An Embedded USB Controller Linux Device Driver: Common Practices and Experience

Dimitris Lampridis

May 13, 2005

Contents

1 Introduction **1**

1.1 Introduction to Embedded Devices . . . 1

1.1.1 Embedded System Platforms . . . 2

1.1.2 Embedded Operating Systems . . . 2

1.1.3 The Need for Device Drivers . . . 3

1.1.4 Linux for Embedded Devices . . . 4

1.2 Device Drivers: Basic Principles . . . 5

1.2.1 The Role of a Device Driver . . . 5

1.2.2 Kernel & User Space . . . 5

1.2.3 Security Issues . . . 6

1.3 Thesis Summary . . . 7

1.3.1 Thesis Goals . . . 7

1.3.2 Thesis Results . . . 8

1.3.3 Thesis Outline . . . 9

2 The Universal Serial Bus **10**

2.1 Background . . . 10

2.1.1 Goals for the Universal Serial Bus . . . 11

2.1.2 USB Features . . . 11

2.2 Architectural Overview . . . 13

2.2.1 USB Topology . . . 13

2.2.2 USB Hosts . . . 13

2.2.3 Dynamic Device Attachment . . . 14

2.3 USB Data Flow Modes . . . 15

2.3.1 Transfer Modes . . . 16

2.4 USB Device Framework . . . 18

2.4.1 USB Hubs . . . 18

2.4.2 USB Functions . . . 19

2.4.3 USB Device Endpoints . . . 20

2.4.4 USB Pipes . . . 21

2.5 USB Descriptors . . . 22

2.5.1 Composition of USB Descriptors . . . 23

2.6 USB Host: Hardware & Software . . . 23

2.6.1 Host Controller Requirements 23
2.6.2 Overview of the USB Host 24
2.6.3 USB Bus Interface 25
2.6.4 USB System 27
2.6.5 Host Controller Driver 27
2.6.6 USB Driver 28
2.7 The Root Hub 28
2.7.1 Hub Requests & Descriptors 29
2.7.2 Hub Architecture & Supported Speeds 30
2.7.3 Hub Connectivity 30
2.7.4 Endpoint Organization 31
2.7.5 Port Change Information Processing 31
2.7.6 Hub and Port Status Change Bitmap 31
2.7.7 Over-current Reporting and Recovery 32
2.7.8 Enumeration Handling 32

3 The Linux USB Subsystem 33
3.1 USB-Standard Types 34
3.1.1 Control Request Support 34
3.1.2 Standard Descriptors 34
3.2 Upper (Host-Side) API Model 37
3.2.1 Host-Side Data Types 38
3.3 USB Core APIs 39
3.3.1 The Structure of USB Request Blocks (URBs) 39
3.3.2 URB Manipulation Functions 43
3.4 Host Controller APIs 44
3.4.1 Host Controller Data Types 44
3.4.2 Host Controller Functions 49

4 Philips ISP1160 Embedded USB Host Controller 51
4.1 Host Controller Internal Registers 52
4.1.1 Root Hub Registers 53
4.2 Microprocessor Bus Interface 54
4.2.1 Programmed I/O (PIO) Addressing Mode 54
4.2.2 DMA Mode 56
4.2.3 Control Registers Access by PIO Mode 57
4.2.4 FIFO Buffer RAM Access by PIO Mode 58
4.2.5 FIFO Buffer RAM Access by DMA Mode 58
4.2.6 Interrupts 58
4.3 Host Controller (HC) Functionality 60
4.3.1 HC's USB States 60
4.3.2 USB Traffic Generation 60
4.3.3 The Structure of Philips Transfer Descriptors 62
4.3.4 Internal FIFO Buffer RAM Data Organization 66

4.3.5 HC Operational Model 68
4.4 The ISP1160 PCI Development Board 69
4.4.1 PCI Bus Overview 70
4.4.2 The PLX9054 PCI I/O Accelerator 71

5 ISP1160 PCI Driver Implementation 73
5.1 Important Data Structures 74
5.2 PCI Module Initialization 78
5.2.1 PCI Device Probing 78
5.2.2 PCI I/O Resource Allocation 79
5.2.3 ISP1160 Initialization 83
5.2.4 PCI Module Cleanup 85
5.3 Data Transfer Management 86
5.3.1 The Interrupt Handler 87
5.3.2 Root Hub Status Change and the "KHUBD" 88
5.3.3 Data Transfer Interrupts: ATLint & SOFint 89
5.3.4 Arrival of new URBs 89
5.3.5 Interrupt Polling Rate 90
5.3.6 Data Transfer to HC's Memory 91
5.3.7 Data Transfer from HC's Memory 92
5.4 Results 93
5.4.1 Insertion & Removal of the Driver 93
5.4.2 Hardware Functionality 94
5.4.3 Driver Policy 95
5.4.4 OS Integration 95
5.4.5 Security 97
5.4.6 Driver Porting 97
5.5 Testing & Performance 98
5.5.1 Testing 98
5.5.2 Performance Evaluation 99

6 Conclusions 106
6.1 Further Work 107

A USB Requests 110
A.1 USB Device Requests 110
A.2 Standard Device Requests 112
A.2.1 Request Descriptions 112
A.3 Hub Requests 118
A.3.1 Standard Requests 118
A.3.2 Class-Specific Requests 118

B USB Descriptors 121

B.1 Device Descriptor 121
B.2 Configuration Descriptor 122
B.3 Interface Descriptor 125
B.4 Endpoint Descriptor 126
B.5 String Descriptor 126
B.6 Class- & Vendor-specific Descriptors 128
B.6.1 Hub Descriptor 129

C ISP1160 Internal Registers 131

C.1 HC Control and Status Registers 131
C.1.1 HcRevision 131
C.1.2 HcControl 131
C.1.3 HcCommandStatus 132
C.1.4 HcInterruptStatus 133
C.1.5 HcInterruptEnable 134
C.1.6 HcInterruptDisable 135
C.2 HC Frame Counter Registers 135
C.2.1 HcFmInterval 135
C.2.2 HcFmRemaining 136
C.2.3 HcFmNumber 137
C.2.4 HcLSThreshold 137
C.3 HC Root Hub Registers 137
C.3.1 HcRhDescriptorA 137
C.3.2 HcRhDescriptorB 139
C.3.3 HcRhStatus 140
C.3.4 HcRhPortStatus(1:2) 142
C.4 HC DMA and Interrupt Control Registers 147
C.4.1 HcHardwareConfiguration 147
C.4.2 HcDMAConfiguration 149
C.4.3 HcTransferCounter 150
C.4.4 HcμPInterrupt 150
C.4.5 HcμPInterruptEnable 152
C.5 HC Miscellaneous Registers 153
C.5.1 HcChipID 153
C.5.2 HcScratch 154
C.5.3 HcSoftwareReset 154
C.6 HC Buffer RAM Control Registers 154
C.6.1 HcITLBufferLength 154
C.6.2 HcATLBufferLength 154
C.6.3 HcBufferStatus 155
C.6.4 HcReadBackITL0Length 155
C.6.5 HcReadBackITL1Length 156
C.6.6 HcITLBufferPort 156

C.6.7 HcATLBufferPort 156

List of Figures

1.1 Embedded System Architectures Survey 2
1.2 Embedded Linux Distributions 4

2.1 Topology of the USB Bus 14
2.2 A Typical USB Hub 18
2.3 Hubs in a Desktop Computer Environment 19
2.4 Addressing of Device Endpoints 20
2.5 Hierarchy of the USB Descriptors 22
2.6 USB Interlayer Communication Model 25
2.7 USB Host Communications 26
2.8 USB Driver Structure 29

3.1 Linux USB Core API Layers 33
3.2 Linux USB Driver Attachment to a USB Device 38

4.1 Block Diagram 55
4.2 ISP1160 Programmed I/O interface 56
4.3 ISP1160 DMA interface 56
4.4 Access to ISP1160's Internal Registers 57
4.5 ISP1160 Interrupt Logic 59
4.6 ISP1160 USB States 60
4.7 ISP1160 HC Transaction Loop 61
4.8 PTD Data in FIFO Buffer RAM 63
4.9 HC Internal FIFO RAM Partitions 67
4.10 HC FIFO RAM Data Organization 68
4.11 ISP1160 PCI Development Board 69
4.12 PCI Configuration Registers 71

5.1 Interrupt Polling Intervals 91
5.2 "USBView" Application Running 96

List of Tables

2.1 USB supported speeds 10

4.1 HC Registers Summary 53
4.2 ISP1160 INT pin configuration 60
4.3 PTD Bit Allocation 63
4.4 PTD Bit Description 64

A.1 Format of Setup Data 111
A.2 Standard Device Requests 113
A.3 Standard Request Codes 114
A.4 Descriptor Types 114
A.5 Hub Responces to Standard Device Requests 118
A.6 Hub Class Requests 118
A.7 Hub Class Request Codes 120

B.1 Device Descriptor Format 123
B.2 Configuration Descriptor Format 124
B.3 Interface Descriptor Format 125
B.4 Endpoint Descriptor Format 127
B.5 String Descriptor Zero Format 128
B.6 Subsequent String Descriptors Format 128
B.7 Hub Descriptor Format 129

C.1 HcRevision Register 131
C.2 HcControl Register 131
C.3 HcCommandStatus Register 133
C.4 HcInterruptStatus Register 133
C.5 HcInterruptEnable Register 135
C.6 HcFmInterval Register 136
C.7 HcFmRemaining Register 136
C.8 HcFmNumber Register 137
C.9 HcLSThreshold Register 137
C.10 HcRhDescriptorA Register 138
C.11 HcRhDescriptorB Register 139
C.12 HcRhStatus Register 140

C.13 HcRhPortStatus(1:2) Register 142
C.14 HcHardwareConfiguration Register 147
C.15 HcDMAConfiguration Register 149
C.16 HcTransferCounter Register 150
C.17 HcμPInterrupt Register 151
C.18 HcμPInterruptEnable Register 152
C.19 HcChipID Register 154
C.20 HcScratch Register 154
C.21 HcSoftwareReset Register 154
C.22 HcITLBufferLength Register 154
C.23 HcATLBufferLength Register 155
C.24 HcBufferStatus Register 155
C.25 HcReadBackITL0Length Register 156
C.26 HcReadBackITL1Length Register 156
C.27 HcITLBufferPort Register 156
C.28 HcATLBufferPort Register 157

Acknowledgements

I'd like to thank my supervisor, associate professor Dionisios N. Pnevmatikatos for his guidance and patience in helping me understand my project, and also for encouraging me to keep on going, at a time when all my efforts seemed futile. I'd like to thank all the open-source community, and especially everyone from the "Linux USB Development" mailing list, for their invaluable support and countless discussions; I have gained much from their insight. I'd also like to thank my remaining committee members, professor Apostolos Dollas and assistant professor Vassilis Samoladas for taking the time to read and evaluate this document.

I would like to thank the Microprocessor and Hardware Laboratory (MHL), of the Electronic and Computer Engineering Department, for the full financial assistance provided in acquiring the required hardware. I am grateful to the director of the laboratory Mr. Markos Kimionis, for providing a perfectly equipped laboratory and helping me with the logic analyzer. I'd also like to mention how much I appreciate the altruistic work of countless members of the open-source community, which provided me with invaluable tools for my work (just to name a few: the Debian GNU/Linux distribution, the GNU tools like GCC and Emacs, the GNOME desktop environment, and of course Latex, Ghostscript and Gimp, used in the creation of this paper).

I'd like to let my mother and father know how much I appreciate them for their many years of emotional and financial support.

And finally, I'd like to thank my girlfriend Katerina for all her love, support and encouragement. Without her, this project would have been much more difficult, if not impossible.

Abstract

Embedded systems design is a rapidly growing field of Computer Engineering. An embedded device may be built from "common of the self components", or use a custom range of hardware, different from the one found in most desktop computers, or a combination of the above. This fact has led to a growing demand for device drivers that will be able to complement the embedded opearating systems in their task to support the hardware. Very often, existing drivers are ported to the embedded architecture; other drivers are written from scratch.

This thesis is an attempt to understand the workings of a device driver in the Linux operating system, a popular choice among the embedded operating systems, and to apply this knowledge to write a driver for the PCI development board of the Philips ISP1160 Embedded USB Host Controller. The device driver is written for the new 2.6 series of the Linux kernel and can handle the Control, Bulk and Interrupt USB transfer modes, supporting a variety of USB devices, including printers, memory sticks, input devices, digital cameras and modems. The driver demonstrates competitive speed compared to the popular UHCI USB host controllers found in most modern desktop computers, using a smaller memory footprint and a simpler driver model.

Chapter 1

Introduction

1.1 Introduction to Embedded Devices

An embedded device is a special-purpose computer system, which is completely encapsulated by the device it controls. An embedded device has specific requirements and performs pre-defined tasks, unlike a general-purpose personal computer. Examples of embedded devices include:

- Automatic Teller Machines (ATMs)
- Cellular telephones and telephone switches
- Computer network equipment, including routers, timeservers and firewalls
- Computer printers
- Handheld calculators
- Household appliances, including microwave ovens, washing machines, television sets, DVD players/recorders
- Inertial Guidance Systems, flight control hardware/software and other integrated systems in aircraft and missiles
- Medical equipment
- Measurement equipment such as digital storage oscilloscopes, logic analyzers, and spectrum analyzers
- Personal digital assistants (PDAs)
- Videogame consoles

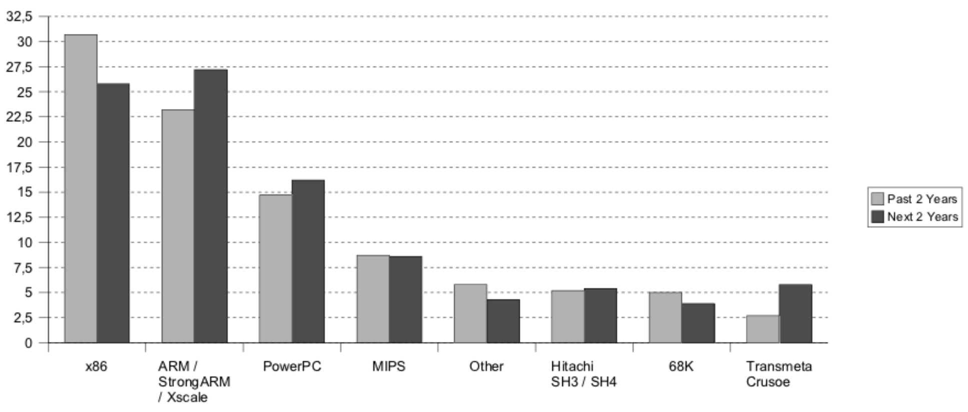


Figure 1.1: Embedded System Architectures Survey

Programs on an embedded system often must run with real-time constraints with limited hardware resources; often there is no disk drive, operating system, keyboard or screen. A flash drive may replace rotating media, and a small keypad and LCD screen may be used instead of a normal keyboard and screen respectively.

1.1.1 Embedded System Platforms

There are many different CPU architectures used in embedded designs. This is in contrast to the desktop computer market, which is limited to just a few competing architectures, mainly the *Intel/AMD x86*, and the *Apple/Motorola/IBM PowerPC*.

Figure 1.1 shows the embedded market trends in CPU architectures[1].

1.1.2 Embedded Operating Systems

Embedded devices may or may not have an operating system (most modern devices do have one). *Firmware* is the name for software that is embedded in hardware devices, e.g. in one or more ROM/Flash memory IC chips.

Embedded systems are routinely expected to maintain reliability while running continuously for long periods of time, sometimes measured in years. Firmware is usually developed and tested to much stricter requirements than is general purpose software. In addition, due to the fact that the embedded system may be outside the reach of humans, embedded firmware must usually be able to self-restart even if some sort of catastrophic data corruption has taken

[1]Source: 2003 Embedded Market Survey, by LinuxDevices.com [1]

place. This last feature often requires external hardware assistance such as a watchdog timer that can automatically restart the system in the event of a software failure.

Many time-critical applications of embedded systems require a *Real Time Operating System* to control the device.

Real-Time Operating Systems

A Real Time Operating System (RTOS) is an operating system that has been developed for real-time applications. A commonly accepted definition of "real-time" performance is that real-world events must be responded to within a defined, predictable, and relatively short time interval.

Popular RTOSes include:

- BeOS
- OS-9
- OSE
- FreeRTOS
- Nucleus
- Windows CE
- RT Linux
- VxWorks
- LynxOS

1.1.3 The Need for Device Drivers

An operating system is of no value without a complete and frequently updated list of device drivers to support the hardware. This is also true for embedded devices.

As mentioned above, an embedded system is designed for a specific task and has special requirements (as compared to a classic desktop computer). During the past few years, the hardware industry has made steps to provide the embedded system designers with tools that fit well with these special needs (e.g., a network interface for the embedded market might not be as fast as a classic one, but it will be able to perform well while consuming less energy).

This special hardware, needs of course device drivers. And although the computer industry is moving towards a unified approach

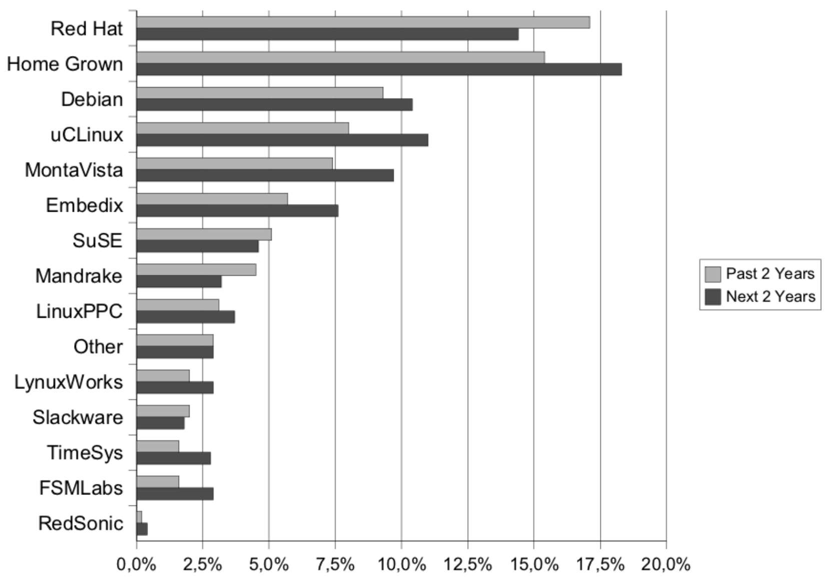


Figure 1.2: Embedded Linux Distributions

in hardware design by adopting specifications (thus greatly reducing the number of different drivers required), the embedded market has its own rules and specifications. As a result, the need for custom device drivers to support the hardware is always a top priority in embedded systems design.

1.1.4 Linux for Embedded Devices

The power, reliability, flexibility, and scalability of Linux, combined with its support for a multitude of microprocessor architectures, hardware devices, graphics support, and communications protocols have established it as an increasingly popular software platform for a vast array of projects and products.

Given that Linux is openly and freely available in source form, many variations and configurations of the operating system and its supporting software components have evolved to meet the diverse needs of the markets and applications to which Linux is being adapted. There are small-footprint versions and real-time enhanced versions. And despite the origins of Linux as a PC architecture operating system, there are now ports to numerous non-x86 CPUs, with and without memory management units, including PowerPC, ARM, MIPS, 68K, and even microcontrollers.

Figure 1.2 shows the Linux distributions that are favored by the embedded market[2]. We can see that there is a shift towards custom-built Linux (listed under "home grown"). Since Linux is open-source software, more and more embedded system manufacturers prefer to download the source of the Linux kernel and modify it to fit their needs.

1.2 Device Drivers: Basic Principles

No matter the hardware addressed, every device driver should adhere to certain basic principles.

1.2.1 The Role of a Device Driver

Most programming problems can be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is easier to develop and to adapt to particular needs.

The role of a device driver is to provide *mechanism*, not *policy*. The driver should deal with making the hardware available, leaving all the issues about *how* to use the hardware to the applications.

Policy-free drivers should have a number of typical characteristics:

- Support both synchronous and asynchronous operation
- Ability to be opened multiple times
- Exploit the full capabilities of the hardware
- Lack of software layers to provide policy-related operations

However, user programs are an integral part of a software package and all drivers should be distributed with configuration files that apply a default behavior to the underlying mechanism.

1.2.2 Kernel & User Space

A device driver runs in the so-called *kernel space*, whereas applications run in *user space*.

[2]Source: 2003 Embedded Market Survey, by LinuxDevices.com [1]

The role of the operating system is to provide programs with a consistent view of the computer's hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This is possible if the CPU enforces protection of system software from the applications.

All current processors have at least two protection levels (the x86 family have more). The kernel (and its device drivers) executes in the highest level where everything is allowed, whereas applications execute in the lowest level where the processor regulates direct access to hardware and unauthorized access to memory.

1.2.3 Security Issues

Security is an increasingly important concern in modern times. It is a problem that can be split in two parts. One security problem is the damage a user can cause through the misuse of existing programs, or by *incidentally* exploiting bugs; a different issue is what kind of misfunctionality a programmer can *deliberately* implement. It is a dangerous to run a program received from somebody else from the administrator account, as it is to give him or her a superuser shell. And since every device driver runs in kernel space, a kernel module is just as powerful as a superuser shell.

When possible, a driver should not encode security policy in its source code. Security is a policy issue that is often best handled at higher levels within the kernel, under the control of the system administrator.

The driver should not include of course security bugs. Many security problems are created by *buffer overrun* errors, when data ends up written beyond the end of an allocated buffer, thus overwriting unrelated data. A few other security ideas include:

- Any input received from user processes should be treated with great suspicion
- Any memory obtained from the kernel should be zeroed or otherwise initialized before being made available to a user process or device.
- If there are specific operations that could affect the system (e.g., formatting a disk), those operations should be restricted to privileged users.

1.3 Thesis Summary

This thesis is an attempt to understand the workings of a device driver in the Linux operating system, a popular choice among the embedded operating systems, and to apply this knowledge to write a driver for the PCI development board of the *Philips ISP1160 Embedded USB Host Controller*. The PCI board was connected to a typical PC x86 32 bit system (AMD Duron), running Debian GNU/Linux, Testing (Sarge).

Although when the project first started, the best choice in Linux kernels was the 2.4 series, a few months later the 2.6 series appeared, which among many other changes, incorporates a very different (and much better) scheme in supporting *hotplug* devices (such as a USB device). Thus, even though the documentation for the 2.6 series was very sparse, it was chosen as the kernel to develop the driver for.

The PCI development board of the ISP1160 ships with a driver for the DOS operating system but lacks a Linux device driver (altough there exists a Linux driver for the ISA version of the board). The drivers shipped with the board (both PCI and ISA) are far from complete, as they only intend to demonstrate some of the capabilities of the host controller. Using the supplied software, the user can monitor the USB downstream ports for changes (e.g. connection of a USB device), get information about the connected devices by reading their descriptors, check the contents of the buffer RAM of the host controller, and try a USB mouse.

1.3.1 Thesis Goals

This driver is an attempt to illustrate the way that a device driver can be written (focusing in the API of Linux kernel 2.6). A set of goals was introduced from the beginning of the development. According to these goals, the driver should:

1. be written using the new API of Linux kernel 2.6
2. integrate with the operating system, making the USB host controller available and functional, just like any other device driver present on the system (especially any USB host controller driver)
3. separate all PCI-specific code from the rest of the source, to ease porting of the driver to non-PCI implementations
4. be fully modular

5. be as policy-free as possible
6. support at least the *Control* and *Bulk* USB transfer modes, and if possible the *Interrupt* and *Isochronous* transfer modes. (see Section 2.3.1, on page 16)
7. be as secure as possible
8. support the PIO mode of operation, and if possible the DMA mode as well.
9. Emphasize in aspects of speed, to demonstrate the importance of a driver in overall system performance

Note however that this driver is not supposed to be a "commercial-grade" software product. As such, some features of a complete device driver might be missing.

1.3.2 Thesis Results

The resulting work of the thesis is a functional driver module for the PCI development board of the IS1160 host controller, which fully integrates with the operating system. Users are able to insert and remove the module from the running kernel, and the host controller is able to coexist with other possibly present USB host controllers, providing the same functionality through standard user-space programs.

The work follows the "separation of policy from mechanism" model presented in Section 1.2.1. The driver makes the hardware available by taking advantage of the API provided by the kernel. This way the kernel registers the host controller with the operating system in a transparent way, leaving all issues of using the hardware to applications. This ensures an adequate level of security, since the driver does not attempt to encode its own security policy, leaving all such matters to the operating system. The Linux OS already includes a strict set of security measures related with the insertion and removal of modules. All input from user space applications is passed to the USBD kernel driver of the Linux USB subsystem, which forwards the request to the correct host controller driver. This way, the driver is able to trust all incoming data, since any user input is first filtered with the security mechanisms of the USB subsystem.

The supported hardware functionality includes Control, Bulk and Interrupt transfer modes, leaving only the Isochronous mode out of the list. This allows the majority of available USB devices to be connected to the host controller. Access to the internal registers and buffer RAM of the host controller is accomplished through PIO operations. DMA access to the buffer RAM was not implemented.

The source code separates the initialization sequence from the actual programming of the ISP1160 , allowing the driver to be easily modified to support other implementations of the ISP1160 as well. Such implemetations need only to alter the way that the hardware resources are made available, by modifying the files that contain the PCI initialization code.

This driver has been tested and works with a custom-built 2.6.10 version of the Linux kernel. A detailed explanation of the results can be found in Section 5.4, on page 93.

1.3.3 Thesis Outline

This document is divided in two parts. The first part (Chapters 2-4) is dedicated to the theoritical knowledge required to understand to second part (Chapter 5), where we analyze the task of writing the actual driver for the ISP1160. The first part begins with an overview of the USB Bus specifications (Chapter 2), followed by a presentation of the Linux USB subsystem (Chapter 3) and the way that it implements the USB standards. Then, we present an architectural and functional overview of the ISP1160 (Chapter 4), focusing on the parts that are important to the device driver. At the end of Chapter 4, we examine the PCI development board that was used. The second part is the presentation of the device driver. Throughout this chapter we examine parts of the source code and explain the key elements of the driver. This is no substitute for the actual source code, and for a full explanation of the way that this driver works, refer to the manual of the source code, available in HTML and PDF format. At the end of the second part, we evaluate the perforance of ther driver and, based on the goals that were set for the thesis, comment on the results. The last part (Chapter 6) contains the conclusions and possible further work on the subject.

Chapter 2

The Universal Serial Bus

This chapter presents a brief description of the background, architecture and operation of the Universal Serial Bus (USB). This is not a full description of the bus specifications. For more information see (5)

2.1 Background

Universal Serial Bus was created when a group of 7 companies: *Compaq*, *Digital Equipment*, *IBM*, *Intel*, *Microsoft* and *Northern Telecom* decided to form a specifications to merge legacy connectivity such as RS232, Printer port, PS2 port into a single common connector to the Personal Computer.

USB version 1.1 was released on 15 January 1996 and supported two speeds, a full speed mode of 12 Mbits/s and a low speed mode of 1.5 Mbits/s. The 1.5 Mbits/s mode is slower and less susceptible to EMI, thus reducing the cost of ferrite beads and quality components. For example, crystals can be replaced by cheaper resonators. USB 2.0, released on 27 April 2000, supports speeds up to 480 Mbits/s. Table 2.1 summarizes the available speeds.

Table 2.1: USB supported speeds

High Speed	480.0 Mbits/s
Full Speed	12.0 Mbits/s
Low Speed	1.5 Mbits/s

2.1.1 Goals for the Universal Serial Bus

The USB is specified to be an industry-standard extension to the PC architecture with a focus on PC peripherals that enable consumer and business applications. The following criteria were applied in defining the architecture for the USB:

1. Ease-of-use for PC peripheral expansion
2. Low-cost solution that supports transfer rates up to 480 Mb/s
3. Full support for real-time data for voice, audio, and video
4. Protocol flexibility for mixed-mode isochronous data transfers and asynchronous messaging
5. Integration in commodity device technology
6. Comprehension of various PC configurations and form factors
7. Provision of a standard interface capable of quick diffusion into product
8. Enabling new classes of devices that augment the PC's capability
9. Full backward compatibility of USB 2.0 for devices built to previous versions of the specification

2.1.2 USB Features

The USB Specification provides a selection of attributes that can achieve multiple price/performance integration points and can enable functions that allow differentiation at the system and component level.

Features are categorized by the following benefits:

1. Easy to use for end user
    - Single model for cabling and connectors
    - Electrical details isolated from end user (e.g., bus terminations)
    - Self-identifying peripherals, automatic mapping of function to driver and configuration
    - Dynamically attachable and reconfigurable peripherals
2. Wide range of workloads and applications

- Suitable for a broad spectrum of bandwidths
- Supports isochronous as well as asynchronous transfer types over the same set of wires
- Supports concurrent operation of many devices (multiple connections)
- Supports up to 127 physical devices
- Supports transfer of multiple data and message streams between the host and devices
- Allows compound devices (i.e., peripherals composed of many functions)
- Lower protocol overhead, resulting in high bus utilization

3. Isochronous bandwidth
   - Guaranteed bandwidth and low latencies appropriate for telephony, audio, video, etc.
4. Flexibility
   - Supports a wide range of packet sizes, which allows a range of device buffering options
   - Allows a wide range of device data rates by accommodating packet buffer size and latencies
   - Flow control for buffer handling is built into the protocol
5. Robustness
   - Error handling/fault recovery mechanism is built into the protocol
   - Dynamic insertion and removal of devices is identified in user-perceived real-time
   - . Supports identification of faulty devices
6. Synergy with PC industry
   - Protocol is simple to implement and integrate
   - Consistent with the PC plug-and-play architecture
   - Leverages existing operating system interfaces
7. Low-cost implementation
   - Low-cost subchannel at 1.5 Mb/s
   - Optimized for integration in peripheral and host hardware

  - Suitable for development of low-cost peripherals
  - Low-cost cables and connectors
  - Uses commodity technologies

8. Upgrade path

  - Architecture upgradeable to support multiple USB Host Controllers in a system

2.2 Architectural Overview

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

2.2.1 USB Topology

The USB connects USB devices with the USB host. The USB physical interconnect is a tiered star topology. A hub is at the center of each star. Each wire segment is a point-to-point connection between the host and a hub or function, or a hub connected to another hub or function (for more inormation on hubs or functions, refer to 2.4.1 and 2.4.2). Figure 2.1 illustrates the topology of the USB.

Due to timing constraints allowed for hub and cable propagation times, the maximum number of tiers allowed is seven (including the root tier). In seven tiers, a maximum of five non-root hubs can be supported in a communication path between the host and any device. A compound device (see Figure 2.1) occupies two tiers; therefore, it cannot be enabled if attached at tier level seven. Only functions can be enabled in tier seven.

2.2.2 USB Hosts

The Universal Serial Bus is host controlled. There can only be one host per bus. The specification in itself, does not support any form of multi-master arrangement. However the On-The-Go specification, which is an addition to USB 2.0, has introduced a Host Negotiation Protocol which allows two devices to negotiate for the role of host. This is aimed at and limited to single point-to-point connections such as a mobile phone and personal organizer and not multiple hub, multiple

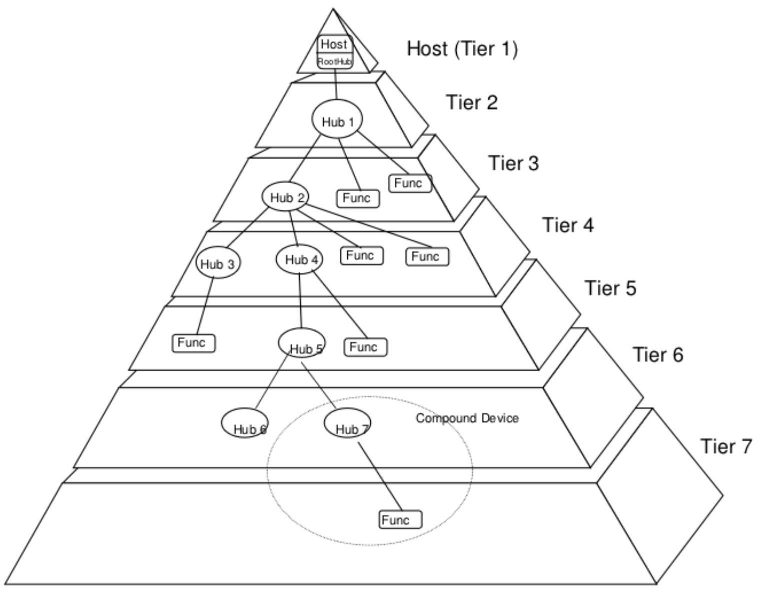


Figure 2.1: Topology of the USB Bus

device desktop configurations. The USB host is responsible for undertaking all transactions and scheduling bandwidth. Data can be sent by various transaction methods using a token-based protocol.

