you should receive, and the packets you should send to set up and start the video stream. I have left it to the reader to create the display portion of this driver.

The next chapter shows the format and description of EHCI Isochronous Transfer Descriptors.

Also, please see the VideoCam utility on the CDROM for source code examples of how to do the things these chapters describe.

Chapter 27 – The EHCI and ISO Transfers

In this chapter, I will explain the format of an EHCI Isochronous Transfer. Chapter 8 explains in more detail the EHCI Stack, and other parts of the EHCI.

The EHCI Stack

The Periodic List is used for all isochronous and interrupt transfers, and is similar to the UHCI frame list. It has a number of frame list pointers, called Frame List Element Pointers, and point to four types of items that include Isochronous Transfer Descriptors, Queue Heads, Split Transaction Isochronous Transfer Descriptors, and Frame Span Traversal Nodes.

The host controller's scheduler reads the current frame pointer, pointed to by the Frame Index register and executes the frame that it points to. At 1mS, the host controller stops executing that frame, increments the Frame Index register, wrapping to 0 at the last entry, and starts executing the next frame. This means, if we have a frame size of 1,024 entries, each frame will be executed approximately once a second, every 1.024 seconds to be exact, or at 1 Hz. The current frame entry to read is calculated by taking the PERIODICLISTBASE register, using the CTRLDSSEGMENT register for 64-bit addressing, and adds the FRINDEX dword index.

For more on this technique, please see Chapter 8 starting with Table 8-1. From this point on, I will assume you know what a frame stack is and know how to insert ISO Transfers.

ISO Transfer Descriptors

The Isochronous Transfer Descriptor is used for audio, video, and other continuous data transfers.

The ISO Transfer is inserted into the Periodical list and depending on the settings of that list, executed up to eight times a second, using micro-frames.

Figure 27-1: EHCI: ISO Transfer Descriptor

bits 31:5				4:3	2:1	0
Next Link Pointer				0	Type	T
31:28	27:16	15	14:12	11:0		
Status	Transaction 0 Len	IOC	Page	Transaction 0 Offset		
Status	Transaction 1 Len	IOC	Page	Transaction 1 Offset		
Status	Transaction 2 Len	IOC	Page	Transaction 2 Offset		
Status	Transaction 3 Len	IOC	Page	Transaction 3 Offset		
Status	Transaction 4 Len	IOC	Page	Transaction 4 Offset		
Status	Transaction 5 Len	IOC	Page	Transaction 5 Offset		
Status	Transaction 6 Len	IOC	Page	Transaction 6 Offset		
Status	Transaction 7 Len	IOC	Page	Transaction 7 Offset		

31:12	11:8	7	6:0
Buffer Pointer (Page 0)	Endpt	R	Dev Addr
Buffer Pointer (Page 1)	I/O	Max Packet Size	
Buffer Pointer (Page 2)	Reserved		Mult
Buffer Pointer (Page 3)	Reserved		
Buffer Pointer (Page 4)	Reserved		
Buffer Pointer (Page 5)	Reserved		
Buffer Pointer (Page 6)	Reserved		

After the controller has executed this transfer descriptor it will modify two fields each of the first eight transaction entries.

Table 27-1: EHCI: ISO Transfer Descriptor

Transfer Descriptor Definition		
Name	HC Access	Description
T	RO	**Terminate**. If set to 1, this tells the controller that this transfer descriptor is the last in the linked list, and that the Next Link Pointer field is not valid.
Next Link Pointer	RO	**Next Link Pointer**. This is bits 31:5 of the link pointer, pointing to the next transfer descriptor in the list. Bits 4:0 are assumed zero. To get this field, read the whole dword and mask off the lower five bits.
TransX Offset	RO[1]	**Transaction (0-7) Offset**. This is the offset within the Buffer Pointer (0-7) of the start of the buffer for this transaction.
PageX	RO[1]	**Page (0-7)**. This is an index into the Buffer Pointer (0-7) entries for this transaction.
IOC	RO	**Interrupt on Complete**. This bit tells the controller to fire an interrupt at the next interrupt threshold.

<table>
<tr><td>TransX Length</td><td>R/W</td><td colspan="2">Transaction (0-7) Length. Before this transaction is executed, this field holds the count of bytes to transfer. After execution, this field will hold the actual bytes transferred.</td></tr>
<tr><td rowspan="5">Status</td><td rowspan="5">R/W</td><td>Bit</td><td>Description</td></tr>
<tr><td>3</td><td>Active. Set to 1 to enable execution of this transaction. The controller will clear this bit after execution of this transfer descriptor.</td></tr>
<tr><td>2</td><td>Data Buffer Error. Set to 1 by the controller if an overrun or underrun is encountered.</td></tr>
<tr><td>1</td><td>Babble Detected. Set to 1 by the controller if a babble was detected.</td></tr>
<tr><td>0</td><td>Transaction Error (XactErr). Set to 1 by the controller if the host did not receive a valid response from the device.</td></tr>
<tr><td>Buffer Pointer</td><td>RO</td><td colspan="2">Buffer Pointer (0-6). This is a physical pointer to a 4k page of memory, aligned to 4k. The transaction offset field points to somewhere within this page, usually zero.</td></tr>
<tr><td>Endpt</td><td>RO</td><td colspan="2">End Point Number. This is the End Point number for this transaction.</td></tr>
<tr><td>R</td><td>RO</td><td colspan="2">Reserved. Zero.</td></tr>
<tr><td>Device Address</td><td>RO</td><td colspan="2">Device Address. The is the Device Address for the transaction.</td></tr>
<tr><td>I/O</td><td>RO</td><td colspan="2">Direction I/O. Set to 1 for an In packet, 0 for and Out.</td></tr>
<tr><td>MPS</td><td>RO</td><td colspan="2">Max Packet Size. Max Packet Size for a transaction. Used with the Milti field, micro-frames, and In packets.</td></tr>
<tr><td>Reserved</td><td>RO</td><td colspan="2">Reserved. Zero.</td></tr>
<tr><td>Multi</td><td>RO</td><td colspan="2">Multiple. The number of transactions that should be executed per micro-frame. Values 1 to 3 are allowed.</td></tr>
<tr><td colspan="4">[1]If I/O indicates an OUT, these fields could be modified by the controller.</td></tr>
</table>

Creating an ISO Transfer

The EHCI Controller has eight micro-frames per every single frame. In other words, each millisecond, a single frame is executed. However, within this frame, eight micro-frames are executed every 125uS. For more information on this, look back in Chapter 7, Tables 7-14 through 7-17.

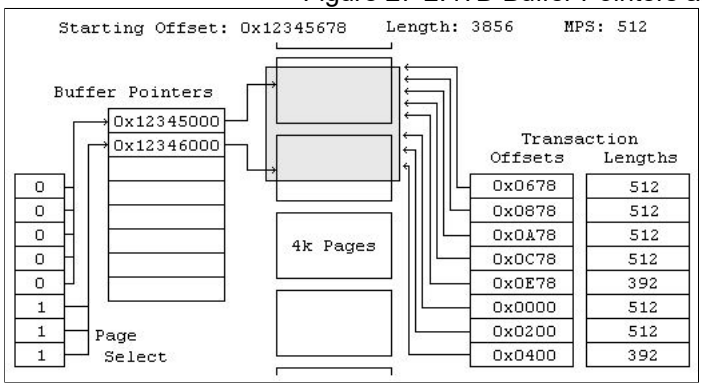
Since there are eight micro-frames per millisecond, the ISO Transfer Descriptor (iTD) has room for eight transactions. It is as if we are stuffing eight full-speed transactions that would take eight milliseconds into a single millisecond high-speed frame. Hence, the High-speed name.

With these eight transactions, we can move up to eight times the max packet size of bytes in a single frame.

Please note that some devices, such as the camera in the previous two chapters, may not allow all eight transactions. For example, that camera only allows a total of up to three per frame.

The figure below shows a buffer of 3,856 bytes to be transmitted, with the buffer crossing a 4k boundary, and a Max Packet Size of 512. The buffer location and size is described in the Transaction Length, Offset, Page Select, and Buffer Pointer fields of the iTD.

Figure 27-2: iTD Buffer Pointers and Lengths



Since the device can only transfer up to 512 bytes per micro-frame, the first five transfers will use the buffer in the first 4k page, with the last three in the second 4k page. Therefore, we only need to use two of the Buffer Pointer fields. The first five Page Select fields will have an index of zero, with the last three having an index of one.

The first five Transaction Offsets and Lengths will make sure that a 4k boundary is not crossed, with the remaining three fields transferring the remaining buffer.

Now the controller knows where to store the 3,856 bytes it is now to retrieve from the device. The remaining fields in the iTD are used for device and Endpoint identification, the I/O bit for direction, and the Multi field for count of transactions per micro-frame.

The camera used in the previous two chapters allows two transactions per microframe, when using the chosen example enpdoint. This count is retrieved from the MaxPacketSize field of the Endpoint Descriptor.

Be sure to fill in the remaining fields with the appropriate values. The Max Packet Size field must not exceed 0x400 even if the device's max packet size does.

Wrap Up

Please see Chapter 8 for more information on the EHCI's periodical frame.

Remember that ISO transfers do not return error information other than the four bits in the status field of each transaction. Since ISO transfers are mostly used for audio and visual transfers, if a single transfer is skipped or missed, the human ear and/or eye should not be able to notice.

Appendix A - Disk Contents

The included disc contains all the mentioned source code, utilities, and other mentioned items within this book. The following is a layout of the included disc.

readme.txt main readme file
main main folder for all books in this series
freedos.img 1.44 floppy image of a bootable DOS
mputimg.exe Simply copied from the MPUTIMG directory
main\usb main folder for this book
main\usb\misc The misc. files folder
attiny2313.fpc The FreePCB file that is included with AppendixQ
pcl_demo0.bin The printer binary file for chapter 20
pcl_demo1.bin The printer binary file for chapter 20
main\usb\utils Utilities folder for this book
csdpmi5b.zip DMPI driver for DOS and FreeDOS
main\usb\utils\detcntlr The Detect Controller Folder
detcntlr.exe Executable file for Detect Controller app
detcntlr.cpp C++ source file for Detect Controller app
detcntlr.h C++ header file for Detect Controller app
main\usb\utils\gdevdesc The Get Device Descriptor folder
common.h Common header file used in all util's below
gd_ohci.exe Executable file for Get Descriptor (OHCI)
gd_ohci.cpp C++ source file for Get Descriptor (OHCI)
gd_ohci.h C++ header file for Get Descriptor (OHCI)
gd_uhci.exe Executable file for Get Descriptor (UHCI)
gd_uhci.cpp C++ source file for Get Descriptor (UHCI)
gd_uhci.h C++ header file for Get Descriptor (UHCI)
gd_ehci.exe Executable file for Get Descriptor (EHCI)
gd_ehci.cpp C++ source file for Get Descriptor (EHCI)
gd_ehci.h C++ header file for Get Descriptor (EHCI)
gd_xhci.exe Executable file for Get Descriptor (xHCI)
gd_xhci.cpp C++ source file for Get Descriptor (xHCI)
gd_xhci.h C++ header file for Get Descriptor (xHCI)
notes.txt Notes on the source in this utility
main\usb\utils\hidparse The HID Parser folder
hidparse.exe Executable file for hidparser app
hidparse.cpp C++ source file for hidparser app
hidparse.h C++ header file for hidparser app
report0.bin example file to parse with hidparser
report1.bin example file to parse with hidparser
report2.bin example file to parse with hidparser
report3.bin example file to parse with hidparser
report4.bin example file to parse with hidparser
main\usb\utils\include The Include folder
ctype.h include file for utilities sources
ehci.h include file for utilities sources
ohci.h include file for utilities sources
pci.h include file for utilities sources
uhci.h include file for utilities sources
usb.h include file for utilities sources
xhci.h include file for utilities sources
main\usb\utils\mputimg The image installer
mputimg.c C++ source file for mputimg app
mputimg.exe install the freedos.img to a floppy disk
mputimg.h C++ header file for mputimg app

wputimg.c	C++ source file for wputimg app (WinXP only)
wputimg.exe	install the freedos.img to a floppy disk (WinXP only)
wputimg.h	C++ header file for wputimg app (WinXP only)
readme.txt	Notes on the source in this utility
main\usb\utils\videocam	The image installer
heap.h	include file for this utility
timer.h	include file for this utility
videocam.c	C++ source file for videocam app
videocam.exe	Get the information from the Video Camera
videocam.h	C++ header file for videocam app

Installing the Source Code on your computer

To install the source and applications from the included CDROM to your computer, I suggest you create a directory of the following:

C:\FYSOS\main\usb\

To install the files from the CDROM to your hard drive, copy the whole contents of the \main\ directory to your hard drive. This will install all the source code and application data you will need for this book.

If you are using DOS, use the following:

```
C:\FYSOS\xcopy D:\FYSOS\MAIN\USB C:\FYSOS\MAIN /E /I /-Y /D<enter>
```

Where the C: drive is your host drive, and the D: drive is the CDROM drive.

The above command will only replace existing files if the source file is a newer file. There is no worry about a file being overwritten by an older version if you have installed another set of files from another book or resource in this series.

Finding the Specification Files

Since most of the specifications mentioned in this book are not to be distributed by anyone outside the company that owns the specification, I have not included them here. However, I will include information on where you can find them.

The following URL's point to pages and/or specifications used through out this book. Please note that the host of the URL may or may not be legally entitled to host these files. Download at your own risk.

Most of the files are available from the USB-IF web-site:
http://www.usb.org/developers

Rev 1.1d of the UHCI specification (uhci11d.pdf)
ftp://ftp.netbsd.org/pub/NetBSD/misc/blymn/uhci11d.pdf

Rev 1.0a of the OHCI specification (ohci.pdf)
ftp://ftp.compaq.com/pub/supportinformation/papers/hcir1_0a.pdf

Rev 0.95 of the EHCI specification (ehci-r095.pdf)
http://www.csit-sun.pub.ro/~cpop/Documentatie_SM/Standarde_magistrale/USB/

Rev 0.96 of the EHCI specification (ehci-r096.pdf)
http://www.csit-sun.pub.ro/~cpop/Documentatie_SM/Standarde_magistrale/USB/

Rev 1.0 of the EHCI specification (ehci-r10.pdf)
http://www.intel.com/technology/usb/download/ehci-r10.pdf

Rev 1.1 of the EHCI specification (ehciv1-1_addendum.pdf)
http://download.intel.com/technology/usb/EHCIv1-1_Addendum.pdf

Rev 1.1 of the xHCI specification (xhci_1_0.pdf)
http://www.intel.com/content/www/us/en/io/universal-serial-bus/extensible-host-controler-interface-usb-xhci.html

Rev 1.11 of the HID specification (hid1_11.pdf)
http://www.usb.org/developers/devclass_docs/HID1_11.pdf

Rev 1.12 of the HID Usage Tables (hut1_12.pdf)
www.usb.org/developers/devclass_docs/Hut1_11.pdf

Multi-Media commands - 6 rev 02f (mmc6r02f.pdf)
http://www.t10.org/

Mass Storage Compliance Specification (msc-compliance-0_9a.pdf)
http://www.usb.org/developers/compliance/

SCSI Primary Commands - 4 rev 18 (spc4r18.pdf)
http://www.t10.org/

Rev 2.0 of the USB specification (usb_20.pdf)
http://www.usb.org/developers/docs/usb_20_101111.zip

Rev 3.0 of the USB specification (usb_30.pdf)
http://www.usb.org/developers/docs/usb_30_spec_092911.zip

Language ID's (usb_langids.pdf)
http://www.usb.org/developers/docs/USB_LANGIDs.pdf

Rev 1.1 MSD Control/Block/Interrupt (usb_msc_cbi_1.1.pdf)
http://www.usb.org/developers/devclass_docs/usb_msc_cbi_1.1.pdf

Rev 1.3b Mass Storage Class (usb_msc_overview_1.3b.pdf)
http://www.usb.org/developers/devclass_docs/usb-msc-overview-1.3b.pdf

Rev 1.1 of the USB specification (usb11.pdf)
http://esd.cs.ucr.edu/webres/usb11.pdf

Rev 1.0 of MSD Bulk transport (usbmassbulk_10.pdf)
http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf

Rev 1.0 of MSD UFI (Floppy) (usbmass-ufi10.pdf)
http://www.usb.org/developers/devclass_docs/usbmass-ufi10.pdf

Rev 1.1 of the USB Printing Devices (usbprint11.pdf)
http://www.usb.org/developers/devclass_docs/usbprint11.pdf

The VIA VT83C572 UHCI controller specs (vt83c572.pdf)
http://www.alldatasheet.com/datasheet-pdf/pdf/98317/ETC/VT83C572.html

Rev 1.1 of the WHCI (WirelessUSB_Specification_r11.pdf)
http://www.usb.org/developers/wusb/docs/

Device Class Documentation
http://www.usb.org/developers/devclass_docs

Video Class Documentation
http://www.usb.org/developers/devclass_docs
USB_Video_Class_1_1.zip
USB_Video_Class_1_5.zip

Interface Association Descriptor
www.usb.org/developers/docs/InterfaceAssociationDescriptor_ecn.pdf
http://www.usb.org/developers/whitepapers/iadclasscode_r10.pdf

USB Device Firmware Upgrade Specification, v1.1
www.usb.org/developers/devclass_docs/DFU_1.1.pdf

Appendix B - Included Utilities/Source Code

The following appendix will describe and show an example output of some of the utilities included on the disc.