USB Host Specifications

The USB host controllers have their own specifications. With USB 1.1, there were two Host Controller Interface Specifications, *UHCI* (Universal Host Controller Interface) developed by *Intel* which puts more of the burden on software and allowing for cheaper hardware and the *OHCI* (Open Host Controller Interface) developed by *Compaq, Microsoft* and *National Semiconductor* which places more of the burden on hardware and makes for simpler software. With the introduction of USB 2.0 a new Host Controller Interface Specification was needed to describe the register level details specific to USB 2.0. The *EHCI* (Enhanced Host Controller Interface) was born. Significant contributors include *Intel, Compaq, NEC, Lucent* and *Microsoft*.

2.2.3 Dynamic Device Attachment

The USB supports USB devices attaching to and detaching from the USB at any time. Consequently, system software must accommodate dynamic changes in the physical bus topology.

Attachment of USB Devices

All USB devices attach to the USB through ports on specialized USB devices known as hubs (refer to 2.4.1, on page 18). Hubs have status bits that are used to report the attachment or removal of a USB device on one of its ports. The host queries the hub to retrieve these bits. In the case of an attachment, the host enables the port and addresses the USB device through the device's control pipe at the default address (pipes are explained in 2.4.4, on page 21). The host assigns a unique USB address to the device and then determines if the newly attached USB device is a hub or a function. The host establishes its end of the control pipe for the USB device using the assigned USB address and endpoint number zero. If the attached USB device is a hub and USB devices are attached to its ports, then the above procedure is followed for each of the attached USB devices. If the attached USB device is a function, then attachment notifications will be handled by host software that is appropriate for the function.

Removal of USB Devices

When a USB device has been removed from one of a hub's ports, the hub disables the port and provides an indication of device removal to the host. The removal indication is then handled by appropriate USB System Software. If the removed USB device is a hub, the USB System Software must handle the removal of both the hub and of all of the USB devices that were previously attached to the system through the hub.

Bus Enumeration

Bus enumeration is the activity that identifies and assigns unique addresses to devices attached to a bus. Because the USB allows USB devices to attach to or detach from the USB at any time, bus enumeration is an on-going activity for the USB System Software. Additionally, bus enumeration for the USB also includes the detection and processing of removals (More information on the enumeration process can be found in section 2.7.8, on page 32).

2.3 USB Data Flow Modes

The USB supports functional data and control exchange between the USB host and a USB device as a set of either unidirectional or bidirectional pipes (refer to 2.4.4, on page 21). USB data transfers take place between host software and a particular endpoint on a USB device. Such associations between the host software and a USB device endpoint are called pipes. In general, data movement though one pipe is independent from the data flow in any other pipe. A given USB device may have many pipes.

2.3.1 Transfer Modes

The USB architecture comprehends four basic types of data transfers:

Control Transfers: Used to configure a device at attach time and can be used for other device-specific purposes, including control of other pipes on the device.

Bulk Data Transfers: Generated or consumed in relatively large and bursty quantities and have wide dynamic latitude in transmission constraints.

Interrupt Data Transfers: Used for timely but reliable delivery of data, for example, characters or coordinates with human-perceptible echo or feedback response characteristics.

Isochronous Data Transfers Occupy a prenegotiated amount of USB bandwidth with a prenegotiated delivery latency.

A pipe supports only one of the types of transfers described above for any given device configuration.

Control Transfers

Control data is used by the USB System Software to configure devices when they are first attached. Other driver software can choose to use control transfers in implementation-specific ways. Data delivery is lossless.

The USB device framework (refer to Section 2.4) defines standard, device class, or vendor-specific requests that can be used to manipulate a device's state. Descriptors are also defined that can be used to contain different information on the device. Control transfers provide the transport mechanism to access device descriptors and make requests of a device to manipulate its behavior.

Bulk Transfers

Bulk data typically consists of larger amounts of data, such as that used for printers or scanners. Bulk data is sequential. Reliable exchange of data is ensured at the hardware level by using error detection in hardware and invoking a limited number of retries in hardware. Also, the bandwidth taken up by bulk data can vary, depending on other bus activities.

Interrupt Transfers

A limited-latency transfer to or from a device is referred to as interrupt data. Such data may be presented for transfer by a device at any time and is delivered by the USB at a rate no slower than is specified by the device.

Interrupt data typically consists of event notification, characters, or coordinates that are organized as one or more bytes. An example of interrupt data is the coordinates from a pointing device. Although an explicit timing rate is not required, interactive data may have response time bounds that the USB must support.

Isochronous Transfers

Isochronous data is continuous and real-time in creation, delivery, and consumption. Timing-related information is implied by the steady rate at which isochronous data is received and transferred. Isochronous data must be delivered at the rate received to maintain its timing. In addition to delivery rate, isochronous data may also be sensitive to delivery delays. For isochronous pipes, the bandwidth required is typically based upon the sampling characteristics of the associated function. The latency required is related to the buffering available at each endpoint.

A typical example of isochronous data is voice. If the delivery rate of these data streams is not maintained, dropouts in the data stream will occur due to buffer or frame underruns or overruns. Even if data is delivered at the appropriate rate by USB hardware, delivery delays introduced by software may degrade applications requiring real-time turnaround, such as telephony-based audio conferencing.

The timely delivery of isochronous data is ensured at the expense of potential transient losses in the data stream. In other words, any error in electrical transmission is not corrected by hardware mechanisms such as retries. USB isochronous data streams are allocated a dedicated portion of USB bandwidth to ensure that data can be delivered at the desired rate.

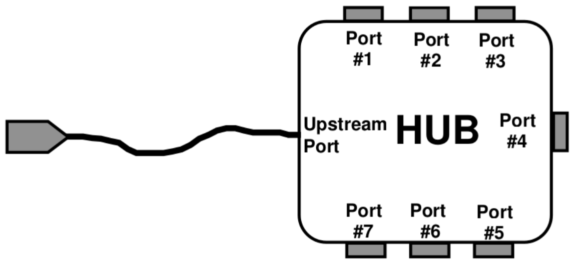


Figure 2.2: A Typical USB Hub

2.4 USB Device Framework

USB devices are divided into device classes such as hub, human interface, printer, imaging, or mass storage device. The hub device class indicates a specially designated USB device that provides additional USB attachment points. USB devices are required to carry information for self-identification and generic configuration. They are also required at all times to display behavior consistent with defined USB device states.

Two major divisions of device classes exist: hubs and functions. Only hubs have the ability to provide additional USB attachment points. Functions provide additional capabilities to the host.

2.4.1 USB Hubs

Hubs are a key element in the plug-and-play architecture of the USB. Figure 2.2 shows a typical hub. Hubs serve to simplify USB connectivity from the user's perspective and provide robustness at relatively low cost and complexity.

Hubs are wiring concentrators and enable the multiple attachment characteristics of the USB. Attachment points are referred to as ports. Each hub converts a single attachment point into multiple attachment points. The architecture supports concatenation of multiple hubs (see also 2.7, on page 28, for details on the root hub).

The upstream port of a hub connects the hub towards the host. Each of the downstream ports of a hub allows connection to another hub or function. Hubs can detect attach and detach at each downstream port and enable the distribution of power to downstream devices. Each downstream port can be individually enabled and attached to either high-, full- or low-speed devices.

Figure 2.3 illustrates how hubs provide connectivity in a typical desktop computer environment.

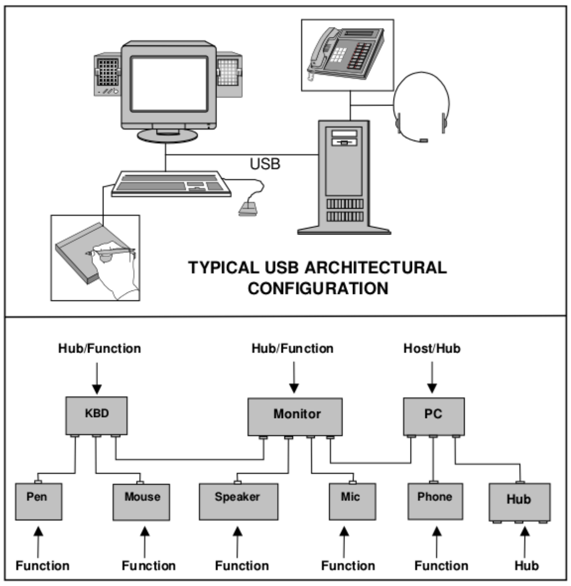


Figure 2.3: Hubs in a Desktop Computer Environment

2.4.2 USB Functions

A function is a USB device that is able to transmit or receive data or control information over the bus. A function is typically implemented as a separate peripheral device with a cable that plugs into a port on a hub. However, a physical package may implement multiple functions and an embedded hub with a single USB cable. This is known as a compound device. A compound device appears to the host as a hub with one or more nonremovable USB devices (see Figure 2.1, on page 14).

Each function contains configuration information that describes its capabilities and resource requirements. Before a function can be used, it must be configured by the host. This configuration includes allocating USB bandwidth and selecting function-specific configuration options. Examples of functions include the following:

- A human interface device such as a mouse, keyboard, tablet, or game controller
- An imaging device such as a scanner, printer, or camera
- A mass storage device such as a CDROM drive, floppy drive, or DVD drive

Most functions have a series of buffers, typically 8 bytes long. Each buffer belongs to an endpoint.

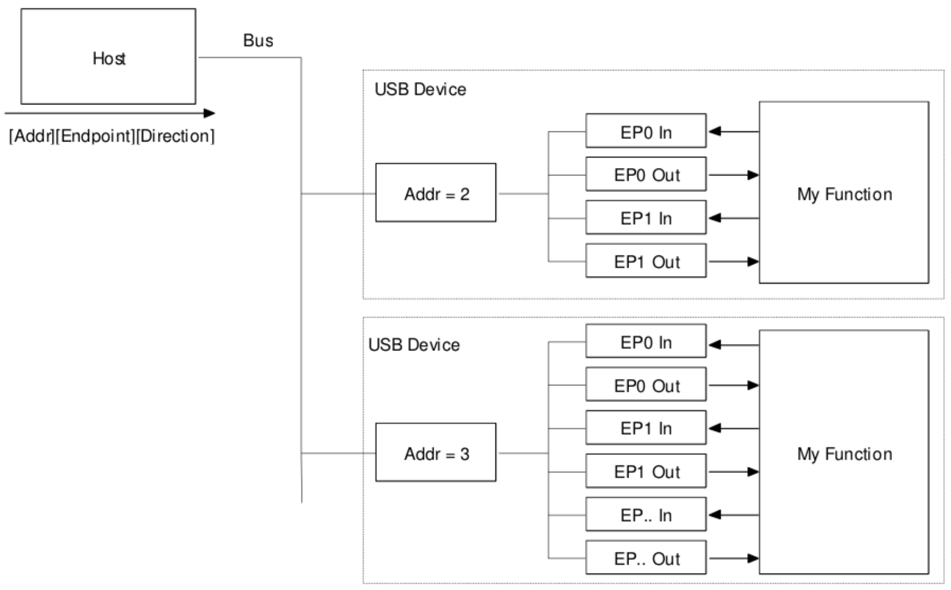


Figure 2.4: Addressing of Device Endpoints

2.4.3 USB Device Endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of a communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints. Each logical device has a unique address assigned by the system at device attachment time. Each endpoint on a device is given at design time a unique device-determined identifier called the endpoint number. Each endpoint has a device-determined direction of data flow. As illustrated in Figure 2.4, the combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced. Each endpoint is a simplex connection that supports data flow in one direction: either input (from device to host) or output (from host to device).

An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software. An endpoint describes itself by:

- Bus access frequency/latency requirement
- Bandwidth requirement
- Endpoint number
- Error handling behavior requirements

- Maximum packet size that the endpoint is capable of sending or receiving

- The transfer type for the endpoint (refer to Section 2.3.1, on page 16 for details)

- The direction in which data is transferred between the endpoint and the host

All devices must support endpoint zero. This is the endpoint which receives all of the devices control and status requests during enumeration and throughout the duration while the device is operational on the bus. Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

2.4.4 USB Pipes

While the device sends and receives data on a series of endpoints, the client software transfers data through pipes. A pipe is a logical connection between the host and endpoint(s). Pipes also have a set of parameters associated with them such as how much bandwidth is allocated to it, what transfer type (Control, Bulk, Iso or Interrupt) it uses, a direction of data flow and maximum packet/buffer sizes. For example the default pipe is a bidirectional pipe made up of endpoint zero in and endpoint zero out with a control transfer type.

USB defines two types of pipes:

Stream Pipes: have no defined USB format. Data flows sequentially and has a predefined direction, either in or out. Stream pipes support bulk, isochronous and interrupt transfer types. Stream pipes can either be controlled by the host or device.

Message Pipes: have a defined USB format. They are host controlled. Data is transferred in the desired direction, dictated by the request. Message pipes allow data to flow in both directions but only support control transfers.

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using control transfers. The USB protocol specifies a standard set of requests that all devices must support. More information on USB standard requests can be found in Appendix A.1, on page 110.

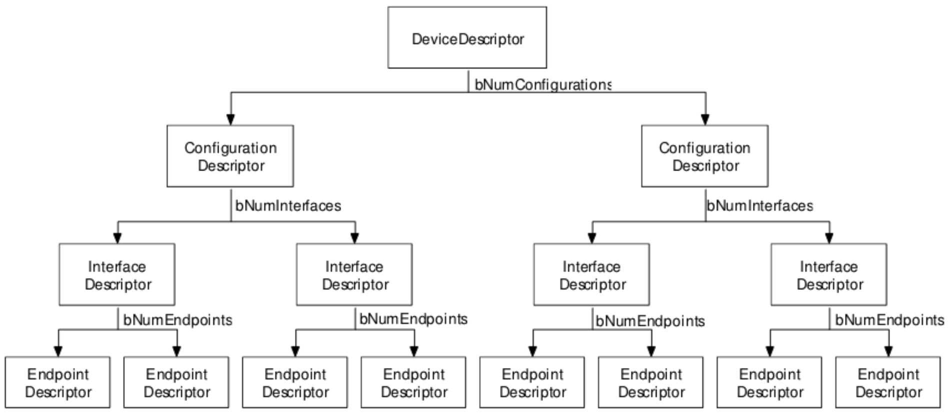


Figure 2.5: Hierarchy of the USB Descriptors

2.5 USB Descriptors

All USB devices have a hierarchy of descriptors which describe to the host information such as what the device is, who makes it, what version of USB it supports, how many ways it can be configured, the number of endpoints and their types etc.

The more common USB descriptors are:

- Device Descriptors
- Configuration Descriptors
- Interface Descriptors
- Endpoint Descriptors
- String Descriptors

The hierarchy of the descriptors is illustrated in Figure 2.5.

USB devices can only have one **device descriptor**. The device descriptor includes information such as what USB revision the device complies to, the Product and Vendor IDs used to load the appropriate drivers and the number of possible configurations the device can have. The number of configurations (*bNumConfigurations* in Figure 2.5) indicates how many configuration descriptors branches are to follow.

The **configuration descriptor** specifies values such as the amount of power this particular configuration uses, if the device is self- or bus-powered and the number of interfaces it has (*bNumInterfaces* in Figure 2.5). When a device is enumerated, the host reads the device descriptors and can make a decision of which configuration to enable. It can only enable one configuration at a time.

The configuration settings are not limited to power differences. Each configuration could be powered in the same way and draw the same current, yet have different interface or endpoint combinations. However it should be noted that changing the configuration requires all activity on each endpoint to stop. While USB offers this flexibility, very few devices have more than 1 configuration.

The **interface descriptor** can be seen as a header or grouping of the endpoints into a functional group performing a single feature of a multi-function device. Unlike the configuration descriptor, there is no limitation as to having only one interface enabled at a time.

Each **endpoint descriptor** is used to specify the type of transfer, direction, polling interval and maximum packet size for each endpoint. Endpoint zero, the default control endpoint is always assumed to be a control endpoint and as such never has a descriptor.

2.5.1 Composition of USB Descriptors

All descriptors are made up of a common format. The first byte specifies the length of the descriptor, while the second byte indicates the descriptor type. If the length of a descriptor is smaller than what the specification defines, then the host shall ignore it. However if the size is greater than expected the host will ignore the extra bytes and start looking for the next descriptor at the end of actual length returned.

For more information on the composition of USB Descriptors, refer to Appendix B, on page 121.

2.6 USB Host: Hardware & Software

The USB interconnect supports data traffic between a host and a USB device. The specifications of the bus, describe the host interfaces necessary to facilitate USB communication between a software client, resident on the host, and a function implemented on a device.

2.6.1 Host Controller Requirements

In all implementations, Host Controllers perform the same basic duties with regard to the USB and its attached devices.The Host Controller has requirements from both the host and the USB. The following is a brief overview of the functionality provided:

State Handling As a component of the host, the Host Controller reports and manages its states.

Serializer/Deserializer For data transmitted from the host, the Host Controller converts protocol and data information from its native format to a bit stream transmitted on the USB. For data being received into the host, the reverse operation is performed.

(micro)Frame Generation The HC produces "Start of Frame" (SOF) tokens at a period of 1 ms when operating with full-speed devices, and at a period of 125 μs when operating with high-speed devices.

Data Processing The Host Controller processes requests for data transmission to and from the host.

Protocol Engine The Host Controller supports the protocol specified by the USB.

Transmission Error Handling All Host Controllers exhibit the same behavior when detecting and reacting to the defined error categories.

Remote Wakeup All Host Controllers must have the ability to place the bus into the Suspended state and to respond to bus wakeup events.

Root Hub The root hub provides standard hub function to link the Host Controller to one or more USB ports.

Host System Interface Provides a high-speed data path between the Host Controller and host system.

2.6.2 Overview of the USB Host

The host and the device are divided into the distinct layers depicted in Figure 2.6. Vertical arrows indicate the actual communication on the host. All communications between the host and device ultimately occur on the physical USB wire. However, there are logical host-device interfaces between each horizontal layer. These communications, between client software resident on the host and the function provided by the device, are typified by a contract based on the needs of the application currently using the device and the capabilities provided by the device. This client-function interaction creates the requirements for all of the underlying layers and their interfaces.

In summary, the host layers provide the following capabilities:

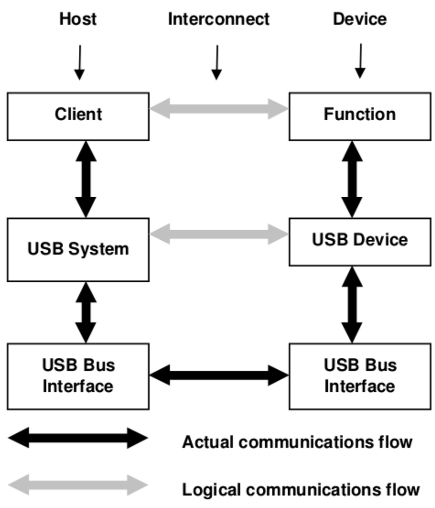


Figure 2.6: USB Interlayer Communication Model

- Detecting the attachment and removal of USB devices
- Managing USB standard control flow between the host and USB devices
- Managing data flow between the host and USB devices
- Collecting status and activity statistics
- Controlling the electrical interface between the Host Controller and USB devices, including the provision of a limited amount of power.

Figure 2.7 illustrates the host's view of its communication with the device. There is only one host for each USB. The major layers of a host consist of the following:

- USB bus interface
- USB System
- Client

2.6.3 USB Bus Interface

The USB bus interface handles interactions for the electrical and protocol layers. From the interconnect point of view, a similar USB bus interface is provided by both the USB device and the host, as exemplified by the Serial Interface Engine (SIE). On the host, however,

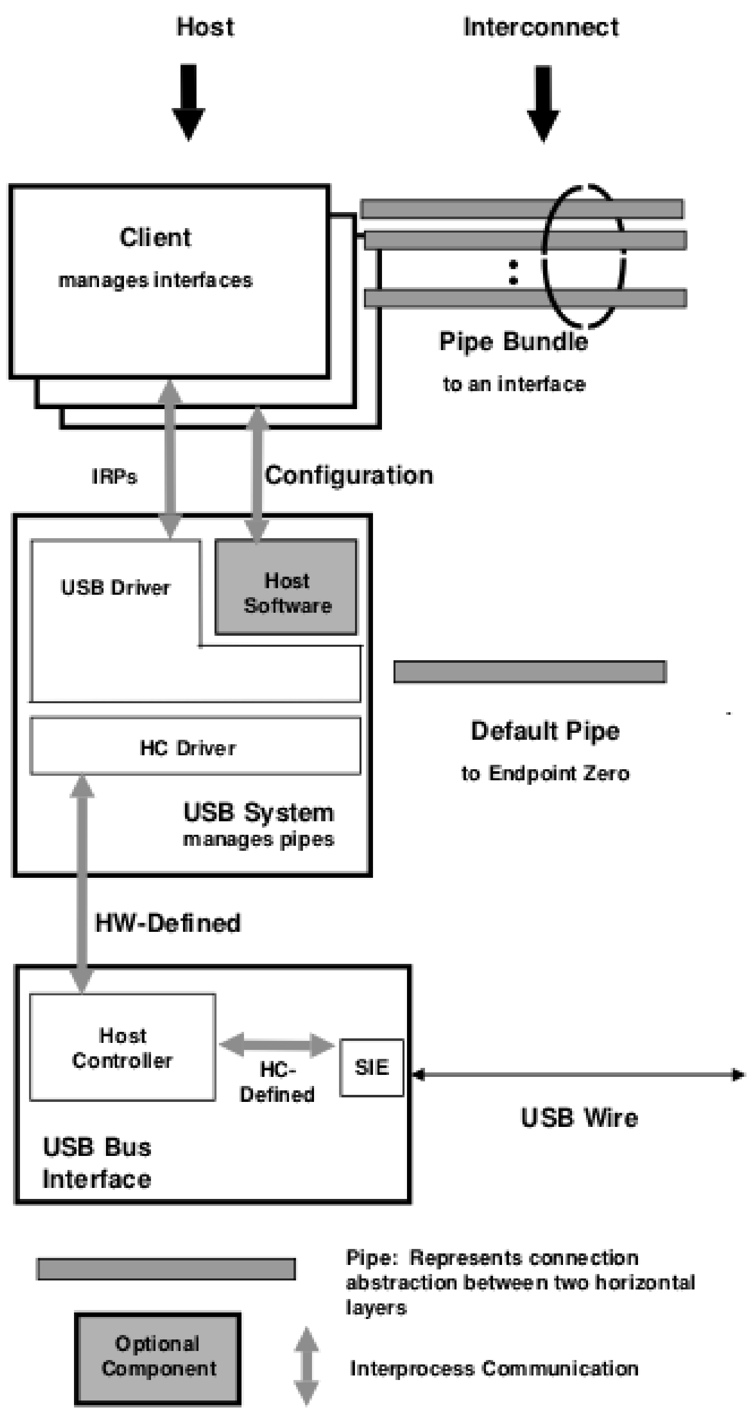


Figure 2.7: USB Host Communications

the USB bus interface has additional responsibilities due to the unique role of the host on the USB and is implemented as the Host Controller. The Host Controller has an integrated root hub providing attachment points to the USB wire.

2.6.4 USB System

The USB System uses the Host Controller to manage data transfers between the host and USB devices. The interface between the USB System and the Host Controller is dependent on the hardware definition of the Host Controller. The USB System, along with the Host Controller, performs the translation between the client's view of data transfers and the USB transactions appearing on the interconnect. This includes the addition of any USB feature support such as protocol wrappers. The USB System is also responsible for managing USB resources, such as bandwidth and bus power, so that client access to the USB is possible.

The USB System has three basic components:

- Host Controller Driver
- USB Driver
- Host Software

2.6.5 Host Controller Driver

The Host Controller Driver (HCD) is an abstraction of Host Controller hardware and the Host Controller's view of data transmission over the USB. The HCD meets the following requirements:

- Provides an abstraction of the Host Controller hardware.
- Provides an abstraction for data transfers by the Host Controller across the USB interconnect.
- Provides an abstraction for the allocation (and de-allocation) of Host Controller resources to support guaranteed service to USB devices.
- Presents the root hub and its behavior according to the hub class definition. This includes supporting the root hub such that the hub driver interacts with the root hub exactly as it would for any hub.

The HCD provides a software interface (HCDI) that implements the required abstractions. The function of the HCD is to provide an abstraction, which hides the details of the Host Controller hardware. Below the Host Controller hardware is the physical USB and all the attached USB devices.

The HCD is the lowest tier in the USB software stack. The HCD has only one client: the Universal Serial Bus Driver (USBD). The USBD maps requests from many clients to the appropriate HCD. A given HCD may manage many Host Controllers. The HCDI is not directly accessible from a client.

2.6.6 USB Driver

The USBD provides a collection of mechanisms that operating system components, typically device drivers, use to access USB devices. The only access to a USB device is that provided by the USBD. The USBD implementations are operating system-specific. The mechanisms provided by the USBD are implemented, using as appropriate and augmenting as necessary, the mechanisms provided by the operating system. For specifics of the USBD operation in the Linux operating system, see Chapter 3.

USBD directs accesses to one or more HCDs that in turn connect to one or more Host Controllers. If allowed, how USBD instancing is managed is dependent upon the operating system environment. However, from the client's point of view, the USBD with which the client communicates manages all of the attached USB devices. Figure 2.8 presents an overview of the USBD structure.

Clients of USBD direct commands to devices or move streams of data to or from pipes. The USBD presents two groups of software mechanisms to clients:

Command mechanisms allow clients to configure and control USBD operation as well as to configure and generically control a USB device. In particular, command mechanisms provide all access to the device's default pipe.

Pipe mechanisms allow a USBD client to manage device specific data and control transfers. Pipe mechanisms do not allow a client to directly address the device's default pipe.

2.7 The Root Hub

Hubs provide the electrical interface between USB devices and the host. Hubs are directly responsible for supporting many of the at-

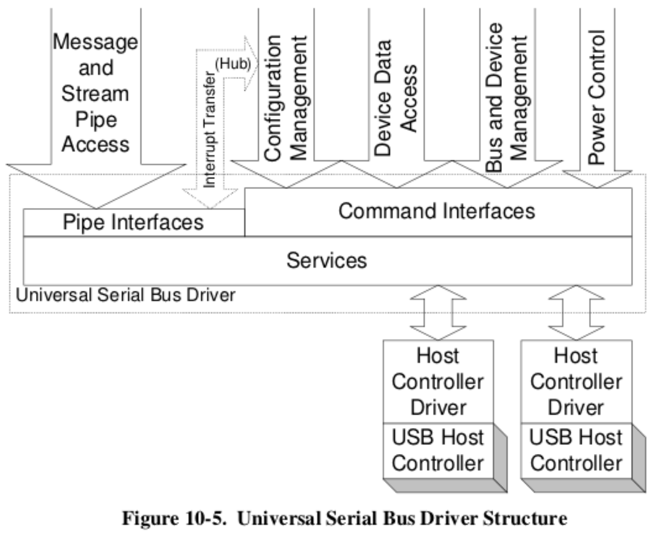


Figure 2.8: USB Driver Structure

tributes that make USB user friendly and hide its complexity from the user. Listed below are the major aspects of USB functionality that hubs must support:

- Connectivity behavior
- Power management
- Device connect/disconnect detection
- Bus fault detection and recovery
- High, full, and lowspeed device support

2.7.1 Hub Requests & Descriptors

Hubs respond to standard device commands as defined in A.3.1, on page 118. In addition, the hub class defines a set of hub-specific requests, listed in Section A.3.2, on page 118.

Likewise, hub descriptors are derived from the general USB framework (refer to Appendix B, on page 121). Hub descriptors define a hub device and the ports on that hub. The hub class defines a hub-specific descriptor, listed in Section B.6, on page 128.

2.7.2 Hub Architecture & Supported Speeds

A hub consists of a Hub Repeater section, a Hub Controller section, and a Transaction Translator section. The hub must operate at high-speed when its upstream facing port is connected at high-speed. The hub must operate at full-speed when its upstream facing port is connected at full-speed. The Hub Repeater is responsible for managing connectivity between upstream and downstream facing ports which are operating at the same speed. The Hub Repeater supports full-/low-speed connectivity and high-speed connectivity. The Hub Controller provides status and control and permits host access to the hub. The Transaction Translator takes high-speed split transactions and translates them to full-/low-speed transactions when the hub is operating at high-speed and has full-/low-speed devices attached. The operating speed of a device attached on a downstream facing port determines whether the Routing Logic connects a port to the Transaction Translator or hub repeater sections (see also Table 2.1, on page 10).

2.7.3 Hub Connectivity

Hubs exhibit different connectivity behavior depending on whether they are propagating packet traffic, or resume signaling, or are in the Idle state.

The Hub Repeater contains one port that must always connect in the upstream direction (referred to as the upstream facing port) and one or more downstream facing ports. Upstream connectivity is defined as being towards the host, and downstream connectivity is defined as being towards a device (see also Figure 2.2, on page 18).

If a downstream facing port is enabled (i.e., in a state where it can propagate signaling through the hub), and the hub detects the start of a packet on that port, connectivity is established in an upstream direction to the upstream facing port of that hub, but not to any other downstream facing ports. This means that when a device or a hub transmits a packet upstream, only those hubs in line between the transmitting device and the host will see the packet.

In the downstream direction, hubs operate in a broadcast mode. When a hub detects the start of a packet on its upstream facing port, it establishes connectivity to all enabled downstream facing ports. If a port is not enabled, it does not propagate packet signaling downstream.

2.7.4 Endpoint Organization

The Hub Class defines one additional endpoint beyond Default Control Pipe, which is required for all hubs: the *Status Change endpoint*. The host system receives port and hub status change notifications through the Status Change endpoint. The Status Change endpoint is an interrupt endpoint. If no hub or port status change bits are set, then the hub returns an NAK when the Status Change endpoint is polled. When a status change bit is set, the hub responds with data, as shown in 2.7.6, indicating the entity (hub or port) with a change bit set. The USB System Software can use this data to determine which status registers to access in order to determine the exact cause of the status change interrupt.

2.7.5 Port Change Information Processing

Hubs report a port's status through port commands on a per-port basis. The USB System Software acknowledges a port change by clearing the change state corresponding to the status change reported by the hub. The acknowledgment clears the change state for that port so future data transfers to the Status Change endpoint do not report the previous event. This allows the process to repeat for further changes.

2.7.6 Hub and Port Status Change Bitmap

The Hub and Port Status Change Bitmap indicates whether the hub or a port has experienced a status change. This bitmap also indicates which port(s) has had a change in status. The hub returns this value on the Status Change endpoint. Hubs report this value in byte-increments. That is, if a hub has six ports, it returns a byte quantity, and reports a zero in the invalid port number field locations.

The USB System Software is aware of the number of ports on a hub (this is reported in the hub descriptor) and decodes the Hub and Port Status Change Bitmap accordingly. The hub reports any changes in hub status in bit zero of the Hub and Port Status Change Bitmap.

The Hub and Port Status Change Bitmap size varies from a minimum size of one byte. Hubs report only as many bits as there are ports on the hub, subject to the byte-granularity requirement (i.e., round up to the nearest byte).

Any time the Status Change endpoint is polled by the host controller and any of the Status Changed bits are non-zero, the Hub and Port Status Change Bitmap is returned.

2.7.7 Over-current Reporting and Recovery

USB devices must be designed to meet applicable safety standards. Usually, this will mean that a self-powered hub implements current limiting on its downstream facing ports. If an over-current condition occurs, it causes a status and state change in one or more ports. This change is reported to the USB System Software so that it can take corrective action.

A hub may be designed to report over-current as either a port or a hub event. The hub descriptor field wHubCharacteristics (refer to B.6.1, on page 129) is used to indicate the reporting capabilities of a particular hub. The over-current status bit in the hub or port status field indicates the state of the over-current detection when the status is returned. The over-current status change bit in the Hub or Port Change field indicates if the overcurrent status has changed.

When a hub experiences an over-current condition, it must place all affected ports in the Powered-off state. If a hub has per-port power switching and per-port current limiting, an over-current on one port may still cause the power on another port to fall below specified minimums. In this case, the affected port is placed in the Powered-off state. If the hub has over-current detection on a hub basis, then an over-current condition on the hub will cause all ports to enter the Powered-off state.

2.7.8 Enumeration Handling

The hub device class commands (refer to Appedix A.3.2, on page 118) are used to manipulate its downstream facing port state. When a device is attached, the device attach event is detected by the hub and reported on the status change interrupt. The host will accept the status change report and request a SetPortFeature(PORT_RESET) on the port. As part of the bus reset sequence, a speed detect is performed by the hub's port hardware.

When the device is detached from the port, the port reports the status change through the status change endpoint and the port will be reconnected to the high-speed repeater. Then the process is ready to be repeated on the next device attach detect.

Chapter 3

The Linux USB Subsystem

The development of the Linux USB subsystem started in 1997 and in the meantime it was redesigned many times. This implied various changes of its internal structure and its API too. So it is hard for device driver developers to keep up to date with all ongoing discussions and current changes.

Inside the Linux kernel, there exists a subsystem called "The USB Core" with a specific API to support USB devices and host controllers. Its purpose is to abstract all hardware or device dependent parts by defining a set of data structures, macros and functions. In fact, the USB core is the implementation of the USBD component of a USB system, as described in the USB specifications (refer to 2.6.6, on page 28).

The USB core contains routines common to all USB device drivers and host controller drivers. These functions can be grouped into an upper (Host-Side) and a lower (Host Controller) API layer. As shown in Figure 3.1 there exists an API for USB device drivers and another one for host controllers.

This Chapter begins by introducing some of the standard data

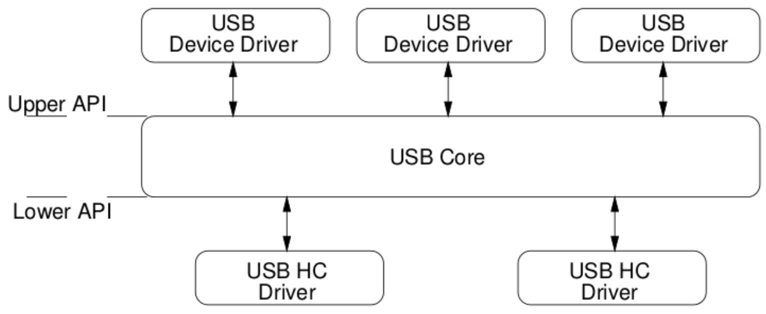


Figure 3.1: Linux USB Core API Layers

types provided by the USB subsystem that conform to the USB specifications. Then the API model of the upper layer is examined, in terms of data structures and functions. The description of the lower layer forms the last part of this chapter. The lower layer is the most important aspect of the USB subsystem in the scope of this work.

3.1 USB-Standard Types

This section describes USB structures that are needed for USB device APIs. These are used by the USB device model, which is defined in chapter 9 of the USB 2.0 specification. Linux has several APIs in C that need these.

Note that the naming scheme followed inside these data structures does not conform to the standard Linux kernel scheme, but matches the one found in the USB specifications.

3.1.1 Control Request Support

struct usb_ctrlrequest

This structure is used to send control requests on a USB device. It matches the different fields of the USB 2.0 Spec. Refer to Table A.1 for a description of the different fields, and what they are used for.

```
struct usb_ctrlrequest {
        __u8 bRequestType;
        __u8 bRequest;
        __le16 wValue;
        __le16 wIndex;
        __le16 wLength;
} __attribute__ ((packed));
```

3.1.2 Standard Descriptors

These are the standard USB Descriptors, defined in 2.5, as returned by the "GET_DESCRIPTOR" command.

Two fields inside these data structures, "bLength" and "bDescriptorType", are common to all descriptors. The other fields are specific to each descriptor.

All multi-byte values here are encoded in little endian byte order on the physical bus. But when exposed through Linux-USB APIs, they've been converted to cpu byte order.

struct usb_device_descriptor

This structure is used to hold a USB device descriptor.

```
struct usb_device_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __u16 bcdUSB;
        __u8  bDeviceClass;
        __u8  bDeviceSubClass;
        __u8  bDeviceProtocol;
        __u8  bMaxPacketSize0;
        __u16 idVendor;
        __u16 idProduct;
        __u16 bcdDevice;
        __u8  iManufacturer;
        __u8  iProduct;
        __u8  iSerialNumber;
        __u8  bNumConfigurations;
} __attribute__ ((packed));
```

For more information on the fields inside this structure, see Appendix B.1, on page 121, particularly Table B.1.

struct usb_config_descriptor

This structure is used to hold a USB configuration descriptor.

```
struct usb_config_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __u16 wTotalLength;
        __u8  bNumInterfaces;
        __u8  bConfigurationValue;
        __u8  iConfiguration;
        __u8  bmAttributes;
        __u8  bMaxPower;
} __attribute__ ((packed));
```

For more information on the fields inside this structure, see Appendix B.2, on page 122, particularly Table B.2.

struct usb_string_descriptor

This structure is used to hold a USB string descriptor.

```
struct usb_string_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __le16 wData[1];
} __attribute__ ((packed));
```

Field *wData(1)* of the data structure, is encoded in Unicode format (UTF-16, Little Endian). For more information on the fields of this structure, see Appendix B.5, on page 126, particularly Table B.5.

struct usb_interface_descriptor

This structure is used to hold a USB interface descriptor.

```
struct usb_interface_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;

        __u8  bInterfaceNumber;
        __u8  bAlternateSetting;
        __u8  bNumEndpoints;
        __u8  bInterfaceClass;
        __u8  bInterfaceSubClass;
        __u8  bInterfaceProtocol;
        __u8  iInterface;
} __attribute__ ((packed));
```

For more information on the fields of this structure, see Appendix B.3, on page 125, particularly Table B.3.

struct usb_endpoint_descriptor

This structure is used to hold a USB endpoint descriptor.

```
struct usb_endpoint_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;
        __u8  bEndpointAddress;
        __u8  bmAttributes;
        __u16 wMaxPacketSize;
        __u8  bInterval;
} __attribute__ ((packed));
```

For more information on the fields of this structure, see Appendix B.4, on page 126, particularly Table B.4.

3.2 Upper (Host-Side) API Model

Within the kernel, host-side drivers for USB devices talk to the USB core APIs. There are two types of public USB core APIs, targetted at two different layers of USB driver. Those are general purpose drivers, exposed through driver frameworks such as block, character, or network devices; and drivers that are part of the core, which are involved in managing a USB bus. Such core drivers include the hub driver, which manages trees of USB devices, and several different kinds of host controller driver (HCD), which control individual busses.

The device model seen by USB drivers is described in detail in Chapter 2, on page 10 (particularly Sections 2.3–2.5). What follows, is a quick description of this device model:

- USB supports four kinds of data transfer (control, bulk, interrupt, and isochronous). Two transfer types use bandwidth as it's available (control and bulk), while the other two types of transfer (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.

- The device description model includes one or more *configurations* per device, only one of which is active at a time. Devices that are capable of high-speed operation must also support full-speed configurations, along with a way to ask about the "other speed" configurations that might be used.

- Configurations have one or more *interface*, each of which may have *alternate settings*. Interfaces may be standardized by USB "Class" specifications, or may be specific to a vendor or device.

- USB device drivers actually bind to interfaces, not devices (see Figure 3.2). Most USB devices are simple, with only one configuration, one interface, and one alternate setting.

- Interfaces have one or more *endpoints*, each of which supports one type and direction of data transfer such as "bulk out" or

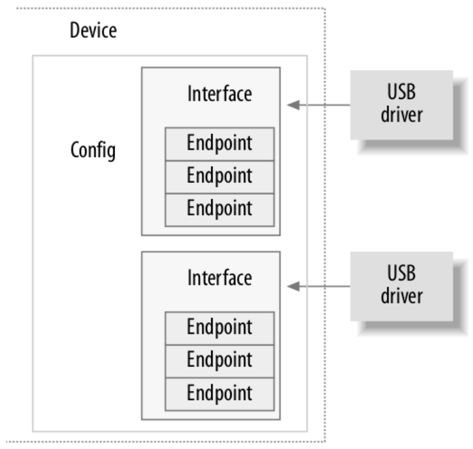


Figure 3.2: Linux USB Driver Attachment to a USB Device

"interrupt in". The entire configuration may have up to sixteen endpoints in each direction, allocated as needed among all the interfaces.

- Data transfer on USB is packetized; each endpoint has a maximum packet size. Drivers must often be aware of conventions such as flagging the end of bulk transfers using "short" (including zero length) packets.

The Linux USB API supports synchronous calls for control and bulk messaging. It also supports asynchnous calls for all kinds of data transfer, using request structures called *USB Request Blocks* (URBs). URBs are covered in Section 3.3.1.

The only drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs. In theory, all HCDs provide the same functionality through the same API. In practice, that's becoming more true with 2.6 Linux kernels, but there are still differences that crop up, especially with fault handling. Different controllers don't necessarily report the same aspects of failures, and recovery from faults (including software-induced ones like unlinking an URB) isn't yet fully consistent.

3.2.1 Host-Side Data Types

The host-side API exposes several layers to drivers, some of which are more necessary than others. These support lifecycle models for host-side drivers and devices, and support passing buffers through USB core to some HCD that performs the I/O for the device driver.

These data types will not be covered in this document, since our focus is on the Host Controller API, needed for Host Controller Drivers. Further information can be found in the documentation of the kernel sources.

3.3 USB Core APIs

There are two basic I/O models in the USB API. The most elemental one is asynchronous: drivers submit requests in the form of an URB, and the URB's completion callback handle the next step. All USB transfer types support that model, although there are special cases for control URBs (which always have setup and status stages, but may not have a data stage) and isochronous URBs (which allow large packets and include per-packet fault reports). Built on top of that is synchronous API support, where a driver calls a routine that allocates one or more URBs, submits them, and waits until they complete. There are synchronous wrappers for single-buffer control and bulk transfers.

USB drivers need to provide buffers that can be used for DMA, although they don't necessarily need to provide the DMA mapping themselves. There are APIs to use used when allocating DMA buffers, which can prevent use of bounce buffers on some systems.

3.3.1 The Structure of USB Request Blocks (URBs)

This structure identifies USB transfer requests.

Data Transfer Buffers

Normally drivers provide I/O buffers allocated with kmalloc() or otherwise taken from the general page pool. That is provided by transfer_buffer (control requests also use setup_packet), and host controller drivers perform a dma mapping (and unmapping) for each buffer transferred. Those mapping operations can be expensive on some platforms, although they're cheap on commodity x86 and ppc hardware.

Alternatively, drivers may pass the URB_NO_xxx_DMA_MAP transfer flags, which tell the host controller driver that no such mapping is needed since the device driver is DMA-aware. For example, a device driver might allocate a DMA buffer with usb_buffer_alloc() or call usb_buffer_map(). When these transfer flags are provided, host controller drivers will attempt to use the dma addresses found in the transfer_dma and/or setup_dma fields rather than determining a dma address themselves.

Initialization

All URBs submitted must initialize the dev, pipe, transfer_flags (may be zero), and complete fields. The URB_ASYNC_UNLINK transfer flag affects later invocations of the usb_unlink_urb() routine. All URBs must also initialize transfer_buffer and transfer_buffer_length. They may provide the URB_SHORT_NOT_OK transfer flag, indicating that short reads are to be treated as errors; that flag is invalid for write requests.

Control URBs must provide a setup_packet. The setup_packet and transfer_buffer may each be mapped for DMA or not, independently of the other. The transfer_flags URB_NO_TRANSFER_DMA_MAP and URB_-NO_SETUP_DMA_MAP indicate which buffers have already been mapped. URB_NO_SETUP_DMA_MAP is ignored for non-control URBs.

Bulk URBs may use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers should always terminate with a short packet, even if it means adding an extra zero length packet.

Interrupt URBs must provide an interval, saying how often (in milliseconds or, for high-speed devices, 125 microsecond units) to poll for transfers. After the URB has been submitted, the interval field reflects how the transfer was actually scheduled. The polling interval may be more frequent than requested.

Isochronous URBs normally use the URB_ISO_ASAP transfer flag, telling the host controller to schedule the transfer as soon as bandwidth utilization allows, and then set start_frame to reflect the actual frame selected during submission. Otherwise drivers must specify the start_-frame and handle the case where the transfer can't begin then. Isochronous URBs have a different data transfer model, in part because the quality of service is only "best effort". Callers provide specially allocated URBs, with number_of_packets worth of iso_frame_desc structures at the end. Each such packet is an individual ISO transfer. Isochronous URBs are normally queued, submitted by drivers to arrange that transfers are at least double buffered, and then explicitly resubmitted in completion handlers, so that data (such as audio or video) streams are as constant a rate as the host controller scheduler can support.

Completion Callbacks

The completion callback is made in_interrupt(), and one of the first things that a completion handler should do is check the status field.

The status field is provided for all URBs. It is used to report unlinked URBs, and status for all non-ISO transfers. It should not be examined before the URB is returned to the completion handler. The context field is normally used to link URBs back to the relevant driver or request state.

When the completion callback is invoked for non-isochronous URBs, the actual_length field tells how many bytes were transferred. This field is updated even when the URB terminated with an error or was unlinked.

ISO transfer status is reported in the status and actual_length fields of the iso_frame_desc array, and the number of errors is reported in error_count. Completion callbacks for ISO transfers will normally (re)submit URBs to ensure a constant transfer rate.

```
struct urb
{
  /* private, usb core and HC only fields in the urb */
  struct kref kref;
  spinlock_t lock;
  void *hcpriv;
  struct list_head urb_list;
  int bandwidth;
  atomic_t use_count;
  u8 reject;

  /* public, fields that can be used by drivers */
  struct usb_device *dev;
  unsigned int pipe;
  int status;
  unsigned int transfer_flags;
  void *transfer_buffer;
  dma_addr_t transfer_dma;
  int transfer_buffer_length;
  int actual_length;
  unsigned char *setup_packet;
  dma_addr_t setup_dma;
  int start_frame;
  int number_of_packets;
  int interval;
  int error_count;
  void *context;
  usb_complete_t complete;
  struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

URB Data Fields Description

urb_list: For use by current owner of the URB.

dev: Identifies the USB device to perform the request.

pipe: Holds endpoint number, direction, type, and more. These values are created with the eight macros available: usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl" (control), "bulk", "int" (interrupt), or "iso" (isochronous). For example usb_sndbulkpipe() or usb_rcvintpipe(). Endpoint numbers range from zero to fifteen. Note that "in" endpoint two is a different endpoint (and pipe) from "out" endpoint two. The current configuration controls the existence, type, and maximum packet size of any given endpoint.

status: This is read in non-iso completion functions to get the status of the particular request. ISO requests only use it to tell whether the URB was unlinked; detailed status for each frame is in the fields of the iso_frame_desc.

transfer_flags: A variety of flags may be used to affect how URB submission, unlinking, or operation are handled. Different kinds of URB can use different flags.

transfer_buffer: This identifies the buffer to (or from) which the I/O request will be performed (unless URB_NO_TRANSFER_DMA_MAP is set). This buffer must be suitable for DMA; it must be allocated with kmalloc() or equivalent. For transfers to "in" endpoints, contents of this buffer will be modified. This buffer is used for the data stage of control transfers.

transfer_dma: If transfer_flags include URB_NO_TRANSFER_DMA_MAP, the device driver is saying that it provided this DMA address, which the host controller driver should use in preference to the transfer_buffer.

transfer_buffer_length: How big is transfer_buffer. The transfer may be broken up into chunks according to the current maximum packet size for the endpoint, which is a function of the configuration and is encoded in the pipe. When the length is zero, neither transfer_buffer nor transfer_dma is used.

actual_length: This is read in non-iso completion functions, and it tells how many bytes (out of transfer_buffer_length) were transferred. It will normally be the same as requested, unless either an error was reported or a short read was performed. The URB_SHORT_NOT_OK transfer flag may be used to make such short reads be reported as errors.

setup_packet: Only used for control transfers, this points to eight bytes of setup data. Control transfers always start by sending this data to the dvice. Then transfer_buffer is read or written, if needed.

setup_dma: For control transfers with URB_NO_SETUP_DMA_MAP set, the device driver has provided this DMA address for the setup packet. The host controller driver should use this in preference to setup_packet.

start_frame: Returns the initial frame for isochronous transfers.

number_of_packets: Lists the number of ISO transfer buffers.

interval: Specifies the polling interval for interrupt or isochronous transfers. The units are frames (milliseconds) for full- and low-speed devices, and microframes (1/8 millisecond) for high-speed ones.

error_count: Returns the number of ISO transfers that reported errors.

context: For use in completion functions. This normally points to request-specific driver context.

complete: Completion handler. This URB is passed as the parameter to the completion function. The completion function may then do what it likes with the URB, including resubmitting or freeing it.

iso_frame_desc: Used to provide arrays of ISO transfer buffers and to collect the transfer status for each buffer.

3.3.2 URB Manipulation Functions

Here's a short overview of the functions provided by the USB core API to manipulate URBs:

usb_alloc_urb(): Creates a URB for the USB driver to use, initializes a few internal structures, incrementes the usage counter, and returns a pointer to URB.

usb_free_urb(): Must be called when a user of a URB is finished with it. When the last user of the URB calls this function, the memory of the URB is freed.

usb_submit_urb(): This submits a transfer request, and transfers control of the URB describing that request to the USB subsystem. Request completion will be indicated later, asynchronously, by calling the completion handler. The three types of completion are success, error, and unlink (a software-induced fault, also called "request cancelation"). The caller must have correctly initialized the URB before submitting it. Functions such as usb_fill_bulk_urb() and usb_fill_control_urb() are available to ensure that most fields are correctly initialized, for the particular kind of transfer, although they will not initialize any transfer flags.

usb_unlink_urb(): This routine cancels an in-progress request. URBs complete only once per submission, and may be canceled only once per submission. Successful cancelation means the requests's completion handler will be called with a status code indicating that the request has been canceled (rather than any other code) and will quickly be removed from host controller data structures.

usb_kill_urb(): This routine cancels an in-progress request. It is guaranteed that upon return all completion handlers will have finished and the URB will be totally idle and available for reuse. These features make this an ideal way to stop I/O in a disconnect callback or close function. If the request has not already finished or been unlinked the completion handler will see: "urb→status == -ENOENT".

usb_control_msg(): This function builds a control urb, sends it off and waits for completion or timeout.

usb_bulk_msg(): This function builds a bulk urb, sends it off and waits for completion or timeout.

3.4 Host Controller APIs

These APIs are only for use by host controller drivers, most of which implement standard register interfaces such as EHCI, OHCI, or UHCI (refer to Section 2.2.2, on page 13). There exist other host controllers. Not all host controllers use DMA; some use PIO. The same basic APIs are available to drivers for all those controllers.

3.4.1 Host Controller Data Types

For historical reasons, the data types available to HCDs are splitted in two layers: *struct usb_bus* is a rather thin layer that became available in the 2.2 kernels, while *struct usb_hcd* is a more featureful layer (available in later 2.4 kernels and in 2.6) that lets HCDs share common code, to shrink driver size and significantly reduce HCD-specific behaviors.

struct hc_driver

This structure holds the hardware-specific hooks for the host controller. While every host controller must ensure compatibility with the USB specifications, the actual implementation varies among the manufacturers. Thus, every host controller requires different handling in terms of initialization, memory allocation, connection of the root hub etc. All these functions are implemented by the HCD and then hooked within this structure. For example, if the kernel needs to reset the host controller, this will be done by a call to hc_driver→reset, without any knowledge of the actual steps required to reset the host controller. This reduces the need for HCD-specific code inside the kernel.

```
struct hc_driver {
  const char    *description;

  irqreturn_t   (*irq) (struct usb_hcd *hcd,
                        struct pt_regs *regs);

  int           flags;
  #define       HCD_MEMORY      0x0001
  #define       HCD_USB11       0x0010
  #define       HCD_USB2        0x0020

  int   (*reset) (struct usb_hcd *hcd);
  int   (*start) (struct usb_hcd *hcd);
  int   (*suspend) (struct usb_hcd *hcd, u32 state);
  int   (*resume) (struct usb_hcd *hcd);
  void  (*stop) (struct usb_hcd *hcd);
  int   (*get_frame_number) (struct usb_hcd *hcd);
  int   (*urb_enqueue) (struct usb_hcd *hcd, struct urb *urb,
                        int mem_flags);
  int   (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
  void  (*endpoint_disable)(struct usb_hcd *hcd,
                            struct hcd_dev *dev,
                            int bEndpointAddress);
  int   (*hub_status_data) (struct usb_hcd *hcd, char *buf);
  int   (*hub_control) (struct usb_hcd *hcd,
                        u16 typeReq, u16 wValue,
                        u16 wIndex, char *buf, u16 wLength);
  int   (*hub_suspend)(struct usb_hcd *);
  int   (*hub_resume)(struct usb_hcd *);
};
```

description: Name of the hcd to use across the operating system.

flags: Flags that describe various aspects of the hardware:

HCD_MEMORY: HC regs use memory (else I/O)

HCD_USB11: HC is compliant with USB 1.1

HCD_USB2: HC is compliant with USB 2.0

**irq*: Hook to the Interrupt Service Routine (see 5.3.1, on page 87).

**reset*: Called to reset the HCD.

**start*: Called to initialize the HCD and the Root Hub.

**suspend*: Called after all devices were suspended.

**resume*: Called before any devices get resumed.

**stop*: Cleanly make HCD stop writing memory and doing I/O.

**get_frame_number*: Returns the current frame number.

**hcd_alloc*: Allocates all memory data structures required by the HCD.

**urb_enqueue*: Enqueues a URB to the list of pending I/O requests.

**urb_dequeue*: Dequeues a URB from the list of pending I/O requests.

**endpoint_disable*: Disables an endpoint.

**hub_status_data*: Reports changes in the root hub (refer to 2.7.5, on page 31. See also 5.3.2, on page 88).

**hub_control*: Performs standard device requests on the root hub.

**hub_suspend*: Suspends the root hub.

**hub_resume*: Resumes the root hub.

struct usb_bus

This structure holds information about the USB bus that is managed by the host controller. In 2.6 kernels, few functions of the HCD need to access directly this structure, as "struct usb_hcd" has largely substitued this one. The most common use for this structure is to keep the link to the "struct device" (every device registered in the Linux kernel has one), in the "controller" field.

```
struct usb_bus {
  struct device                  *controller;
  int                            busnum;
  char                           *bus_name;
  int                            devnum_next;
  struct usb_devmap             devmap;
  struct usb_operations         *op;
  struct usb_device             *root_hub;
  struct list_head              bus_list;
  void                          *hcpriv;
  int                           bandwidth_allocated;
  int                           bandwidth_int_reqs;
  int                           bandwidth_isoc_reqs;
  struct class_device           class_dev;
  void (*release)(struct usb_bus *bus);
};
```

controller: Host/master side hardware.

busnum: Bus number (assigned in order of registration).

bus_name: Stable ID (PCI slot_name etc).

devnum_next: Next open device number in "Round-Robin" allocation.

devmap: Device address allocation map.

op: File Operations (specific to the HC).

root_hub: The Root hub is a USB device (refer to 2.4, on page 18).

bus_list: List of registered busses in the kernel.

hcpriv: Host Controller private data.

bandwidth_allocated: How much of the time reserved for periodic (intr/iso) requests is used, on average.

bandwidth_int_reqs: Number of Interrupt requests.

bandwidth_isoc_reqs: Number of Isochronous requests.

class_dev: The USB class device for this bus.

**release*: Function hook to destroy this bus's memory.

struct usb_hcd

This structure is the "USB Host Controller Driver framework". It's purpose is to hold all the information specific to the host controller, such as information about the hardware and it's current state, locations of allocated memory and devices attached to the host.

```
struct usb_hcd {
  struct usb_bus         self;
  const char             *product_desc;
  const char             *description;
  struct timer_list      rh_timer;
  struct list_head       dev_list;
  struct hc_driver       *driver;
  unsigned               saw_irq : 1;
  unsigned               can_wakeup:1;
  unsigned               remote_wakeup:1;
  int                    irq;
  void __iomem           *regs;