Important notice: Most of the utilities on the disc will only work in a true DOS environment with a third party DOS Protected Mode Interface (DPMI) driver installed. They will not work in a DOS session under a Windows environment.

Included on the disc is a FreeDOS (www.freedos.org) 1.44meg floppy image and a small utility to write it to a disk. You can then copy any of these utilities and/or your own code to the floppy, reboot in your test machine, and test your code. The image has a DPMI driver already installed.

If you have a test machine with a free disk partition, you can get the full version of FreeDOS from www.freedos.org and install on that partition. Then find the csdpmi5b.zip file in the UTILS directory of the included disc. Unzip it using the shareware pk250dos.exe utility (Google for pk250dos.exe), and follow the directions to install the DPMI driver included. Now set your BIOS to boot to the FreeDOS partition you just created, and you can start to run these utilities and any code that you create to test your knowledge.

Obtaining a C/C++ Compiler

Unless otherwise noted, most of the C source code is compiled with the GCC ported C Compiler named DJGPP found at http://www.delorie.com/djgpp/. Go to the “Zip Picker” at http://www.delorie.com/djgpp/zip-picker.html and select a FTP site, “Build and Run programs with DJGPP”, the operating system you will use (Windows2000/XP), choose where you want to read the online documentation, then make sure that “C”, “C++”, “Objective C”, and “Assembler” are checked. Then choose the remaining items as you would like and click on “Tell me which files I need”. This will give you a list of files, with links, and instructions on how to install and configure the compiler.

Obtaining an Assembler

You will not need an assembler for any of the source files included with this disc. However, you may want to create your own x86 assembly source files and need to assemble them. You may obtain a free x86 assembler from the author's website at http://www.fysnet.net/newbasic.htm. This assembler is free for non-profit use and has documentation and example source code.

PKSFX(R) FAST! Self Extract Utility - v2.50

The pk250dos.exe file mentioned before is an installer by PKWARE that will install PKZIP, PKUNZIP, and a few other utilities for use with zipping and unzipping .zip files, and is shareware. It is used to archive files.

This utility is not included on the cdrom since PKWare has changed ownership. You can still get the old version by using your favorite web search engine and search for pk250dos.exe.

Create an empty directory on your hard drive, copy the pk250dos.exe file to it, change to that directory and run the installer.

The self-extracting installer should extract the utilities needed and have a similar display as the following.

```
PKSFX (R)   FAST!   Self Extract Utility   Version 2.50   03-01-1999
Copr. 1989-1999 PKWARE Inc.  All Rights Reserved.  Shareware Version
PKSFX Reg. U.S. Pat. and Tm. Off.

Searching EXE: C:/PKWARE/PK250DOS.EXE
  Inflating: ADDENDUM.TXT  -AV
  Inflating: HINTS.TXT     -AV
  Inflating: LICENSE.TXT   -AV
  Inflating: MANUAL.TXT    -AV
  Inflating: OMBUDSMN.ASP  -AV
  Inflating: ORDER.TXT     -AV
  Inflating: PKUNZIP.EXE   -AV
  Inflating: PKUNZJR.COM   -AV
  Inflating: PKZIP.EXE     -AV
  Inflating: PKZIPFIX.EXE  -AV
  Inflating: README.TXT    -AV
  Inflating: WHATSNEW.TXT  -AV
  Inflating: ZIP2EXE.EXE   -AV

Authentic files Verified!   # PKW655
PKWARE Inc.
```

The PKUNZIP.EXE file is used to unzip an archive with the .zip extension. Be sure to set your PATH= environment variable to point to this directory.

To view the contents of the csdpmi5b.zip archive, use the following command line.

C:\>PKUNZIP CSDPMI5B.ZIP /vbm<*enter*>

This will display the files included in the archive. Example listed below.

```
PKUNZIP (R)    FAST!    Extract Utility    Version 2.50   03-01-1999
Copr. 1989-1999 PKWARE Inc.  All Rights Reserved.  Shareware Version
PKUNZIP Reg. U.S. Pat. and Tm. Off.

. Pentium  class CPU detected.
. XMS version 2.00 detected.
. DPMI version 0.90 detected.

Searching ZIP: CSDPMI5B.ZIP

Length  Method   Size  Ratio    Date     Time   Name
------  ------   ----- -----    ----     ----   ----
     0  Stored       0   0%  10-22-2000  16:02  BIN/
     0  Stored       0   0%  10-22-2000  16:02  MANIFEST/
 20125  DeflatX  12835  37%  10-22-2000  12:55  BIN/CWSDPMI.EXE
 19335  DeflatX  12264  37%  10-22-2000  12:54  BIN/CWSDPR0.EXE
  8167  DeflatX   3450  58%  10-18-2000  20:54  BIN/CWSDPMI.DOC
 13124  DeflatX   8429  36%  10-22-2000  12:54  BIN/CWSPARAM.EXE
  6176  DeflatX   2641  58%  10-22-2000  09:38  BIN/CWSPARAM.DOC
    50  Stored      50   0%  06-06-1999  22:19  MANIFEST/CSDPMI5B.VER
```

```
  151  DeflatX     72  53%  10-08-2000  19:39  MANIFEST/CSDPMI5B.MFT
 21504  DeflatX  13731  37%  10-22-2000  12:55  BIN/CWSDSTUB.EXE
------          ------  ---                    -------
 88632           53472  40%                         10
```

To unzip the files using the directory structure, use the following command line.

C:\>PKUNZIP CSDPMI5B.ZIP -d<*enter*>

DetCntlr -- Detect Controllers - v1.00.00

This utility was created specifically for this book and shows how to find USB controllers on the PCI bus. It shows how to program the PCI without using the BIOS or outside libraries. C source code is included.

Running detcntlr.exe as-is will produce something like the following example.

C:\>DETCNTLR<*enter*>

```
Detect USB Controllers on a PCI Bus.  v1.00.00
Forever Young Software  -- Copyright 1984-2014

Found a USB compatible device entry.
Bus =  0, device =  7, function = 2 IO Base: 0x0000D401, IRQ:   10

Found a USB compatible device entry.
Bus =  0, device =  8, function = 0 IO Base: 0xE1005000, IRQ:   11

Found a USB compatible device entry.
Bus =  0, device =  8, function = 1 IO Base: 0xE1000000, IRQ:   11

Found a USB compatible device entry.
Bus =  0, device =  8, function = 2 IO Base: 0xE1001000, IRQ:   11

Found a USB compatible device entry.
Bus =  0, device =  8, function = 3 IO Base: 0xE1002000, IRQ:   11
```

Running detcntlr.exe with a command line parameter of /type will produce something like the following example.

C:\>DETCNTLR /type<*enter*>

```
Detect USB Controllers on a PCI Bus.  v1.00.00
Forever Young Software -- Copyright 1984-2014

Found a USB compatible device entry. (UHCI)
Bus =  0, device =  7, function = 2 IO Base: 0x0000D401, IRQ:   10

Found a USB compatible device entry. (OHCI)
Bus =  0, device =  8, function = 0 IO Base: 0xE1005000, IRQ:   11

Found a USB compatible device entry. (OHCI)
Bus =  0, device =  8, function = 1 IO Base: 0xE1000000, IRQ:   11

Found a USB compatible device entry. (OHCI)
Bus =  0, device =  8, function = 2 IO Base: 0xE1001000, IRQ:   11

Found a USB compatible device entry. (OHCI)
Bus =  0, device =  8, function = 3 IO Base: 0xE1002000, IRQ:   11
```

The difference is that the /type parameter will tell this utility to print the type of controller found, UHCI, OHCI, EHCI, or xHCI.

GDevDesc -- Get Device Descriptor
GD_UHCI - UHCI: Get Device Descriptor - v1.00.00
GD_OHCI - OHCI: Get Device Descriptor - v1.10.00
GD_EHCI - EHCI: Get Device Descriptor - v1.10.00
GD_XHCI - xHCI: Get Device Descriptor - v1.00.00

These utilities will find all the USB controllers, respectively of each type above, then communicate with the controller to find any attached devices on each of the controllers root hub ports. If an attached device is found, it will display the device's Device Descriptor.

These utilities show how to reset the controller and root hub, set up the controller and root hub for communication, set up a stack frame, and communicate with the attached device through the Default Control Endpoint.

These utilities will only communicate with the device attached to the root hub port. If you have an external hub attached, these utilities will not get the device descriptor of any devices down stream of the external hub, though they will return the device descriptor of the external hub itself.

These utilities assume you have an Intel compatible 486 or higher processor with the CPUID and RDTSC instructions. Please verify that your test machine meets this qualification before you use any of these utilities.

The following is an example output produced by one of these utilities.

```
GD_OHCI -- OHCI: Get Device Descriptor  v1.10.00
Forever Young Software       Copyright 1984-2014

Found a USB compatible device entry: Bus = 0, device = 7, function = 2
  Found UHCI controller at 0x0000D400
Found a USB compatible device entry: Bus = 0, device = 8, function = 0
  Found OHCI controller at 0xE1005000
Found Device Descriptor:
              len: 18
             type: 1
          version: 1.10
            class: 0
         subclass: 0
         protocol: 0
   max packet size: 8
        vendor id: 0x04FC
       product id: 0x0003
      release ver: 01.01
 manufacture index: 0 (index to a string)
     product index: 2
      serial index: 0
 number of configs: 1
Found a USB compatible device entry: Bus = 0, device = 8, function = 1
Found OHCI controller at 0xE1000000
Found a USB compatible device entry: Bus = 0, device = 8, function = 2
Found OHCI controller at 0xE1001000
Found a USB compatible device entry: Bus = 0, device = 8, function = 3
Found OHCI controller at 0xE1002000
```

HIDParser -- Parse a HID Report Descriptor - v1.00.00

This utility will take a given HID Report Descriptor, parse and display each item.

This utility was compiled for a 32-bit DPMI operating system such as a WinXP DOS session. Source code is included so that you may compile it for a different platform.

Running HIDParse.exe with a filename as a parameter will produce something like the following example.

C:\>HIDParse input.bin<*enter*>

```
HIDParser -- Parse a HID Report Descriptor  v1.00.00
Forever Young Software  Copyright 1984-2014

05 01         Usage Page (Generic Desktop)
09 02         Usage (Mouse)
A1 01         Collection (Application)
09 01           Usage (Pointer)
A1 00           Collection (Physical)
05 09             Usage Page (Button)
19 01             Usage min (1)
29 03             Usage max (3)
15 00             Logical Min (0)
25 01             Logical Max (1)
75 01             report size (1)
95 03             report count (3)
81 02             Input (Data,Var,Abs,NoW,Lin,PState,NoNull,Bit)
75 05             report size (5)
95 01             report count (1)
81 01             Input (Cons,Array,Abs,NoW,Lin,PState,NoNull,Bit)
05 01             Usage Page (Generic Desktop)
09 30             Usage (X)
09 31             Usage (Y)
09 38             Usage (Wheel)
15 81             Logical Min (-127)
25 7F             Logical Max (127)
75 08             report size (8)
95 03             report count (3)
81 06             Input (Data,Var,Rel,NoW,Lin,PState,NoNull,Bit)
C0              End Collection
09 3C           Usage (Motion Wakeup)
05 FF           Usage Page (Reserved)
09 01           Usage (Unsupported Usage Page)
15 00           Logical Min (0)
25 01           Logical Max (1)
75 01           report size (1)
95 02           report count (2)
B1 22           Feature (Data,Var,Abs,NoW,Lin,NoPre,NoNull,Bit)
75 06           report size (6)
95 01           report count (1)
B1 01           Feature (Cons,Array,Abs,NoW,Lin,PState,NoNull,Bit)
C0            End Collection
```

MPutImg -- Write Disk Image to Floppy Drive - v0.10.01

This utility was created for use with images created for the Bochs emulator found at http://bochs.sourceforge.net/. It will take a filename of a floppy disk image and write it to a floppy disk. With this you can take the disk image you have been emulating in Bochs, write it to a floppy disk, and boot it on physical hardware.

C source code is included. You can get the latest version and other utilities of this kind at http://www.fysnet.net/mtools.htm.

I am including it here so that you can write the FreeDOS image included on the disc to a floppy disk for use with your tests.

To use it, you simply do the following command line to write the freedos.img image file to the floppy disk in drive A:.

C:\>MPutImg freedos.img a:<*enter*>

Running mputimg.exe as-is with no command line parameters, as listed below, will produce something like the example below.

C:\>MPutImg<*enter*>

```
MTOOLS   Put Image  v00.10.00    Forever Young Software 1984-2014

Filename [floppy.img]: freedos.img

Found 1.44M image with 2880 sectors.  Continue? [Y]
Please choose a drive letter (A, B, C, D, ...) [A]:

      Writing file:  freedos.img
        Cylinders:  80
            Sides:  2
    Sectors/Track:  18
    Total Sectors:  2880
             Size:  1.44M
 Is this correct? (Y or N) [Y]:

Writing cyl 12 (of 80), side 1
```

You can use the /v command line parameter to skip all of the prompts and simply write the image to the disk.

C:\>MPutImg freedos.img a: /v <*enter*>

Appendix C - List of Tables and Figures

Table 2-1: Format of PCI Config Space
Table 2-2: Format of PCI Address Register
Table 2-3: The PCI PM Status and Control Register
Table 3-1: The UHCI controllers I/O registers
Table 3-2: The UHCI Command Register
Table 3-3: The UHCI Status Register
Table 3-4: The UHCI Interrupt Enable Register
Table 3-5: The UHCI Frame Number Register
Table 3-6: The UHCI Frame Base Register
Table 3-7: The UHCI Start of Frame Register
Table 3-8: The UHCI Port Status Register
Table 3-9: The VIA VT83C572 Misc. Control Registers
Table 4-1: The UHCI Frame Pointer
Table 4-2: The UHCI Transfer Descriptor
Table 4-3: The UHCI Transfer Descriptor Definition
Table 4-4: The UHCI Queue
Table 5-1: The OHCI controller's I/O registers
Table 5-2: The OHCI HcRevision Register
Table 5-3: The OHCI HcControl Register
Table 5-4: The OHCI HcCommandStatus Register
Table 5-5: The OHCI HcInterruptStatus Register
Table 5-6: The OHCI HcInterruptEnable Register
Table 5-7: The OHCI HcInterruptDisable Register
Table 5-8: The OHCI HcHCCA Register
Table 5-9: The OHCI Host Controller Communications Area
Table 5-10: The OHCI HcPeriodCurrentED Register
Table 5-11: The OHCI HcCurrentHeadED Register
Table 5-12: The OHCI HcControlCurrentED Register
Table 5-13: The OHCI HcBulkHeadED Register
Table 5-14: The OHCI HcBulkCurrentED Register
Table 5-15: The OHCI HcDoneHead Register
Table 5-16: The OHCI HcFmInterval Register
Table 5-17: The OHCI HcFrameRemaining Register
Table 5-18: The OHCI HcFrameNumber Register
Table 5-19: The OHCI HcPeriodicStart Register
Table 5-20: The OHCI HcLSThreshold Register
Table 5-21: The OHCI HcRhDescriptorA Register
Table 5-22: The OHCI HcRhDescriptorB Register
Table 5-23: The OHCI HcRhStatus Register
Table 5-24: The OHCI HcRhPortStatus Register
Table 5-25: OHCI Port Power Options
Table 6-1: OHCI Endpoint Descriptor Format
Table 6-2: OHCI Endpoint Descriptor Definition
Table 6-3: OHCI Transfer Descriptor Format
Table 6-4: OHCI Transfer Descriptor Definition
Table 7-1: EHCI Capability Registers
Table 7-2: The EHCI CAPLENGTH Register
Table 7-3: EHCI Reserved
Table 7-4: The EHCI HCIVERSION Register
Table 7-5: The EHCI HCSPARAMS Register
Table 7-6: The EHCI HCCPARAMS Register
Table 7-7: The EHCI HCSP-PORTROUTE Register
Table 7-8: The EHCI Operational Registers
Table 7-9: The EHCI Command Register