  #ifdef CONFIG_PCI
  int                    region;
  #endif

  #define HCD_BUFFER_POOLS 4
  struct dma_pool        *pool [HCD_BUFFER_POOLS];
  int                    state;
};
```

- Housekeeping

  self: usb_hcd is a wrapper around usb_bus.

  product_desc:Product/Vendor string.

  description: Name of the hcd to use across the operating system.

  rh_timer: Kernel timer (9, Chapter 6, page 200) that polls the root hub for changes.

  dev_list: List of devices attached on this bus.

- Hardware Info/State

  driver: Hardware-specific hooks.

  irq: Number of reserved IRQ channel.

  regs: Device's base memory/IO region address.

  region: If the device is located on the PCI bus, this is the address of the region for access to the internal device registers.

  HCD_BUFFER_POOLS: Number of DMA pools.

pool (HCD_BUFFER_POOLS): Array of memory-mapped locations of the DMA pools.

state: The functional state of the HCD (USBOperational, USBSuspend, USBResume and USBReset).

3.4.2 Host Controller Functions

In order to facilitate and ensure correct behaviour of HCDs, a number of functions have been implemented inside the kernel that help in the creation and maintainance of the data structures defined in Section 3.4.1. This is a short (and by no means complete) overview of these functions:

usb_bus_init(): Used to initialize a usb_bus structure, memory for which is separately managed.

usb_register_bus(): Assigns a bus number, and links the controller into the "USB core" data structures so that it can be seen by scanning the bus list.

usb_deregister_bus(): Recycles the bus number, and unlinks the controller from the "USB core" data structures so that it won't be seen by scanning the bus list.

usb_register_root_hub(): The USB host controller calls this function to register the root hub with the USB subsystem. It sets up the device properly in the device tree and stores the root_hub pointer in the bus structure, then calls usb_new_device() to register the usb device. It also assigns the root hub's USB address (always 1).

usb_calc_bus_time(): Returns approximate bus time in nanoseconds for a periodic transaction. Only periodic transfers need to be scheduled in software, and this function is only used for such scheduling.

usb_hcd_giveback_urb(): This hands the URB from HCD to its USB device driver, using its completion function. The HCD has freed all per-urb resources (and is done using urb→hcpriv). It has also released all HCD locks; the device driver won't cause problems if it frees, modifies, or resubmits this URB.

usb_hcd_irq(): When registering a USB bus through the HCD framework code, this function is used to handle interrupts.

hcd_buffer_create(): This function is called as part of initializing a host controller that uses the dma memory allocators. It initializes some pools of dma-coherent memory that will be shared by all drivers using that controller.

hcd_buffer_destroy(): This function frees the buffer pools created by hcd_buffer_create.

Chapter 4

Philips ISP1160 Embedded USB Host Controller

The ISP1160 is an embedded Universal Serial Bus (USB) Host Controller (HC) that complies with Universal Serial Bus Specification, supporting data transfer at full-speed (12 Mbit/s) and low-speed (1.5 Mbit/s). The ISP1160 provides two downstream ports. The downstream ports for the HC can be connected with any USB compliant USB devices and USB hubs that have USB upstream ports.

The ISP1160 is suited for embedded systems and portable devices that require a USB host. Popular applications of the ISP1160 include:

- Personal Digital Assistants (PDA)
- Digital Cameras
- Third-generaration (3G) Mobile Phones
- Photo Printers
- MP3 Jukeboxes
- Game Consoles

Features

- Complies with Universal Serial Bus Specification Rev 2.0 (5)
- Supports data transfer at full-speed (12 Mbit/s) and low-speed (1.5 Mbit/s)
- Adapted from Open Host Controller Interface Specification for USB (2)
- Selectable one or two downstream ports for HC

- High-speed parallel interface to most of the generic microprocessors and Reduced Instruction Set Computer (RISC) processors
- Maximum 15 MByte/s data transfer rate between the microprocessor and the HC
- Supports single-cycle and burst mode DMA operations
- Built-in FIFO buffer RAM for the HC (4 KBytes)
- Endpoints with double buffering to increase throughput and ease real-time data transfer for isochronous (ISO) transactions
- 6 MHz crystal oscillator with integrated PLL for low EMI
- Built-in software selectable internal 15 KΩ pull-down resistors for HC downstream ports
- Dedicated pins for suspend sensing output and wake-up control input for flexible applications
- Operation at either +5 V or +3.3 V power supply voltage
- Operating temperature range from -40^{o}C to +85^{o}C

4.1 Host Controller Internal Registers

The Host Controller (HC) contains a set of on-chip control registers. These registers can be read or written by the Host Controller Driver (HCD). The Control and Status register sets (C.1, page 131), Frame Counter register sets (C.2, page 135), and Root Hub register sets (C.3, page 137) are grouped under the category of HC Operational registers. These operational registers are made compatible to OpenHCI [2] operational registers. This allows the OpenHCI HCD to be easily ported to the ISP1160.

Reserved bits may be defined in future releases of the chip specification. To ensure interoperability, the HCD must not assume that a reserved field contains logic 0. Furthermore, the HCD must always preserve the values of the reserved field. When a R/W register is modified, the HCD must first read the register, modify the bits desired, and then write the register with the reserved bits still containing the original value. Alternatively, the HCD can maintain an in-memory copy of previously written values that can be modified and then written to the HC register. When a "write to set" or "clear the register" is performed, bits written to reserved fields must be logic 0.

As shown in Table 4.1, the addresses (the commands for reading registers) are similar to the offsets defined in the OHCI specification (2, chapter 7) with the addresses being equal to offset divided by 4.

Table 4.1: HC Registers Summary

Read	Write	Register	Width	Reference
00	N/A	HcRevision	32	C.1.1, page 131
01	81	HcControl	32	C.1.2, page 131
02	82	HcCommandStatus	32	C.1.3, page 132
03	83	HcInterruptStatus	32	C.1.4, page 133
04	84	HcInterruptEnable	32	C.1.5, page 134
05	85	HcInterruptDisable	32	C.1.6, page 135
0D	8D	HcFmInterval	32	C.2.1, page 135
0E	N/A	HcFmRemaining	32	C.2.2, page 136
0F	N/A	HcFmNumber	32	C.2.3, page 137
11	91	HcLSThreshold	32	C.2.4, page 137
12	92	HcRhDescriptorA	32	C.3.1, page 137
13	93	HcRhDescriptorB	32	C.3.2, page 139
14	94	HcRhStatus	32	C.3.3, page 140
15	95	HcRhPortStatus(1)	32	C.3.4, page 142
16	96	HcRhPortStatus(2)	32	C.3.4, page 142
20	A0	HcHardwareConfiguration	16	C.4.1, page 147
21	A1	HcDMAConfiguration	16	C.4.2, page 149
22	A2	HcTransferCounter	16	C.4.3, page 150
24	A4	HcμPInterrupt	16	C.4.4, page 150
25	A5	HcμPInterruptEnable	16	C.4.5, page 152
27	N/A	HcChipID	16	C.5.1, page 153
28	A8	HcScratch	16	C.5.2, page 154
N/A	A9	HcSoftwareReset	16	C.5.3, page 154
2A	AA	HcITLBufferLength	16	C.6.1, page 154
2B	AB	HcATLBufferLength	16	C.6.2, page 154
2C	N/A	HcBufferStatus	16	C.6.3, page 155
2D	N/A	HcReadBackITL0Length	16	C.6.4, page 155
2E	N/A	HcReadBackITL1Length	16	C.6.5, page 156
40	C0	HcITLBufferPort	16	C.6.6, page 156
41	C1	HcATLBufferPort	16	C.6.7, page 156

4.1.1 Root Hub Registers

These registers are dedicated to the USB Root Hub, which is an integral part of the HC although it is functionally a separate entity.

The Host Controller Driver (HCD) emulates USBD accesses to the Root Hub via a register interface. The HCD maintains many USB-defined hub features that are not required to be supported in hardware. For example, the Hub's Device, Configuration, Interface, and Endpoint Descriptors (refer to Appendix B), as well as some static fields of the Class Descriptor (refer to B.6, on page 128), are maintained only in the HCD. The HCD also maintains and decodes the Root Hub's device address as well as other minor operations more suited for software than hardware.

Four 32 bit registers have been defined:

1. HcRhDescriptorA
2. HcRhDescriptorB
3. HcRhStatus
4. HcRhPortStatus[1:Number of Downstream Ports (NDP)]

Each register is read and written as a DWORD. These registers are only written during initialization to correspond with the system implementation. The HcRhDescriptorA and HcRhDescriptorB registers are writeable regardless of the HC's USB states. HcRhStatus and HcRhPortStatus are writeable during the USBOperational state only.

More information on the use of the root hub registers can be found in Appendix C.3, on page 137.

4.2 Microprocessor Bus Interface

Figure 4.1 shows the block diagram of the ISP1160. The microprocessor bus interface is a high-speed parallel interface that connects the ISP1160 to an external microprocessor. Both a Programmed I/O and a DMA interface are defined.

4.2.1 Programmed I/O (PIO) Addressing Mode

A generic PIO interface is defined for speed and ease-of-use. It also allows direct interfacing to most microcontrollers. To a microcontroller, the ISP1160 appears as a memory device with a 16 bit data bus and uses the A0 address line to access internal control registers and FIFO buffer RAM. Therefore, the ISP1160 occupies only two I/O ports or two memory locations of a microprocessor. External microprocessors can read from or write to the ISP1160's internal control registers and FIFO buffer RAM through the Programmed I/O (PIO) operating mode. Figure 4.2 shows the Programmed I/O interface between a microprocessor and the ISP1160.

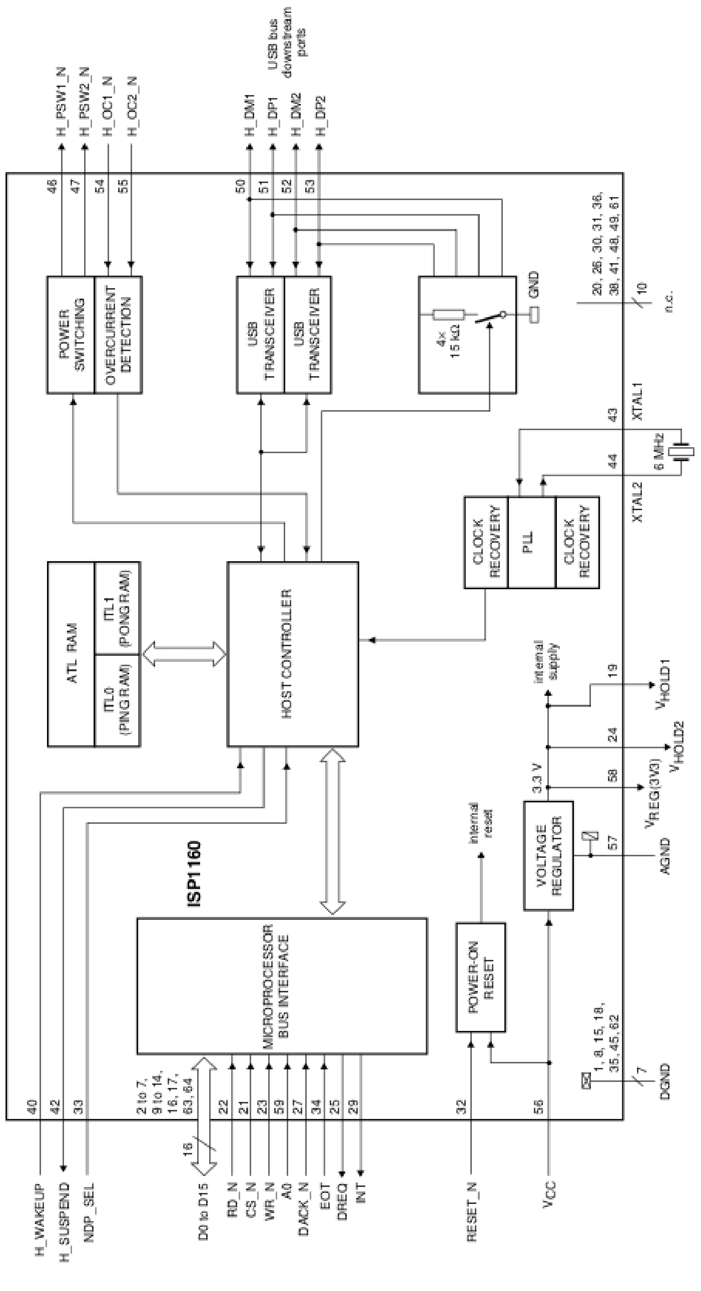


Figure 4.1: Block Diagram

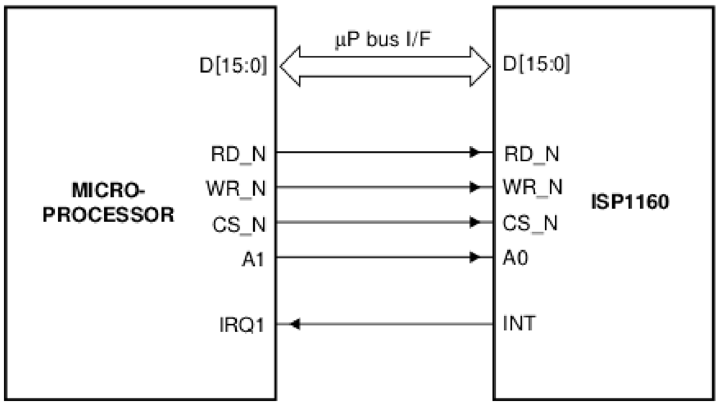


Figure 4.2: ISP1160 Programmed I/O interface

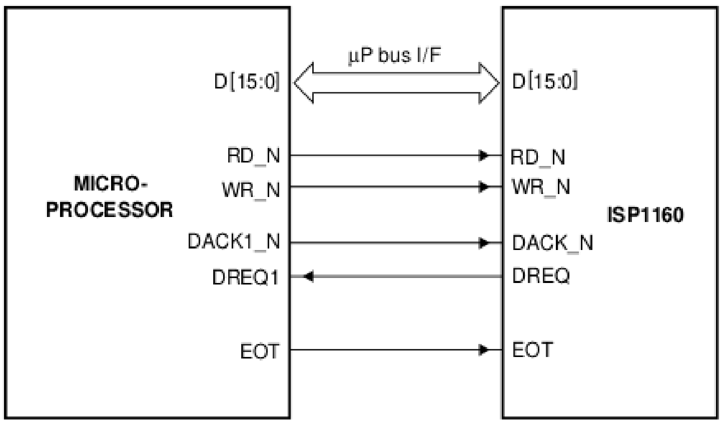


Figure 4.3: ISP1160 DMA interface

4.2.2 DMA Mode

The ISP1160 also provides the DMA mode for external microprocessors to access its internal FIFO buffer RAM. Data can be transferred by the DMA operation between a microprocessor's system memory and the ISP1160's internal FIFO buffer RAM.

Figure 4.3 shows the DMA interface between a microprocessor system and the ISP1160. The ISP1160 provides a DMA channel controlled by DREQ for DACK_N signals for the DMA transfer between a microprocessor's system memory and the ISP1160 HC's internal FIFO buffer RAM.

The EOT signal is an external end-of-transfer signal used to termi-

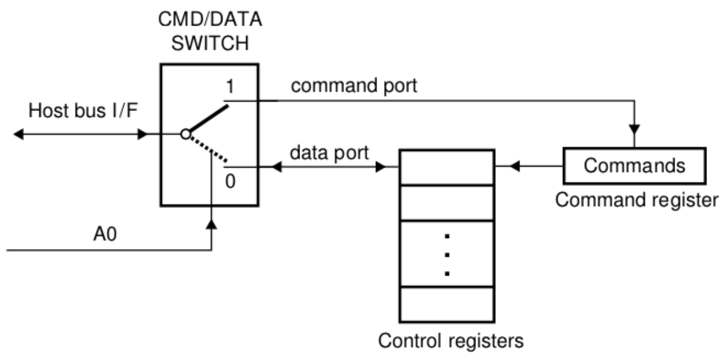


Figure 4.4: Access to ISP1160's Internal Registers

nate the DMA transfer. Some microprocessors may not have this signal. In this case, the ISP1160 provides an internal EOT signal to terminate the DMA transfer as well. Setting the HcDMAConfiguration register (C.4.2, page 149) enables the ISP1160's HC internal DMA counter for the DMA transfer. When the DMA counter reaches the value set in the HcTransferCounter register (C.4.3, page 150), an internal EOT signal will be generated to terminate the DMA transfer.

4.2.3 Control Registers Access by PIO Mode

The ISP1160's register structure is a command-data register pair structure. A complete register access cycle comprises a command phase followed by a data phase. The command (also known as the index of a register) points the ISP1160 to the next register to be accessed. A command is 8 bits long. On a microprocessor's 16 bit data bus, a command occupies the lower byte, with the upper byte filled with zeros.

Figure 4.4 illustrates how an external microprocessor accesses the ISP1160's internal control registers. In a complete 16 bit register access cycle, the microprocessor writes a command code to the command port, and then reads from or writes the data word to the data port.

Most of the ISP1160's internal control registers are 16 bit wide. Some of the internal control registers, however, are 32 bit wide (consult Table 4.1, on page 53). The complete cycle of accessing a 32 bit register consists of a command phase followed by two data phases. In the two data phases, the microprocessor first reads or writes the lower 16 bit data, followed by the upper 16 bit data.

4.2.4 FIFO Buffer RAM Access by PIO Mode

Since the ISP1160's internal memory is structured as a FIFO buffer RAM, the FIFO buffer RAM is mapped to dedicated register fields. Therefore, accessing the internal FIFO buffer RAM is similar to accessing the internal control registers in multiple data phases.

For a write cycle, the microprocessor first writes the FIFO buffer RAM's command code to the command port, and then writes the data words one by one to the data port until half of the transfer's byte count is reached. The HcTransferCounter register is used to specify the byte count of a FIFO buffer RAM's read cycle or write cycle. Every access cycle must be in the same access direction. The read cycle procedure is similar to the write cycle.

4.2.5 FIFO Buffer RAM Access by DMA Mode

When doing a DMA transfer, at the beginning of every burst the ISP1160 outputs a DMA request to the microprocessor via pin DREQ. After receiving this signal, the microprocessor will reply with a DMA acknowledge to the ISP1160 via pin DACK_N, and at the same time, execute the DMA transfer through the data bus. In the DMA mode, the microprocessor must issue a read or write signal to the ISP1160's pins RD_N or WR_N. The ISP1160 will repeat the DMA cycles until it receives an EOT signal to terminate the DMA transfer.

The ISP1160 supports both external and internal EOT signals. The external EOT signal is received as input on pin EOT, and generally comes from the external microprocessor. The internal EOT signal is generated inside the ISP1160. The ISP1160 supports either single-cycle DMA operation or burst mode DMA operation.

4.2.6 Interrupts

The ISP1160 has an interrupt request pin INT. As shown as in Figure 4.5, there are many interrupt events associated with the INT pin.

There are two groups of interrupts represented by group 1 and group 2 in Figure 4.5. A pair of registers control each group:

Group 2 contains six possible interrupt events, recorded in the HcInterruptStatus register (C.1.4, page 133). On occurrence of any of these events, the corresponding bit would be set to logic 1; and if the corresponding bit in the HcInterruptEnable register (C.1.5, page 134) is also logic 1, the 6-input OR gate would output a logic 1. This output is AND-ed with the value of MIE (bit 31 of HcInterruptEnable). Logic 1 at the AND gate will cause

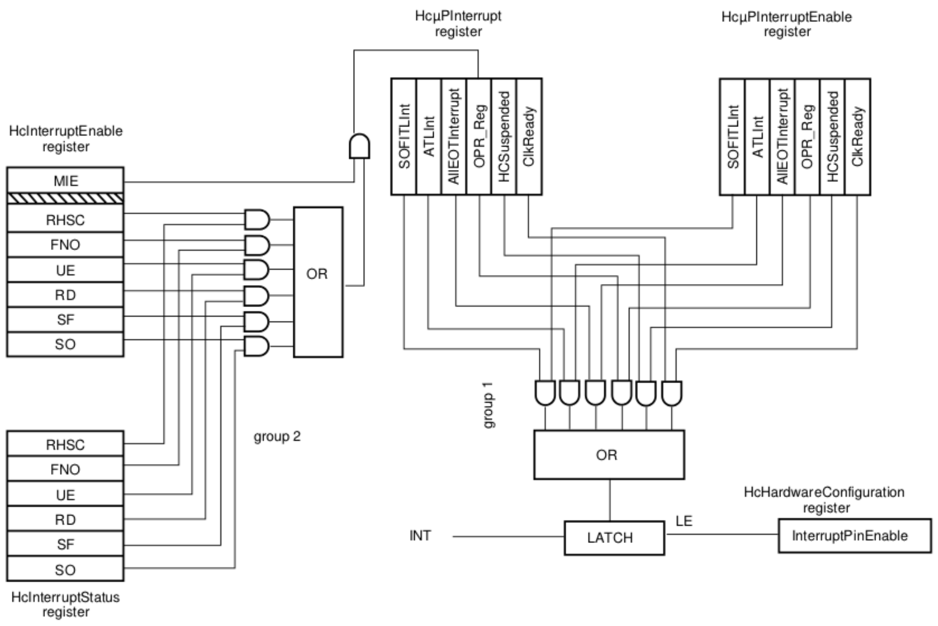


Figure 4.5: ISP1160 Interrupt Logic

the OPR bit in the HcμPInterrupt register (C.4.4, page 150) to be set to logic 1.

Group 1 contains six possible interrupt events, one of which is the output of group 2 interrupt sources. The HcμPInterrupt and HcμPInterruptEnable registers work in the same way as the HcInterruptStatus and HcInterruptEnable registers in the interrupt group 2. The output from the 6-input OR gate is connected to a latch, which is controlled by bit 0 (InterruptPinEnable) of the HcHardwareConfiguration register (C.4.1, page 147).

Pin Configuration

The interrupt output signals have four configuration modes, as shown in Table 4.2. They are programmable through the HcHardwareConfiguration register.

Table 4.2: ISP1160 INT pin configuration

Mode 0	level trigger	active LOW
Mode 1	level trigger	active HIGH
Mode 2	edge trigger	active LOW
Mode 3	edge trigger	active HIGH

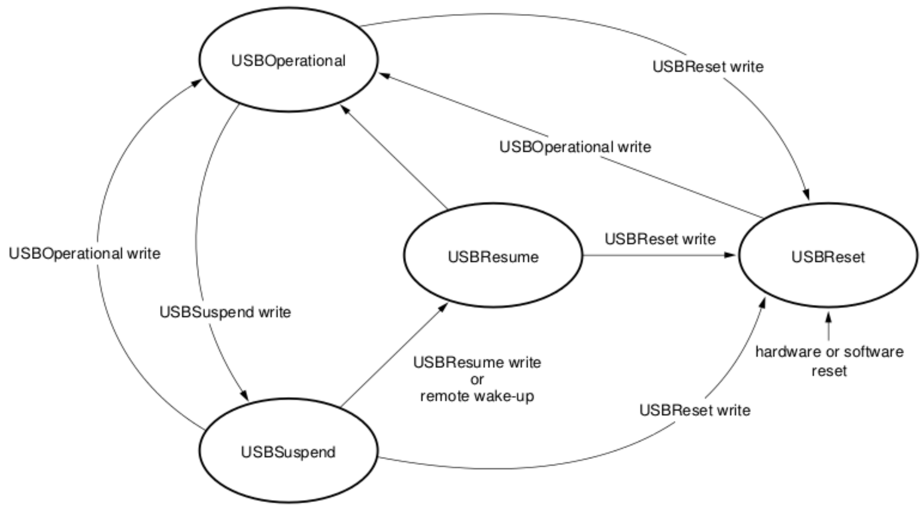


Figure 4.6: ISP1160 USB States

4.3 Host Controller (HC) Functionality

4.3.1 HC's USB States

The ISP1160's USB HC has four USB states: USBOperational, USBReset, USBSuspend and USBResume. These states define the HC's USB signalling and bus states responsibilities. The signals are visible to the Host Controller Driver (HCD) via the ISP1160 USB HC's control registers.

The USB states are reflected in the HostControllerFunctionalState field of the HcControl register (C.1.2, page 131). The HCD can perform only the USB state transitions shown in Figure 4.6.

4.3.2 USB Traffic Generation

USB traffic can be generated only when the ISP1160 USB HC is in the USBOperational state. Therefore, the HCD must set the HostCon-

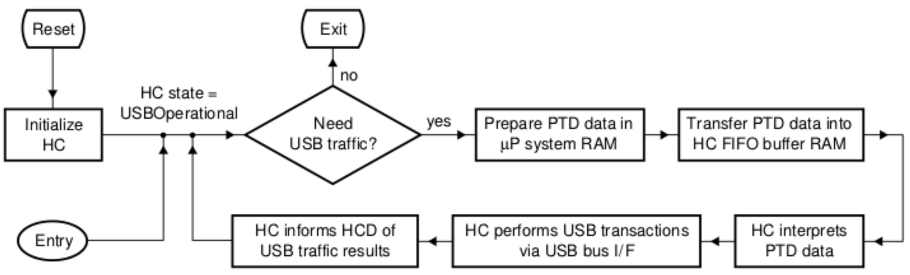

Reset
Exit
no
HC state = USBOperational
Initialize HC
Need USB traffic?
yes
Prepare PTD data in µP system RAM
Transfer PTD data into HC FIFO buffer RAM
Entry
HC informs HCD of USB traffic results
HC performs USB transactions via USB bus I/F
HC interprets PTD data

Figure 4.7: ISP1160 HC Transaction Loop

trollerFunctionalState field of the HcControl register before generating USB traffic.

A simplistic flow diagram showing when and how to generate USB traffic is shown in Figure 4.7. For greater accuracy, refer to the Universal Serial Bus Specification (5) and the ISP1160 USB HC's product data (8).

Description of Figure 4.7:

1. **Reset**. This includes hardware reset by pin RESET_N and software reset by the HcSoftwareReset command (C.5.3, page 154). The reset function will clear all the HC's internal control registers to their reset status. After reset, the HCD must initialize the ISP1160 USB HC by setting some registers.

2. **Initialize HC**:

   (a) Set the physical size for the HC's internal FIFO buffer RAM by setting the HcITLBufferLength register (C.6.1, page 154) and the HcATLBufferLength register (C.6.2, page 154).

   (b) Set the HcHardwareConfiguration register according to requirements.

   (c) Clear interrupt events, if required.

   (d) Enable interrupt events, if required.

   (e) Set the HcFmInterval register (C.2.1, page 135).

   (f) Set the HC's Root Hub registers.

   (g) Set the HcControl register to move the HC into the USBOperational state.

3. **Entry**. The normal entry point. The microprocessor returns to this point when there are HC requests.

4. **Need USB traffic**. USB devices need the HC to generate USB traffic when they have USB traffic requests such as:

   (a) Connecting to or disconnecting from downstream ports
   (b) Issuing the Resume signal to the HC.

   To generate USB traffic, the HCD must enter the USB transaction loop.

5. **Prepare PTD data in system RAM**. The communication between the HCD and the ISP1160 HC is in the form of Philips Transfer Descriptor (PTD) data. The PTD data provides USB traffic information about the commands, status and USB data packets. The physical storage media of PTD data for the HCD is the microprocessor's system RAM. For the ISP1160's HC, the storage media is the internal FIFO buffer RAM. The HCD prepares PTD data in the microprocessor's system RAM for transfer to the ISP1160's HC internal FIFO buffer RAM.

6. **Transfer PTD data into HC's FIFO buffer RAM**.When PTD data is ready in the microprocessor's system RAM, the HCD must transfer the PTD data from the microprocessor's system RAM into the ISP1160's internal FIFO buffer RAM.

7. **HC interprets PTD data**. The HC determines what USB transactions are required based on the PTD data that has been transferred into the internal FIFO buffer RAM.

8. **HC performs USB transactions via the USB bus interface**. The HC performs the USB transactions with the specified USB device endpoint through the USB bus interface.

9. **HC informs HCD of the USB traffic results**. The USB transaction status and the feedback from the specified USB device endpoint will be put back into the ISP1160's HC internal FIFO buffer RAM in PTD data format. The HCD can read back the PTD data from the internal FIFO buffer RAM.

4.3.3 The Structure of Philips Transfer Descriptors

The Philips Transfer Descriptor (PTD) data structure provides communication between the HCD and the ISP1160's USB HC. The PTD data contains information required by the USB traffic. PTD data consists of a PTD followed by its payload data, as shown in Figure 4.8.

The PTD data structure is used by the HC to define a buffer of data that will be moved to or from an endpoint in the USB device. This data buffer is set up for the current frame (1 ms frame) by the HCD. The payload data for every transfer in the frame must have a

Table 4.3: PTD Bit Allocation

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Byte 0	ActualBytes[7:0]							
Byte 1	CompletionCode[3:0]				Active	Toggle	ActualBytes[9:8]	
Byte 2	MaxPacketSize[7:0]							
Byte 3	EndpointNumber[3:0]				Last	Speed	MaxPacketSize[9:8]	
Byte 4	TotalBytes[7:0]							
Byte 5	reserved		B5_5	reserved	DirectionPID[1:0]		TotalBytes[9:8]	
Byte 6	Format	FunctionAddress[6:0]						
Byte 7	reserved							

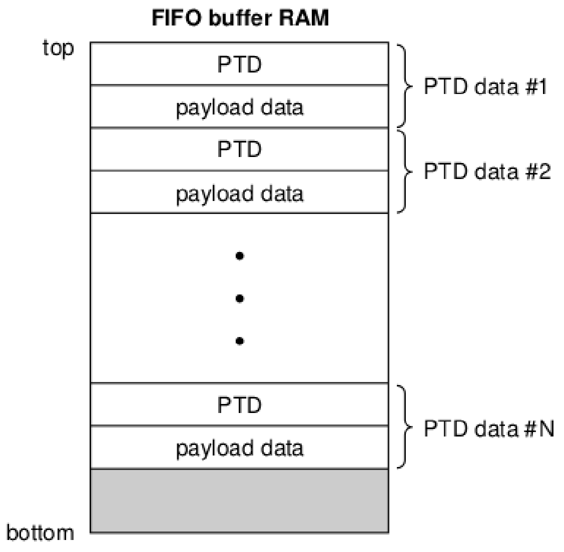


Figure 4.8: PTD Data in FIFO Buffer RAM

PTD as the header to describe the characteristic of the transfer. The PTD data is DWORD (double-word or 4 byte) aligned.

The PTD forms the header of the PTD data. It tells the HC the transfer type, where the payload data should go, and the actual size of the payload data. A PTD is an 8 byte data structure that is very important for HCD programming. Table 4.3 shows the PTD bit allocation, while Table 4.4 summarizes the role of each bit.

Table 4.4: PTD Bit Description

Symbol	**Access**	**Description**		
ActualBytes(9:0)	R/W	Contains the number of bytes that were transferred for this PTD		
CompletionCode(3:0)	R/W	0000	NoError	Data packet processing completed with no detected errors.
		0001	CRC	Last data packet from endpoint contained a CRC error.
		0010	Bit Stuffing	Last data packet from endpoint contained a bit stuffing violation.
		0011	DataToggle Mismatch	Last packet from endpoint had data toggle PID that did not match the expected value.
		0100	Stall	TD was moved to the Done queue because the endpoint returned a STALL PID.
		0101	Device Not Responding	Device did not respond to token (IN) or did not provide a handshake (OUT).
		0110	PIDCheck Failure	Check bits on PID from endpoint failed on data PID (IN) or handshake (OUT).

table continued on next page...

Table 4.4: PTD Bit Description (continued)

Symbol	Access	Description
CompletionCode(3:0)	R/W	0111 Unexpected PID: Received PID was not valid when encountered or PID value is not defined. 1000 Data Overrun: The amount of data returned exceeded either the size of the maximum data packet allowed from the endpoint or the remaining buffer size. 1001 Data Underrun: The amount of data was not sufficient to fill the specified buffer. 1010 reserved: – 1011 reserved: – 1100 BufferOverrun: During an IN, the HC received data from an endpoint faster than it could be written to system memory. 1101 BufferUnderrun: During an OUT, the HC could not retrieve data from the system memory fast enough to keep up with the data rate.
Active	R/W	Set to logic 1 by firmware to enable the execution of transactions by the HC. When the transaction is completed, the HC sets this bit to logic 0.
Toggle	R/W	Used to generate or compare the data PID value (DATA0 or DATA1). It is updated after each successful transmission or reception of a data packet.
MaxPacketSize(9:0)	R	The maximum number of bytes that can be sent to or received from the endpoint in a single data packet.
EndpointNumber(3:0)	R	USB address of the endpoint within the function.
Last	R	Last PTD of a list (ITL or ATL). Logic 1 indicates that the PTD is the last PTD.
Speed	R	Speed of the endpoint: • 0 – full speed • 1 – low speed
TotalBytes(9:0)	R	Specifies the total number of bytes to be transferred with this data structure. For Bulk and Control only, this can be greater than MaximumPacketSize.

table continued on next page...

Table 4.4: PTD Bit Description (continued)

Symbol	Access	Description
DirectionPID(1:0)	R	00 SETUP 01 OUT 10 IN 11 reserved
B5_5	R/W	This bit is logic 0 at power-on reset. When this feature is not used, software used for the ISP1160 is the same for the ISP1161 and the ISP1161A. When this bit is set to logic 1 in this PTD for interrupt endpoint transfer, only one PTD USB transaction will be sent out in 1 ms.
Format	R	The format of this data structure. If this is a Control, Bulk or Interrupt endpoint, then Format = 0. If this is an Isochronous endpoint, then Format = 1.
FunctionAddress(6:0)	R	This is the USB address of the function containing the endpoint that this PTD refers to.

4.3.4 Internal FIFO Buffer RAM Data Organization

The HC's internal FIFO buffer RAM has a physical size of 4 KBytes. This internal FIFO buffer RAM is used for transferring data between the microprocessor and USB peripheral devices. This on-chip buffer RAM can be partitioned into two areas: Acknowledged Transfer List (ATL) buffer and Isochronous (ISO) Transfer List (ITL) buffer. The ITL buffer is a Ping-Pong structured FIFO buffer RAM that is used to keep the payload data and their PTD header for Isochronous transfers. The ATL buffer is a non Ping-Pong structured FIFO buffer RAM that is used for the other three types of transfers.

The ITL buffer can be further partitioned into ITL0 and ITL1 for the Ping-Pong structure. The ITL0 and ITL1 buffers always have the same size. The microprocessor can put ISO data into either the ITL0 buffer or the ITL1 buffer. When the microprocessor accesses an ITL buffer, the HC can take over the other ITL buffer at the same time. This architecture improves the ISO transfer performance.

The HCD can assign the logical size for the ATL buffer and ITL buffers at any time, but normally at initialization after power-on reset. This is done by setting the HcATLBufferLength register and the HcITLBufferLength register, respectively. The total length (ATL buffer + ITL buffer) should not exceed the maximum RAM size of 4 KBytes. Figure 4.9 shows the partitions of the internal FIFO buffer RAM.

When the embedded system wants to initiate a transfer to the USB bus, the data needed for one frame is transferred to the ATL buffer or the ITL buffer. The microprocessor detects the buffer status

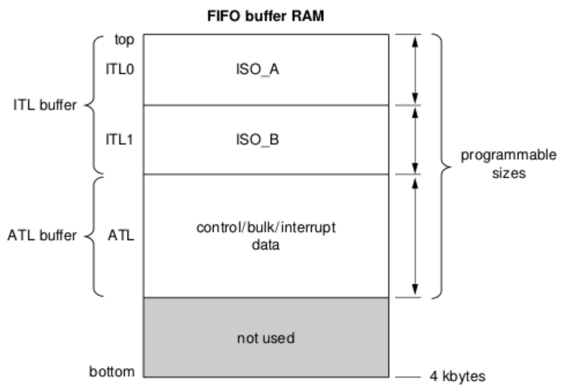


Figure 4.9: HC Internal FIFO RAM Partitions

through interrupt routines. When the HcBufferStatus register (C.6.3, page 155) indicates that the buffer is empty, then the microprocessor writes data into the buffer. When the HcBufferStatus register indicates that the buffer is full, the data is ready on the buffer, and the microprocessor needs to read data from the buffer.

The data transfer can be done via the PIO mode or the DMA mode. The data transfer rate can go up to 15 MByte/s. In the DMA operation, the single-cycle or multi-cycle burst modes are supported. Multi-cycle burst modes of 1, 4 or 8 cycles per burst are supported for the ISP1160.

Data Organization

PTD data is used for every data transfer between a microprocessor and the USB bus, and the PTD data resides in the buffer RAM. For an OUT or SETUP transfer, the payload data is placed just after the PTD, after which the next PTD is placed. For an IN transfer, RAM space is reserved for receiving a number of bytes that is equal to the total bytes of the transfer. After this, the next PTD and its payload data are placed (see Figure 4.10).

The PTD data (PTD header and its payload data) is a structure of DWORD alignment. This means that the memory address is organized in blocks of 4 bytes. Therefore, the first byte of every PTD and the first byte of every payload data are located at an address that is a multiple of 4.

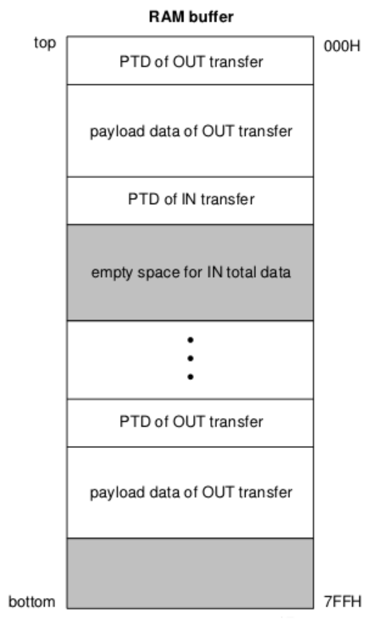

RAM buffer
top
000H
PTD of OUT transfer
payload data of OUT transfer
PTD of IN transfer
empty space for IN total data
PTD of OUT transfer
payload data of OUT transfer
bottom
7FFH

Figure 4.10: HC FIFO RAM Data Organization

4.3.5 HC Operational Model

Upon power-up, the HCD sets up all operational registers (32 bit). The FSLargestDataPacket field (bits 30 to 16) of the HcFmInterval register and the HcLSThreshold register (C.6.3, page 155) determine the end of the frame for full-speed and low-speed packets. By programming these fields, the effective USB bus usage can be changed. Furthermore, the size of the ITL and ATL buffers is programmed.

If a USB frame contains both ISO and AT packets, two interrupts will be generated per frame. One interrupt is issued concurrently with the SOF. This interrupt (ITLInt is set in the HcμPInterrupt register) triggers reading and writing of the ITL buffer by the microprocessor, after which the interrupt is cleared by the microprocessor. Next the programmable ATL Interrupt (bit ATLInt is set in the HcμPInterrupt register) is issued, which triggers reading and writing of the ATL buffer by the microprocessor, after which the interrupt is cleared by the microprocessor. If the microprocessor cannot handle the ISO interrupt before the next ISO interrupt, disrupted ISO traffic can result.

To be able to send more than one packet to the same Control or Bulk endpoint in the same frame, the Active bit and the TotalBytes field are introduced. Bit Active is cleared only if all data of the Philips Transfer Descriptor (PTD) have been transferred or if a transaction at

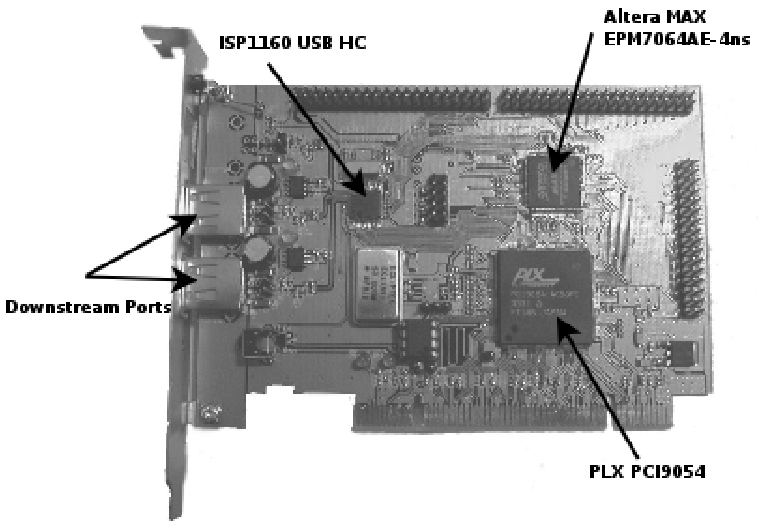


Figure 4.11: ISP1160 PCI Development Board

that endpoint contained a fatal error. If all PTDs of the ATL are serviced once and the frame is not over yet, the HC starts looking for a PTD with bit Active still set. If such a PTD is found and there is still enough time in this frame, another transaction is started on the USB bus for this endpoint.

For ISO processing, the HCD also has to take care of the Buffer-Status register for the ITL buffer RAM operations. After the HCD writes ISO data into ITL buffer RAM, the ITL0BufferFull or ITL1BufferFull bit (depending on whether it is ITL0 or ITL1) will be set to logic 1. After the HC processes the ISO data in the ITL buffer RAM, the corresponding ITL0BufferDone or ITL1BufferDone bit will automatically be set to logic 1. The HCD can clear the buffer status bits by a read of the ITL buffer RAM. This must be done within the 1 ms frame from which ITL0BufferDone or ITL1BufferDone was set. Failure to do so will cause the ISO processing to stop and a power-on reset or software reset will have to be applied to the HC, a USB reset to the USB bus must not be made.

4.4 The ISP1160 PCI Development Board

This work was done using the ISP1160 PCI development board. This board incorporates the ISP1160 in a PCI device.

As shown in Figure 4.11, the development board includes:

- The ISP1160 HC

- The PLX9054 PCI I/O Accelerator
- An Altera CPLD
- 2 Downstream Ports

In order to understand the functionality of this board, an overview of the PCI bus is required. For a full description of the PCI specification, see (7).

4.4.1 PCI Bus Overview

The PCI architecture was designed as a replacement for the ISA standard, with three main goals:

1. Better performance
2. Platform independency
3. Simple insertion/removal of peripherals

PCI is currently used extensively on IA-32, Alpha, PowerPC, SPARC64, and IA-64 systems, and some other platforms as well.

What is more relevant to the driver writer, however, is the support for autodetection of interface boards. PCI devices are jumperless and are automatically configured at boot time. Every PCI motherboard is equipped with PCI-aware firmware.

The device driver must be able to access configuration information in the device, in order to complete initialization. This information is stored in a 256 byte location, called the "PCI configuration space".

PCI Configuration Space

The layout of the configuration registers is device independent. PCI devices feature a 256 byte address space. The first 64 bytes are standarized, while the rest are device dependent. Figure 4.12 shows the layout of the registers.

As the figure shows, some of the registers are required and some are optional. All registers are always little-endian.

Hardware Resources on the PCI Bus

Devices on the PCI bus use the PCI configuration space to report the location of their hardware resources. These resources include:

- I/O Ports

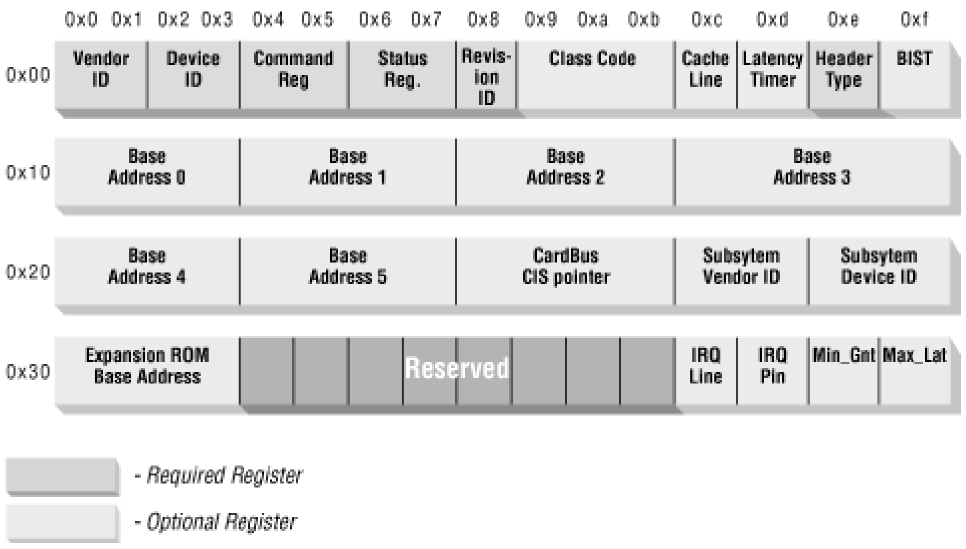


Figure 4.12: PCI Configuration Registers

- I/O Memory
- Interrupt Lines

I/O and Memory Spaces A PCI device implements up to six I/O address regions. Each region consists of either memory or I/O locations. The interface reports the size and current location (as assigned by firmware at boot time) of its regions, using configuration registers– the six 32 bit registers shown in Figure 4.12, Base Address 0 through Base Address 5.

PCI Interrupts By the time the operating system boots, the firmware has already assigned a unique interrupt number to the device. The interrupt number is stored in the configuration space. The register holding the interrupt number is one byte wide. This allows for as many as 256 interrupt lines, but the actual limit depends on the CPU being used. All PCI interrupts are "level-triggered", "active-low" and must be shared.

4.4.2 The PLX9054 PCI I/O Accelerator

As shown in Section 4.2, on page 54, the ISP1160 offers a PIO or DMA interface to access its internal registers and buffer RAM. In the case of the PCI development board, these ports must be mapped to the PCI

I/O space. Furthermore, the INT pin of the ISP1160 must be mapped to one of the INT pins of the PCI interface.

This is the role of the PLX9054 PCI I/O accelerator. As shown in Figure 4.11, the PLX9054's physical location on the board is just above the connector to the PCI slot. This chip is able to connect an arbitrary bus (named "local bus") to the PCI bus. This versatility comes at a cost: the configuration of the PLX9054 is very complicated and needs updating for almost every event on the "local bus" side.

The setup of the PLX9054 can be accomplished from both sides (local & PCI). This is the role of the Altera CPLD, shown in Figure 4.11. From the "local bus" side, the CPLD configures the PLX9054. The exact behaviour of the CPLD is not known, but it can be summarized in two functions:

1. During a hardware reset, the CPLD sets the PLX9054 to its initial values.

2. During normal operation, the CPLD monitors the ISP1160 and configures accordingly the PLX9054

Note: Under normal circumstances this would mean that the development board is ready for use. However, there is one single register of the PLX9054 that is not configured from the CPLD. The register in question is the Interrupt Control/Status register, bit 11 (Local Interrupt Input Enable). This bit defaults to 0, thus not allowing a local interrupt to assert a PCI interrupt. To complete the setup of the PLX9054 we need to set this bit to 1, using the PCI side configuration.

For more information on the role, functionality and registers of the PLX9054, see (3).

Chapter 5

ISP1160 PCI Driver Implementation

This chapter presents the actual work done to create the Host Controller Driver. Several parts of the source code will be exposed here, in order to understand the functionality of the driver. The full source code is available with this document, along with a manual created with the "doxygen" tool and presented in both *HTML* and *PDF* format.

This work is based on the book "Linux Device Drivers" (9). The second edition of this book is available for free on the Internet, by request of the authors. This book covers all the basic knowledge needed to write a device driver for the Linux operating system. However, device drivers for host controllers (for IDE, SCSI, USB etc.) are considered a special case and require handling that is not covered in the scope of the book "Linux Device Drivers".

The main difference between a normal device driver and a host controller driver is that the latter needs to manage all the devices that lay on the controller's bus, as well as itself (the host controller device). This expands to:

1. Enumerating any new device
2. Keeping track of the state of connected devices
3. Frame scheduling of any device that needs communication

As always, the best place to look for information when writing a Linux device driver, is the Linux kernel. This is because the kernel is frequently updated/modified and any attempt to document the kernel internals is bound to be outdated.

This work is also heavily based on the OHCI driver code, standard part of the Linux kernel, written by *Roman Weissgaerber* and *David*

Brownell. The reason is that the OHCI driver is known to be created "by the book" and thus is considered the best example to follow when writing a new HCD. Another reason is the similarity between the design of the ISP1160 and the OpenHCI (2).

Philips, the designer of the ISP1160, has also published a programming guide (6), that describes a typical hardware initialization sequence and presents a method to accomplish basic data flow on the USB bus. However, most of the methods presented are just a simplified version of the OpenHCI specification.

5.1 Important Data Structures

Before proceeding further in the analysis of the HCD, we must present a set of very important data structures to the driver. These structures are the basis of the driver development.

In Section 3.4.1, on page 44, we presented *struct usb_hcd*. As mentioned there, this structure is very important to the HCD. In order to enhance its functionality, we present *struct isp116x*, a structure that wraps around usb_hcd.