Table 7-10: EHCI: Size of Frame List
Table 7-11: EHCI: Maximum Interrupt Interval
Table 7-12: The EHCI Status Register
Table 7-13: The EHCI Interrupt Enable Register
Table 7-14: The EHCI Frame Index Register
Table 7-15: EHCI: Frame Index Register: 1024
Table 7-16: EHCI: Frame Index Register: 512
Table 7-17: EHCI: Frame Index Register: 256
Table 7-18: The EHCI Ctrl/Data Structure Segment Register
Table 7-19: The EHCI Periodic List Address Register
Table 7-20: The EHCI Async List Address Register
Table 7-21: The EHCI Configure Flag Register
Table 7-22: The EHCI Port Status and Control Register
Table 7-23: EHCI: Port Indicator Color
Table 8-1: Frame List Link Pointer
Table 8-2: EHCI: Frame List Item Type
Table 8-3: EHCI: Queue Head Transfer Descriptor
Table 8-4: EHCI: Transfer Descriptor
Table 8-5: EHCI: Queue Head
Table 8-6: EHCI: Queue Head Definition
Table 9-1: xHCI Capability Registers
Table 9-2: The xHCI CAPLENGTH Register
Table 9-3: Reserved
Table 9-4: The xHCI HCIVERSION Register
Table 9-5: The xHCI HCSPARAMS1 Register
Table 9-6: The xHCI HCSPARAMS2 Register
Table 9-7: The xHCI HCSPARAMS3 Register
Table 9-8: The xHCI HCCPARAMS1 Register
Table 9-9: The xHCI Doorbell Offset Register
Table 9-10: The xHCI Runtime Register Base Offset Register
Table 9-11: The xHCI HCCPARAMS2 Register
Table 9-12: Format of xHCI Extended Capabilities List Entry
Table 9-13: Valid Extended Capabilities List IDs
Table 9-14: Format of xHCI USB Legacy Support: First DWORD
Table 9-15: Format of xHCI USB Legacy Support: Second DWORD
Table 9-16: Format of xHCI USB Protocol Support
Table 9-17: The xHCI Operational Registers
Table 9-18: The xHCI Command Register
Table 9-19: The xHCI Status Register
Table 9-20: The xHCI Page Size Register
Table 9-21: The xHCI Device Notification Control Register
Table 9-22: The xHCI Command Ring Control Register
Table 9-23: The xHCI Device Context Base Address Array Register
Table 9-24: The xHCI Configure Register
Table 9-25: xHCI Memory Mapped Address Space Usage
Table 9-26: The xHCI Port Register Set
Table 9-27: The xHCI Port Status and Control Register
Table 9-28: The xHCI Port Speed values
Table 9-29: xHCI: Port Indicator Control
Table 9-30: The xHCI PM Port Status and Control Register (USB2)
Table 9-31: xHCI: L1 Status Field
Table 9-32: The xHCI PM Port Status and Control Register (USB3)
Table 9-33: The xHCI PORTLI Register (USB2)
Table 9-34: The xHCI PORTLI Register (USB3)
Table 9-35: The xHCI PROTHLPMC Register (USB2)
Table 9-36: The xHCI PROTHLPMC Register (USB3)

Table 9-37: The xHCI Host Runtime Register Set
Table 9-38: The xHCI Interrupter Register Set
Table 9-39: The xHCI Interrupter Management Register
Table 9-40: The xHCI Interrupter Moderation Register
Table 9-41: The xHCI Event Ring Segment Table Size Register
Table 9-42: xHCI Event Ring Segment Table Base Address Register
Table 9-43: The xHCI Event Ring Dequeue Pointer Register
Table 9-44: The xHCI Command Doorbell Register
Table 9-45: The xHCI Device Slot Doorbell Register
Table 9-46: Doorbell Target Notification values
Table 10-1: Data Structure Memory Requirements
Table 10-2: xHCI Slot Context: DWORD 0
Table 10-3: Route String
Table 10-4: xHCI Slot Context: DWORD 1
Table 10-5: xHCI Slot Context: DWORD 2
Table 10-6: xHCI Slot Context: DWORD 2: TT Think Time
Table 10-7: xHCI Slot Context: DWORD 3
Table 10-8: xHCI Slot Context: DWORD 3: Slot State
Table 10-9: xHCI Endpoint Context: DWORD 0
Table 10-10: xHCI Endpoint Context: DWORD 0: EP State
Table 10-11: xHCI Endpoint Context: DWORD 1
Table 10-12: xHCI Endpoint Context: DWORD 1: EP Type
Table 10-13: xHCI Endpoint Context: DWORDs 2 & 3
Table 10-14: xHCI Endpoint Context: DWORD 4
Table 10-15: xHCI Normal TRB Format: DWORD 0 & 1
Table 10-16: xHCI Normal TRB Format: DWORD 2
Table 10-17: xHCI Normal TRB Format: DWORD 3
Table 10-18: xHCI Setup Stage TRB Format: DWORD 0
Table 10-19: xHCI Setup Stage TRB Format: DWORD 1
Table 10-20: xHCI Setup Stage TRB Format: DWORD 2
Table 10-21: xHCI Setup Stage TRB Format: DWORD 3
Table 10-22: Control Transfer Type
Table 10-23: xHCI Data Stage TRB Format: DWORD 0 & 1
Table 10-24: xHCI Data Stage TRB Format: DWORD 2
Table 10-25: xHCI Data Stage TRB Format: DWORD 3
Table 10-26: xHCI Status Stage TRB Format: DWORD 0 & 1
Table 10-27: xHCI Status Stage TRB Format: DWORD 2
Table 10-28: xHCI Status Stage TRB Format: DWORD 3
Table 10-29: xHCI Command Completion TRB Format: DWORD 0 & 1
Table 10-30: xHCI Command Completion TRB Format: DWORD 2
Table 10-31: xHCI Command Completion TRB Format: DWORD 3
Table 10-32: xHCI Event Ring Segment Table Entry: DWORD 0 & 1
Table 10-33: xHCI Event Ring Segment Table Entry: DWORD 2
Table 10-34: xHCI Event Ring Segment Table Entry: DWORD 3
Table 11-1: Device Descriptor
Table 11-2: Device Request Packet
Table 11-3: String Language Descriptor
Table 11-4: String Descriptor
Table 11-5: Configuration Descriptor
Table 11-6: Interface Descriptor
Table 11-7: Endpoint Descriptor
Table 12-1: Device Descriptor
Table 12-2: Device Request Packet
Table 12-3: String Language Descriptor
Table 12-4: String Descriptor
Table 12-5: Configuration Descriptor

Table 12-6: Interface Descriptor
Table 12-7: Endpoint Descriptor
Table 13-1: Device Descriptor
Table 13-2: Device Request Packet
Table 13-3: String Language Descriptor
Table 13-4: String Descriptor
Table 13-5: Configuration Descriptor
Table 13-6: Interface Descriptor
Table 13-7: Endpoint Descriptor
Table 14-1: Corresponding Cable Wiring Colors
Table 14-2: Device Request Packet
Table 14-3: String Language Descriptor
Table 14-4: String Descriptor
Table 15-1: Device Descriptor
Table 15-2: String Language Descriptor
Table 15-3: String Descriptor 2
Table 15-4: Configuration Descriptor
Table 15-5: Interface Descriptor
Table 15-6: Default Mouse Input Report
Table 15-7: Interface's HID Descriptor
Table 15-8: Interface's HID Descriptor
Table 15-9: Endpoint Descriptor 1
Table 15-10: Set Idle Request
Table 15-11: Get Report Descriptor Request
Table 15-12: Mouse Input Report
Table 15-13: Set Protocol Request
Table 16-1: Device Descriptor
Table 16-2: String Language Descriptor
Table 16-3: String Descriptor 1
Table 16-4: String Descriptor 2
Table 16-5: Configuration Descriptor
Table 16-6: Interface Descriptor
Table 16-7: String Descriptor 5
Table 16-8: Default Keyboard Input Report
Table 16-9: Default Keyboard Output Report (1 byte)
Table 16-10: Interface's HID Descriptor
Table 16-11: Interface's HID Descriptor
Table 16-12: Endpoint Descriptor 1
Table 16-13: Set Idle Request
Table 16-14: Get Report Descriptor Request
Table 16-15: Default Keyboard Input Report
Table 16-16: Default Keyboard Output Report (1 byte)
Table 16-17: Set Protocol Request
Table 16-18: Set Report Request
Table 17-1: Device Descriptor
Table 17-2: String Language Descriptor
Table 17-3: String Descriptor 1
Table 17-4: String Descriptor 2
Table 17-5: String Descriptor 3
Table 17-6: Configuration Descriptor
Table 17-7: Interface Descriptor
Table 17-8: Endpoint Descriptor 1
Table 17-9: Endpoint Descriptor 2
Table 17-10: Endpoint Descriptor 3
Table 17-11: Device Qualifier Request
Table 17-12: Device Qualifier Descriptor

Table 17-13: Get Max LUN's Request
Table 17-14: Partial SBC-3 Command Code List
Table 17-15: Command Block Wrapper
Table 17-16: Command Status Wrapper
Table 17-17: Inquiry Command
Table 17-18: Inquiry Command Return Data
Table 17-19: Command Status Wrapper
Table 17-20: Read Format Capacities Command
Table 17-21: Capacity List Header
Table 17-22: Capacity Descriptor
Table 17-23: Capacities Descriptor Code
Table 17-24: Command Status Wrapper
Table 17-25: Request Sense Command
Table 17-26: Request Sense Return Data
Table 17-27: Sense Key, ASC, and ASCQ
Table 17-28: Command Status Wrapper
Table 17-29: Read Capacity(10) Command
Table 17-30: Read Capacity (32-bit)
Table 17-31: Read Capacity(16) Command
Table 17-32: Read Capacity (64-bit)
Table 17-33: Command Status Wrapper
Table 17-34: Read(10) Command
Table 17-35: Read(12) Command
Table 17-36: Read(16) Command
Table 17-37: Command Status Wrapper
Table 17-38: Reset Bulk-only Request
Table 18-1: Device Descriptor
Table 18-2: String Language Descriptor
Table 18-3: String Descriptor 1
Table 18-4: String Descriptor 2
Table 18-5: Configuration Descriptor
Table 18-6: Interface Descriptor
Table 18-7: Endpoint Descriptor 1
Table 18-8: Endpoint Descriptor 2
Table 18-9: Endpoint Descriptor 3
Table 18-10: Device Qualifier Descriptor
Table 18-11: Get Max LUN's Request
Table 18-12: UFI Command Codes
Table 18-13: CB/CBI Command Request
Table 18-14: Inquiry Command
Table 18-15: Inquiry Command Return Data
Table 18-16: Read Format Capacities Command
Table 18-17: Capacity List Header
Table 18-18: Capacity Descriptor
Table 18-19: Capacities Descriptor Code
Table 18-20: Formattable Capacity Descriptors returned
Table 18-21: Supported Capacity Descriptors
Table 18-22: Request Sense Command
Table 18-23: Request Sense Return Data
Table 18-24: Sense Key, ASC, ASCQ
Table 18-25: Read(10) Command
Table 18-26: Read(12) Command
Table 18-27: Send Diagnostic/Reset Command
Table 19-1: Device Descriptor
Table 19-2: String Language Descriptor
Table 19-3: String Descriptor 1

Table 19-4: Configuration Descriptor
Table 19-5: Interface Descriptor
Table 19-6: Endpoint Descriptor 1
Table 19-7: Interrupt Endpoint Return Data
Table 19-8: Get Hub Descriptor Request
Table 19-9: Hub Descriptor
Table 19-10: Get (Hub) Status Request
Table 19-11: Status Request Return Data
Table 19-12: Set Configuration Request
Table 19-13: Set_Feature/Clear_Feature
Table 19-14: Feature Selector
Table 19-15: Set Feature Request
Table 19-16: Clear Feature Request
Table 19-17: Get (Port) Status Request
Table 19-18: Port Status Field
Table 19-19: Port Change Status Field
Table 19-20: Set Feature Request
Table 19-21: Clear Feature Request
Table 20-1: Device Descriptor
Table 20-2: String Language Descriptor
Table 20-3: String Descriptor 1
Table 20-4: String Descriptor 2
Table 20-5: String Descriptor 3
Table 20-6: Configuration Descriptor
Table 20-7: Interface Descriptor 1
Table 20-8: Endpoint Descriptor 1 of Interface 1
Table 20-9: Endpoint Descriptor 2 of Interface 1
Table 20-10: Endpoint Descriptor 3 of Interface 1
Table 20-11: Interface Descriptor 2
Table 20-12: Endpoint Descriptor 1 of Interface 2
Table 20-13: Endpoint Descriptor 2 of Interface 2
Table 20-14: Interface Descriptor 3
Table 20-15: Endpoint Descriptor 1 of Interface 3
Table 20-16: Get Port Status Request
Table 20-17: Get Port Status Return
Table 20-18: Set Configuration Request
Table 20-19: Set Interface Request
Table 20-20: Soft Reset Request
Table 20-21: Get Device ID Request
Table 21-1: Device Descriptor
Table 21-2: String Language Descriptor
Table 21-3: String Descriptor 1
Table 21-4: String Descriptor 2
Table 21-5: BOS Descriptor
Table 21-6: BOS Capability Stub
Table 21-7: BOS Descriptor: USB 2.0 Extension
Table 21-8: BOS Descriptor: Super Speed USB Device
Table 21-9: BOS Descriptor: Container-ID
Table 21-10: Configuration Descriptor
Table 21-11: Interface Descriptor
Table 21-12: Endpoint Descriptor 1
Table 21-13: Interrupt Endpoint Return Data
Table 21-14: Endpoint Companion Descriptor
Table 21-15: Set Hub Depth Request
Table 21-16: Get USB 3.0 Hub Descriptor Request
Table 21-17: USB 3.0 Hub Descriptor

Table 21-18: Set Configuration Request
Table 22-1: Device Descriptor
Table 22-2: String Language Descriptor
Table 22-3: String Descriptor 1
Table 22-4: String Descriptor 2
Table 22-5: String Descriptor 3
Table 22-6: BOS Descriptor
Table 22-7: BOS Descriptor: USB 2.0 Extension
Table 22-8: BOS Descriptor: Super Speed USB Device
Table 22-9: Set Select Request Packet
Table 22-10: Set Select Data Out
Table 22-11: Configuration Descriptor
Table 22-12: Interface Descriptor
Table 22-13: Endpoint Descriptor 1
Table 22-14: Endpoint Companion Descriptor
Table 22-15: Endpoint Descriptor 2
Table 22-16: Endpoint Companion Descriptor
Table 24-1: Get Report Descriptor Request
Table 24-2: Items 1-byte Prefix
Table 24-3: Mouse Input Report
Table 25-1: Device Descriptor
Table 25-2: String Language Descriptor
Table 25-3: String Descriptor 1
Table 25-4: String Descriptor 2
Table 25-5: Configuration Descriptor
Table 25-6: Interface Association Descriptor
Table 25-7: Interface Descriptor: 0
Table 25-8: Function Descriptor: Video Control
Table 25-9: Function Descriptor: VC Input
Table 25-10: Function Descriptor: VC Processing Unit
Table 25-11 Function Descriptor: VC Output
Table 25-12: Endpoint Descriptor 1
Table 25-13: Function Descriptor: VC Processing Unit
Table 25-14: Interface Descriptor: 1
Table 25-15: Function Descriptor: Video Streaming
Table 25-16: Function Descriptor: VS Format Uncompressed
Table 25-17: Function Descriptor: VS Frame Uncompressed (1)
Table 25-18: Function Descriptor: VS Frame Uncompressed (2)
Table 25-19: Function Descriptor: VS Frame Uncompressed (3)
Table 25-20: Function Descriptor: VS Frame Uncompressed (4)
Table 25-21: Function Descriptor: VS Frame Uncompressed (5)
Table 25-22: Function Descriptor: VS Still Frame
Table 25-23: Function Descriptor: VS Color Matching
Table 25-24: Alt Interface Descriptor: 1
Table 25-26: Device Qualifier Request
Table 25-27: Device Qualifier Descriptor
Table 26-1: VS_Probe_Control GET_CUR Request
Table 26-2: VS_Probe_Control Return
Table 26-3: VS_Probe_Control SET_CUR Request
Table 26-4: VS_Commit_Control GET_CUR Request
Table 26-5: Set Alternate Interface 1.6
Table 26-6: Stream Header
Table 27-1: EHCI: ISO Transfer Descriptor
Table E-1: Sense Key, ASC, and ASCQ
Table G-1: Transfer Request Block Type Codes
Table G-2: Transfer Request Block Completion Codes

Table G-3: Setup Packet Format
Table G-4: bmRequestType Format
Table G-5: bRequest
Table G-6: Descriptor Types
Table I-1: USB Class Codes
Table J-1: DFU Interface Descriptor
Table J-2: DFU Function Descriptor
Table Q-1: Breakout Board Components
Table Q-2: Toggle Pin Request
Table Q-3: Get Report Descriptor Request