```
struct isp116x {
  struct usb_hcd        hcd;
  spinlock_t            lock;
  atomic_t              atl_finishing;

  struct plx9054_regs   __iomem *plxbase;

  u16                   addr_reg;
  u16                   data_reg;

  u32                   intenb;
  u16                   irqenb;
  u32                   rhdesca;
  u32                   rhdescb;
  u32                   rhstatus;
  u32                   rhport[2];

  struct list_head      async;

  u16                   load[PERIODIC_SIZE];
  struct isp116x_ep     *periodic[PERIODIC_SIZE];
  unsigned              periodic_count;
  u16                   fmindex;
  struct isp116x_ep      *atl_active;
  int                    atl_buflen;
  int                    atl_bufshrt;
  int                    atl_last_dir;
};
```

hcd: *isp116x* is a wrapper around *usb_hcd*.

lock: Spinlock variable to hold, in order to avoid race conditions.

atl_finishing: This variable is of type "atomic_t", defined in the Linux kernel. It is used to share the integer value between the interrupt handler and other functions. If *atl_finishing* is non-zero then the HCD is still busy finishing previous ATL transfers. Variables of type "atomic_t" must be handled using the functions defined in 'asm/atomic.h', such as *atomic_set(), atomic_read(), atomic_inc()* and *atomic_dec()*.

plxbase: Memory-mapped location of the PLX9054 registers (refer to 4.4.2, on page 71).

addr_reg: Location of the ISP1160 command port.

data_reg: Location of the ISP1160 data port.

intenb, irqenb, rhdesca, rhdescb, rhstatus, rhport(2): To avoid frequent access to the registers of the ISP1160, the driver keeps an in-memory copy of the interrupt enable registers (HcInterruptEnable (C.1.5, page 134) and HcμPInterruptEnable (C.4.5, page 152)) and the Root Hub registers (C.3, page 137). These variables are updated every time the driver writes a new value to these registers.

async: List containing pending Control/Bulk (asynchronous)transfers.

load(PERIODIC_SIZE): Current data load of a list with pending Interrupt Transfers. This is used to calculate if a new Interrupt transfer request can be added to the list without exceeding the maximum periodic load.

periodic(PERIODIC_SIZE): Array of lists containing pending Interrupt transfers. The position inside the array of a specific transfer request, marks its polling interval (see 5.3.5,on page 90).

periodic_count: This is used to count the number of periodic transfers pending. Apart from that, the HCD uses this counter to switch from SOFint to ATLint interrupt when there are no periodic transfers left (see HcμPInterrupt register C.4.4, on page 150). This helps in minimizing the number of interrupts per frame.

fmindex: The current frame. This is used for marking the starting frame of an ous transfer.

atl_active: List containg the ATL transfers that are active and due for the current frame.

atl_buflen: Size of data in ATL buffer.

atl_bufshrt: This is used instead of "atl_buflen" when the last PTD in a transfer is of "IN" direction. This helps to minimize chip access by not copying the data of the last PTD in a frame unnecessarily.

atl_last_dir: Direction (IN/OUT) of the last PTD in an ATL transfer.

Another important structure of the ISP1160 HCD, useful in scheduling the transfers, is "struct isp116x_ep". This data structure holds information about the data pending for transfer from/to a specific device endpoint (refer to 2.4.3, on page 20). More information about data scheduling and transfer can be found in Section 5.3.

```
struct isp116x_ep {
  struct list_head      queue;
  struct usb_device     *udev;
  struct ptd            ptd;

  u8                    maxpacket;
  u8                    epnum;
  u8                    nextpid;
  u16                   error_count;
  u16                   length;
  unsigned char         *data;

  struct isp116x_ep     *active;

  u16                   period;
  u16                   branch;
  u16                   load;
  struct isp116x_ep     *next;
```

```
    struct list_head        schedule;
};
```

queue: This is the queue where new URBs (refer to 3.3.1, on page 39) are submitted.

udev: Pointer to the USB device that owns this endpoint.

ptd: Philips Transfer Descriptor header (refer to 4.3.3, on page 62) for the current endpoint transfer.

maxpacket: Maximum packet size for this endpoint.

epnum: The endpoint number.

nextpid: Direction and type of data transfer to be executed (IN/OUT/SETUP/ACK).

error_count: Counter to keep track of transaction errors.

length: Length of data for the current frame. If the data size is greater that the maximum packet size, the length is set to the maximum packet size.

data: Pointer to the memory location where the transfer data is to be read/written.

active: Pointer to the next endpoint in the list of active endpoints for the current frame.

period: In case of an Interrupt endpoint, this variable holds the interrupt polling interval (see 5.3.5,on page 90).

branch: In case of an Interrupt endpoint, this variable holds the selected schedule branch.

load: Approximate time needed for the transfer. This is calculated with the help of the USB core kernel function *usb_calc_bus_time()*.

next: Pointer to the next endpoint in the list of all registered endpoints.

schedule: This is the list where scheduled transactions are placed.

5.2 PCI Module Initialization

The first thing the driver must do is to probe for the device and initialize it. This includes the allocation of I/O resources and interrupt line. Since our device is located on the PCI bus, we should take advantage of the PCI architecture that simplifies the task of probing. For more information regarding the PCI bus, refer to 4.4.1, on page 70. For a full description of the PCI bus, see (7).

5.2.1 PCI Device Probing

A new way of probing for PCI devices has been introduced in Linux kernel 2.6. This "new-style PCI driver framework" tries to unify all hot-plug devices (PCI, USB, Firewire etc.) under the same mechanism.

In the center of this mechanism lies a data structure called *struct pci_driver*. This structure points to the functions that the kernel should use to probe, remove, suspend or resume the PCI hardware and it is passed to the kernel from the entry point of the driver.

```
static struct pci_driver isp116x_driver = {
  .name          = (char *)hcd_name,
  .id_table      = isp116x_table,

  .probe         = isp116x_pci_probe,
  .remove        = isp116x_pci_remove,

  .suspend       = isp116x_pci_suspend,
  .resume        = isp116x_pci_resume,
};
```

More important to the probing task, this structure holds the "PCI device ID table" of the driver. This table describes the device that is being probed, by using known information such as the ID of the vendor, the product or even the device class. Fields can be left empty to match any device. This way, a driver can state that it can handle for example any device of a specific class or vendor. Every PCI driver should provide such a table.

The ISP1160 HCD uses its vendor (0x10b5) and product ID (0x5406) to probe for the host controller. If the kernel finds a PCI device that matches the given criteria, it bounds the driver to that device and proceeds to call the probing function.

The purpose of the probing function is largely left to the driver author to decide. When this function returns, the hardware should be initialized and ready for operation.

5.2.2 PCI I/O Resource Allocation

The first thing the driver must do, is to establish communication with the ISP1160 Host Controller. In order to be able to access the internal registers of the ISP1160, the driver should locate and request two 16 bit-wide I/O ports, the command and the data port (refer to Section 4.2.3, on page 57). These two ports are made available on the PCI bus through the PLX9054 chip (refer to Section 4.4.2, on page 71).

By the time the operating system has booted, all PCI I/O resources are uniquely assigned by the firmware. The driver needs only to read the assigned values from the PCI configuration space and then request the resources from the operating system.

Here's a snapshot of the PCI bus, as reported by the "lspci" command, running on the system that was used for the development of the driver:

```
0000:00:00.0 Host bridge: VIA Technologies, Inc.
VT8363/8365 [KT133/KM133] (rev 02)

0000:00:01.0 PCI bridge: VIA Technologies, Inc.
VT8363/8365 [KT133/KM133 AGP]

0000:00:07.0 ISA bridge: VIA Technologies, Inc.
VT82C686 [Apollo Super South] (rev 22)

0000:00:07.1 IDE interface: VIA Technologies, Inc.
VT82C586A/B/VT82C686/A/B/VT823x/A/C
PIPC Bus Master IDE (rev 10)

0000:00:07.2 USB Controller: VIA Technologies, Inc.
VT82xxxxx UHCI USB 1.1 Controller (rev 10)

0000:00:07.3 USB Controller: VIA Technologies, Inc.
VT82xxxxx UHCI USB 1.1 Controller (rev 10)

0000:00:07.4 Bridge: VIA Technologies, Inc.
VT82C686 [Apollo Super ACPI] (rev 30)

0000:00:0b.0 Ethernet controller: Linksys NC100
Network Everywhere Fast Ethernet 10/100 (rev 11)

0000:00:0c.0 Bridge: PLX Technology, Inc.:
Unknown device 5406 (rev 0b)
```

```
0000:00:0e.0 Multimedia audio controller:
Ensoniq 5880 AudioPCI (rev 02)

0000:01:00.0 VGA compatible controller:
nVidia Corporation NV11
[GeForce2 MX/MX 400] (rev a1)
```

The hardware we're interested is located in address "0000:00:0c.0". At this point, we already know this information, since the probing that took place returned a pointer to this device. Note that we located the device using the PCI vendor and device ID, without explicitly stating its address on the PCI bus. This way, we can be sure that even if the firmware decides to assign a different address to the device (something that usually doesn't happen unless a device is added or removed), the driver will be able to locate the device.

Every PCI device must be enabled before requesting its I/O resources. This is done with a call to the "pci_enable_device()" function, with a single argument pointing the the PCI device data structure. After this is done, the device is ready for use.

A closer look to the ISP1160 PCI evaluation board can be acquired using a more verbose version of the "lspci" command:

```
0000:00:0c.0 Bridge: PLX Technology, Inc.:
                     Unknown device 5406 (rev 0b)
        Subsystem: PLX Technology, Inc.:
                   Unknown device 9054
        Flags: bus master, medium devsel,
               latency 32, IRQ 9
        - Memory    at df100000
          (32-bit, non-prefetchable) [size=256]
        - I/O ports at e000 [size=256]
        - I/O ports at e400 [size=256]
        - Memory    at df000000
          (32-bit, non-prefetchable) [size=1M]
        Capabilities: [40] Power Management v1
        Capabilities: [48] #06 [0000]
        Capabilities: [4c] Vital Product Data
```

As we can see, the PCI configuration space of the device reports the interrupt channel (IRQ 9) and four I/O locations:

1. I/O memory at **df100000**. These are the memory-mapped registers of the PLX9054. They are used for the configuration of the chip.

2. I/O ports at **e000**. This is an alternative way to access the registers of the PLX9054, if memory-mapping is not possible.

3. I/O ports at **e400**. This is the most important I/O location for the Host Controller Driver. Address e400 is connected to the "data port" of the ISP1160. 2 bytes further, address e402 is connected to the "command port" of the ISP1160.

4. I/O memory at **df000000**. The purpose of this location is not known. However, since the evaluation board contains an CPLD, we can speculate that this is the way to program the CPLD.

I/O Ports and Memory

In order to initialize and use our hardware, we need to allocate the memory-mapped registers of the PLX9054, and the I/O ports of the ISP1160. The Linux kernel presents a complete set of functions to accomplish this task. These functions take an argument to the PCI Base Address that is requested (i.e. BASE_ADDRESS_0,1 etc). This way we can request the I/O regions without using their actual address. This is the right thing to do, since the firmware can change the assigned numbers. Thus, a simple allocation of the above resources, will look like this (with no error checking):

```
mem_region  = 0;
mem_resource  = pci_resource_start (dev, mem_region);
mem_len  = pci_resource_len (dev, mem_region);
request_mem_region (mem_resource, mem_len);
plxbase = ioremap_nocache (mem_resource, mem_len);

port_region  = 2;
port_resource = pci_resource_start (dev, port_region);
port_len  = pci_resource_len (dev, port_region);
request_region (port_resource, port_len);
base          = (void *) port_resource;

isp116x->data_reg = (int) base;
isp116x->addr_reg = isp116x->data_reg + 2;
isp116x->plxbase = (struct plx9054_regs *)plxbase;
```

After the allocation is done, we can see that the resources are bound to the driver, by checking the "ioports" and "iomem" entries of the "proc" filesystem:

/proc/ioports
```
e000-e0ff : 0000:00:0c.0
e400-e4ff : 0000:00:0c.0
  e400-e4ff : isp116x-hcd
```

/proc/iomem
```
df000000-df0fffff : 0000:00:0c.0
df100000-df1000ff : 0000:00:0c.0
  df100000-df1000ff : isp116x-hcd
```

Interrupt Channel

The driver needs to be bound to an interrupt channel. The number of the channel can be retrieved from the PCI configuration space. Again, we will not explicitly request a number, but rather allocate the number assigned to the device by the firmware. the function that requests the interrupt, takes as an argument a pointer to the routine to be executed when this interrupt occurs. This routine is the "Interrupt Handler" (see 5.3.1, on page 87).

After the request is complete, we can watch the interrupt channel at the "interrupts" entry of the "proc" filesystem:

/proc/interrupts
```
           CPU0
  0:     553043          XT-PIC  timer
  1:       1473          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:          0          XT-PIC  Ensoniq AudioPCI
  7:          0          XT-PIC  parport0
  8:          4          XT-PIC  rtc
 10:      32557          XT-PIC  uhci_hcd, uhci_hcd,
                                 nvidia, isp116x-hcd
 11:          0          XT-PIC  acpi
 12:      17362          XT-PIC  i8042
 14:      10854          XT-PIC  ide0
 15:         13          XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

As mentioned in Section 4.4.1, all PCI interrupts must be shared. The above snapshot of the system shows that the ISP1160 HCD is able to share its interrupt line (IRQ 10) with the UHCI HCD of the system and the graphics accelerator.

Before proceeding to the initialization of the ISP1160, the driver sets the PLX9054, using the memory-mapped registers. As noted in Section 4.4.2, the driver needs to enable the local interrupts of the board.

5.2.3 ISP1160 Initialization

The initialization of the ISP1160 requires a sequence of events that can be found inside the "ISP1160 Embedded Programming Guide" (see (6, section 5.4)). The order of events presented in (6) is not the only possible. Our driver uses a different one (only to the order of events), that fits better with the Linux device driver model for PCI devices[1].

Device Reset

The initialization of the ISP1160 begins with a reset of the device. This is done in two steps:

1. A value of 0xF6 is written in the HcSoftwareReset register (C.5.3, page 154)
2. Bit 0 (HCR) of the HcCommandStatus register (C.1.3, page 132), is set to initiate a software reset.

After that, the driver waits 20 msec for the reset operation to complete (upon completion, the HC will clear the HCR bit), before the function returns.

Wait for Clock

With the reset operation done, the HCD must wait for the clock to be ready. This is done by waiting for Bit 6 (ClkReady) of the HcμPInterrupt register (C.4.4, page 150) to be set.

HC Detection

To ensure that the device is indeed an ISP1160 HC, the driver uses the following three-step procedure:

1. HcChipID register (C.5.1, page 153) is read. The register's high byte must read 0x61 (in the case of the evaluation board, the whole register reads 0x6123).

[1]The example followed in the guide refers to an ISP1160 chip connected on the ISA bus

2. The driver writes an arbitrary value to the HcScratch register (C.5.2, page 154), and then reads the contents of the register, expecting to read the value that was written.

3. The previous step is repeated with the same value inverted.

If all three steps succeed, the function returns with no error and the driver continues to set the HC to the USBOperational state.

Root Hub Timer Initialization

The root hub is monitored for events by the kernel, through "KHUBD" (see 5.3.2, on page 88). This process uses a kernel timer, that needs to be initialized, using the kernel-provided function "init_timer()".

Device Start

In order to generate USB traffic, the HC must move to USBOperational. This is accomplished with the following steps:

1. Clear all interrupts (HcInterruptStatus and HcμPInterrupt)

2. Partition the internal buffer RAM (HcITLBufferLength and HcATLBufferLength):

   - ITL size = 0
   - ATL size = 4096

3. Set the HcHardwareConfiguration register (C.4.1, page 147) to enable interrupts, as level-triggered, active-low (as required for PCI interrupt lines

4. Set the HcFmInterval register (C.2.1, page 135). The recommended values are:

   - FrameInterval = 0x2edf
   - FSLargestDataPacket = 0x2778

5. Root Hub configuration by means of the root hub registers (refer to Appendix C.3, on page 137). This includes:

   - Setting POTPGT to 25, resulting in 50 ms of wait for the root hub ports to become operational.
   - Setting the number of downstream ports (NDP) to 2.

- The driver accepts a single boolean argument, which controls the way that the downstream ports are powered (*always on*, or *individually powered* when there is a device connected). Depending on the value of the module argument (passed during module loading), the driver sets the power switching mode (bit PSM) and overcurrent protection mode (OCPM) of the HcRhDescriptorA, and the port power control mask (bits PPCM(2:0)) of the HcRhDescriptorB.
- After the ports are configured, the driver enables port power, by setting bit LPSC of the HcRhStatus

6. Connect the virtual Root Hub. The kernel requires the root hub to be reported as a USB device, so the driver calls the function "usb_alloc_dev()" (using a special argument which denotes that the new device is a root hub) to allocate and initialize the device data structure. Next function "hcd_register_root()" is called to register the newly allocated device as a root hub, and pass the device's control to the kernel.

7. The last thing to do before entering USBOperational, is to enable the interrupt events that are needed for normal operation. These include:

   - Master Interrupt Enable
   - Root Hub Status Change
   - Unrecoverable Error
   - ATL interrupt
   - OPR interrupts

   If any other interrupts are needed, they can be enabled later.

With the above steps completed, the driver moves the host controller to USBOperational, by writing a value of 2 to the field HCFS(2:0) of the HcControl Register (C.1.2, page 131). Upon entering this state, the Host Controller will begin generating Start of Frame signals and will be able to handle USB traffic.

5.2.4 PCI Module Cleanup

The most important step in the modularization of a driver, is its removal from the system. A "clean" removal allows the driver to be reloaded many times without compromising the stability of the operating system. All steps performed during initialization must be reversed.

The ISP1160 HCD registers its removal function from the "pci_driver" data structure (refer to Section 5.2.1). This function is called from a thread context, normally "rmmod", "apmd", or something similar.

When this function is called, it goes through the following steps:

1. Disconnect the Root Hub (The root hub is considered a normal USB device, embedded in the host controller, so we call the "usb_disconnect()" function).

2. Disable all interrupts on the board

3. Power down all downstream ports

4. Software reset the chip (see also 5.2.3)

5. Detach the driver from the allocated interrupt line

6. Unmap and deallocate all I/O resources (both memory and ports)

7. Unregister the USB bus from the system

8. Deallocate the data structures used by the HCD

Note that this sequence of events is applicable only if the driver has been correctly initialized prior to its removal. In case of abnormal initialization, this function is not called. It is the responsibility of the probing function to clean all the steps that were completed before the error. This is accomplished in the ISP1160 HCD with the help of "goto" commands. For more information, refer to (9, Chapter 2, pages 30-32) and the source code.

5.3 Data Transfer Management

After the driver has been successfully initialized and registered with the operating system, the driver module sits in the background and waits for transfer requests on the USB bus. As mentioned in Chapter 3, the Linux USB subsystem uses the USB Core for communication between the USB devices and the USB host controllers (see also Figure 3.1, on page 33). This communication is in the form of URBs. During module initialization, the driver informs the kernel of its methods for accepting URBs. When an application needs to communicate with a USB device, it submits a request to the USB Core, which in turn delivers the URB to the HCD, using the declared functions.

These transfer requests are stored in linked lists, managed by the HCD. When the host controller is ready to process new requests, an interrupt is asserted (either a SOFint or ATLint, depending on the configuration). As soon as an interrupt appears, the *Interrupt Handler* (also known as the Interrupt Service Routine) is called to service the interrupt. Through the interrupt handler, the driver translates the pending URBs to data accepted by the ISP1160 host controller (i.e. PTDs) and moves the data to the host controller's internal FIFO buffer RAM. When the host controller is done with the processing, a new interrupt is asserted to mark the arrival of new data. Again, the interrupt handler is called and the data are read back in system memory. After the new data are translated back to URBs, they are delivered to USB Core and in turn to the application that requested the transaction.

Before proceeding in the analysis of traffic scheduling and management, an introduction to the ISP1160 HCD interrupt handler is in order.

5.3.1 The Interrupt Handler

The interrupt handler is the center of interest for the ISP1160 HCD. Apart from initialization and cleanup, almost every function that gets called is connected to the interrupt handler. This is because the ISP1160 host controller is based on *interrupt driven I/O*. The HC uses interrupts to communicate with the driver and synchronize the transmission of data.

The interrupt handler of the ISP1160 HCD is a rather simple function on its own. Upon entering the function, the first thing the driver does is to get a hold of the spinlock. This is crucial to avoid race conditions, since we don't know the context of the module that was interrupted to call the handler. Next, the driver checks the status of the HcμPInterrupt register to see if this group of interrupts has any events pending (see also Section 4.2.6, on page 58). If the "OPR" bit is set, the driver checks the status of the HcInterruptStatus register as well, for the second group of interrupts.

Since our device is a PCI one, we must make sure that the interrupt handler can share its interrupt channel with other devices. This is accomplished with a simple method: upon entering the interrupt handler, the driver reads the contents of the HcμPInterrupt register. If there is no interrupt event (i.e. there was an interrupt on the line but it is *not* for this device), the handler returns immediately with a value of *IRQ_NONE*.

5.3.2 Root Hub Status Change and the "KHUBD"

Changes to the Root Hub include connection of new devices, port powering, detection of over-current situations, etc. On occurrence of any of these events an interrupt is asserted and the RHSC bit of the HcInterruptStatus register is set. This marks a change in the Root Hub registers (refer to C.3, on page 137). The interrupt handler clears the interrupt and updates the in-memory copies of the root hub registers. This way, it is assured that the copies of the registers in system memory will always reflect the correct values.

The real work that needs to be done for a status change in the Root Hub is left to a kernel thread called "KHUBD". As illustrated in Section 3.4.1, on page 44, a data structure named "hc_driver" registers several functions of the driver with the Linux kernel. Among these functions, the ones that are of interest for the manipulation of the Root Hub, are *hub_status_data()* and *hub_control()*.

The basic idea behind "KHUBD" is that all root hubs are compliant with the USB specifications and as such, they provide the same functionality. If we can describe the way that our hardware implements this functionality, we can leave the rest to an automated process that can manage all root hubs on the system.

isp116x_hub_status_data()

This function reports any changes in the root hub. It does so by looking into the in-memory copies of the internal root hub registers of the host controller. One of the arguments for this function is a pointer to a buffer. This must be filled, as dictated by KHUBD, and reflects any changes in the root hub. The function returns 1 if a status change was found, negative number on error, 0 otherwise.

This function is called by KHUBD. For this mechanism to work, the driver must use the "struct usb_hcd" HCD device framework. Inside this data structure lies a kernel timer (see [9, Chapter 6, page 200]) that the driver must initialize. Using this timer, the kernel polls the root hub for changes, by calling this function. If a change is reported, KHUBD proceeds to call the "hub_control" function of the driver.

Note: Obviously it is important to make sure that the in-memory copies of the root hub registers are always up to date. But it is worth the extra code, since otherwise every invocation of the timer of KHUBD would require I/O with the host controller. This way, we can take full advantage of the RHSC interrupt and read the registers only when there is a change in their values.

isp116x_hub_control()

This function implements the standard commands that the root hub must support according to the USB specifications. Upon entering this function, KHUBD already knows the reason for the status change on the hub and the course of action (a series of standard commands) that should be taken.

For example, upon connecting a new USB device to the second downstream port, the HC will change the CSC bit of the HcRhPort-Status(2) register, to reflect this change, and will also set the RHSC bit of the HcInterruptStatus register, asserting an interrupt. The kernel acknowledges the interrupt and invokes the interrupt handler that is connected to this interrupt line. The interrupt handler clears the RHSC bit and updates the values of the in-memory copies of the root hub registers. Later, KHUBD will poll the hub for changes by reading the copies of the root hub registers and will find out that there is a new device connected on downstream port 2. KHUBD will clear the change from the root hub registers and will update the in-memory copies to reflect this change (this is instructed by the isp116x_hub_control() function). The course of action from this point is irrelevant to the driver since all the transactions to register the new device are the responsibility of KHUBD and the USB Core (but they will both use the HCD to communicate with the new USB device, using control transfers).

5.3.3 Data Transfer Interrupts: ATLint & SOFint

If the interrupt handler finds out that the cause of the interrupt was either ATLint or SOFint bit, it proceeds to transmit/receive data to/the host controller. The course of action is determined by a read of the HcBufferStatus register. If the bit ATLBufferDone is set, the HC has finished processing the data and the HCD may read them back. If ATLBufferFull is not set, the the buffer is empty and the HCD can send new data to the HC.

The driver uses the ATLint when there are no INT transfers pending, otherwise it switches to SOFint to be able to poll the INT endpoints periodically.

The whole process of managing, scheduling and transmitting USB data is examined in the next sections.

5.3.4 Arrival of new URBs

In this section we examine the action that is taken by the driver when a new URB arrives, in order to schedule it. This is done asynchronously and does not require the invocation of the interrupt handler, but it should prepare the data for transmission when the interrupt handler is called.

Once again, the "hc_driver" data structure is the key to the organization of the process. The "urb_enqueue" hook, is the function responsible for accepting newly-arrived URBs. ISP1160 HCD declares the "isp116x_urb_enqueue()" function. Its purpose is to receive the URBs from the USB Core and schedule them for transmission.

Since the driver doesn't support isochronous transfers, if the URB is an isochronous one the HCD signals an error and drops the packet. Otherwise, if this is the first request to the given USB device endpoint, the driver initializes a new "struct isp116x_ep" to keep track of the events specific to the endpoint.

After the correct endpoint is found, the driver checks the type of the request (CONTROL, BULK or INTERRUPT). If this the first bulk transfer pending in the "schedule" list of the endpoint, the whole list is added to the "async" list of the "isp116x" data structure. Otherwise, the request is added to the end of the "schedule" list. If this is an interrupt transfer, the correct branch for the interrupt polling interval (see Section 5.3.5) is determined. If there is no other interrupt transfer (meaning that the driver until now was using ATLint interrupts), the driver switches to SOFint to be able to service periodically the interrupt endpoint. Finally, the request is added to the queue of pending requests.