Figures

Figure 4-1: A UHCI Example
Figure 4-2: The UHCI Stack Frame
Figure 4-3: The UHCI Process
Figure 6-1: OHCI Endpoint / Transfer Descriptors
Figure 6-2: OHCI Periodic ED List
Figure 6-3: OHCI Stack Outline
Figure 8-1: EHCI: Buffer Address
Figure 8-2: EHCI Stack Frame
Figure 8-3: The EHCI Stack Outline
Figure 9-1: xHCI Extended Capabilities List
Figure 9-2: Port Routing and Control
Figure 9-3: The xHCI Doorbell Register Set
Figure 10-1: Memory Boundaries
Figure 10-2: Device Context Memory Buffers
Figure 10-3: Device Context Data Block Example
Figure 10-4: xHCI Slot Context Format
Figure 10-5: Route String Example
Figure 10-6: xHCI Endpoint Context Format
Figure 10-7: xHCI Normal TRB Format
Figure 10-8: xHCI Setup Stage TRB Format
Figure 10-9: xHCI Data Stage TRB Format
Figure 10-10: xHCI Status Stage TRB Format
Figure 10-11: xHCI Command Completion TRB Format
Figure 10-12: xHCI Transfer Ring Cycle Bit
Figure 10-13: Command Ring Example
Figure 10-14: xHCI Ring TRB Advance with Link TRB
Figure 10-15: xHCI Command Ring TRB Advance with Link TRB
Figure 10-16: xHCI Event Ring Segment Table Entry
Figure 11-1: UHCI Request
Figure 11-2: Request Return
Figure 11-3: UHCI Short Packet Detect
Figure 12-1: OHCI Request
Figure 12-2: Request Return
Figure 12-3: ED Insert
Figure 12-4: OHCI Outline
Figure 12-5: Request Return
Figure 12-6: Set Address
Figure 13-1: EHCI Request
Figure 13-2: Queue Head Insert
Figure 13-3: Request Return
Figure 13-4: Set Address
Figure 14-1: USB 3.0 Socket and Plug
Figure 14-2: USB 3.0 Cable Cross-section
Figure 14-3: Port Routing and Control
Figure 14-4: No Op Command TRB
Figure 14-5: No Op Command Event TRB
Figure 14-6: Enable Slot Command TRB
Figure 14-7: Slot Enabled Command Event TRB
Figure 14-8: Set Address Command TRB
Figure 14-9: Set Address Command Event TRB
Figure 14-10: Input Context Buffer
Figure 14-11: Input Control Context
Figure 14-12: Setup Stage TRB
Figure 14-13: Data Stage TRB
Figure 14-14: Event Data TRB #1

Figure 14-15: Status Stage TRB
Figure 14-16: Event Data TRB #2
Figure 14-17: Configure EP Command TRB
Figure 14-18: Configure EP Command Event TRB
Figure 14-19: Example Input Context Buffer
Figure 17-1: Bulk Only Protocol
Figure 19-1: Single- and Multi-TT
Figure 20-1: Print Demo 1
Figure 20-2: Print Demo 2
Figure 23-1: Analyzer Hook-up
Figure 23-2: Analyzer Listing 1
Figure 23-3: Analyzer Listing 2
Figure 24-1: Input/Output Reports
Figure 24-2: Short/Long Items
Figure 25-1: Interface Association Descriptor
Figure 27-1: EHCI: ISO Transfer Descriptor
Figure 27-2: iTD Buffer Pointers and Lengths
Figure K-1: Type C Connector
Figure Q-1: Our ATTiny2313 Breakout Board
Figure Q-2: Breakout Board Schematic
Figure Q-3: LED Resistance Calculator
Figure Q-4: AVR Directory Structure
Figure Q-5: Blink Device
Figure Q-6: Output Device
Figure Q-7: Input Device

Listings

Listing 2-1: Read value from the PCI
Listing 3-1: UHCI Host Controller Reset
Listing 3-2: UHCI Port Reset
Listing 3-3: UHCI Port Present
Listing 4-1: UHCI Queue Insertion
Listing 4-2: UHCI Queue Removal
Listing 5-1: OHCI Host Controller Reset
Listing 5-2: OHCI Port Reset
Listing 6-1: OHCI Schedule Creation
Listing 7-1: EHCI Port Read
Listing 7-2: EHCI Host Controller Reset
Listing 7-3: OHCI Port Reset
Listing 8-1: EHCI Queue Insertion
Listing 8-2: EHCI Queue Removal
Listing 8-3: EHCI Schedule Creation
Listing 8-4: EHCI Async Schedule Creation
Listing 9-1: xHCI Host Controller Reset
Listing 9-2: xHCI Paired Protocol
Listing 9-3: xHCI Port Reset
Listing 13-1: Release BIOS Ownership
Listing 14-1: Supported Protocol Capability
Listing 14-2: Offsets and Lengths
Listing 14-3: Release BIOS Ownership
Listing 25-1: The Configuration Descriptor Dump
Listing Q-1: Blink: main.c
Listing Q-2: Blink: Makefile
Listing Q-3: Output: main.c
Listing Q-4: Output: Makefile
Listing Q-5: Output: main.c

Appendix D - Notes for all Controller Types

The following appendix will describe and/or show information about all controller types that might be worth knowing, but not pertinent to include in each chapter. Each subject in this appendix may or may not have any connection with each other. These are just items that are of interest for each controller type, but I didn't want to include in each section, hence it would take up too much space.

Port Connection Status Change

Each controller's port registers have a connection status change bit that indicates that there has been a change in the connection status of that particular port. This bit is set by the controller to indicate that a connection change has taken place. Your driver must be careful with this status change. This is not a count of how many times the change has taken place, but simply that a change has happened. For example, let's say there is a device attached to a port on the root hub. Your driver has reset and enumerated it. Later the user disconnects the device, and the controller sets the connect status change bit. However, before your driver has a chance to handle the change, the user plugs the device back in. It can happen. Now the connect status change and the current connect status bits are both set. As far as your driver is concerned, nothing happened, right? Wrong. Since the device was disconnected, hence losing power, it is no longer in the configured state and does not have an address number, therefore it will not respond to data requests at the original address value.

Anytime the connect status change bit is set, your driver needs to assume that it must reset the port and re-enumerate the device, whether it is the same device or not.

ISO Transfers

Isochronous transfers are used in devices such as audio and video devices. Devices that continue to send or receive data at a fixed rate. ISO transfers are single data transfers that do not have a setup or status packet like a control transfer does.

For example, if you have an audio device and are sending data to it, if your transfer drops some packets due to control/status issues, you will most likely here a pause or static in the outcome. Therefore, ISO transfers do not have a handshake/stall stage. Your driver will simply send ISO transfers at the specified rate and the audio device handles them as they come across the wire. If one or two are dropped, most likely you will not hear the pause or static since the time interval between packets is very small.

The same is for video data being received by a device. Your driver will receive packets of ISO transfers and display them accordingly. Since the data is updated on the screen at least 14 times a second, and most of the time, must faster, if you miss a packet or two, you will not see much of a difference.

If the hardware spent the time to send and receive control and status packets, the chance of static would increase dramatically.

For more information on isochronous transfers, please see Part 5 of this book where I discuss how to enumerate and receive live video from a small video camera.

Super-speed (not counting burst sizes) and high-speed ISO transfers may have a max payload size of 1024, full-speed may have 1023 byte transfers, while low-speed does not allow ISO transfers at all.

Controller Quirks

All controllers are not created equal and some have little quirks that make them not work correctly or have an undocumented function. For example, if the xHCI controller has a version of 0x95, the Chain bit in the Link TRB isn't handled correctly by the controller.

There are quirks that are specific to a controller type while other quirks are specific to a specific brand and/or make of controller. I will try to explain some of these quirks within this section.

The information within this section is gathered from many sources and research. In most cases, I have not verified that these quirks actually exist. I am just passing on information that I have found from these sources. Thanks go to all of those who have done the research and reported these quirks to the public.

The heading of each quirk below will start with the controller type, next have a brand name and/or model, and then have the controller version number this quirk pertains to.

xHCI: All: Version 0x95

Different sections of version 0.95 of the specification had opposing requirements for the chain bit in a link TRB. The chain bit is used to indicate that adjacent TRBs are all part of the same TD. A link TRB can be anywhere within the TD, including the last TRB.

Sections 4.11.5.1 and 6.4.4.1 both stated that the Link TRB "shall have the chain bit set to 1", indicating that it is always chained to the next TRB. However, section 4.6.9 on the Stop Endpoint command has specific cases for what the hardware must do for a Link TRB with the chain bit set to zero. Version 0.96 of the specification cleared up this issue by fixing those sections to state that a Link TRB could have the chain bit set or clear.

The problem is some controllers with a version of 0.95 will stop processing the ring when it encounters a Link TRB with the chain bit cleared, even though the TD continues.

If your driver finds a controller with a version of 0.95, you should always have the Link TRB's chain bit set.

xHCI: AMD: Version 0x96

Setting the chain bit in the Link TRB of a isochronous transfers is required by AMD brand xHCI controllers with a version of 0.96 to successfully transverse multi-TRB TD's that span through different memory segments.

If your driver finds a controller with a version of 0.96 and is manufactured by AMD, you should make sure that all ISO TRB's have the chain bit set.

xHCI: NEC: uPD720200

The NEC uPD720200 xHCI controller needs a reset after resuming from a suspended state. The controller may become unstable afterward if the reset is not issued after the resume.

xHCI: NEC: All

The NEC xHCI controller will return an error if the Interval value for full-speed ISO endpoints is less than 3. Since this value is expressed in exponent form for full-speed ISO endpoints, ex: 2^3 = 8 micro-frames, a value less than 3 will indicate an interval smaller than one frame. It is impossible for a full-speed device to have an interval smaller than one frame.

xHCI: Fresco Logic: Device ID 0x1000, rev 0x00

This Fresco Logic xHCI controller revision puts bad data into the output endpoint context after a Reset Endpoint command. It needs a Configure Endpoint command, instead of a Set TR Dequeue Pointer command, after the Reset Endpoint command. This bug was only in prototype

hardware that was given to other companies for evaluation purposes. Fresco Logic's newer chip revision should have this bug fixed.

xHCI: Fresco Logic: Device ID 0x1000

This Fresco Logic xHCI controller advertises MSI, but fails to actually generate MSI interrupts. If your driver finds this controller, you should disable MSI interrupts. Later revisions of this chip may not be affected by this quirk.

xHCI: Texas Instruments: Device ID 0x8241, rev <= 0x01

A TI xHCI controller with this Device ID and a revision number less than or equal to 0x01, needs to clear bit 17 of four undocumented registers to reset correctly. These register offsets are 0xC2C0, 0xC2C4, 0xC2C8, and 0xC2CC. It is unknown at this time what these registers do.

xHCI: Texas Instruments: Some

A TI xHCI controller may not function correctly if the Interval value for an interrupt endpoint is greater than 7. Make sure that all Interval values for an interrupt endpoint is no more than 7.

xHCI: All: Version 1.00 and before

I don’t know of any xHCI controller, with a version less than 1.00, and very few with a version of 1.00, that would handle the save and restore function correctly. All of the controllers that I have checked will allow the Save State bit to be set, but would never set the Save State Status bit to indicate that it was done. One manufacturer told me that the save/restore function of the controller was not fully completed at time of release, though this was a prototype chip.

EHCI: MosChip: MCS9990

This EHCI controller updates its microframe counter just before the frame counter. When this controller increments the frame-index register, it changes the three low-order bits, the microframe counter, before changing the higher order bits, the frame counter. If the register is read at just the wrong time, the value obtained is too low by a value of eight. To work-around this, read the frame-index register a second time if the first value's three low-order bits are zero.

UHCI: HP: iLO2 (Lights Out 2)

The Hewlett Packard iLO2 UHCI controller needs a longer delay of 250uS after a port reset before the port may be used.

OHCI: Compaq: ZFMicro

Compaq’s ZFMicro chipset needs short delays before the control or bulk queues get re-activated.

OHCI: AMD: 756

The AMD-756 OHCI needs to have hub power always on. Make sure you don’t use port power switching. Leave power on to all ports.

OHCI: NEC: uPD9210 (uPD720100A)

The Power On to Power Good field in the HcRhDescriptorA Register has a value of 0xFF, which gives a wait time of 510 mS. Odd that it is 0xFF when most are 2 to 20mS.

OHCI: Toshiba: Portege 4000

This controller has a problem with the suspend state.

The BOS Descriptor

The Binary Device Object Store (BOS) Descriptor is used to add device level capabilities to the standard descriptors. This descriptor is accessed very similar to the configuration descriptor. It has a stub that contains the total length of all of the included descriptors within the BOS descriptor. You get the descriptor just like the device descriptor, however sending 0x0F instead of 0x01.

Even though this descriptor is now common with most USB 3.0 devices, this descriptor was defined and allowed starting with USB version 2.10.

For more information on this descriptor, see the description in Chapter 21, page 5.

Device Address Number

The Device Address you assign to a device must be in the range of 1 to 127. However, you have probably seen that a well-known modern operating system starts with a value of 2. Without doing a lot of research, my guess is that this operating system gives a value of 1 to its internal root hub data block, treating it as an external hub, giving it an address of 1. Then the first device attached gets the address of 2. This is just my opinion.

String Descriptors with an Index of 0xEE

Vendors have started to include a string descriptor with an index value of 0xEE. This is a "Microsoft OS String Descriptor". Every time you plug a device in, the OS will check for this string descriptor. The first time a device is plugged in with this string descriptor index, it will create a registry entry for this device. Then every time this device is plugged in after this, it doesn't have to do the driver check. For more information on this, please see the following web sites:

http://msdn.microsoft.com/en-us/library/windows/hardware/ff537430(v=vs.85).aspx
http://msdn.microsoft.com/en-us/windows/hardware/gg463179

You can also search for "Microsoft OS Descriptor" using your favorite web search.

Just for fun, you can check all of your devices for this descriptor by requesting for string index 0xEE. If the device does not support this string index, it will stall. The stall will be cleared on the next SETUP packet the device receives.

The Configuration Descriptor's Bus Powered Bit

If you have noticed, all of the devices you have seen in this book, and the devices you have used with your work, bit 7 is set in the bmAttributes field of the Configuration Descriptor. Up to and including version 1.0 of the USB specification, this was the "Bus Powered" bit. Starting with version 1.1, this bit is set for "historical" reasons. All devices are bus-powered up to a combined 500mA.

Appendix E - Request Sense Return Data

The following appendix will list the most common Sense Key, Additional Sense Code (ASC), and Additional Sense Code Qualifier (ASCQ) values from a Request Sense Command for Mass Storage Devices, which include the Floppy Disk Drive and Thumb Drive explained in the chapters within this book.

For a complete list, see the T10.org website at:

http://www.t10.org/lists/2asc.htm.

Table E-1: Sense Key, ASC, and ASCQ

Sense Key	ASC value	ASCQ value	Description
00	00	00	No Sense
01	17	01	Recovered data with retries
	18	00	Recovered data with ECC
02	04	01	Logical drive not ready - Becoming ready
		02	Logical drive not ready - Initialization required
		04	Logical Unit not ready - Format in progress
		FF	Logical Unit not ready - Device is busy
	06	00	No reference position found
	08	00	Logical unit communication error
		01	Logical unit communication time-out
		80	Logical unit communication overrun
	3A	00	Medium not present
	54	00	USB to host system interface failure
	80	00	Insufficient resources
	FF	FF	Unknown error
03	02	00	No seek complete
	03	00	Write fault
	10	00	ID CRC error
	11	00	Unrecovered read error
	12	00	Address mark not found for ID field
	13	00	Address mark not found for data field
	14	00	Recorded Entity not found
	30	01	Cannot read medium - unknown format
	31	01	Format command failed
04	40	80-FF	Diagnostic error on component *n*
05	1A	00	Parameter list length error
	20	00	Invalid command operation code
	21	00	Logical block address out of range
	24	00	Invalid field in command packet
	25	00	Logical unit not supported
Sense Key	ASC value	ASCQ value	Description
05	26	00	Invalid field in parameter list
		01	Parameter not supported
		02	Parameter value invalid

	39	00	Saving parameters not supported
06	28	00	Not ready to ready transition - Media changed
	29	00	Power on reset or bus device reset occurred
	2F	00	Commands cleared by another initiator
07	27	00	Write protected media
0B	4E	00	Overlapped command attempted

Appendix F - Brief History of the USB

In the latter part of 1995, at version 1.0 and 1.1 of the USB, there were two competing host controller implementations, the OHCI and the UHCI. The USB-IF adopted the OHCI specification from Compaq since it was “open”, hence its name. Intel, on the other hand, created the UHCI and demanded a license fee for any hardware chipset implementers, for which VIA Technologies was one of these implementers.

The main difference between the two controllers was that UHCI demanded more work via the software, while the OHCI was more work via the hardware. With this in mind, the UHCI was more processor hungry, but was cheaper to implement, if you didn’t count the licensing fee.

However, it backfired on them. The two controller types caused software and hardware vendors to develop, test, and maintain each implementation which increased the cost of hardware and software.

In 2002, when the USB-IF started to implement USB 2.0, they insisted that there would be only one implementation. To do this, they used the EHCI controller to control all high-speed devices, and included the UHCI and OHCI controllers as the companion controllers to control the lower speed devices. Intel and VIA continued to include the UHCI while other manufactures included the OHCI as the companion controllers.

Then in late 2007, when the xHCI was announced, it was to do away with any companion controllers and would handle all speeds with one controller. It would also be less work for software when it came to the different speeds, by handling all for speeds almost independent of the software that was driving it.

I have included a brief history outline below.

Specification/Hardware Release History

I have tried to make this as accurate as possible, but am unsure as to exact dates on some items.

November 1994: USB 0.7
December 1994: USB 0.8
1995: USB Implementers Forum formed with 340 members
April 1995: USB 0.9
August 1995: USB 0.99
September 1995: OHCI Release Candidate
November 1995: USB 1.0 Release Candidate
November 1995: UHCI 1.0
December 1995: OHCI 1.0
: First OHCI host controller released[9]
January 1996: USB 1.0
March 1996: UHCI 1.1d
October 1996: OHCI 1.0a[10]
September 1998: USB 1.1
July 1999: EHCI 0.8a[11]
April 2000: USB 2.0
: First EHCI host controller released[1]
June 2000: First EHCI hub controller released[1]
December 2001: USB OTG released
March 2002: EHCI 1.0
August 2005: WHCI 0.84
April 2007: First WHCI host controller released[1]
September 2007: xHCI announced[12]
August 2008: xHCI 0.9
November 2008: USB 3.0
May 2009: xHCI 0.96
: NEC announces standalone controller[13]
June 2009: First xHCI host controller released[1]
May 2010: xHCI 1.0
September 2010: Wireless USB Specification 1.1
2010: USB OTG 2.0 released
July 2013: USB 3.1
December 2013: xHCI 1.1 released
December 2013: Type C Connector announced
August 2014: Type C Connector specification released

Brad Hosler - Founder of USB

Brad Hosler was a founder of the USB technology, and spent many hours on its implementation. He passed away in late August of 2007 and I have included his obituary in this section, which is found at http://www.usb.org/inmemory.

In Memory of Brad Hosler
Our very dear friend and colleague Brad Hosler passed away on August 31, 2007, at his home in Portland, Oregon, after several years of battling cancer.

Brad was an alumnus of Bucknell and completed his graduate studies at Carnegie Mellon University. He joined Intel in 1980. Brad's signature accomplishments at Intel are associated with the USB family of technologies. He was a core contributor to the USB 1.0 initiative where he led the host controller specification and implemented an end-user focused USB Logo Compliance program. Spanning the next decade, Brad drove the wired and wireless USB evolutions. For both programs he set architectural standards and managed the team delivering the technology and logo compliance programs. Brad received two Intel Achievement Awards, one in 2003 for the USB 2.0 extension and another in 2006 for the architecture and enabling of the Wireless USB protocol over the Ultra Wideband (UWB) spectrum. The success of the USB interface and market of platforms and peripherals that sell in multiple billion units today is a measure of his impact.

Brad was promoted to Principal Engineer last year in recognition of his long roadmap of accomplishments in the field of I/O architecture and compliance. He was vested with the informal authority of Chief Technical Officer and served as Treasurer for the USB Implementers Forum. For over a decade, Brad spoke frequently at USB-IF and industry forums, where his talks were widely solicited and acclaimed.

xHCI TRB Types and Completion Codes

Table G-1: Transfer Request Block Type Codes

Allowed TRB Types			ID	TRB Name
Command Ring	**Event Ring**	**Transfer Ring**		
			0	Reserved
		Allowed	1	Normal
		Allowed	2	Setup Stage
		Allowed	3	Data Stage
		Allowed	4	Status Stage
		Allowed	5	Isochronous
Allowed		Allowed	6	Link
		Allowed	7	Event Data
		Allowed	8	No-Op
Allowed			9	Enable Slot
Allowed			10	Disable Slot
Allowed			11	Address Device
Allowed			12	Configure Endpoint
Allowed			13	Evaluate Context
Allowed			14	Reset Endpoint
Allowed			15	Stop Endpoint
Allowed			16	Set TR Dequeue Pointer
Allowed			17	Reset Device
Allowed			18	Force Event[1]
Allowed			19	Negotiate Bandwidth[2]
Allowed			20	Set Latency Tolerance Value[2]
Allowed			21	Get Port Bandwidth
Allowed			22	Force Header
Allowed			23	No-Op Command
			24-31	Reserved
	Allowed		32	Transfer Event
	Allowed		33	Command Completion Event
	Allowed		34	Port Status Change Event
	Allowed		35	Bandwidth Request Event[2]
	Allowed		36	Doorbell Event[1]
	Allowed		37	Host Controller Event
	Allowed		38	Device Notification Event
	Allowed		39	MFINDEX Wrap Event
			40-47	Reserved
Optional	Optional	Optional	48-63	Vendor Defined

[1] Optional -- used with virtualization only.

[2] Optional.

Vendor defined codes may be different for each vendor. Therefore, you must match the vendor-defined code with the vendor and device ids from the PCI configuration space of the device.

Table G-2: Transfer Request Block Completion Codes

Value	Name	Description
0	Invalid	This is an invalid completion code. The producer has not updated this field yet.
1	Success	The TRB was completed successfully.
2	Data Buffer Error	The controller is not able to keep up with the incoming data (overrun) or is unable to supply data fast enough (underrun).
3	Babble Detected Error	A babble error was generated.
4	USB Transaction Error	The controller did not receive a valid response from the device.
5	TRB Error	The controller detected an invalid parameter in the TRB.
6	Stall Error	A stall condition was detected. Will also be asserted when an error has occurred and you need to clear the stall via the control endpoint.
7	Resource Error	Used with the Configure Endpoint command if there are not enough resources to enable the requested amount of endpoints.
8	Bandwidth Error	Used with the Configure Endpoint command if periodic endpoints are declared and the controller is not able to allocate the bandwidth.
9	No Slots Available	Used with the Enable Slot command and returned if no more slots are available.
10	Invalid Stream Type	Sent if an invalid Stream Context Type is detected.
11	Slot Not Enabled	Sent if a command is sent to a slot that is not yet enabled.
12	Endpoint Not Enabled	Sent if a doorbell is rung for an endpoint that is not yet enabled.
13	Short Packet	Sent if the number of bytes received was less than the requested amount from the TD.
14	Ring Underrun	Sent in a Transfer Event TRB if the Transfer Ring is empty when an enabled Isochronous endpoint is scheduled to transmit data.
15	Ring Overrun	Sent in a Transfer Event TRB if the Transfer Ring is empty when an enabled Isochronous endpoint is scheduled to receive data.
16	VF Event Ring	Used with a Force Event command if the

	Full	target VF's Event Ring is full.
17	Parameter Error	Sent by a command if a command Context's parameter is invalid.
18	Bandwidth Overrun	Sent when an Isochronous transfer TD exceeds the bandwidth allocated to the endpoint.
19	Context State Error	Sent when a command is issued to transition from an illegal context state.
20	No Ping Response	Sent if the controller was unable to complete a periodic data transfer associated within the ESIT because it did not receive a Ping Response in time.
21	Event Ring Full	Sent if the Event Ring is full.
22	Incompatible Device	Sent if the controller detects that the attached device is was not accessed successfully.
23	Missed Service	Sent if the controller was unable to service an Isochronous endpoint within the Interval time.
24	Command Ring Stopped	Sent in a Command Completion Event when the Command Stop bit is set.
25	Command Aborted	Sent in a Command Completion Event when the Command Abort bit is set.
26	Stopped	Sent in a Transfer Event if the transfer was terminated by a Stop Endpoint Command.
27	Stopped - Length Invalid	Sent in a Transfer Event if the transfer was terminated by a Stop Endpoint Command and the Length field was invalid.
28	Reserved	Reserved.
29	Max Exit Latency Too Large	Sent by the Evaluate Context Command if the Max Exit Latency would not allow the periodic endpoints of the Device Context to be scheduled.
30	Reserved	Reserved.
31	Isochronous Buffer Overrun	Sent if an Isochronous TD on an IN endpoint is less than the MAS ESIT Payload in size and the device attempts to send more data than it can hold.
32	Event Lost	Sent when a controller internal event overrun condition happens.
33	Undefined	Will be sent by the controller when another error does not apply.
34	Invalid Stream ID	Sent when an invalid Stream ID is received.
35	Secondary Bandwidth Error	Sent by the Configure Endpoint Command if periodic endpoints are declared and the controller is not able to allocate the required bandwidth due to a Secondary Bandwidth Domain.
36	Split	Sent when an error is detected on a

	Transaction Error	USB2 protocol endpoint for a split transaction.
37-191	Reserved	Reserved.
192-223	Vendor Defined Error	Vendor specific.
224-255	Vendor Defined Information	Vendor specific. Used to give information to the host.

Setup Packet

The following table describes the format of the standard Setup Request Packet.

Table G-3: Setup Packet Format

Offset	Field	Size	Description
0	bmRequestType	1	Request Type. See Table G-4.
1	bRequest	1	Request. See Table G-5.
2	wValue	2	Specific to the Request.
4	wIndex	2	Specific to the Request.
6	wLength	2	Specific to the Request.

bmRequestType

The following table describes the allowed values in the bmRequestType byte, the first byte of a Setup Packet.

Table G-4: bmRequestType Format

bmRequestType Format	
Bits	Description
7	Data Transfer Direction: 0 = Host to Device 1 = Device to Host
6:5	Type: 00 = Standard 01 = Class 10 = Vendor 11 = Reserved
4:0	Recipient: 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4-31 = Reserved

bRequest

The following table describes the allowed values in the bRequest byte, also listing the allowed requests.

Table G-5: bRequest

bRequest		
Value	USB Version[1]	Description
0	1.1	Get Status
1	1.1	Clear Feature
2		Reserved
3	1.1	Set Feature
4		Reserved
5	1.1	Set Address
6	1.1	Get Descriptor
7	1.1	Set Descriptor
8	1.1	Get Configuration
9	1.1	Set Configuration
10	1.1	Get Interface
11	1.1	Set Interface
12	1.1	Synch Frame
13	3.1	Set Encryption
14	3.1	Get Encryption
15	3.1	Set Handshake
16	3.1	Get Handshake
17	3.1	Set Connection
18	3.1	Set Security Data
19	3.1	Get Security Data
20	3.1	Set WUSB Data
21	3.1	Loopback Data Write
22	3.1	Loopback Data Read
23	3.1	Set Interface DS
24-47		Reserved
48	3.0	Set Select
49	3.0	Set Isoch Delay
50-255		Reserved

[1]If a version is greater than 1.1, this value is reserved in any version before the one listed.

Descriptor Types

The following table lists the general descriptor types.

Table G-6: Descriptor Types

Descriptor Types		
Value	USB Version[1]	Description
0		Reserved
1	1.1	Device
2	1.1	Configuration
3	1.1	String
4	1.1	Interface
5	1.1	Endpoint
6	2.0	Device Qualifier
7	2.0	Other Speed Configuration
8	2.0	Interface Power
9	3.0	On The Go
10	3.0	Debug
11	3.0	Interface Association[2]
12-14		Reserved
15	3.0	Binary Device Object Store (BOS)
16	3.0	Device Capability
17-47		Reserved
48	3.0	Super Speed Endpoint Companion
49	3.1	Super Speed ISO Endpoint Companion

[1]If a version is greater than 1.1, this value is reserved in any version before the one listed.

[2]Before version 3.0, this was class 0xEF, subclass 0x02, protocol 0x01.

Appendix H - USB OTG (On The Go)

USB OTG, or On the Go, was introduced in December of 2001 for devices such as USB cameras. For example, you may have a digital camera that has a USB interface. You plug this into your computer and are able to read from the camera's storage device similar to a Mass Storage Device. You can read, copy, and even delete images from the camera.

What about printing these images? You would have to plug the camera into your computer, then load your favorite image viewer, and if that viewer had printing capabilities, you would be set.

OTG makes it a bit quicker to print these images. For example, if all you want to do is print certain images from your camera, you can simply plug the printer into your camera, and with a few presses of a button or two, you are printing images from your camera.

With this, your camera is a device when plugged into the computer but it plays the role of the host when plugged into the printer. Therefore, the camera has to be able to play both roles.

The USB OTG allows this to happen.

Another example would be to plug two cameras into each other to share images. The OTG allows one camera to be the host and one to be the device. All of the negotiating is done via the OTG and is not visible to the user.

Since the OTG specification is used for devices and not for root hubs via your computer, I will not go into any more detail than this appendix. If you would like to know more about OTG, there are many places on the web to search and read, starting with the USB-IF at www.usb.org.

Appendix I - USB Class Codes

The following tables contain the defined class codes for Device and Interface Descriptors. The first table has a list of base classes while the following tables describe each class and their subclasses and protocols.

Table I-1: USB Class Codes

Base Class	Descriptor	Description
0x00	Device	Use class info in Interface Descriptor
0x01	Interface	Audio
0x02	Both	Communications and CDC Control
0x03	Interface	Human Interface Device
0x05	Interface	Physical Interface Device
0x06	Interface	Image
0x07	Interface	Printer
0x08	Interface	Mass Storage
0x09	Device	Hub
0x0A	Interface	CDC-Data
0x0B	Interface	Smart Card
0x0D	Interface	Content Security
0x0E	Interface	Video
0x0F	Interface	Personal Healthcare
0x10	Interface	Audio/Video
0x11	Device	Billboard
0xDC	Both	Diagnostic Device
0xE0	Interface	Wireless Controller
0xEF	Both	Miscellaneous
0xFE	Interface	Application Specific
0xFF	Both	Vendor Specific

The following tables describe the subclasses and protocols in each class above. The items in bold below are defined in the USB specifications, while the others are values I have found through out my research. Official class code documents can be found at the following web-site:

http://www.usb.org/developers/devclass_docs

Base Class 0x00

If this value is found in the Device Descriptor, you are to use the class code in the interface descriptor.

Base Class 0x01: Audio

This class code defines devices that conform to the Audio Device Class Specification.

Class	Subclass	Protocol	Description
0x01	**0x00**	**0x00**	Undefined
	0x01	**0x00**	Audio Control
	0x02	**0x00**	Audio Streaming
	0x03	**0x00**	MIDI Streaming

Base Class 0x02: Communications and CDC Control

This class code defines devices that conform to the Communications and CDC Control Class Specification.

Class	Subclass	Protocol	Description
0x02	**0x00**	See Below	Reserved
	0x01		Direct Line Control Model
	0x02		Abstract Control Model
	0x03		Telephone Control Model
	0x04		Multi-Channel Control Model
	0x05		CAPI Control Model
	0x06		Ethernet Networking Control Model
	0x07		ATM Networking Control Model
	0x08		Wireless Handset Control Model
	0x09		Device Management
	0x0A		Mobile Direct Line Model
	0x0B		OBEX
	0x0C		Ethernet Emulation Model
	0x0D		Network Control Model

Each of the above subclasses may use one of the following protocol values.

Class	Subclass	Protocol	Description
0x02	See Above	**0x00**	Reserved
		0x01	AT Commands for V.250 etc
		0x02	AT Commands for PCCA-101
		0x03	AT Commands for PCCA-101 & Annex O
		0x04	AT Commands for GSM 07.07
		0x05	AT Commands for 3GPP 27.007
		0x06	AT Commands for TIA for CDMA
		0x07	Ethernet Emulation Model
		0xFE	External Protocol
		0xFF	Vendor Specific

Base Class 0x03: HID (Human Interface Device)

This class code defines devices that conform to the Human Interface Device Class Specification. Please remember that actual subclass and protocol codes would be too restrictive for this type of class. Therefore, the following codes are only used by the boot utility.

Class	Subclass	Protocol	Description
0x03	**0x00**	**0x00**	Undefined
	0x01	Boot Interface	
		0x00	None
		0x01	Keyboard
		0x02	Mouse

Base Class 0x05: PID (Physical Interface Device)

This class code defines devices that conform to the Physical Class Specification and is an extension of the HID Class Specification.

Class	Subclass	Protocol	Description

0x05	**0x00**	**0x00**	PID

Base Class 0x06: Image
This class code defines devices that conform to the Imaging Device Class Specification.

Class	Subclass	Protocol	Description
0x06	**0x01**	**0x01**	Imaging Device

Base Class 0x07: Printer
This class code defines devices that conform to the Printer Device Class Specification.

Class	Subclass	Protocol	Description
0x07	**0x01**	Printers	
		0x00	Undefined
		0x01	Unidirectional
		0x02	Bi-directional
		0x03	1284.4 bi-directional
		0xFF	Vendor Specific

Base Class 0x08: Mass Storage Device
This class code defines devices that conform to the Mass Storage Device Class Specification.

Class	Subclass	Protocol	Description
0x08	**0x00**	See Below	SCSI Command Set (De facto use)
	0x01		RBC
	0x02		MMC-5 (ATAPI)
	0x03		QIC-157 (Obsolete)
	0x04		UFI (Floppy disks)
	0x05		SFF-8070i (Obsolete)
	0x06		SCSI Command Set
	0x07		LSD FS
	0x08		IEEE 1667
	0xFF		Vendor Specific

Each of the above subclasses may use one of the following protocol values.