5.3.5 Interrupt Polling Rate

With interrupt URBs submitted, the driver works with SOF interrupt enabled and ATL interrupt disabled. After the PTDs are written to FIFO RAM, the chip starts FIFO processing and USB transfers after the next SOF and continues until the transfers are finished (succeeded or failed) or the frame ends. Therefore, the transfers occur only in every second frame, while FIFO reading/writing and data processing occur in every other second frame. With the frame duration set to 1 ms, this results in a minimum 2 ms polling interval.

The driver keeps a 32-entry array. Each entry (branch) represents a list of interrupt endpoints. Every second frame, the driver services one list of endpoints. Therefore we can schedule an interrupt endpoint to be serviced with an interval ranging from 2 ms to 64 ms. This is accomplished by submitting the endpoint to multiple lists. For example, if the URB requests a polling interval of 8 ms, the driver will submit the request to every 4th list in the array. This is illustrated in Figure 5.1.

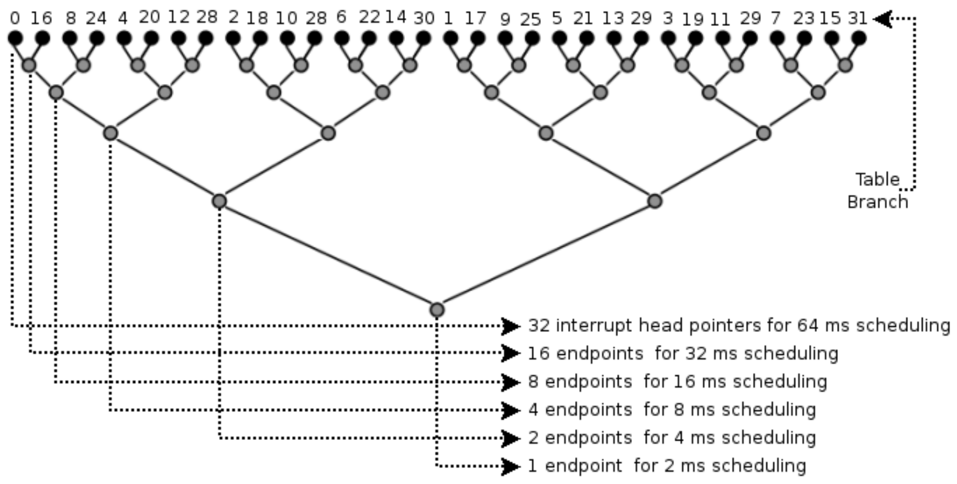

0 16 8 24 4 20 12 28 2 18 10 28 6 22 14 30 1 17 9 25 5 21 13 29 3 19 11 29 7 23 15 31
Table Branch
32 interrupt head pointers for 64 ms scheduling
16 endpoints for 32 ms scheduling
8 endpoints for 16 ms scheduling
4 endpoints for 8 ms scheduling
2 endpoints for 4 ms scheduling
1 endpoint for 2 ms scheduling

Figure 5.1: Interrupt Polling Intervals

5.3.6 Data Transfer to HC's Memory

When the host controller is ready to accept new data, an interrupt is asserted and the interrupt handler starts any pending transfers.

Transfer List Management

The first thing to check is whether the FIFO memory is empty. This is accomplished by checking the status of ATLBufferFull bit of the HcBuffer-Status register. If the memory is empty the driver proceeds to initialize the list of active transfers for the current frame, and the counters for the length of the transfers.

Next all INT transfers that are due for this frame are brought to the "active" queue. After all INT transfers are scheduled, the drivers scans the list of asynchronous transfers (Bulk and Control). If a pending transfer is found and there is enough time in the current frame, the transfer is added to the end of the "active" list.

If no transfer is brought to the "active" queue, the function returns and no further processing occurs for the current frame.

Data Organization in PTDs

All the members of the "active" queue are due for transfer in the current frame. However, the communication between the HC and the HCD must be in the form of *Philips Transfer Descriptors* (PTDs). All the transfer requests must be translated to PTDs before the actual transaction occurs.

To accomplish this, the drivers scans the "active" queue and fills a PTD for each entry. The values for the various fields of the PTD are extracted from the URB structure, with the help of the functions provided by the USB Core.

PTD Transfer to Host Controller

After all pending transfers have their own PTD, the driver rescans the list and begins transmission of data. This is accomplished in the following sequence, for each entry in the list:

1. The driver writes the size of the transfer to HcTransferCounter register (see C.4.3,on page 150).
2. The driver writes the address of the ATL port (0xC1 for write) to the command port of the host controller.
3. The driver sends the header of the PTD followed by its payload data to the ATL port (see C.6.7,on page 156).

When all data have been sent, the function returns and the HC is left to analyze the newly-arrived data and perform the actual transmission on the USB Bus.

5.3.7 Data Transfer from HC's Memory

When the host controller has finished the pending transfers, an interrupt is asserted and the interrupt handler gets the results from the HC's memory.

The first thing to check is whether the FIFO memory is done. This is accomplished by checking the status of ATLBufferDone bit of the HcBufferStatus register. If indeed the HC has finished all transfer requests, the driver proceeds to load the results to system RAM. This is accomplished in the following steps:

1. The driver writes the size of the transfer to HcTransferCounter register.
2. The driver writes the address of the ATL port (0x41 for read) to the command port of the host controller.
3. The driver receives the header of the PTD followed by its payload data from the ATL port.

Next, the driver reads back the information contained inside the PTDs and reforms a URB for each transfer completed. The driver also reads the "completion code" of each PTD and keeps track of errors and transfers needing retransmission.

After the URBs are recreated, containing the results for each transfer request, the driver calls *usb_hcd_giveback_urb()* (refer to 3.4.2, on page 49) to deliver the URB to the USB Core.

5.4 Results

The resulting device driver was tested and works with a custom-built 2.6.10 version of the Linux kernel, for *x86* architectures.

5.4.1 Insertion & Removal of the Driver

The driver is written as a module for the Linux 2.6 kernel. The new way of writing modules allows the same code to be compiled either as part of the kernel, or as module. If the source code is added to the kernel source tree (under the *drivers/usb/host* directory), and the respective *Makefile* and *Kconfig* files are modified to include the module, the driver can be loaded using the "modprobe" interface, or even by an automatic hardware detection system, such as the "discover" application. Otherwise, the driver can be loaded and unloaded with the standard "insmod" and "rmmod" commands. There is a module parameter that controls they way that the downstream ports of the host controller are powered. This is passed as an argument to the command that will load the module.

Here's what happens when the driver module is inserted to the running kernel:

```
driver isp116x-hcd, Apr  4 2005
ACPI: PCI interrupt 0000:00:0a.0[A] ->
                          GSI 5 (level, low) -> IRQ 5
isp116x-hcd: PCI region 0 (start: df101000, length: 256),
                          successfully requested
isp116x-hcd: PCI region 2 (start: e000, length: 256),
                          successfully requested
isp116x-hcd: PCI device 10b5:5406
                          (PLX Technology, Inc.), irq 5
isp116x-hcd: new USB bus registered,
                          assigned bus number 3
isp116x-hcd: Power root hub - Waiting 50 msec...
isp116x-hcd: Root hub operational
```

```
isp116x-hcd: Root hub power: global switching
usb.agent[3404]:      usbcore: already loaded
hub 3-0:1.0: USB hub found
hub 3-0:1.0: 2 ports detected
```

Notice that if the module parameter is omitted, the driver defaults to global power switching.

Here's the output of the "lsmod" command, showing the loaded modules (only the USB-specific modules are shown):

```
Module       Size  Used by
usb_storage 32384  1
isp1160_hcd 21508  0
uhci_hcd    33808  0
usbcore    121208  4 usb_storage,isp1160_hcd,uhci_hcd
sd_mod      16912  2
scsi_mod   129408  2 usb_storage,sd_mod
```

As shown, the "usbcore" module is used by both host controllers, while the "usb_storage" and SCSI modules are loaded to support a USB flash disk. Notice the smaller size of the ISP1160 driver (21508 Bytes, vs. 33808 Bytes for the UHCI driver).

Here's what happens when the driver module is removed from the kernel, while a device is attached on the root hub:

```
usb 3-1: USB disconnect, address 2
isp116x-hcd 0000:00:0a.0: remove, state 1
usb usb3: USB disconnect, address 1
isp116x-hcd 0000:00:0a.0: USB bus 3 deregistered
```

5.4.2 Hardware Functionality

The driver supports three USB transfer modes:

Control transfers are required, to be able to communicate with the USB devices.

Bulk transfers are used to communicate with USB devices such as printers and memory sticks.

Interrupt transfers are used for synchronous communication with USB devices such as HID devices (keyboards, mice etc.)

The driver does not support the *Isochronous* transfer mode,which is used for streaming media USB devices (e.g. USB speakers). If such a device is connected to the host controller, the driver will display an error message, and all further communication with the device will be silently dropped.

The HCD performs accesses to the internal registers of the ISP1160 HC by means of PIO. The same is true for the HC's memory. DMA access to the memory was not implemented.

5.4.3 Driver Policy

As explained in Section 1.2.1, on page 5, the driver tries not to implement any "mechanism". In other words, the driver just makes the hardware available, without imposing restrictions to the way the hardware will be used. This was accomplished by using only the functions provided by the kernel to register the driver as an HCD. This is also the way that the other HCDs of Linux work (EHCI, UHCI and OHCI).

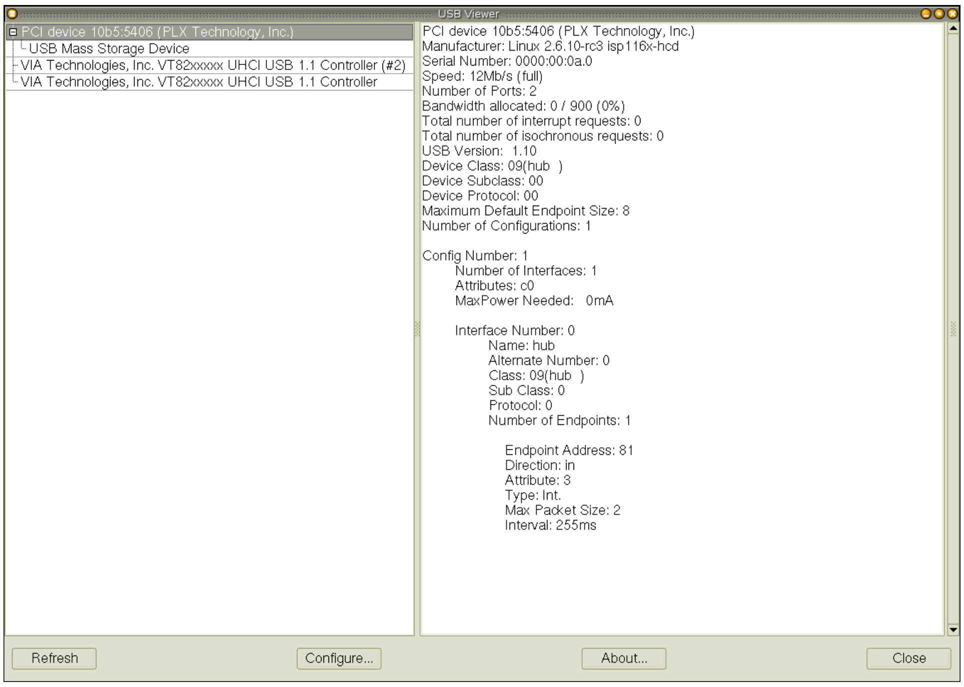
Another important point is the implementation of the Root Hub. By choosing to use the supplied kernel functions, we surrender the control of the root hub to the kernel. This way, any event in the root hub triggers a standard response from the operating system.

5.4.4 OS Integration

The host controller driver integrates very well with the operating system. When the module is loaded, the kernel automatically adds the driver to the "sysfs" pseudo-filesystem. The *sysfs* is the successor of the *proc* filesystem. It is mounted under */sys* and provides a snapshot of the system, including loaded drivers and the hardware they control, reserved resources and information on the hardware present. In the case of the ISP1160 USB Host Controller, we can also read the contents of the various descriptors, either directly or use an application that takes advantage of the sysfs.

In Figure 5.2, we can see the "USBView" application running: on the left side of the screen are listed the USB devices present in the system (two UHCI host controllers, plus the ISP1160 with a USB flash disk attached), and on the right side we can read information about the selected device (the ISP1160 PCI development board).

When a new device is attached to the host controller, the kernel automatically enumerates the device without any effort from the part of the HCD, thanks to the KHUBD daemon (refer to Section 5.3.2, on page 88). The kernel detects the class of the attached device (e.g. a USB printer) and automatically loads (if not already loaded) the appropriate driver to handle the new device. Furthermore, the new device is added to the sysfs, as connected under the ISP1160



```
PCI device 10b5:5406 (PLX Technology, Inc.)
  └ USB Mass Storage Device
├ VIA Technologies, Inc. VT82xxxxx UHCI USB 1.1 Controller (#2)
└ VIA Technologies, Inc. VT82xxxxx UHCI USB 1.1 Controller
```

```
PCI device 10b5:5406 (PLX Technology, Inc.)
Manufacturer: Linux 2.6.10-rc3 isp116x-hcd
Serial Number: 0000:00:0a.0
Speed: 12Mb/s (full)
Number of Ports: 2
Bandwidth allocated: 0 / 900 (0%)
Total number of interrupt requests: 0
Total number of isochronous requests: 0
USB Version:  1.10
Device Class: 09(hub  )
Device Subclass: 00
Device Protocol: 00
Maximum Default Endpoint Size: 8
Number of Configurations: 1

Config Number: 1
        Number of Interfaces: 1
        Attributes: c0
        MaxPower Needed:   0mA

        Interface Number: 0
                Name: hub
                Alternate Number: 0
                Class: 09(hub  )
                Sub Class: 0
                Protocol: 0
                Number of Endpoints: 1

                    Endpoint Address: 81
                    Direction: in
                    Attribute: 3
                    Type: Int.
                    Max Packet Size: 2
                    Interval: 255ms
```

Figure 5.2: “USBView” Application Running

host controller. Upon detection of a device removal, if the driver is no longer needed (e.g. by another device of the same class), the operating system unloads the driver and removes the sysfs entry of the device.

Here’s what happens when a USB device is attached to the host controller:

```
usb 3-1: new full speed USB device using isp116x-hcd
         and address 2
Initializing USB Mass Storage driver...
usb.agent[3590]:      usb-storage: loaded successfully
scsi0 : SCSI emulation for USB Mass Storage devices
usbcore: registered new driver usb-storage
USB Mass Storage support registered.
usb-storage: device found at 2
usb-storage: waiting for device to settle before scan

   Vendor: WinFast Model: Disk 128MB Rev:
   Type:   Direct-Access        ANSI SCSI revision: 02

SCSI device sda: 253952 512-byte hdwr sectors (130 MB)
sda: Write Protect is off
```

```
sda: Mode Sense: 03 00 00 00
sda: assuming drive cache: write through
sda: sda1
Attached scsi removable disk sda at
         scsi0, channel 0, id 0, lun 0
usb-storage: device scan complete
scsi.agent[3710]: disk at /devices/pci0000:00/
                           0000:00:0a.0/usb3/
                           3-1/3-1:1.0/host0/
                           target0:0:0/0:0:0:0
```

Notice the way that the operating system is able to understand the "class" of the device (USB Mass Storage) and load the appropriate driver (usb-storage). The USB flash disk is managed by "SCSI" emulation. It is interesting to note that although the ISP1160 is used to communicate with the USB device, none of the above is part of the ISP1160 driver, thanks to the way this driver is written (i.e. to take advantage of the Linux USB subsystem).

5.4.5 Security

Since the driver is created as a standard module of the Linux kernel, we don't need any special security precautions for loading and unloading the driver. The tools used for that purpose ("insmod" and "modprobe"), if configured in the proper way[2], provide the required security mechanisms.

The way that the driver registers itself with the kernel us allows to trust input from user space programs. This is due to the fact that the Linux USB subsystem (refer to Chapter 3, on page 33) sits in between the HCD and the applications, functioning as a filter for the transmitted data. This system already contains security mechanisms that check the received data before propagating them to the HCD, eliminating the need for security mechanisms inside the HCD.

Furthermore, the driver strives to make sure that all new data structures are initialised before being used, and that all data written to buffers will not exceed the allocated buffer sizes.

5.4.6 Driver Porting

The source code file "isp1160_hcd-pci.c" contains all PCI-specific code that is needed for the initialisation of the PCI development board. All

[2]most Linux distributions come with these tools already configured

the other files refer to the ISP1160 HC and can be reused in other implementations of the host controller. In order to successfully port the driver, one needs only modify the initialisation code to fit his hardware, and make sure that after the initialisation, the same resources will be available to the driver (i.e. I/O ports and interrupt channel).

There are some implementations of the ISP1160 that use memory-mapped regions insted of I/O ports. In this case, the driver writer must also alter the contents of the "isp1160_hcd-pio.c" file, where the functions that access the hardware are declared using the naming scheme *isp116x_(read,write)_reg(16,32)*.

5.5 Testing & Performance

5.5.1 Testing

The driver's stability was tested using a variety of devices. All supported transfer modes were tested as follows:

- *Control* transfers were guaranteed since the host controller is always able to detect the attachment of a new USB device and enumerate it. They are also used in all the subsequent tests.
- To test the stability of *Interrupt* transfers, we connected a USB mouse and used it as the system's mouse (in a graphical user interface), for about 40 hours.
- Two USB printers[3] were connected simultaneously to the ISP1160 to test *Bulk* transfers. Files were sent to both printers at once(both text and graphics), and they were successfully printed. Furthermore, a USB flash disk[4] was attached and it was used to transfer more than 3 GBytes of data.

Due to the fact that the above tests were performed during a 2 month period, we cannot state the exact number of times or hours we performed them. A very good sign though is that during this period, all transfers were always successful. There is still however, a stability problem, related to the unregistration of the driver when the module is removed from a running kernel, especially if this happens due to a loading error. If this happens, all subsequent attempts to reload the module to the kernel will fail, and a reset of the operating system will be required. This is a typical situation when a driver fails

[3]Hewlett Packard 840C and 930C
[4]Leadtek Winfast 128MB

to free the allocated resources (I/O ports, interrupts and memory), since they will still be in use when the module is reloaded.

A variety of USB devices were used to test the driver. These are included in the following list:

- *Flash Disk*: Leadtek WinFast 128MB, USB 2.0
- *Mouse*: Logitech, N48/M-BB48 (FirstMouse Plus)
- *Printer*: Hewlett-Packard DeskJet 840C
- *Printer*: Hewlett-Packard DeskJet 930C
- *Digital Camera*: Olympus Camedia C-40 Zoom

5.5.2 Performance Evaluation

The performance of the driver is hard to measure, since the only means is through the use of a USB flash disk. But, while performing timed read or write on the flash disk, we also evaluate the performance of the operating system, memory and caches, the USB host controller and of course the USB flash disk. We can disable the operating system's role by making sure that we always synchronise the data between the system's memory and the storage location. Furthermore, we have no official performance statistics from the manufacturer of the ISP1160 (apart from the typical 12 Mbits/sec) to compare our results. Thus, the best solution was to use the same system to compare the performance of the ISP1160 and the system's USB controller, using the same USB flash disk. This way, the only parts that are different between the tests, are the host controller and its driver.

For this purpose we wrote a script, which measures the time required to write/read a 2 MB file to the USB flash disk, using record sizes ranging from 4 KB, up to 2000 KB. The same procedure is repeated for a 4 MB file, and then for 8, 16, 32 and 64 MB files (each time with the record size limit set to the file's size).

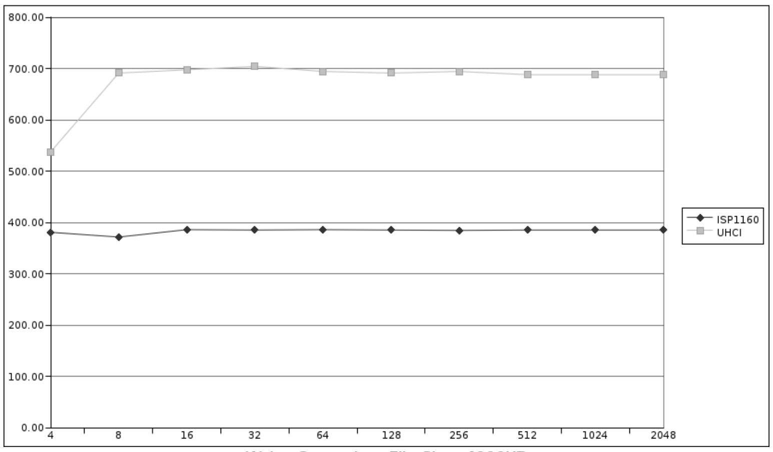
The above test script was performed on the following USB controllers:

- VIA VT82xxxxx UHCI USB 1.1 Controller, using the "uhci-hcd" linux driver
- PHILIPS ISP1160 PCI development board, using the "isp116x-hcd" driver.

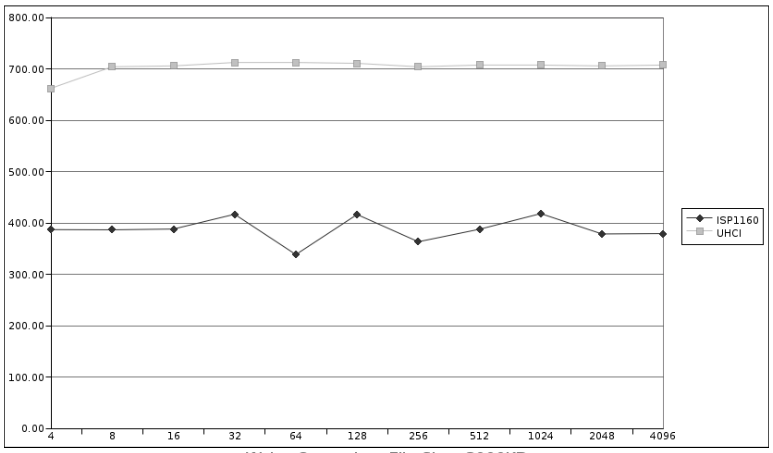
The results of the performance evaluation are summarised in the following charts.

Write Operation Performance

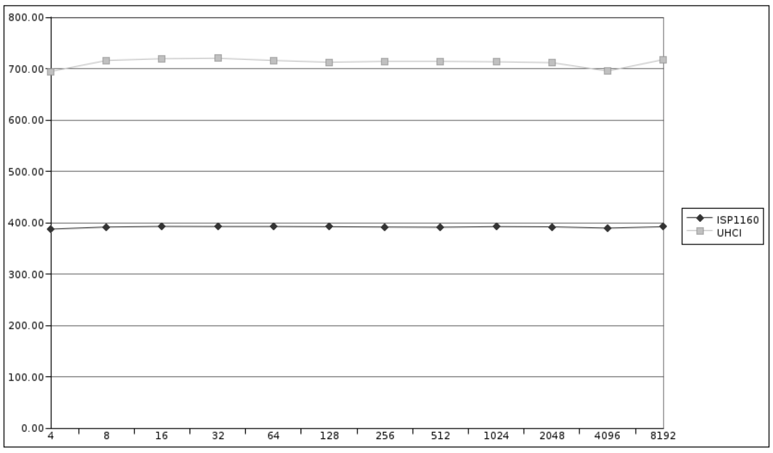
Write Operation, File Size: 2000KB

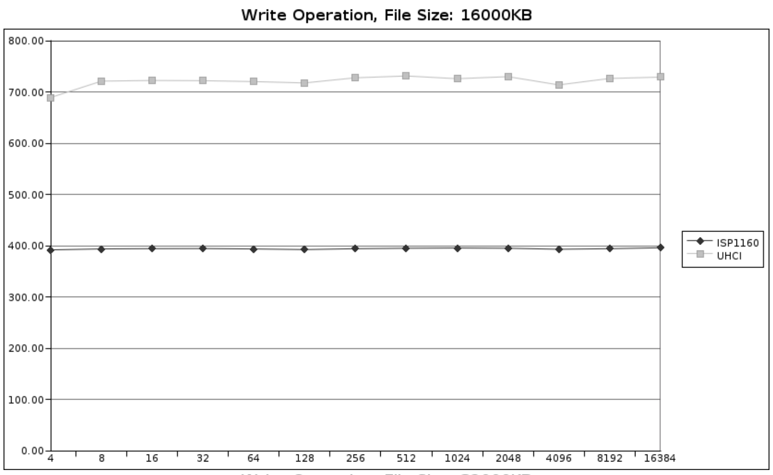


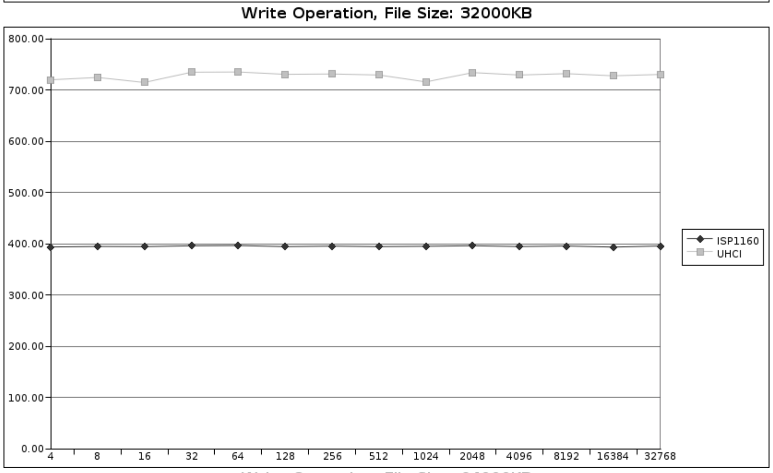
Write Operation, File Size: 4000KB

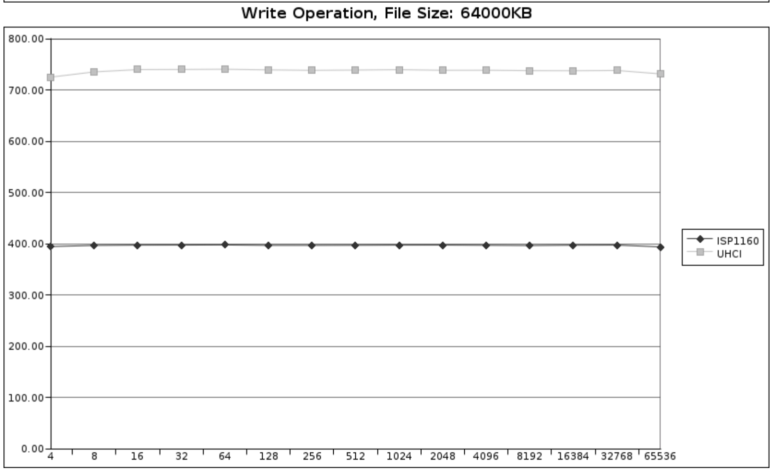


Write Operation, File Size: 8000KB









As shown in the above charts, the performance of the ISP1160 is lower than that of the UHCI host controller. Some reasons behind this fact might be:

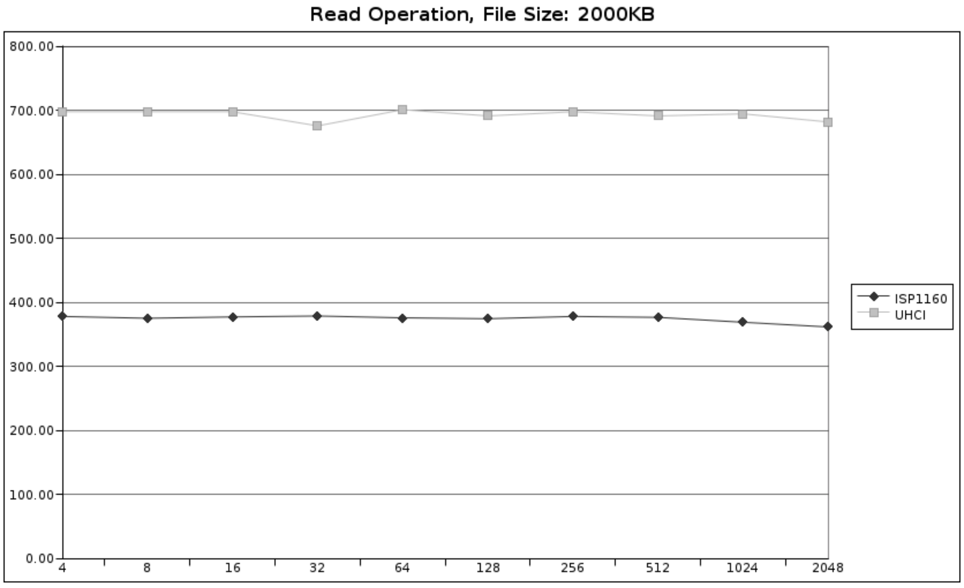
- The UHCI host controller is directly connected to the *southbridge* of the motherboard's chipset, while every external PCI device (including our host controller) must first pass through a PCI bridge.

- The driver makes no use of the DMA capabilities provided. If DMA was used, the transfer of data from/to the host controller would be faster.

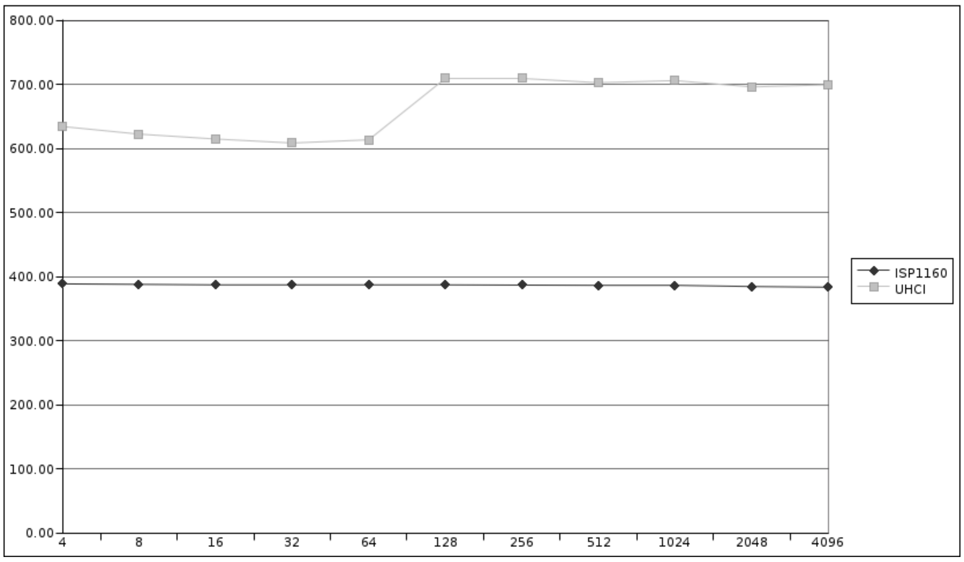
- The UHCI driver has been in the Linux kernel for many years (written by Linus Torvalds in 1999), and it has undergone many changes that improved the performance of the initial source code, while the ISP1160 driver can be still considered "under development".

- *This is just an assumption:* The ISP1160 is a USB host controller targeting the embedded market. As such, it might not perform as fast as other USB host controllers. However, the ISP1160 offers energy saving, a very important factor in the design of embedded systems.