Class	Subclass	Protocol	Description
0x08	See Above	**0x00**	CBI (command completion interrupt)
		0x01	CBI (no command completion interrupt)
		0x02	Obsolete
		0x50	BBB (MSD Bulk-only Transport)
		0x62	UAS
		0xFF	Vendor Specific

Base Class 0x09: Hub
This class code defines devices that conform to the Hub Device Class Specification.

Class	Subclass	Protocol	Description
0x09	**0x00**	**0x00**	Full-speed Hub

		0x01	Hi-speed Hub with Single TT
		0x02	Hi-speed Hub with Multiple TT's
		0x03	Super-speed Hub

Base Class 0x0A: CDC Data

This class code defines devices that conform to the Communications Device Class Specification.

Class	Subclass	Protocol	Description
0x0A	**0x00**	**0x00**	Reserved
		0x01	Network Transfer Block
		0x30	Physical Interface proto for ISDN BRI
		0x31	ISO/IEC 3309-1993 - HDLC
		0x32	Transparent
		0x50	Management protocol for Q.921
		0x51	Data link protocol of Q.931
		0x52	TEI-multiplexor for Q.921
		0x90	Data compression procedures
		0x91	Euro-ISDN protocol control
		0x92	V.120 - V.24 rate adaptation to ISDN
		0x93	CAPI commands (2.0)
		0xFD	Host Based Driver
		0xFE	Protocol Unit Functional Descriptors
		0xFF	Vendor Specific

Base Class 0x0B: Smart Card

This class code defines devices that conform to the Smart Card Device Class Specification.

Class	Subclass	Protocol	Description
0x0B	**0x00**	**0x00**	Circuit Card Interface Device (CCID)
		0x01	Integrated Circuit Card Device (ICCD)
		0x02	Integrated Circuit Card Device (ICCD)

Base Class 0x0D: Content Security

This class code defines devices that conform to the Content Security Device Class Specification.

Class	Subclass	Protocol	Description
0x0D	**0x00**	**0x00**	Content Security Device

Base Class 0x0E: Video

This class code defines devices that conform to the Video Device Class Specification.

Class	Subclass	Protocol	Description
0x0E	**0x00**	**0x00**	Undefined

	0x01	**0x00**	Video Control
	0x02	**0x00**	Video Streaming
	0x03	**0x00**	Video Interface Collection

Base Class 0x0F: Personal Healthcare

This class code defines devices that conform to the Personal Healthcare Device Class Specification.

Class	Subclass	Protocol	Description
0x0F	**0x00**	**0x00**	Personal Healthcare Device

Base Class 0x10: Audio/Video Devices

This class code defines devices that conform to the Audio/Video Device Class Specification, devices that are composite devices using audio and video such as a video camera.

Class	Subclass	Protocol	Description
0x10	**0x01**	**0x00**	A/V Control Device
0x10	**0x02**	**0x00**	A/V Video Streaming
0x10	**0x01**	**0x00**	A/V Audio Streaming

Base Class 0x11: Billboard Device

This class code defines devices that conform to the Billboard Device Class Specification.

Class	Subclass	Protocol	Description
0x11	**0x00**	**0x00**	Billboard Device

Base Class 0xDC: Diagnostic Device

This class code defines devices that are diagnostic devices.

Class	Subclass	Protocol	Description
0xDC	**0x01**	**0x01**	Diagnostic Device

Base Class 0xE0: Wireless Controller

This class code defines devices that are wireless controllers.

Class	Subclass	Protocol	Description
0xE0	**0x01**	**0x01**	Bluetooth Programming
		0x02	UWB Radio Control
		0x03	Remote NDIS
		0x04	Bluetooth AMP Controller
	0x02	**0x01**	Host Wire Adapter Control/Data
		0x02	Device Wire Adapter Control/Data
		0x03	Diver Wire Adapter Isochronous

Base Class 0xEF: Miscellaneous

This class code defines devices that are miscellaneous devices or uses multiple interfaces such as the Interface Association Descriptor combination, EF:02:01.

Class	Subclass	Protocol	Description
0xEF	**0x01**	**0x01**	Active Sync Device
		0x02	Palm Sync

	0x02	**0x01**	Interface Association Descriptor
		0x02	Wire Adapter Multifunction Peripheral
	0x03	**0x01**	Cable Based Association Framework
	0x04	**0x01**	RNDIS over Ethernet
		0x02	RNDIS over WiFi
		0x03	RNDIS over WiMax
		0x04	RNDIS over Wwan
		0x05	RNDIS for Raw IPv4
		0x06	RNDIS for Raw IPv6
		0x07	RNDIS for GPRS
	0x05	**0x00**	USB3 Vision Control Interface
		0x01	USB3 Vision Event Interface
		0x02	USB3 Vision Streaming Interface

Base Class 0xFE: Application Specific

This class code defines devices that conform to several classes on the USB-IF web-site.

Class	Subclass	Protocol	Description
0xEF	**0x01**	**0x01**	Device Firmware Upgrade
	0x02	**0x00**	IRDA Bridge Device
	0x03	**0x00**	USB Test and Measurement Device
		0x01	USB Test and Measurement Device (USNTMC USB 488)

Base Class 0xFF: Vendor Specific

This class code defines devices that have Vendor Specific Class Codes.

Class	Subclass	Protocol	Description
0xFF	xx	xx	Vendor Specific

Appendix J - Device Firmware Upgrade

The USB standard allows for a device to support upgrading its firmware. For example, if a device needs to update its controller's firmware to function with new hosts, rather than buying a new device, you can update the firmware on the device to work with the new features of the host. The device must support this feature, though.

This specification is called "Device Class Specification for Device Firmware Upgrade", version 1.1, 5 Aug 2004.

A device will inform the host that it has this capability by placing two additional interface descriptors in the list of descriptors. The first is a standard Interface Descriptor with a class code of 0xFE, "Application Specific Class", a sub-class of 0x01, and a Protocol of 0x01. The second descriptor is a DFU Function descriptor.

Table J-1: DFU Interface Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x04**	Interface Descriptor
2	Interface Num	1	**XX**	Number of Interface.
3	Alternate Set	1	**0x00**	Used to select alt. setting.
4	Endpoints	1	**0x00**	Number of Endpoints: 0
5	Class Code	1	**0xFE**	Class Code: 0xFE = Application Specific
6	Sub Class	1	**0x01**	Sub Class Code: 0x01 = Device Firmware Upgrade.
7	Protocol	1	**0x01**	Protocol Code: 01h
8	Interface Str	1	**XX**	String descriptor

Table J-2: DFU Function Descriptor

Offset	Field	Size	Value	Description
0	Length	1	**0x09**	Size of descriptor in bytes.
1	Type	1	**0x21**	Interface Descriptor
2	Attributes	1	**XX**	Bits 7:4 Reserved Bit 3 Will Detach 0 = must reset 1 = performs Detach Bit 2 Manifestation 0 = must reset 1 = is tolerant Bit 1 Upload Capable 0 = No 1 = Yes Bit 0 Download Capable 0 = No 1 = Yes
3	Detach Timeout	2	**XXXX**	Millisecond time out to wait for detach sequence
5	Size	2	**XXXX**	Transfer Size: Max Packet Size per UDF transfer
7	DFU Version	2	**XXXX**	Version of the specification this supports

Upgrade Process

To start the process, you use this interface and send the Detach request. Depending on bit 3 in the Attributes field, you may or may not need to reset the port the device is attached to. You can then send and receive other packets to update the firmware on the device.

If the device supports a download of the firmware, bit 0, you can download the firmware, disassemble it, and see how the device works. However, I would bet that most commercial devices would not support this feature.

See the specification for more information. See Appendix A to see where you can get this specification.

Wrap up and Comments

There is a flaw with this specification's use. A virus or other type of malware that wishes to install a different firmware on the device, can at will.

The reason it would want to do this is to infect any machine that this USB device would be attached to.

For example, a virus could install a firmware set onto a thumb drive that will now make any host think it is a keyboard. As soon as the host enumerates it as a keyboard, the device's new firmware now sends keys to the host that could do anything it wishes.

It could even read from its own Mass Storage area to infect the new host with the same virus that wrote to this device.

My suggestion is to never insert an unknown device into your computer. Only trust devices that you know are clean.

For more information on this, see the following web sites.

http://www.bigmessowires.com/2014/10/04/badusb-and-the-hidden-microcontroller/
https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf
http://hackaday.com/2014/10/05/badusb-means-were-all-screwed/#more-134046

Appendix K - New Type C USB Connector

Have you ever tried to plug in your phone in the dark, or reached way back behind a desktop or tower and tried to plug in a device, only to find out you have the plug upside down?

The USB-IF recently made this a thing of the past introducing the Type C Connector which can be plugged in right side up, or upside down.

Figure K-1: Type C Connector

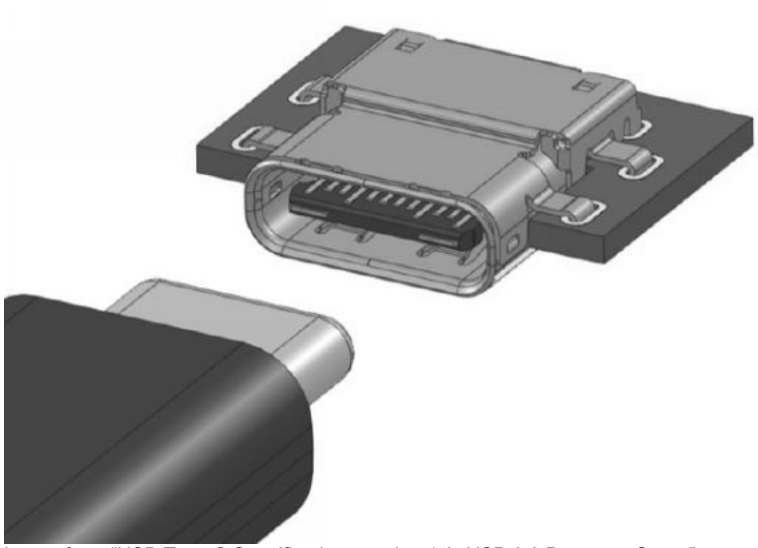


Image from “USB Type C Specification, version 1.0, USB 3.0 Promoter Group”

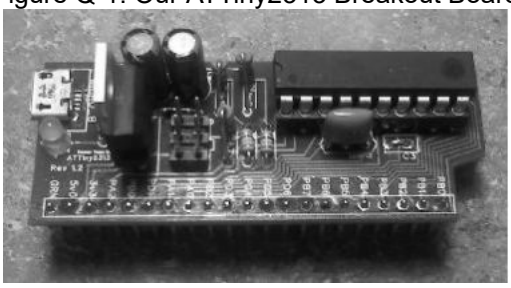
With this reversible connector, you don’t have to worry anymore if you have the connector right side up, nor do you have to worry that you have the correct end of the cable.

Appendix Q - Building a Simple USB Device

In this “chapter”, we will build a small USB device, learn how to build it using the Atmel® ATTINY2313[14] micro-controller, program the controller to accept USB commands, then communicate with our newly built device using the information you learned within the chapters of this book.

All USB devices have a common requirement, the USB communications interface. That is the hardware that is used to send and receive USB data. Therefore, the first thing we will do is create this common requirement using a few electronic components and solder them to a PCB with 21 pins that you can insert into a breadboard ready to build your USB device.

Figure Q-1: Our ATTiny2313 Breakout Board

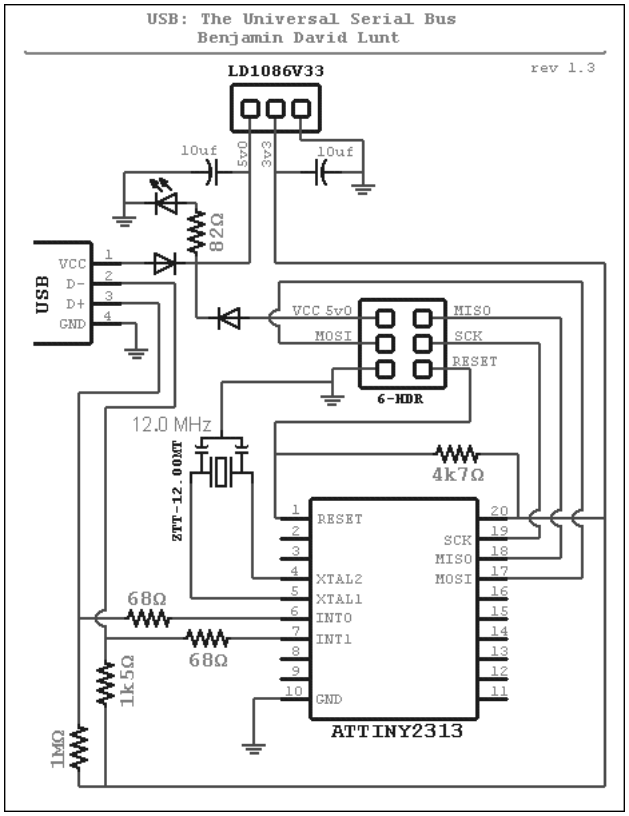


In the figure above, we have the USB connector on the left, a green LED and resistor to indicate that we have power from the USB connector, the 6 pin programming header, two diodes, the voltage regulator and its two capacitors, five resistors, a 12-Mhz resonator, and the ATTiny2312 micro-controller that does all the work. Then at the bottom of the PCB there are 21 pins that allow us to place these into a breadboard allowing us to build USB devices.

Included on the disc, in the misc folder, is the file I used with FreePCB to create the PCB above.

Throughout the remaining sections of this chapter, I will explain what you will need to build the board, how to build and program it, then how to use it to create your own USB devices.

Figure Q-2: Breakout Board Schematic


USB: The Universal Serial Bus
Benjamin David Lunt
rev 1.3
LD1086V33
10uf
5v0
3v3
10uf
82Ω
USB
VCC
D-
D+
GND
1
2
3
4
VCC 5v0
MOSI
MISO
SCK
RESET
6-HDR
12.0 MHz
ZTT-12.00MT
4k7Ω
RESET
XTAL2
XTAL1
INT0
INT1
GND
SCK
MISO
MOSI
68Ω
68Ω
1k5Ω
1MΩ
ATTINY2313

Our ATTiny2313 Breakout Board

Our breakout board uses a USB connector, a voltage regulator, the micro-controller, and a 12-Mhz resonator, along with a few other components used with each of these items. Figure Q-2, on the previous page is the schematic of the breakout board.

The following table shows the list of the components used. Please note that the price of the components was taken at time of this writing and does not include any applicable taxes or shipping.

Table Q-1: Breakout Board Components

	Part Number[1]	Description	Cost
1	USBTinyISP	The Programmer (www.fysnet.net/blog/2012/12/)	15.62
1	Breakout Board	Our PCB (fys@fysnet.net)	7.50
1	538-105017-0001	USB Type B Micro Connector	1.14
1	859-LTL-1CHG	Green LED	0.08
1	791-RC1/4-820JB	82 Ohm 1/4w resistor[2]	0.29
1	511-LD1086V33	Voltage Regulator	0.95
2	667-ECA-1HHG100	10 uF Capacitor	0.35
1	538-10-89-7062	6 pin (3x2) program header	0.67
2	78-BZX55C5V1-TAP	5v1 zener diode	0.07
1	791-RC1/4-105JB	1M Ohm 1/4w resistor[2]	0.29
1	791-RC1/4-152JB	1k5 Ohm 1/4w resistor[2]	0.29
1	791-RC1/4-472JB	4k7 Ohm 1/4w resistor[2]	0.29
2	660-CFS1/4CT52R680J	68 Ohm 1/4w resistor	0.15
1	575-199320	Optional 20 pin Socket	0.69
1	556-ATTINY2313A-PU	ATTiny2313 micro-controller	1.31
1	520-ZTT1200MT	12Mhz Resonator	0.50
1	649-68004-236	36 pin (1x36) header	0.83
		Total in USD	31.59

[1]All but the top 2 items are from www.mouser.com

[2]Resistor is not the one shown in Figure Q-1 but works just the same.

When you acquire these items, I recommend that you acquire multiple quantities of all but the top two items, especially the LED's and then even different colored LED's. You will need a few of them in the examples later in this chapter.

The first component needed for any USB device is of course the USB connector. Here I use a USB Type B Micro connector. It is a surface mounted connector and needs to be soldered in numerous places to the PCB so that it does not come loose, since it may have a cable attached and unattached numerous times.

In a previous revision of this breakout board, I used a standard Type B connector, however, it used up a lot of room on the board. By using the Micro connector, I was able to free up some space and place an LED indicator next to it.

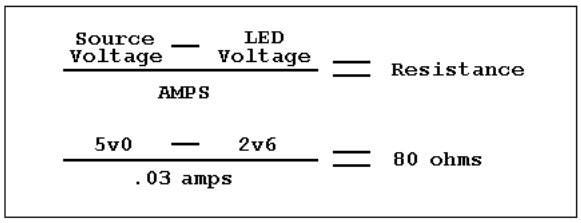
The next component is the green LED used to indicate that we have current at the connector. Since the USB provides about 5.0 volts, we must use a voltage resistor with the LED so that the 5v0 will not burn out the 2v6 LED.

In the electronics world, when a voltage, ohm reading, or other value is listed, the 'v' in voltage is used for the point in 5.0 volts. For example, 5.0 volts would be listed

as 5v0, while 4.5k ohms would be listed at 4k5 ohms. Therefore, the 2v6 above indicates 2.6 volts.

A resistor will limit the amount of current to the LED, hence not allowing the LED to burn out. To calculate the amount of ohms needed for the resistor and LED pair, you take the source voltage of 5v0 and subtract the amount of voltage the LED requires, 2v6, and then divide by the amount of amps the LED will use. This will give you the resistance needed.

Figure Q-3: LED Resistance Calculator



With the LED I use here, it requires an 80-ohm resistor. Since it does not have to be exact, and common ohm values are much cheaper and more available, I rounded to an 82-ohm resistor. If you use a different LED than I do, make sure to calculate the correct resistance you will need for the resistor.

Since the USB provides 5v0 and the USB data lines for the micro-controller requires 3v3 volts, we need to use a voltage regulator to drop the voltage to the required 3v3. I used the Low Dropout LD1086, but any voltage regulator that outputs 3v3 will do. The specification for this regulator requires a 10uF capacitor on the 5v0 input as well as the 3v3 output.

You can get away without using the voltage regulator by using a couple of 3v6 diodes and a resistor on the data lines. However, this would not give us the two voltage sources for our breadboard.

The next component is the 6-pin programming header used by the programmer to program the micro-controller. See later in this chapter for more on programming the controller.

If you have the breakout board plugged in to a USB port, you should never have the programmer plugged in to the board, and visa-versa. However, just to be sure, I have placed two diodes on the board to not allow one voltage source to contaminate the other source if you accidentally have both items plugged in at the same time.

The next components are five resistors. Two are used as pull-up resistors for the two USB data lines, two more on the data lines themselves, while the fifth is used to pull up the Reset pin of the micro-controller.

With this project, I put a 1k5 Ohm resistor from 3v3 to the D- data line. This indicates to the hub that this device is a Low-speed device. Put the 1k5 Ohm on the D+ data line to indicate a Full-speed device, swapping the 1m0 Ohm resister to the D- side.

The next component is the 12-Mhz Resonator used by the micro-controller.

The last component is the micro-controller itself, the ATTiny2313. We will use two pins as data lines for the USB interface and eight I/O pins to control our USB device. See later in this chapter for how we do this. Next, soldering the components.

Soldering the Components

To solder the components, you will need a 25-Watt soldering iron with a small tip. The tip should be small enough not to get in the way of other pins, but large enough to hold good heat. The tip should be clean and tinned before and after use.

> The term “tinned” means that it is covered with solder. This is to keep the iron’s tip from corrosion.

You should have a good solder paste. This will clean and flux the components once heat is applied. I use Oatey’s *No. 5 Lead Free Solder Paste* No. 30011. It is good practice to always clean the residue paste from the component once it has cooled. You can clean it with denatured alcohol and a small flat paint brush.

When soldering, apply a small amount of paste to the pin and solder pad of the PCB. Place a little dab of solder on the tip of your warmed iron, then place the tip of the iron on the pin and solder pad. Once the pin and solder pad are warm enough, the solder will flow from the iron to the pin and pad. Hold the iron on the pin and pad for a split second longer to be sure the solder takes, then remove the iron. Continue with each component with this technique, snipping the excess lead lengths from the PCB once the solder has cooled. Be sure to use sharp cutters to cut the leads. Dull cutters may “shock” the lead, breaking the connection from the solder.

> Another solder technique is to hold the iron on the pin and pad, then applying solder to the other side of the pad. Once the pad and pin are warm enough, the solder will flow toward the iron. This is considered a “better” technique. However, this uses both hands to solder where the first technique can be done with one hand holding the project.

Once you have soldered all the components to the board, finally solder the 21-pin header to the backside so that the pins point down. I left the drill holes a little large in the PCB for this header so that you can solder it at a slight angle forward. This will “lift” the PCB from the breadboard a little when inserted.

The Programmer

The programmer I use is a stripped down version of the USBtinyISP AVR Programmer Kit (USB SpokePOV Dongle) - v2.0 from Adafruit Industries, https://learn.adafruit.com/usbtinyisp that I made myself. Any 6-pin ISP programmer designed for the ATTiny series will work.

I use the Windows® version of the driver located at the URL above, by clicking on the *Download* link and downloading the *Windows USBtinyISP* driver. Unzip the file to an empty folder, then plug in the programmer. When prompted, point the Windows' "Found New Hardware" installer to this folder. When the install is complete, unplug the programmer then plug it back in to be able to use it.

The Cross Compiler

Next, you will need an AVR cross compiler. I use the one at http://winavr.sourceforge.net/. Install the compiler by simply running the install file and allowing it to install to an empty folder. Make sure your PATH environment variable points to this WinAVR\bin folder.

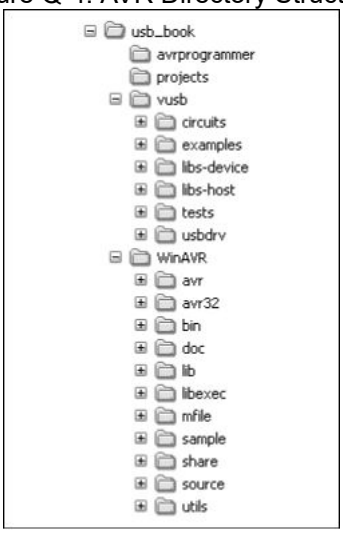
You will also need the USB-LIB code found at

http://www.obdev.at/products/vusb/download.html

This download contains the USBDRV folder that you will need for each device you build. Unzip it to the same group of folders.

You should now have a directory structure similar to the figure shown here.

Figure Q-4: AVR Directory Structure



The *avrprogrammer* folder is where you might have put the driver files for the programmer, the *vusb* folder is where you might have put the USB Library files, which includes the needed *usbdrv* folder, and the *WinAVR* folder is where you might have installed the Cross Compiler. The *Projects* folder is one that you might want to create to hold all of your projects.

Once you have the AVR compiler and the USB library installed, you can now start to write, compile, and flash code to the ATTiny2313.

Fusing and Flashing

One of the first things we will do is create a simple blinking LED app just so that we know that the programmer and the breakout board works, and that we soldered everything correctly.

> In the electronics world, a blinking LED project is the same as the "*Hello, World*!" project in the programming world.

In the *projects* folder, create a folder called *blink*.

Now create a single source file called *main.c*.

Listing Q-1: Blink: main.c

```
#include <avr/io.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

int main(void) {

 // make bottom 2 PORTB pins output
 DDRB = 3;  // bits 0 and 1

 while (1) {        // main event loop
   PORTB = 1; // Turn LED on PB0 on and PB1 off
   _delay_ms(500);
   PORTB = 2; // Turn LED on PB0 off and PB1 on
   _delay_ms(500);
 }

 return 0;
}
```

You will now need a Makefile to be able to build and flash the project. Create a file called *Makefile* with the contents in the Listing below.

Listing Q-2: Blink: Makefile

```
CC = avr-gcc
OBJCOPY = avr-objcopy
DUDE = avrdude
RM = del

CFLAGS = -Wall -Os -Iusbdrv -mmcu=attiny2313 -DF_CPU=12000000
OBJFLAGS = -j .text -j .data -O ihex
DUDEFLAGS = -p attiny2313 -c usbtiny

# Object files for the firmware
OBJECTS = main.o
# By default, build the firmware, but do not flash
all: main.hex

# Flash the firmware by typing "make flash" on command-line
flash: main.hex
        $(DUDE) $(DUDEFLAGS) -v -U flash:w:$<

# This will set the FUSE bits of the chip
fuse:
```

```
	$(DUDE) $(DUDEFLAGS) -U lfuse:w:0xef:m

# Housekeeping if you want it
clean:
	$(RM) *.o *.hex *.elf

# From .elf file to .hex
%.hex: %.elf
	$(OBJCOPY) $(OBJFLAGS) $< $@

# Main.elf requires additional objects, not just main.o
main.elf: $(OBJECTS)
	$(CC) $(CFLAGS) $(OBJECTS) -o $@

# From C source to .o object file
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

# From assembler source to .o object file
%.o: %.S
	$(CC) $(CFLAGS) -x assembler-with-cpp -c $< -o $@
```

Now by typing MAKE at the DOS command line, you should be able to make the firmware file *main.hex*.

Make sure you use the MAKE.EXE from the WinAVR install. If you have other compilers installed and your PATH environment variable has a pointer to one of them first, it may try to use that MAKE.EXE. This may produce errors.

Now you need to set the fuse bits of the micro-controller. This only needs to be done once per project and is explained below.

The ATTiny2313 has an internal Oscillator (crystal) that is set by default to 1-Mhz, less than the 12-Mhz we desire. Therefore, by setting the CKSEL bits (bits 3:0) to any value of 1000 to 1111, we tell the micro-controller to use our external 12-Mhz clock attached to XTAL1 and XTAL2. We also set the SUT0 and SUT1 bits, and the CKOUT bit, but clear the CKDIV8 bit. Remember that clearing a bit indicates that fuse is set. See page 160, Table 68 of the ATTiny2313 specification.

Make sure you have the programmer plugged into your host and installed correctly. Now plug the programmer's 6-pin cable to the programming header of the breakout board. Make sure that the red line on the ribbon is on the side with the circle on the breakout board. i.e.: The ribbon should be lying over the regulator, not the resistors (see Figure Q-5). The red line side should be near the two capacitors. If the green LED on the breakout board is not lit, something is wrong.

At the C:\ prompt enter the following:

C:\>MAKE FUSE<*enter*>

This will write 0xEF to the low fuse byte. If everything is working correctly, you should have not seen any errors displayed. If you saw any errors displayed, double check all your connections and solder joints and try again.

Now you can flash the firmware to the micro-controller.

At the C:\ prompt enter the following:

C:\>MAKE FLASH<*enter*>

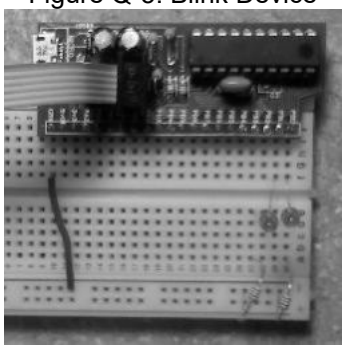
Again, watch for any errors that might be displayed. If no errors were found, you are ready to build the *Blink* device.

Build Blink Device

To build the *Blink* device, simply plug the breakout board into your breadboard so that you can plug an LED lead into the PB0 and another into the PB1 pin groups. Then put the other lead of the LED's into an empty group. Place a resistor between this group of pins and the GRN pin group on the breadboard. Be sure to connect the GRN pin to the GRN group as show in Figure Q-5 below.

Remember that the longer lead on the LED is the positive side.

Figure Q-5: Blink Device



Note that the programming cable is still attached in the figure above. This is to show you which way it should be attached. You should always disconnect the programmer after programming the micro-controller.

Now plug in the breakout board to a USB power supply, either a charging unit or an actual USB port on a computer.

If you plug the device into a port on a computer, the host operating system will complain that you attached an unknown device. This is okay for now, we haven't gotten to the point of an actual USB device yet. We just want the 5v0 current it provides.

If you did everything correctly, the two LED's should flash alternating between the two about every half-second. If this is the case, we can move on to creating an actual USB device, sending and receiving data.

However, if you have not been able to get the LED's to flash, you need to go back and check all of your connections, solder joints, component orientation, program code, etc.

Troubleshooting

This section is incase you did not get the *Blink* project to work, listing a few questions to make sure you did everything correctly. The top of the next page shows a list.

- When you plugged in the programmer, did your Host accept the device? Was there an error message displayed? If you go to the Systems Menu, does the device show within the list of USB devices attached?
- When the programmer is plugged in to the Host, does the programmer's green LED light up? If not, there is no power to the programmer.
- When the programmer is plugged in to the Host and is plugged in to the breakout board, does the green LED on the breakout board light up? If not, is the ribbon coming from the regulator side?
- When you type MAKE FUSE on the command line, are there any errors on the screen due to the HOST not finding the AVR BIN directory?
- When you type MAKE FUSE on the command line, does the AVR_DUDE command show a "Check Connections" error?

If all check points above have been passed, and you still can not get the programmer to program your breakout board, you will need to check all of the solder joints and components on the board, both the breakout board and, if you assembled the programmer, the programmer's board. Take an ohm tester and place one lead on one pin, and follow the traces on the board checking that there is continuity to each pin on that trace. Also make sure that all components are oriented correctly. The resonator and resistors can be either direction, but the diodes, LED, capacitors, regulator, and micro-controller must be oriented correctly.

Building an Actual USB Device

Fortunately, someone has already written the code we need for our firmware to be able to have our USB device communicate with the host. This code is in the *usbdrv* directory shown in Figure Q-4. Please read all applicable license documents in that folder.

Since we include that code with our projects, we don't have to worry about the actual USB communications. All we have to do is write code to use the functions it provides for us.

The first device we will build is a small example of applying power to the eight pins of the micro-controller, in turn lighting eight LED's. The second device will show how to read input from these eight pins. With this information, you should be able to create most any simple USB device, which would require current to run a motor or LED, or require input such as a sensor.

The attiny2313 micro-controller we are using actually has more than eight available pins. It has eight on Port B and seven on Port D. However, we are using four of the seven on Port D for communication and timing. Therefore, we can actually have a combination of eleven I/O pins.

Build *OUTPUT* Device

Create a directory called *output* in your *projects* folder. Copy the *usbdrv* folder from the *vusb* folder to this folder. This is the code required to have USB communication. Within this folder, rename the *usbconfig-prototype.h* file to *usbconfig.h*. This is the configuration file. In most cases, you will only need to modify this file, leaving the rest of the files within this folder unmodified.

For this project, make the following modifications to *usbconfig.h* starting with the "---- Hardware Config ----" section.

```
#define USB_CFG_IOPORTNAME   D
#define USB_CFG_DMINUS_BIT   3
#define USB_CFG_DPLUS_BIT    2
```

The above is telling the USB code to use Port D, pins PD3 and PD2 as the USB communications pins.

```
#define USB_CFG_CLOCK_KHZ    (F_CPU/1000)
```

Above defines F_CPU in the *Makefile* as 12 Mhz. (modification may not be needed)

```
#define USB_CFG_IMPLEMENT_FN_READ    0
#define USB_CFG_IMPLEMENT_FN_WRITE   0
```

The Host is not transferring data from the device other than the eight bytes within the SETUP request packet. If we were using control-in or control-out packets, other than the SETUP packet, we would need to set these to 1, as in the next example later in this chapter.

```
#define USB_CFG_VENDOR_ID       0xc0, 0x16
#define USB_CFG_DEVICE_ID       0xdc, 0x05
#define USB_CFG_DEVICE_VERSION  0x00, 0x01
#define USB_CFG_VENDOR_NAME     'V', 'O', 'T', 'I'
#define USB_CFG_VENDOR_NAME_LEN 4
#define USB_CFG_DEVICE_NAME   \
    'U', 'S', 'B', 'E', 'x', 'a', 'm', 'p', 'l', 'e'
#define USB_CFG_DEVICE_NAME_LEN 10
```

The above lines define the Vendor and Device ID's and string values used within the device's Device Descriptor. However, there are a few guidelines. Please see the *USB-IDs-for-free.txt* file for more information. If you will never distribute your device, you may use whatever values you wish.

```
#define USB_CFG_DEVICE_CLASS       0xFF
#define USB_CFG_DEVICE_SUBCLASS    0
```

The above defines are used for the Class and Subclass fields in the device's Device Descriptor. We will just use the Vendor Specific value for this example.

```
#define USB_CFG_INTERFACE_CLASS    0
#define USB_CFG_INTERFACE_SUBCLASS 0
#define USB_CFG_INTERFACE_PROTOCOL 0
```

The above lines are used for the Class, Subclass, and Protocol fields in the device's Interface Descriptor.

- /* #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH 24 */

Make sure this #define is commented out.

Any other values should be zero or left as is.

Now create the *main.c* source file with the contents listed in Listing Q-3 below.

Listing Q-3: Output: main.c

```
#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

#include "usbdrv.h"

// ------------------------------------------------------------

// this gets called when custom control message is received
USB_PUBLIC uchar usbFunctionSetup(uchar data[8]) {
  usbRequest_t *rq = (void *) data;

  if ((rq->bRequest >= 0) && (rq->bRequest <= 7)) {
    PORTB ^= (1 << rq->bRequest); // toggle LED on pin
    return 0;
  }

  return 0;
}

int main(void) {
  wdt_enable(WDTO_1S);
  usbInit();

  // make all PORTB pins output
  DDRB = 0xFF;
  PORTB = 0;

  sei();
  while (1) {        // main event loop
    wdt_reset();
    usbPoll();
  }

  return 0;
}
```

The code in Listing Q-3 above simply catches all SETUP packets, and if the Request is zero through seven, toggles that respective bit in the PORTB register. If the bit is on, power is applied to that pin.