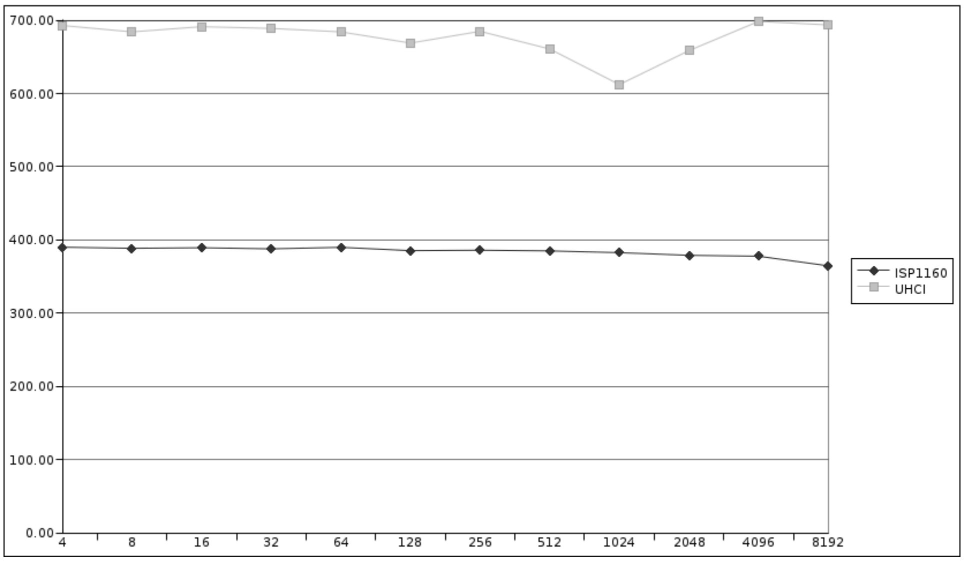
Read Operation Performance



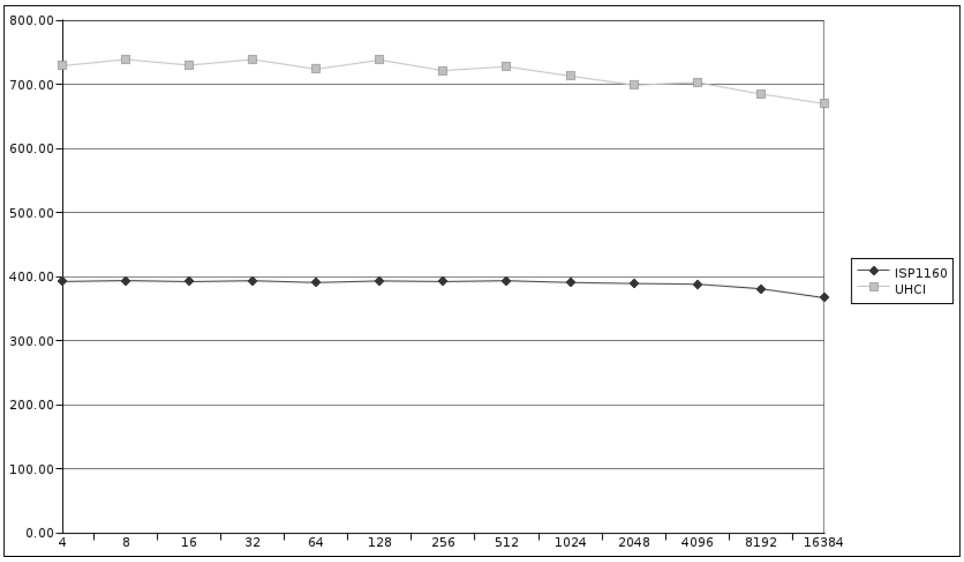
Read Operation, File Size: 4000KB



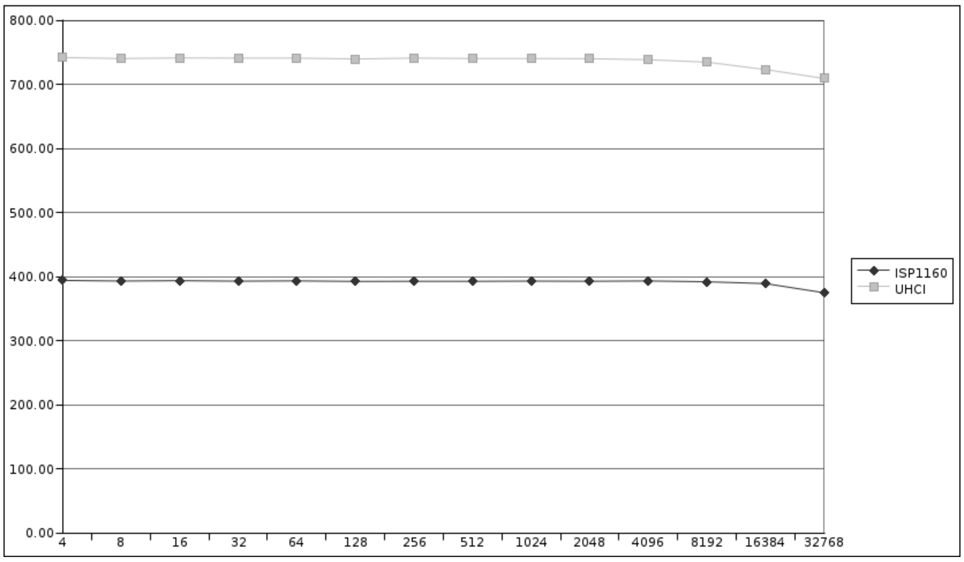
Read Operation, File Size: 8000KB

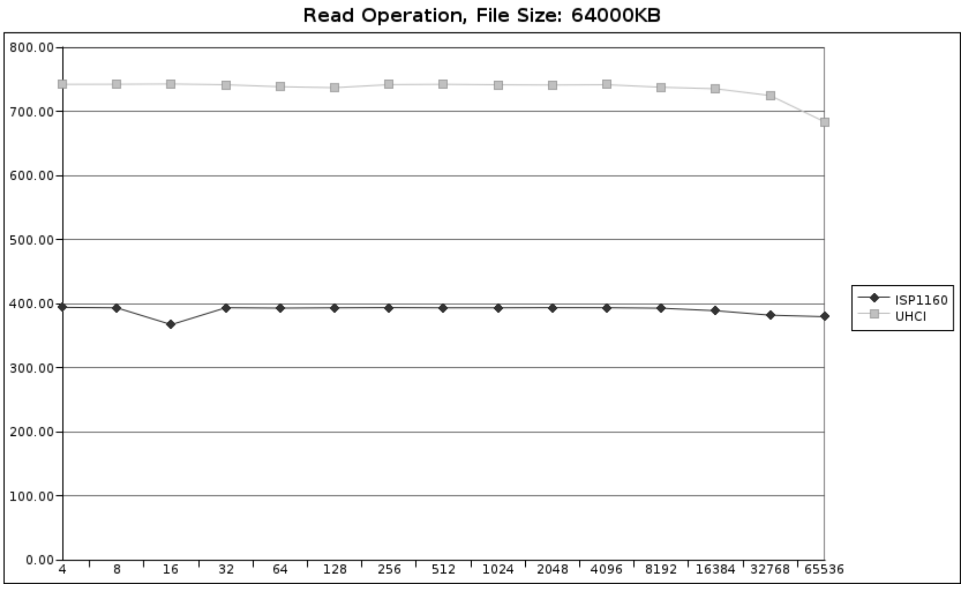


Read Operation, File Size: 16000KB



Read Operation, File Size: 32000KB





As shown in the above charts, the overall reading performance of the ISP1160 is similar to the one observed in the writing tests, so we can assume that the same reasons apply also here. As a result, we can say that the ISP1160 HCD performs well, and a future addition of DMA support promises better performance.

Chapter 6

Conclusions

The purpose behind this thesis was to understand and demonstrate the right way of writing a device driver under the Linux operating system. It is very important to explain not only what should be done, but what should NOT be done as well, when undertaking such a task.

The choice of implementing a host controller driver, was not a simple one. The programmer needs to focus on multiple subjects (e.g. device probing, handling of attached devices, scheduling of data transfers). A big mistake that I made from the early steps of the development was that, as soon as I had understood the basics, I tried to solve the whole problem at once. After many months of hard work, I found myself again right in the beginning, since I had created a full non-working driver and I was not sure what part of it was causing me the trouble. I had not taken the time to test whether the hardware was fully functional (I had only tested the I/O ports), so I was not even sure whether I had faulty hardware or not.

The correct attitude would have been (and eventually was. . .) to read and understand the basics, then begin testing the hardware, combining step by step all conquered knowledge and facts into more functional units. For example, after testing the I/O ports, we are ready to send the first commands to the host controller. With the correct sequence of commands, we can bring the controller to the USB Operational mode, thus triggering the transmission of interrupt signals to mark the beginning of every new frame. This is the correct time to test whether the controller is indeed generating pulses through the INT pin. This last action was considered as a possibility after several months of work, and I was shocked to realize that the operating system was not acknowledging any interrupts from the PCI board. Using a logical analyser on the development board, I monitored the interrupt pin and concluded that the ISP1160 was indeed generating interrupts (i.e. it had successfully entered the operational mode), but these signals were blocked somewhere on the board, thus never reaching the system's interrupt controller. The research that followed uncovered all the information about the PLX9054 chip that is present in Section 4.4, and of course, the fact that the assertion of local interrupts was disabled on the development board.

While in the beginning of my work, one of the first problems that I encountered was to find the location of the command and data ports of the ISP1160. The solution came one day, when I took a closer look at the DOS driver that came along with the host controller. It had not occurred to me before that the specifications of the PCI bus are independent of the system's architecture and operating system. Inside the DOS driver, I found the part where the resources were allocated, and it was just a matter of translating it.

Note however that both this problem and the previous one, are tightly connected to the fact that the development board was not accompanied by any manual at all. The absence of the manual led me to read the PCI bus specifications and the PLX9054 manual, and I was overwhelmed by the size of the new field that I was forced to enter. So, another advice is to be sure you have access to all available manuals and support, before you begin working.

As a last piece of advice, two personal thoughts on the subject, fruit of countless hours of despair:

1. *When in doubt whether the problem is caused by faulty hardware or not, it is usually your fault.*

2. *When in doubt whether it is the kernel's fault or not (after all, open source comes with "absolutely no warranty", right?), it is* always *your fault...*

6.1 Further Work

As mentioned before, the driver implemented for the PCI development board can easily be converted to serve another device that is based on the ISP1160, covering the basic functionality of the hardware. There is however a list of tasks that still remain to be done, if this driver is to be considered "fully working":

- All stability issues related to module loading/unloading should be solved. This needs closer examination from an experienced driver programmer. The problem is that this issue is related to the PCI initialisation and cleanup code, and I was not able to find anyone else working with the same device (i.e. the PCI development board). However, when the driver was tested under the Linux kernel 2.6.8, it didn't show any of the above signs, so we might say that this is a minor incompatibility issue of the PCI development board and the Linux kernel 2.6.10. Thus, as a first step, it is advised to wait for the next kernel release and reevaluate the situation.

- The driver needs to support Isochronous transfers. This requires a good deal of work and a lot of testing. The problem lies in the fact that there exist few USB devices that support this kind of data transfer, and it is unlikely that you will be having one at hand (as opposed to USB mice, flash disks and printers).
- The ISP1160 is an embedded USB host controller, focusing on low-power consumption. This driver should support this feature by implementing the *suspend* and *resume* features.
- Direct Memory Access to the internal RAM of the host controller, if implemented, will increase the performance of the device, and at the same time demand less time from the system's CPU.
- All data transfers to and from the host controller, are triggered by an interrupt and served by the Interrupt Service Routine. I suspect that we can boost the driver's performance if the ISR is rewritten with the help of *tasklets*. This will split the ISR into the "upper" and "bottom" halves. The first half of the ISR will be executed "in interrupt" and will just acknowledge the reason of the interrupt, scheduling the real work as a tasklet to be latter executed. This way we can minimize the time spent on running the ISR, thus reducing the time that all the other driver's functions are blocked waiting for the ISR to release the spinlock.
- The Linux kernel already maintains a set of memory pools of objects that are all the same size (named "Lookaside Caches"). The kernel allows us to create our own memory pools. It would probably be a good idea to create this sort of pool for the PTDs and minimize the time required to allocate them, every time they are needed (i.e. for every data transfer on the USB bus). A side benefit to using lookaside caches, is that the kernel maintains statistics on cache usage.
- The actual driver for the ISP1160 is only half the work of this thesis. The other half was about understanding and learning the general method of writing Linux device drivers. The gained knowledge can be applied to many other kernel-related issues. This thesis presents the opportunity to create a group of undergraduate students, supervised by postgraduates and professors of

the E.C.E. department, that will contribute on the evolution of open source kernels (like Linux, BSD and GNU Hurd). This initiative can offer invaluable knowledge to the students in the fields of hardware and system software, strengthen the bonds between the hardware and software laboratories of the department, and promote the image of the university in the scientific community.

Appendix A

USB Requests

A.1 USB Device Requests

All USB devices respond to requests from the host on the device's Default Control Pipe. These requests are made using control transfers. The request and the request's parameters are sent to the device in the Setup packet. The host is responsible for establishing the values passed in the fields listed in Table A.1. Every Setup packet has eight bytes.

bmRequestType This bitmapped field identifies the characteristics of the specific request. In particular, this field identifies the direction of data transfer. The state of the Direction bit is ignored if the wLength field is zero, signifying there is no Data stage. The USB specification defines a series of standard requests that all devices must support. These are enumerated in Table A.2. In addition, a device class may define additional requests. A device vendor may also define requests supported by the device. Requests may be directed to the device, an interface on the device, or a specific endpoint on a device. This field also specifies the intended recipient of the request. When an interface or endpoint is specified, the wIndex field identifies the interface or endpoint.

bRequest This field specifies the particular request. The Type bits in the bmRequestType field modify the meaning of this field. This specification defines values for the bRequest field only when the bits are reset to zero, indicating a standard request (refer to Table A.2).

Table A.1: Format of Setup Data

Offset	**Field**	**Size**	**Value**	**Description**
0	bmRequestType	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host to device 1 = Device to host D6. . . 5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4. . . 0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4. . . 31 = Reserved
1	bRequest	1	Value	Specific request (refer to Table A.2)
2	wValue	2	Value	Word-sized field that varies according to request
4	wIndex	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	wLength	2	Count	Number of bytes to transfer if there is a Data stage

wValue & wIndex The contents of these fields vary according to the request. They are used to pass parameters to the device, specific to the request.

wLength This field specifies the length of the data transferred. The direction of data transfer (host-to-device or device-to-host) is indicated by the Direction bit of the bmRequestType field. If this field is zero, there is no data transfer phase. On an input request, a device must never return more data than is indicated by the wLength value; it may return less. On an output request, wLength will always indicate the exact amount of data to be sent by the host. Device behavior is undefined if the host should send more data than is specified in wLength.

A.2 Standard Device Requests

This section describes the standard device requests defined for all USB devices. Table A.2 outlines the standard device requests, while Table A.3 and Table A.4 give the standard request codes and descriptor types, respectively.

USB devices must respond to standard device requests, even if the device has not yet been assigned an address or has not been configured.

A.2.1 Request Descriptions

Clear Feature

This request is used to clear or disable a specific feature. Feature selector values in wValue must be appropriate to the recipient. A ClearFeature() request that references a feature that cannot be cleared, that does not exist, or that references an interface or endpoint that does not exist, will cause the device to respond with a Request Error. If wLength is non-zero, then the device behavior is not specified.

Get Configuration

This request returns the current device configuration value. If the returned value is zero, the device is not configured. If wValue, wIndex, or wLength are not as specified above, then the device behavior is not specified.

Table A.2: Standard Device Requests

bmRequest Type	**bRequest**	**wValue**	**wIndex**	**wLength**	**Data**
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Config-uration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descrip-tor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Config-uration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descrip-tor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Table A.3: Standard Request Codes

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

Table A.4: Descriptor Types

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5
DEVICE_QUALIFIER	6
OTHER_SPEED_CONF	7
INTERFACE_POWER	8

Get Descriptor

This request returns the specified descriptor if the descriptor exists.

The wValue field specifies the descriptor type in the high byte (refer to Table A.4) and the descriptor index in the low byte. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device. For other standard descriptors that can be retrieved via a GetDescriptor() request, a descriptor index of zero must be used. The range of values used for a descriptor index is from 0 to one less than the number of descriptors of that type implemented by the device.

The wIndex field specifies the Language ID for string descriptors or is reset to zero for other descriptors. The wLength field specifies the number of bytes to return. If the descriptor is longer than the wLength field, only the initial bytes of the descriptor are returned. If the descriptor is shorter than the wLength field, the device indicates the end of the control transfer by sending a short packet when further data is requested. A short packet is defined as a packet shorter than the maximum payload size or a zero length data packet.

All devices must provide a device descriptor and at least one configuration descriptor. If a device does not support a requested descriptor, it responds with a Request Error.

Get Interface

This request returns the selected alternate setting for the specified interface.

Some USB devices have configurations with interfaces that have mutually exclusive settings. This request allows the host to determine the currently selected alternate setting. If wValue or wLength are not as specified above, then the device behavior is not specified. If the interface specified does not exist, then the device responds with a Request Error.

Get Status

This request returns status for the specified recipient. The Recipient bits of the bmRequestType field specify the desired recipient. The data returned is the current status of the specified recipient. If wValue or wLength are not as specified above, or if wIndex is non-zero for a device status request, then the behavior of the device is not specified. If an interface or an endpoint is specified that does not exist, then the device responds with a Request Error.

Set Address

This request sets the device address for all future device accesses. The wValue field specifies the device address to use for all subsequent accesses. If the specified device address is greater than 127, or if wIndex or wLength are non-zero, then the behavior of the device is not specified.

Set Configuration

This request sets the device configuration. The lower byte of the wValue field specifies the desired configuration. This configuration value must be zero or match a configuration value from a configuration descriptor. If the configuration value is zero, the device is placed in its Address state. The upper byte of the wValue field is reserved. If wIndex, wLength, or the upper byte of wValue is non-zero, then the behavior of this request is not specified.

If the specified configuration value matches the configuration value from a configuration descriptor, then that configuration is selected and the device remains in the Configured state. Otherwise, the device responds with a Request Error.

Set Descriptor

This request is optional and may be used to update existing descriptors or new descriptors may be added. The wValue field specifies the descriptor type in the high byte (refer to Table A.4) and the descriptor index in the low byte. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device. For other standard descriptors that can be set via a SetDescriptor() request, a descriptor index of zero must be used. The range of values used for a descriptor index is from 0 to one less than the number of descriptors of that type implemented by the device.

The wIndex field specifies the Language ID for string descriptors or is reset to zero for other descriptors. The wLength field specifies the number of bytes to transfer from the host to the device. The only allowed values for descriptor type are device, configuration, and string descriptor types. If this request is not supported, the device will respond with a Request Error.

Set Feature

This request is used to set or enable a specific feature. Feature selector values in wValue must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

A SetFeature() request that references a feature that cannot be set or that does not exist causes a STALL to be returned in the Status stage of the request. If wLength is non-zero, then the behavior of the device is not specified. If an endpoint or interface is specified that does not exist, then the device responds with a Request Error.

Set Interface

This request allows the host to select an alternate setting for the specified interface.

Some USB devices have configurations with interfaces that have mutually exclusive settings. This request allows the host to select the desired alternate setting. If a device only supports a default setting for the specified interface, then a STALL may be returned in the Status stage of the request. This request cannot be used to change the set of configured interfaces (the SetConfiguration() request must be used instead).

If the interface or the alternate setting does not exist, then the device responds with a Request Error. If wLength is nonzero, then the behavior of the device is not specified.

Synch Frame

This request is used to set and then report an endpoint's synchronization frame. When an endpoint supports isochronous transfers, the endpoint may also require per-frame transfers to vary in size according to a specific pattern. The host and the endpoint must agree on which frame the repeating pattern begins. The number of the frame in which the pattern began is returned to the host. If a high-speed device supports the Synch Frame request, it must internally synchronize itself to the zeroth microframe and have a time notion of classic frame. Only the frame number is used to synchronize and reported by the device endpoint (i.e., no microframe number). The endpoint must synchronize to the zeroth microframe.

This value is only used for isochronous data transfers using implicit pattern synchronization. If wValue is non-zero or wLength is not two,

Table A.5: Hub Responces to Standard Device Requests

bRequest	Hub Response
CLEAR_FEATURE	Standard
GET_CONFIGURATION	Standard
GET_DESCRIPTOR	Standard
GET_INTERFACE	Undefined. Hubs are allowed to support only one interface.
GET_STATUS	Standard
SET_ADDRESS	Standard
SET_CONFIGURATION	Standard
SET_DESCRIPTOR	Optional
SET_FEATURE	Standard
SET_INTERFACE	Undefined. Hubs are allowed to support only one interface.
SYNCH_FRAME	Undefined. Hubs are not allowed to have isochronous endpoints.

then the behavior of the device is not specified. If the specified endpoint does not support this request, then the device will respond with a Request Error.

A.3 Hub Requests

A.3.1 Standard Requests

All hubs respond to standard usb requests (see A.2) as outlined in Table A.5.

A.3.2 Class-Specific Requests

The hub class defines requests to which hubs respond, as outlined in Table A.6. Table A.7 defines the hub class request codes. All requests in Table A.6 except SetHubDescriptor() are mandatory.

Table A.6: Hub Class Requests

Request	bmRequestType	bRequest	wValue	wIndex	wLength	Data
ClearHubFeature	00100000B	CLEAR_FEATURE	Feature Selector	Zero	Zero	None
ClearPortFeature	00100011B	CLEAR_FEATURE	Feature Selector	Selector, Port	Zero	None

table continued on next page...

Table A.6: Hub Class Requests (continued)

Request	bmRequestType	bRequest	wValue	wIndex	wLength	Data
ClearTTBuffer	00100011B	CLEAR_TT_BUFFER	Dev_Addr, EP_Num	TT_port	Zero	None
GetHubDescriptor	10100000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
GetHubStatus	10100000B	GET_STATUS	Zero	Zero	Four	Hub Status and Change Status
GetPortStatus	10100011B	GET_STATUS	Zero	Port	Four	Port Status and Change Status
ResetTT	00100011B	RESET_TT	Zero	Port	Zero	None
SetHubDescriptor	00100000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
SetHubFeature	00100000B	SET_FEATURE	Feature Selector	Zero	Zero	None
SetPortFeature	00100011B	SET_FEATURE	Feature Selector	Selector, Port	Zero	None
GetTTState	10100011B	GET_TT_STATE	TT_Flags	Port	TT State Length	TT State
StopTT	00100011B	STOP_TT	Zero	Port	Zero	None

Table A.7: Hub Class Request Codes

Request	**Value**
GET_STATUS	0
CLEAR_FEATURE	1
RESERVED	2
SET_FEATURE	3
Reserved for future use	4-5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
CLEAR_TT_BUFFER	8
RESET_TT	9
GET_TT_STATE	10
STOP_TT	11

Appendix B

USB Descriptors

USB devices report their attributes using descriptors. A descriptor is a data structure with a defined format. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor followed by a byte-wide field that identifies the descriptor type. Using descriptors allows concise storage of the attributes of individual configurations because each configuration may reuse descriptors or portions of descriptors from other configurations that have the same characteristics. In this manner, the descriptors resemble individual data records in a relational database.

Where appropriate, descriptors contain references to string descriptors that provide displayable information describing a descriptor in human-readable form. The inclusion of string descriptors is optional. However, the reference fields within descriptors are mandatory. If a device does not support string descriptors, string reference fields must be reset to zero to indicate no string descriptor is available. If a descriptor returns with a value in its length field that is less than defined by this specification, the descriptor is invalid and should be rejected by the host. If the descriptor returns with a value in its length field that is greater than defined by this specification, the extra bytes are ignored by the host, but the next descriptor is located using the length returned rather than the length expected.

B.1 Device Descriptor

The device descriptor of a USB device represents the entire device. As a result a USB device can only have one device descriptor. It specifies some basic, yet important information about the device such as the supported USB version, maximum packet size, vendor and product IDs and the number of possible configurations the device can have. The format of the device descriptor is shown in Table B.1.

- The *bcdUSB* field reports the highest version of USB the device supports. The value is in binary coded decimal with a format of 0xJJMN where JJ is the major version number, M is the minor version number and N is the sub minor version number. e.g. USB 2.0 is reported as 0x0200, USB 1.1 as 0x0110 and USB 1.0 as 0x0100.

- The *bDeviceClass*, *bDeviceSubClass* and *bDeviceProtocol* are used by the operating system to find a class driver for the device. Typically only the bDeviceClass is set at the device level. Most class specifications choose to identify itself at the interface level and as a result set the bDeviceClass as 0x00. This allows for the one device to support multiple classes.

- The *bMaxPacketSize* field reports the maximum packet size for endpoint zero. All devices must support endpoint zero.

- The *idVendor* and *idProduct* are used by the operating system to find a driver for your device. The Vendor ID is assigned by the USB.org.

- The *bcdDevice* has the same format as the bcdUSB and is used to provide a device version number. This value is assigned by the developer.

- Three string descriptors exist to provide details of the manufacturer, product and serial number. There is no requirement to have string descriptors. If no string descriptor is present, a index of zero should be used.

- *bNumConfigurations* defines the number of configurations the device supports at its current speed.

B.2 Configuration Descriptor

A USB device can have several different configurations although the majority of devices are simple and only have one. The configuration descriptor specifies how the device is powered, what the maximum power consumption is, the number of interfaces it has. Therefore it is possible to have two configurations, one for when the device is bus powered and another when it is mains powered. As this is a "header" to the Interface descriptors, its also feasible to have one

Table B.1: Device Descriptor Format

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

Table B.2: Configuration Descriptor Format

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLenght	2	Number	Total Length of Data Returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor Describing this configuration
7	bmAttributes	1	Bitmap	D7 — Bus Powered
				D6 — Self Powered
				D5 — Remote Wakeup
				D4. . . 0 — Reserved (0)
8	bMaxPower	1	mA	Maximum Power Consumption

configuration using a different transfer mode to that of another configuration. Once all the configurations have been examined by the host, the host will send a SetConfiguration command with a non-zero value which matches the bConfigurationValue of one of the configurations. This is used to select the desired configuration. The format of the configuration descriptor is shown in Table B.2.

- When the configuration descriptor is read, it returns the entire configuration hierarchy which includes all related interface and endpoint descriptors. The *wTotalLength* field reflects the number of bytes in the hierarchy.

- *bNumInterfaces* specifies the number of interfaces present for this configuration.

- *bConfigurationValue* is used by the SetConfiguration request to select this configuration.

- *iConfiguration* is a index to a string descriptor describing the configuration in human readable form.

- *bmAttributes* specify power parameters for the configuration. If a device is self powered, it sets D6. Bit D7 was used in USB 1.0 to indicate a bus powered device, but this is now done by bMaxPower. If a device uses any power from the bus, whether

Table B.3: Interface Descriptor Format

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

it be as a bus powered device or as a self powered device, it must report its power consumption in bMaxPower. Devices can also support remote wakeup which allows the device to wake up the host when the host is in suspend.

- *bMaxPower* defines the maximum power the device will drain from the bus. This is in 2 mA units, thus a maximum of approximately 500 mA can be specified. The specification allows a high powered bus powered device to drain no more than 500 mA from Vbus. If a device loses external power, then it must not drain more than indicated in bMaxPower. It should fail any operation it cannot perform without external power.

B.3 Interface Descriptor

The interface descriptor could be seen as a header or grouping of the endpoints into a functional group performing a single feature of the device. The interface descriptor conforms to the format shown in Table B.3.

- *bInterfaceNumber* indicates the index of the interface descriptor. This should be zero based, and incremented once for each new interface descriptor.
- *bAlternativeSetting* can be used to specify alternative interfaces. These alternative interfaces can be selected with the SetInterface request.

- *bNumEndpoints* indicates the number of endpoints used by the interface. This value should exclude endpoint zero and is used to indicate the number of endpoint descriptors to follow.
- *bInterfaceClass*, *bInterfaceSubClass* and *bInterfaceProtocol* can be used to specify supported classes (e.g. HID, communications, mass storage etc.) This allows many devices to use class drivers, preventing the need to write individual drivers for every device.
- *iInterface* allows for a string description of the interface.

B.4 Endpoint Descriptor

Endpoint descriptors are used to describe endpoints other than endpoint zero. Endpoint zero is always assumed to be a control endpoint and is configured before any descriptors are even requested. The host will use the information returned from these descriptors to determine the bandwidth requirements of the bus. The format of the configuration descriptor is shown in Table B.4.

- *bEndpointAddress* indicates what endpoint this descriptor is describing.
- *bmAttributes* specifies the transfer type. This can either be Control, Interrupt, Isochronous or Bulk Transfer. If an Isochronous endpoint is specified, additional attributes can be selected such as the Synchronisation and usage types.
- *wMaxPacketSize* indicates the maximum payload size for this endpoint.
- *bInterval* is used to specify the polling interval of certain transfers. The units are expressed in frames, thus this equates to either 1 ms for low/full speed devices and 125 μs for high speed devices.

B.5 String Descriptor

String descriptors provide human readable information and are optional. If they are not used, any string index fields of descriptors must be set to zero indicating there is no string descriptor available. The strings are encoded in the Unicode format and products can be made to support multiple languages. String Index 0 should return a

Table B.4: Endpoint Descriptor Format

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (7 bytes)
1	bDescriptionType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	Endpoint Address 0..3b Endpoint Number 4..6b Reserved 7b Direction 0 = Out , 1 = In
3	bmAttributes	1	Bitmap	• Bits 0..1 Transfer Type – 00 = Control – 01 = Isochronous – 10 = Bulk – 11 = Interrupt • Bits 3..2 = Synchronisation Type (Iso Mode) – 00 = No Synchonisation – 01 = Asynchronous – 10 = Adaptive – 11 = Synchronous • Bits 5..4 = Usage Type (Iso Mode) – 00 = Data Endpoint – 01 = Feedback Endpoint – 10 = Explicit Feedback Data Endpoint – 11 = Reserved
4	wMaxPacketSize	2	Number	Maximum Packet Size this endpoint is capable of sending or receiving
6	bInterval	1	Number	Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Iso must equal 1 and field may range from 1 to 255 for interrupt endpoints.

Table B.5: String Descriptor Zero Format

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	String Descriptor (0x03)
2	wLANGID[0]	2	Number	Supported Language Code Zero
3	wLANGID[1]	2	Number	Supported Language Code One
4	wLANGID[2]	2	Number	Supported Language Code x

Table B.6: Subsequent String Descriptors Format

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

list of supported languages. A list of USB Language IDs can be found in (4).

The String Descriptor shown in Table B.5 shows the format of String Descriptor Zero. The host should read this descriptor to determine what languages are available. If a language is supported, it can then be referenced by sending the language ID in the wIndex field of a GetDescriptor(String) request. All subsequent strings take on the format shown in Table B.6.

B.6 Class- & Vendor-specific Descriptors

A device may return class- or vendor-specific descriptors in two ways:

1. If the class or vendor specific descriptors use the same format as standard descriptors, they must be returned interleaved with standard descriptors in the configuration information returned by a GetDescriptor(Configuration) request. In this case, the class- or vendor-specific descriptors must follow a related standard descriptor they modify or extend.

2. If the class- or vendor-specific descriptors are independent of configuration information or use a non-standard format, a GetDescriptor() request specifying the class or vendor specific descriptor type and index may be used to retrieve the descriptor from the device. A class or vendor specification will define the appropriate way to retrieve these descriptors.

B.6.1 Hub Descriptor

Table B.7 outlines the various fields contained by the hub descriptor.

Table B.7: Hub Descriptor Format

Offset	Field	Size	Description
0	bDescLength	1	Number of bytes in this descriptor, including this byte
1	bDescriptorType	1	Descriptor Type, value: 29H for hub descriptor
2	bNbrPorts	1	Number of downstream facing ports that this hub supports
3	wHubCharacteristics	2	• D1. . . D0: Logical Power Switching Mode – 00: Ganged Power – 01: Individual Port – 1X: No power switching • D2: Identifies a Compound Device – 0: Hub is not a compound device. – 1: Hub is a compound device. • D4. . . D3: Over-current Protection Mode – 00: Global – 01: Individual Port – 1X: No protection. • D6...D5: TT Think Time – 00: at most 8 FS bit times. – 01: at most 16 FS bit times. – 10: at most 24 FS bit times. – 11: at most 32 FS bit times. • D7: Port Indicators Supported – 0: Port Indicators are not supported – 1: Port Indicators are supported • D15. . . D8: Reserved
5	bPwrOn2PwrGood	1	Time (in 2 ms intervals) from the time the poweron sequence begins on a port until power is good on that port.

table continued on next page. . .

Table B.7: Hub Descriptor Format (continued)

Offset	Field	Size	Description
6	bHubContrCurrent	1	Maximum current requirements of the Hub Controller electronics in mA.
7	DeviceRemovable	Variable	This is a bitmap corresponding to the individual ports on the hub: • Bit 0: Reserved for future use. • Bit 1: Port 1 • Bit 2: Port 2 • Bit n: Port n (implementation-dependent, up to a maximum of 255 ports). Bit value definition: • 0B - Device is removable. • 1B - Device is non-removable
Variable	PortPwrCtrlMask	Variable	This field exists for reasons of compatibility with software written for 1.0 compliant devices. All bits in this field should be set to 1B.

Appendix C

ISP1160 Internal Registers

C.1 HC Control and Status Registers

C.1.1 HcRevision

Table C.1: HcRevision Register

Bit	Symbol	Description
31...8	–	reserved
7...0	REV(7:0)	This read-only field contains the BCD representation of the version of the HCI spefication (equal to 10H) that is implemented by the HC.

C.1.2 HcControl

The HcControl register defines the operating modes of the HC.

Table C.2: HcControl Register

Bit	Symbol	Description
31...11	–	reserved
10	RWE	**RemoteWakeupEnable:** This bit is used by the HCD to enable or disable the remote wake-up feature upon the detection of upstream resume signaling. When this bit is set and the ResumeDetected bit in HcInterruptStatus is set, a remote wake-up is signaled to the host system. Setting this bit has no impact on the generation of hardware interrupt.

table continued on next page...