Now create the make file, *Makefile,* with the contents listed in Listing Q-4 below.

Listing Q-4: Output: Makefile

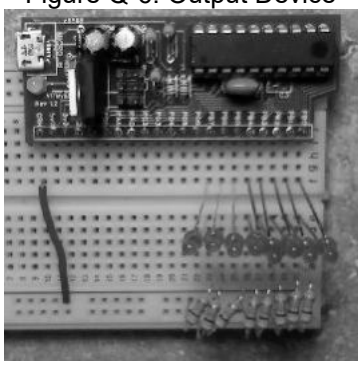
```
CC = avr-gcc
```

```
OBJCOPY = avr-objcopy
DUDE = avrdude
RM = del
CFLAGS = -Wall -Os -Iusbdrv -mmcu=attiny2313 -DF_CPU=12000000
OBJFLAGS = -j .text -j .data -O ihex
DUDEFLAGS = -p attiny2313 -c usbtiny

# Object files for the firmware
OBJECTS = usbdrv/usbdrv.o usbdrv/usbdrvasm.o main.o

# By default, build the firmware, but do not flash
all: main.hex

# With this, you can flash the firmware by just
#   typing "make flash" on command-line
flash: main.hex
	$(DUDE) $(DUDEFLAGS) -v -U flash:w:$<

# This will set the FUSE byte of the chip
fuse:
	$(DUDE) $(DUDEFLAGS) -U lfuse:w:0xef:m

# Housekeeping if you want it
clean:
	$(RM) *.o *.hex *.elf usbdrv\*.o

# From .elf file to .hex
%.hex: %.elf
	$(OBJCOPY) $(OBJFLAGS) $< $@

# Main.elf requires additional objects to the firmware,
#   not just main.o
main.elf: $(OBJECTS)
	$(CC) $(CFLAGS) $(OBJECTS) -o $@

# Without this dependence, .o files will not be recompiled
#  if you change the config file.
$(OBJECTS): usbdrv/usbconfig.h

# From C source to .o object file
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

# From assembler source to .o object file
%.o: %.S
	$(CC) $(CFLAGS) -x assembler-with-cpp -c $< -o $@
```

Notice that we now include the USB code in the *usbdrv* directory with our build.

Now create the device. Plug the breakout board into the bread-board as shown in Figure Q-6 below, placing eight LED's in a row starting from the right with PB0, moving left, ending with PB7. Then place eight resistors, one lead in the same group for each respective LED with the other lead in the Ground group. Then place a single jumper from the GND pin on the breakout board to the Ground group on the breadboard.

Figure Q-6: Output Device



Connect the programmer to the Host and the breakout board.

At the C:\ prompt enter the following:

C:\>MAKE<*enter*>

If you didn't receive any errors, continue with:

C:\>MAKE FUSE<*enter*>
C:\>MAKE FLASH<*enter*>

Disconnect the programmer.

Now connect the breakout board to the host and simply send a Setup Request Packet, defined in Table Q-2 shown below, to the breakout board with a request value of zero through seven, turning on or off the respected pin's LED.

Table Q-2: Toggle Pin Request

Offset	Field	Size	Value	Description
0	Request type	1	**0xC0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard 1 = Class **2 = Vendor** 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0-7**	Pin (0, 1, 2, ..., 7)
2	Value	2	**0x0000**	High byte: 0x00 = not used Low byte: 0x00 = not used
4	Index	2	**0x00**	Interface number (0x00)
6	Length	2	**0x00**	Length

Build *INPUT* Device

Create a directory called *input* in your *projects* folder. Again, copy the *usbdrv* folder from the *vusb* folder to this folder and rename the *usbconfig-prototype.h* file to *usbconfig.h*.

> Once you get more familiar with the way the code works and how the makefile's work, you can copy only the *usbconfig.h* file for each project, modifying the *makefile* slightly. Since the actual code within the *usbdrv* folder will be the same for every project, there is no need to copy it every time. Just be sure to re-compile the code for each project since the *usbconfig.h* file will change the #defines and how it is compiled.

For this project, make the same modifications you made for the *output* project before, with the following addition modifications.

- #define USB_CFG_IMPLEMENT_FN_READ 1
- #define USB_CFG_IMPLEMENT_FN_WRITE 0

The Host will be transferring data from the device.

- #define USB_CFG_INTERFACE_CLASS 3
- #define USB_CFG_INTERFACE_SUBCLASS 0
- #define USB_CFG_INTERFACE_PROTOCOL 0

The above lines are used for the Class, Subclass, and Protocol fields in the device's Interface Descriptor.

- #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH 24

Uncomment the above line and make sure it is set to 24.
Any other values should be the same as the *output* project, zero, or left as is.

Now create the *main.c* source file with the contents listed in Listing Q-5 listed below.

Listing Q-5: Output: main.c

```
#include <avr/io.h>
#include <avr/wdt.h>
#include <avr/interrupt.h>  // for sei()

#include <util/delay.h>     // for _delay_ms()

#include <avr/eeprom.h>
#include <avr/pgmspace.h>   // required by usbdrv.h

#include "usbdrv.h"

static struct {
  uint8_t pins[8];
} state_report;

PROGMEM char usbHidReportDescriptor[42] = {
  0x06, 0x00, 0xFF,     // USAGE_PAGE (Generic Desktop)
  0x09, 0x01,           // USAGE (Vendor Usage 1)
  0xA1, 0x01,           // COLLECTION (Application)
  0x15, 0x00,           // LOGICAL_MINIMUM (0)
  0x26, 0xFF, 0x00,     // LOGICAL_MAXIMUM (255)
```

```
  0x75, 0x08,           // REPORT_SIZE (8)

  // read state
  0x85, 0x00,           // REPORT_ID (0)
  0x95, sizeof(state_report), // REPORT_COUNT
  0x09, 0x00,           // USAGE (Undefined)
  0xB2, 0x02, 0x01,     // FEATURE (Data,Var,Abs,Buf)

  0xC0                  // END_COLLECTION
};

// ---------------------------------------------------------------

#define CONFIG_TOP ((uint8_t *) 0)
#define CONFIG_LEN 128

static uchar currentAddress;
static uchar bytesRemaining;
static uchar report_id;

void buildStateReport() {
  uchar pins;

  pins = PINB;
  state_report.pins[0] = (pins & _BV(PB0)) ? 1 : 0;
  state_report.pins[1] = (pins & _BV(PB1)) ? 1 : 0;
  state_report.pins[2] = (pins & _BV(PB2)) ? 1 : 0;
  state_report.pins[3] = (pins & _BV(PB3)) ? 1 : 0;
  state_report.pins[4] = (pins & _BV(PB4)) ? 1 : 0;
  state_report.pins[5] = (pins & _BV(PB5)) ? 1 : 0;
  state_report.pins[6] = (pins & _BV(PB6)) ? 1 : 0;
  state_report.pins[7] = (pins & _BV(PB7)) ? 1 : 0;
}

uchar usbFunctionRead(uchar *data, uchar len) {
    if (len > bytesRemaining) len = bytesRemaining;
    eeprom_read_block(data, CONFIG_TOP + currentAddress, len);
    currentAddress += len;
    bytesRemaining -= len;
    return len;
}

usbMsgLen_t usbFunctionSetup(uchar data[8]) {
  usbRequest_t *rq = (void *)data;

  if ((rq->bmRequestType & USBRQ_TYPE_MASK) == USBRQ_TYPE_CLASS)
  {  // HID class request
    report_id = rq->wValue.bytes[0];
    if (rq->bRequest == USBRQ_HID_GET_REPORT) {
      if (report_id == 0) {
        buildStateReport();
        usbMsgPtr = (void *) &state_report;
        return sizeof(state_report);
      }
    }
  }
```

```
  return 0;
}

int main(void) {
  wdt_enable(WDTO_1S);
  usbInit();

  // make all PORTB pins input
  DDRB = 0;

  sei();
  while (1) {        // main event loop
    wdt_reset();
    usbPoll();
  }

  return 0;
}
```

The code in Listing Q-5 simply waits for a SETUP packet from the host, that sends the values in Table Q-3 below.

Table Q-3: Get Report Descriptor Request

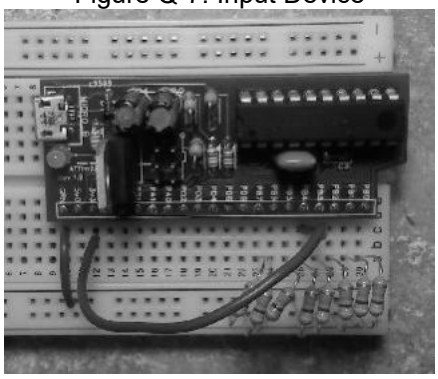
Offset	Field	Size	Value	Description
0	Request type	1	**0xA0**	Bit 7 Transfer Direction 0 = Host to device **1 = Device to host** Bit 6:5 Type 0 = Standard **1 = Class** 2 = Vendor 3 = Reserved Bit 4:0 Recipient **0 = Device** 1 = Interface 2 = Endpoint 3 = Other 4..31 = Reserved
1	Request	1	**0x01**	HID Get Report
2	Value	2	**0x0000**	High byte: 0x00 = not used Low byte: 0x00 = index 0
4	Index	2	**0x00**	Interface number (0x00)
6	Length	2	**0x08**	Descriptor length retrieve from HID Descriptor above.

The code will then create an 8-byte packet that holds a boolean value for each respective pin's input and place it at address zero of the chip. Then when the host sends the IN packet, it will transfer the requested amount of bytes, up to eight, to the Host.

Now copy the *Makefile* from the *Output* project folder to this folder. You should not have to make any modifications to it. Make the code and flash it to the breakout board. Then setup the

breakout board and breadboard similar to Figure Q-7 below, but without the wire connecting 3v3 and PB4 as shown, for now.

Figure Q-7: Input Device



Once you have connected the device to the host, send the request in Table Q-3. You should get an 8-byte array of all zeros. Now, take the long lead wire and connect the breakout board's 3v3 pin to PB4 on the breadboard, as shown in the figure above, and request the packet again. You should now get an array of seven zeros and a one. Try connecting the lead to other pins and requesting the packet again.

The eight resistors are needed to pull-down the eight pins. The Attiny2313 will have small voltages on the pins due to various other reasons, which will give false values in your 8-byte array packet. Therefore, if you use a 47k Ohm resistor to pull the pin to ground, you will eliminate this issue. Since the resistor has a high resistance, when you apply voltage to the pin group, very little voltage will be lost through the resistor to ground. I would use as large of a resistance as you can and still get good results. I have found that a 47k Ohm resistor works well.

Conclusion

With the information you now have, you should be able to use your imagination and create such things as a wheeled robot with proximity sensors, or other items such as an LED cube. You are only limited by your imagination.

Please note that some motors need additional circuits to control them correctly. Search for “stepper motor control” via your favorite web search tool.

Other Comments

Something that you might also want to look into is the ATTiny85 or similar chip. The ATTiny2313 used in this chapter does not have an ADC, Analog to Digital Converter, where as the ATTiny85 does. The ATTiny45 and ATTiny85 chips communicate via USB just as the ATTiny2313 used here with only a few modifications to the source code and Makefile, however they do not have as many I/O pins.

The specification for the ATTiny2313 used in this chapter can be found at.

http://www.atmel.com/Images/doc2543.pdf.

I also want to thank the following people for their help in one way or another on this chapter.

Joonas Pihlajamaa http://codeandlife.com/
Dan Stahlke http://www.stahlke.org/dan/
Bernhard Schornak http://thepoolofhumanity.blogspot.com/

You can find more information about this project at:

http://www.fysnet.net/attiny2313.htm
http://www.fysnet.net/The_Universal_Serial_Bus.htm

Copyright © Forever Young Software © Benjamin David Lunt © 1984-2014
This Appendix may be freely distributed as long as it is
unmodified and in its entirety.
All rights reserved

Latest reversion: 10 Oct 2014

Appendix X - For More Information

Where to get the CDROM that is included with this book

The current form of publication does not include a CDROM when publishing a book. However, if you use the contact information at the URL listed below, you can contact the author for a copy of the CDROM via the US Postal Service or an ISO image file via email. There is a shipping cost for all CDROM's shipped via USPS, however, no cost is added if sent via email.

Where to find more information on this book

For more information, extra documentation, and information on all of the specifications used within this book, use the URL listed below.

Where to get an erratum if one is needed

If and when an erratum is needed, it will be posted to the URL below with dated versions.

Where to get more examples

For more examples and source code, go to the URL below.

http://www.fysnet.net/the_universal_serial_bus.htm

Bibliography

One or more of the following sources may have been used for the information for this book, tables and/or figures within it.

- Universal Serial Bus Specification, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, Revision 2.0, April 27, 2000.
- Universal Host Controller Interface (UHCI) Design Guide, Intel, Revision 1.1, March 1996.
- Open Host Controller Interface Specification for USB, Compaq, Microsoft, National Semiconductor, Revision 1.0a, 09/14/1999 2:33pm
- Enhanced Host Controller Interface Specification for Universal Serial Bus, Intel, Revision 1.0, March 12, 2002.
- Extended Host Controller Interface Specification for Universal Serial Bus, Intel, Revision 1.0, May 21, 2010.
- Language Identifiers (LANGIDs), 2000 USB Implementers' Forum, Version 1.0, 2/29/2000
- Device Descriptor Class codes: http://www.usb.org/developers/defined_class
- http://america2.renesas.com/usb/business.html
- http://www.intel.com/technology/comms/wusb/whci.htm
- http://www2.renesas.com/usb/en/product/upd720102.html
- https://patchwork.kernel.org/patch/51185/
- http://kerneltrap.org/mailarchive/linux-kernel/2010/7/1/4589465
- http://www.intel.com/technology/comms/wusb/whci.htm
- Other files and/or specifications as listed in Appendix A "Finding the Specification Files"

Contributors

The following list of people (in no particular order) have helped in one way or another, by using their machine as a test machine, sent in bug reports, and/or giving advice in one way or another. I am forever grateful to them. I am truly sorry if I have forgotten to list someone here. If you helped with this book in any way and are not listed, please let me know.

Loren "Boreal" Blaney
Steve Dubrovich
James Harris
Tyrel Haveman
Wolfgang Kern
Rod Pemberton
Tony Prisk
Anssi Saari
Maxim S. Shatskih
Aharon Lavie
Mauro Dicicco
Steve -- DexOS
Stephan R. Novosad
Clemens Ladisch
Sushma
Larry Lee
Mike Gonta
Walter's World
Jordi Sanfeliu
Tim Cropley

The following list of people or groups of people may have directly or indirectly helped in one way or another.

Sarah Sharpe
alt.os.development group
gmane.linux.usb.general
Bruce M. Hamilton, Tech Coast Sales, Laguna Hills, CA

The following companies, in no particular order, donated hardware or contributed in some way or another.

Total Phase
- **http://www.totalphase.com/products/beagle_usb12/**
- Donated the Beagle USB12 for my research. I want to thank Total Phase for their patience. I requested this item as one of the first things I did toward this book. It has been a few years.

Texas Instruments
- **www.ti.com**
- Donated a PCIe xHCI Host Controller. They were very generous with their offer and sent it my way, quickly and efficiently. The sales rep also sent me numerous emails with information about the controller and other items I may be interested in pertaining to the controller.

Digital Innovations
- **www.digitalinnovations.com**
- Gladly gave permission to use the 4-port hub, and Micro Mouse

J-Tech Digital
- Gladly gave permission to use the Super Speed 4-port hub

Kingston Technology
- **http://www.kingston.com/us/**
- Gladly gave permission to use the high-speed thumb drive used in Chapter 17.

Thank you to all, Ben

---

[1] A PCI bus is used for the UHCI, OHCI, and UHCI controllers. I use a PCIe bus for a PCIe xHCI controller, though you may use a PCI bus for a PCI xHCI controller.
[2] Staring with USB 3.0, a device can draw more than 500mA. More on this in a later chapter.
[3] PCI Power Management Interface Specification, revision 1.2, section 5.4
[4] If you are using a 32-bit machine, you must use two consecutive 32-bit reads or writes.
[5] When using 32-byte context structures. See further in this chapter why this might be a 2048-byte block.
[6] Two hundred fifty six entries of 64-bits each
[7] http://www.kingston.com/datasheets/kusbdti_us.pdf
[8] https://lists.one-eyed-alien.net/pipermail/usb-storage/2004-September/000795.html
[9] http://america2.renesas.com/usb/business.html
[10] The actual specification states September of 1999, but Compaq says different.
[11] There were 14 releases of EHCI before version 1.0 was released.
[12] http://arstechnica.com/old/content/2007/09/intel-announces-demonstrates-usb-3-0.ars
[13] http://www.theinquirer.net/inquirer/news/1137399/nec-announces-usb-host-controller
[14] http://www.atmel.com/