Table C.2: HcControl Register (continued)

Bit	Symbol	Description
9	RWC	**RemoteWakeupConnected:** This bit indicates whether the HC supports remote wake-up signaling. If remote wake-up is supported and used by the system, it is the responsibility of system firmware to set this bit. The HC clears the bit upon a hardware reset but does not alter it upon a software reset.
8	–	reserved
7. . . 6	HCFS	**HostControllerFunctionalState** for USB: **00B —** USBReset **01B —** USBResume **10B —** USBOperational **11B —** USBSuspend A transition to USBOperational from another state causes start-of-frame (SOF) generation to begin 1 ms later. The HCD may determine whether the HC has begun sending SOFs by reading the StartofFrame field of HcInterruptStatus. This field can be changed by the HC only when in the USBSuspend state. The HC can move from the USBSuspend state to the USBResume state after detecting the resume signaling from a downstream port. The HC enters USBReset after a software reset and a hardware reset. The latter also resets the Root Hub and asserts subsequent reset signaling to downstream ports.
5. . . 0	–	reserved

C.1.3 HcCommandStatus

The HcCommandStatus register is used by the HC to receive commands issued by the HCD, and it also reflects the HC's current status. To the HCD, it appears to be a "write to set" register. The HC must ensure that bits written as logic 1 become set in the register while bits written as logic 0 remain unchanged in the register. The HCD may issue multiple distinct commands to the HC without concern for corrupting previously issued commands. The HCD has normal read access to all bits.

Table C.3: HcCommandStatus Register

Bit	Symbol	Description
31. . . 18	–	reserved
17. . . 16	SOC(1:0)	**SchedulingOverrunCount:** The field is incremented on each scheduling overrun error. It is initialized to 00B and wraps around at 11B. It will be incremented when a scheduling overrun is detected even if SchedulingOverrun in HcInterruptStatus has already been set. This is used by the HCD to monitor any persistent scheduling problems.
15. . . 1	–	reserved
0	HCR	**HostControllerReset:** This bit is set by the HCD to initiate a software reset of the HC. Regardless of the functional state of the HC, it moves to the USBSuspend state in which most of the operational registers are reset, except those stated otherwise, and no Host bus accesses are allowed. This bit is cleared by the HC upon the completion of the reset operation. The reset operation must be completed within 10 μs. This bit, when set, does not cause a reset to the Root Hub and no subsequent reset signaling should be asserted to its downstream ports.

C.1.4 HcInterruptStatus

This register provides the status of the events that cause hardware interrupts. When an event occurs, the HC sets the corresponding bit in this register. When a bit is set, a hardware interrupt is generated if the interrupt is enabled in the HcInterruptEnable register (C.1.5) and bit MasterInterruptEnable is set. The HCD can clear individual bits in this register by writing logic 1 to the bit positions to be cleared, but cannot set any of these bits. Conversely, the HC can set bits in this register, but cannot clear these bits.

Table C.4: HcInterruptStatus Register

Bit	Symbol	Description
31. . . 7	–	reserved
6	RHSC	**RootHubStatusChange:** This bit is set when the content of HcRhStatus or the content of any of HcRhPortStatus(1:2) has changed.

table continued on next page. . .

Table C.4: HcInterruptStatus Register (continued)

Bit	Symbol	Description
5	FNO	**FrameNumberOverflow:** This bit is set when the MSB of HcFmNumber (bit 15) changes value.
4	UE	**UnrecoverableError:** This bit is set when the HC detects a system error not related to USB. The HC does not proceed with any processing nor signaling before the system error has been corrected. The HCD clears this bit after the HC has been reset.
3	RD	**ResumeDetected:** This bit is set when the HC detects that a device on the USB is asserting resume signaling from a state of no resume signaling. This bit is not set when the HCD enters USBResume state.
2	SF	**StartOfFrame:** At the start of each frame, this bit is set by the HC and a SOF is generated.
1	-	reserved
0	SO	**SchedulingOverrun:** This bit is set when USB schedules for current frame overrun. A scheduling overrun will also cause the SchedulingOverrunCount of HcCommandStatus to be incremented.

C.1.5 HcInterruptEnable

Each enable bit in the HcInterruptEnable register corresponds to an associated interrupt bit in the HcInterruptStatus register (C.1.4). The HcInterruptEnable register is used to control which events generate a hardware interrupt. A hardware interrupt is requested on the host bus when three conditions occur:

1. A bit is set in the HcInterruptStatus register
2. The corresponding bit in the HcInterruptEnable register is set
3. Bit MasterInterruptEnable is set.

Writing a logic 1 to a bit in this register sets the corresponding bit, whereas writing a logic 0 to a bit in this register leaves the corresponding bit unchanged. On a read, the current value of this register is returned.

Table C.5: HcInterruptEnable Register

Bit	Symbol	Description
31	MIE	**MasterInterruptEnable:** A logic 0 is ignored by the HC. A logic 1 enables interrupt generation by events specified in other bits of this register.
30. . . 7	–	reserved
6	RHSC	Enable interrupt generation due to Root Hub Status Change
5	FNO	Enable interrupt generation due to Frame Number Overflow
4	UE	Enable interrupt generation due to Unrecoverable Error
3	RD	Enable interrupt generation due to Resume Detect
2	SF	Enable interrupt generation due to Start of Frame
1	-	reserved
0	SO	Enable interrupt generation due to Scheduling Overrun

C.1.6 HcInterruptDisable

Each disable bit in the HcInterruptDisable register corresponds to an associated interrupt bit in the HcInterruptStatus register (C.1.4). The HcInterruptDisable register is coupled with the HcInterruptEnable register (C.1.5). Thus, writing a logic 1 to a bit in this register clears the corresponding bit in the HcInterruptEnable register, whereas writing a logic 0 to a bit in this register leaves the corresponding bit in the HcInterruptEnable register unchanged. On a read, the current value of the HcInterruptEnable register is returned.

C.2 HC Frame Counter Registers

C.2.1 HcFmInterval

The HcFmInterval register contains a 14 bit value which indicates the bit time interval in a frame (that is, between two consecutive SOFs), and a 15 bit value indicating the full-speed maximum packet size that the HC may transmit or receive without causing a scheduling overrun.

Table C.6: HcFmInterval Register

Bit	Symbol	Description
31	FIT	**FrameIntervalToggle:** The HCD toggles this bit whenever it loads a new value to FrameInterval.
30. . . 16	FSMPS(14:0)	**FSLargestDataPacket:** Specifies a value which is loaded into the Largest Data Packet Counter at the beginning of each frame. The counter value represents the largest amount of data in bits which can be sent or received by the HC in a single transaction at any given time without causing a scheduling overrun. The field value is calculated by the HCD.
15. . . 14	-	reserved
13. . . 0	FI(13:0)	**FrameInterval:** Specifies the interval between two consecutive SOFs in bit times. The default value is 11999. The HCD must save the current value of this field before resetting the HC. Setting the HostControllerReset field of the HcCommandStatus register will cause the HC to reset this field to its default value. HCD may choose to restore the saved value upon completing the reset sequence.

C.2.2 HcFmRemaining

The HcFmRemaining register is a 14 bit down counter showing the bit time remaining in the current frame.

Table C.7: HcFmRemaining Register

Bit	Symbol	Description
31	FRT	**FrameRemainingToggle:** This bit is loaded from the FrameIntervalToggle field of the HcFmInterval register whenever FrameRemaining reaches 0. This bit is used by the HCD for synchronization between FrameInterval and FrameRemaining.
30. . . 14	-	reserved
13. . . 0	FR(13:0)	**FrameRemaining:** This counter is decremented at each bit time. When it reaches zero, it is reset by loading the FrameInterval value specified in the HcFmInterval register at the next bit time boundary. When entering the USBOperational state, the HC reloads it with the content of the FrameInterval part of the HcFmInterval register and uses the updated value from the next SOF.

C.2.3 HcFmNumber

The HcFmNumber register is a 16 bit counter. It provides a timing reference for events happening in the HC and the HCD. The HCD may use the 16 bit value specified in this register and generate a 32 bit frame number without requiring frequent access to the register.

Table C.8: HcFmNumber Register

Bit	Symbol	Description
31...16	–	reserved
15...0	FN(15:0)	FrameNumber: This is incremented when HcFm-Remaining is reloaded. It rolls over to 0000H after FFFFH. When the USBOperational state is entered, this will be incremented automatically. The HC will set bit StartofFrame in the HcInterruptStatus register.

C.2.4 HcLSThreshold

The HcLSThreshold register contains an 11 bit value used by the HC to determine whether to commit to the transfer of a maximum of 8 byte LS packet before EOF. Neither the HC nor the HCD is allowed to change this value.

Table C.9: HcLSThreshold Register

Bit	Symbol	Description
31...11	–	reserved
10...0	LST(10:0)	**LSThreshold:** Contains a value that is compared to the FrameRemaining field before a low-speed transaction is initiated. The transaction is started only if FrameRemaining is greater or equal to this field. The value is calculated by the HCD, which considers transmission and set-up overhead.

C.3 HC Root Hub Registers

C.3.1 HcRhDescriptorA

The HcRhDescriptorA register is the first register of two describing the characteristics of the Root Hub. Reset values are implementation-specific. The descriptor length, descriptor type and hub controller current fields of the hub Class Descriptor(refer to B.6.1, on page 129) are emulated by the HCD. All other fields are located in registers HcRhDescriptorA and HcRhDescriptorB (C.3.2).

Table C.10: HcRhDescriptorA Register

Bit	Symbol	Description
31...24	POTPGT(7:0)	**PowerOnToPowerGoodTime:** This byte specifies the duration HCD has to wait before accessing a powered-on port of the Root Hub. The unit of time is 2 ms.
23...13	-	reserved
12	NOCP	**NoOverCurrentProtection:** This bit describes how the overcurrent status for the Root Hub ports are reported. When this bit is cleared, the OverCurrentProtectionMode field specifies global or per-port reporting. **0 —** overcurrent status is reported collectively for all downstream ports **1 —** no overcurrent reporting supported
11	OCPM	**OverCurrentProtectionMode:** This bit describes how the overcurrent status for the Root Hub ports is reported. At reset, this field reflects the same mode as PowerSwitchingMode. This field is valid only if the NoOverCurrentProtection field is cleared. **0 —** overcurrent status is reported collectively for all downstream ports **1 —** overcurrent status is reported on a per-port basis.
10	DT	**DeviceType:** This bit specifies that the Root Hub is not a compound device. This field should always read/write 0.

table continued on next page...

Table C.10: HcRhDescriptorA Register (continued)

Bit	Symbol	Description
9	NPS	**NoPowerSwitching:** This bit is used to specify whether power switching is supported or ports are always powered. When this bit is cleared, bit PowerSwitchingMode specifies global or per-port switching. **0 —** ports are power switched **1 —** ports are always powered on when the HC is powered on
8	PSM	**PowerSwitchingMode:** This bit is used to specify how the power switching of the Root Hub ports is controlled. This field is valid only if the NoPowerSwitching field is cleared. **0 —** all ports are powered at the same time **1 —** each port is powered individually. This mode allows port power to be controlled by either the global switch or per-port switching. If bit PortPowerControlMask is set, the port responds to only port power commands (Set/ClearPortPower). If the port mask is cleared, then the port is controlled only by the global power switch (Set/ClearGlobalPower).
7. . . 2	-	reserved
1. . . 0	NDP(1:0)	**NumberDownstreamPorts:** These bits specify the number of downstream ports supported by the Root Hub. The maximum number of ports supported by the ISP1160 is 2.

C.3.2 HcRhDescriptorB

The HcRhDescriptorB register is the second register of two describing the characteristics of the Root Hub. These fields are written during initialization to correspond with the system implementation. Reset values are implementation-specific.

Table C.11: HcRhDescriptorB Register

Bit	Symbol	Description
31. . . 19	-	reserved

table continued on next page. . .

Table C.11: HcRhDescriptorB Register (continued)

Bit	Symbol	Description
18. . . 16	PPCM(2:0)	**PortPowerControlMask:** Each bit indicates whether a port is affected by a global power control command when PowerSwitchingMode is set. When set, the port's power state is only affected by per-port power control (Set/ClearPortPower). When cleared, the port is controlled by the global power switch (Set/ClearGlobalPower). If the device is configured to global switching mode (PowerSwitchingMode = 0), this field is not valid. **Bit 0 —** reserved **Bit 1 —** Ganged-power mask on Port #1 **Bit 2 —** Ganged-power mask on Port #2
15. . . 3	-	reserved
2. . . 0	DR(2:0)	**DeviceRemovable:** Each bit is dedicated to a port of the Root Hub. When cleared, the attached device is removable. When set, the attached device is not removable. **Bit 0 —** reserved **Bit 1 —** Device attached to Port #1 **Bit 2 —** Device attached to Port #2

C.3.3 HcRhStatus

The HcRhStatus register is divided into two parts. The lower word of a DWORD represents the Hub Status field and the upper word represents the Hub Status Change field. Reserved bits should always be written as logic 0.

Table C.12: HcRhStatus Register

Bit	Symbol	Description
31	CRWE	*On write,* **ClearRemoteWakeupEnable:** Writing a logic 1 clears DeviceRemoveWakeupEnable. Writing a logic 0 has no effect.
30. . . 18	-	reserved

table continued on next page. . .

Table C.12: HcRhStatus Register (continued)

Bit	Symbol	Description
17	OCIC	**OverCurrentIndicatorChange:** This bit is set by hardware when a change has occurred to the OCI field of this register. The HCD clears this bit by writing a logic 1. Writing a logic 0 has no effect.
16	LPSC	*On read,* **LocalPowerStatusChange:** The Root Hub does not support the local power status feature. Therefore, this bit is always read as logic 0. *On write,* **SetGlobalPower:** In global power mode (PowerSwitchingMode = 0), this bit is written to logic 1 to turn on power to all ports (set PortPowerStatus). In per-port power mode, it sets PortPowerStatus only on ports whose bit PortPowerControlMask is not set. Writing a logic 0 has no effect.
15	DRWE	*On read,* **DeviceRemoteWakeupEnable:** This bit enables the bit ConnectStatusChange as a resume event, causing a state transition from USBSuspend to USBResume and setting the ResumeDetected interrupt. **0 —** ConnectStatusChange is not a remote wake-up event **1 —** ConnectStatusChange is a remote wake-up event *On write,* **SetRemoteWakeupEnable:** Writing a logic 1 sets DeviceRemoveWakeupEnable. Writing a logic 0 has no effect.
14. . . 2	-	reserved
1	OCI	**OverCurrentIndicator:** This bit reports overcurrent conditions when global reporting is implemented. When set, an overcurrent condition exists. When clear, all power operations are normal. If per-port overcurrent protection is implemented this bit is always logic 0.

table continued on next page. . .

Table C.12: HcRhStatus Register (continued)

Bit	Symbol	Description
0	LPS	*On read*, **LocalPowerStatus:** The Root Hub does not support the local power status feature. Therefore, this bit is always read as logic 0. *On write*, **ClearGlobalPower:** In global power mode (PowerSwitchingMode = 0), this bit is written to logic 1 to turn off power to all ports (clear PortPowerStatus). In per-port power mode, it clears PortPowerStatus only on ports whose bit PortPowerControlMask is not set. Writing a logic 0 has no effect.

C.3.4 HcRhPortStatus[1:2]

The HcRhPortStatus(1:2) register is used to control and report port events on a per-port basis. NumberDownstreamPorts represents the number of HcRhPortStatus registers that are implemented in hardware. The lower word is used to reflect the port status, whereas the upper word reflects the status change bits.

Table C.13: HcRhPortStatus(1:2) Register

Bit	Symbol	Description
31. . . 21	-	reserved
20	PRSC	**PortResetStatusChange:** This bit is set at the end of the 10 ms port reset signal. The HCD writes a logic 1 to clear this bit. Writing a logic 0 has no effect. **0 —** port reset is not complete **1 —** port reset is complete
19	OCIC	**PortOverCurrentIndicatorChange:** This bit is valid only if overcurrent conditions are reported on a per-port basis. This bit is set when Root Hub changes the PortOverCurrentIndicator bit. The HCD writes a logic 1 to clear this bit. Writing a logic 0 has no effect. **0 —** no change in PortOverCurrentIndicator **1 —** PortOverCurrentIndicator has changed

table continued on next page. . .

Table C.13: HcRhPortStatus(1:2) Register (continued)

Bit	Symbol	Description
18	PSSC	**PortSuspendStatusChange:** This bit is set when the full resume sequence has been completed. The HCD writes a logic 1 to clear this bit. Writing a logic 0 has no effect. This bit is also cleared when ResetStatusChange is set. **0 —** resume is not completed **1 —** resume is completed
17	PESC	**PortEnableStatusChange:** This bit is set when hardware events cause the PortEnableStatus bit to be cleared. Changes from HCD writes do not set this bit. The HCD writes a logic 1 to clear this bit. Writing a logic 0 has no effect. **0 —** no change in PortEnableStatus **1 —** change in PortEnableStatus
16	CSC	**ConnectStatusChange:** This bit is set whenever a connect or disconnect event occurs. The HCD writes a logic 1 to clear this bit. Writing a logic 0 has no effect. If CurrentConnectStatus is cleared when a SetPortReset, SetPortEnable, or SetPortSuspend write occurs, this bit is set to force the driver to reevaluate the connection status since these writes should not occur if the port is disconnected. **0 —** no change in CurrentConnectStatus **1 —** change in CurrentConnectStatus **Remark:** If bit DeviceRemovable(NDP) is set, this bit is set only after a Root Hub reset to inform the system that the device is attached.
15. . . 10	-	reserved

table continued on next page. . .

Table C.13: HcRhPortStatus(1:2) Register (continued)

Bit	Symbol	Description
9	LSDA	*On read*, **LowSpeedDeviceAttached:** This bit indicates the speed of the device connected to this port. When set, a low-speed device is connected to this port. When clear, a full-speed device is connected to this port. This field is valid only when the CurrentConnectStatus is set. **0 —** full-speed device attached **1 —** low-speed device attached *On write*, **ClearPortPower:** The HCD clears bit PortPowerStatus by writing a logic 1 to this bit. Writing a logic 0 has no effect.
8	PPS	*On read*, **PortPowerStatus:** This bit reflects the port power status, regardless of the type of power switching implemented. This bit is cleared if an overcurrent condition is detected. The HCD sets this bit by writing SetPortPower or SetGlobalPower. The HCD clears this bit by writing ClearPortPower or ClearGlobalPower. Which power control switches are enabled is determined by PowerSwitchingMode. In the global switching mode (PowerSwitchingMode = 0), only Set/ClearGlobalPower controls this bit. In per-port power switching (PowerSwitchingMode = 1), if bit PortPowerControlMask(NDP) for the port is set, only Set/ClearPortPower commands are enabled. If the mask is not set, only Set/ClearGlobalPower commands are enabled. When port power is disabled, CurrentConnectStatus, PortEnableStatus, PortSuspendStatus, and PortResetStatus should be reset. **0 —** port power is off **1 —** port power is on *On write* **SetPortPower:** The HCD writes a logic 1 to set bit PortPowerStatus. Writing a logic 0 has no effect.
7...5	-	reserved

table continued on next page...

Table C.13: HcRhPortStatus(1:2) Register (continued)

Bit	Symbol	Description
4	PRS	*On read*, **PortResetStatus:** When this bit is set, port reset signaling is asserted. When reset is completed, this bit is cleared when PortReset-StatusChange is set. This bit cannot be set if CurrentConnectStatus is cleared. **0 —** port reset signal is not active **1 —** port reset signal is active *On write*, **SetPortReset:** The HCD sets the port reset signaling by writing a logic 1 to this bit. Writing a logic 0 has no effect. If Current-ConnectStatus is cleared, this write does not set PortResetStatus but instead sets Connect-StatusChange. This informs the driver that it attempted to reset a disconnected port.
3	POCI	*On read*, **PortOverCurrentIndicator:** This bit is valid only when the Root Hub is configured in such a way that overcurrent conditions are reported on a per-port basis. If cleared, all power operations are normal for this port. If set, an overcurrent condition exists on this port. **0 —** no overcurrent condition **1 —** overcurrent condition detected *On write*, **ClearSuspendStatus:** The HCD writes a logic 1 to initiate a resume. Writing a logic 0 has no effect. A resume is initiated only if Port-SuspendStatus is set.

table continued on next page. . .

Table C.13: HcRhPortStatus(1:2) Register (continued)

Bit	Symbol	Description
2	PSS	*On read*, **PortSuspendStatus:** This bit indicates whether the port is suspended or in the resume sequence. It is cleared when PortSuspendStatusChange is set at the end of the resume interval. **0 —** port is not suspended **1 —** port is suspended *On write*, **SetPortSuspend:** The HCD sets bit PortSuspendStatus by writing a logic 1 to this bit. Writing a logic 0 has no effect. If CurrentConnectStatus is cleared, this write does not set PortSuspendStatus; instead it sets ConnectStatusChange. This informs the driver that it attempted to suspend a disconnected port.
1	PES	*On read*,**PortEnableStatus:** This bit indicates whether the port is enabled or disabled. The Root Hub may clear this bit when an overcurrent condition, disconnect event, switched-off power, or operational bus error is detected. This change also causes PortEnabledStatusChange to be set. The HCD sets this bit by writing SetPortEnable and clears it by writing ClearPortEnable. This bit cannot be set when CurrentConnectStatus is cleared. This bit is also set at the completion of a port reset when ResetStatusChange is set or port is suspended when SuspendStatusChange is set. **0 —** port is disabled **1 —** port is enabled *On write*, **SetPortEnable:** The HCD sets PortEnableStatus by writing a logic 1. Writing a logic 0 has no effect. If CurrentConnectStatus is cleared, this write does not set PortEnableStatus, but instead sets ConnectStatusChange. This informs the driver that it attempted to enable a disconnected port.

table continued on next page. . .

Table C.13: HcRhPortStatus(1:2) Register (continued)

Bit	Symbol	Description
0	CCS	*On read,* **CurrentConnectStatus:** This bit reflects the current state of the downstream port. **0 —** no device connected **1 —** device connected *On write,* **ClearPortEnable:** The HCD writes a logic 1 to this bit to clear bit PortEnableStatus. Writing a logic 0 has no effect. CurrentConnectStatus is not affected by any write. **Remark:** This bit always reads logic 1 when the attached device is non-removable (DeviceRemoveable(NDP)).

C.4 HC DMA and Interrupt Control Registers

C.4.1 HcHardwareConfiguration

Table C.14: HcHardwareConfiguration Register

Bit	Symbol	Description
15. . . 13	-	reserved
12	2_Downstream Port 15K resistorsel	**0 —** use external 15 KΩ resistors for downstream ports **1 —** use built-in resistors for downstream ports
11	Suspend Clk NotStop	**0 —** clock can be stopped **1 —** clock can not be stopped
10	Analog OC Enable	**0 —** use external OC detection; digital input **1 —** use on-chip OC detection; analog input
9	-	reserved

table continued on next page. . .

Table C.14: HcHardwareConfiguration Register (continued)

Bit	Symbol	Description
8	DACK Mode	**0 —** normal operation; pin DACK_N is used with read and write signals **1 —** reserved
7	EOTInput Polarity	**0 —** active LOW **1 —** active HIGH
6	DACKInput Polarity	**0 —** active LOW **1 —** reserved
5	DREQOutput Polarity	**0 —** active LOW **1 —** active HIGH
4. . . 3	DataBus Width(1:0)	These bits are fixed at logic 0 and logic 1 for the ISP1160. **01 —** 16 bits **Others —** reserved
2	Interrupt Output Polarity	**0 —** active LOW **1 —** active HIGH
1	Interrupt Pin Trigger	**0 —** interrupt is level-triggered **1 —** interrupt is edge-triggered

table continued on next page. . .

Table C.14: HcHardwareConfiguration Register (continued)

Bit	Symbol	Description
0	Interrupt Pin Enable	This bit is used as pin INT's master interrupt enable and should be used together with register HcμPInterruptEnable to enable pin INT. **0 —** pin INT is disabled **1 —** pin INT is enabled

C.4.2 HcDMAConfiguration

Table C.15: HcDMAConfiguration Register

Bit	Symbol	Description
15. . . 7	-	reserved
6. . . 5	BurstLen(1:0)	**00 —** single-cycle burst DMA **01 —** 4-cycle burst DMA **10 —** 8-cycle burst DMA **11 —** reserved
4	DMAEnable	This bit will be reset to logic 0 when DMA transfer is completed. **0 —** DMA is terminated **1 —** DMA is enabled
3	-	reserved
2	DMA Counter Select	HcTransferCounter register must be filled with non-zero values for DREQ to be raised after bit DMA Enable is set. **0 —** DMA counter not used. External EOT must be used **1 —** enables the DMA counter for DMA transfer.

table continued on next page. . .

Table C.15: HcDMAConfiguration Register (continued)

Bit	Symbol	Description
1	ITL_ATL_Data Select	**0 —** ITL buffer RAM selected for ITL data **1 —** ATL buffer RAM selected for ATL data
0	DMA Read-Write Select	**0 —** read from the HC FIFO buffer RAM **1 —** write to the HC FIFO buffer RAM

C.4.3 HcTransferCounter

This register holds the number of bytes of a PIO or DMA transfer. For a PIO transfer, the number of bytes being read or written to the Isochronous Transfer List (ITL) or Acknowledged Transfer List (ATL) buffer RAM must be written into this register. For a DMA transfer, the number of bytes must be written into this register as well. However, for this counter to be read into the DMA counter, the HCD must set bit 2 (DMACounterSelect) of the HcDMAConfiguration register (C.4.2). The counter value for ATL must not be greater than 1000H, and for ITL it must not be greater than 800H. When the byte count of the data transfer reaches this value, the HC will generate an internal EOT signal to set bit 2 (AllEOTInterrupt) of the HcμPInterrupt register (C.4.4), and also update the HcBufferStatus register (C.6.3).

Table C.16: HcTransferCounter Register

Bit	Symbol	Description
15. . . 0	Counter value	The number of data bytes to be read to or written from RAM.

C.4.4 HcμPInterrupt

All the bits in this register will be active on power-on reset. However, none of the active bits will cause an interrupt on the interrupt pin (INT) unless they are set by the respective bits in the HcμPInterruptEnable register (C.4.5), and together with bit 0 of the HcHardwareConfiguration register (C.4.1).

After this register is read, the bits that are active will not be reset, until logic 1 is written to the bits in this register to clear it. To clear all the enabled bits in this register, the HCD must write FFH to this register.

Table C.17: HcμPInterrupt Register

Bit	Symbol	Description
15. . . 7	-	reserved
6	ClkReady	**0 —** no event **1 —** clock is ready. After a wakeup is sent, there is a wait for clock ready. Maximum is 1 ms, and typical is 160 μs.
5	HCSuspend	**0 —** no event **1 —** the HC has been suspended and no USB activity is sent from the microprocessor for each ms. When the microprocessor wants to suspend the HC, the microprocessor must write to the HcControl register. And when all downstream devices are suspended, then the HC stops sending SOF; the HC is suspended by having the HcControl register written into.
4	OPR_Reg	**0 —** no event **1 —** there are interrupts from HC side. Need to read HcControl and HcInterrupt registers to detect type of interrupt on the HC.
3	-	reserved
2	AllEOTInt	**0 —** no event **1 —** implies that data transfer has been completed via PIO transfer or DMA transfer. Occurrence of internal or external EOT will set this bit.

table continued on next page. . .

Table C.17: HcμPInterrupt Register (continued)

Bit	Symbol	Description
1	ATLInt	**0 —** no event **1 —** implies that the microprocessor must read ATL data from the HC. This requires that the HcBufferStatus register must first be read. The time for this interrupt depends on the number of clocks bit set for USB activities in each ms.
0	SOFITLInt	**0 —** no event **1 —** implies that SOF indicates the 1 ms mark. The ITL buffer that the HC has handled must be read. To know the ITL buffer status, the HcBufferStatus register must first be read. This is for the microprocessor to get ISO data to or from the HC.

C.4.5 HcμPInterruptEnable

The bits 6:0 in this register are the same as those in the HcμPInterrupt register (C.4.4). They are used together with bit 0 of the HcHardwareConfiguration register (C.4.1) to enable or disable the bits in the HcμPInterrupt register.

On power-on, all bits in this register are masked with logic 0. This means no interrupt request output on the interrupt pin INT can be generated. When the bit is set to logic 1, the interrupt for the bit is not masked but enabled.

Table C.18: HcμPInterruptEnable Register

Bit	Symbol	Description
15. . . 7	-	reserved
6	ClkReady	**0 —** power-up value **1 —** enables ClkReady interrupt

table continued on next page. . .

Table C.18: HcμPInterruptEnable Register (continued)

Bit	Symbol	Description
5	HC Suspended Enable	**0 —** power-up value **1 —** enables HC suspended interrupt.
4	OPR Interrupt Enable	**0 —** power-up value **1 —** enables the 32 bit operational register's interrupt.
3	-	reserved
2	EOT Interrupt Enable	**0 —** power-up value **1 —** enables the EOT interrupt which indicates an end of a read/write transfer
1	ATL Interrupt Enable	**0 —** power-up value **1 —** enables ATL interrupt. The time for this interrupt depends on the number of clock bits set for USB activities in each ms.
0	SOF Interrupt Enable	**0 —** power-up value **1 —** enables the interrupt bit due to SOF (for the microprocessor DMA to get ISO data from the HC by first accessing the HcDMAConfiguration register)

C.5 HC Miscellaneous Registers

C.5.1 HcChipID

This register contains the ID of the ISP1160 silicon chip. The higher byte stands for the product name. The lower byte indicates the revision number of the product.

Table C.19: HcChipID Register

Bit	Symbol	Description
15. . . 0	ChipID(15:0)	ISP1160's chip ID

C.5.2 HcScratch

This register is for the HCD to save and restore values when required.

Table C.20: HcScratch Register

Bit	Symbol	Description
15. . . 0	Scratch(15:0)	Scratch register value

C.5.3 HcSoftwareReset

This register provides a means for software reset of the HC. To reset the HC, the HCD must write a reset value of F6H to this register. Upon receiving the reset value, the HC resets all the registers except its buffer memory.

Table C.21: HcSoftwareReset Register

Bit	Symbol	Description
15. . . 0	Reset(15:0)	Writing a reset value of F6H will cause the HC to reset all the registers except its buffer memory.

C.6 HC Buffer RAM Control Registers

C.6.1 HcITLBufferLength

This register is written to assign the ITL buffer size in bytes: ITL0 and ITL1 are assigned the same value.

Table C.22: HcITLBufferLength Register

Bit	Symbol	Description
15. . . 0	ITLBufferLength(15:0)	Assigns ITL buffer length

C.6.2 HcATLBufferLength

This register is written to assign the ATL buffer size in bytes.

Table C.23: HcATLBufferLength Register

Bit	Symbol	Description
15. . . 0	ATLBufferLength(15:0)	Assigns ATL buffer length

C.6.3 HcBufferStatus

Table C.24: HcBufferStatus Register

Bit	Symbol	Description
15. . . 6	-	reserved
5	ATLBufferDone	**0 —** ATL Buffer not read by HC yet **1 —** ATL Buffer read by HC
4	ITL1BufferDone	**0 —** ITL1 Buffer not read by HC yet **1 —** ITL1 Buffer read by HC
3	ITL0BufferDone	**0 —** ITL0 Buffer not read by HC yet **1 —** ITL0 Buffer read by HC
2	ATLBufferFull	**0 —** ATL Buffer is empty **1 —** ATL Buffer is full
1	ITL1BufferFull	**0 —** ITL1 Buffer is empty **1 —** ITL1 Buffer is full
0	ITL0BufferFull	**0 —** ITL0 Buffer is empty **1 —** ITL0 Buffer is full

C.6.4 HcReadBackITL0Length

This register's value stands for the current number of data bytes inside the ITL0 buffer to be read back by the microprocessor. The HCD must set the HcTransferCounter (C.4.3) equivalent to this value before reading back the ITL0 buffer RAM.

Table C.25: HcReadBackITL0Length Register

Bit	Symbol	Description
15. . . 0	RdITL0BufferLength(15:0)	The number of bytes for ITL0 data to be read back by the microprocessor

C.6.5 HcReadBackITL1Length

This register's value stands for the current number of data bytes inside the ITL1 buffer to be read back by the microprocessor. The HCD must set the HcTransferCounter (C.4.3) equivalent to this value before reading back the ITL1 buffer RAM.

Table C.26: HcReadBackITL1Length Register

Bit	Symbol	Description
15. . . 0	RdITL1BufferLength(15:0)	The number of bytes for ITL1 data to be read back by the microprocessor

C.6.6 HcITLBufferPort

This is the ITL buffer RAM read/write port. The bits 15:8 contain the data byte that comes from the ITL buffer RAM's even address. The bits 7:0 contain the data byte that comes from the ITL buffer RAM's odd address.

Table C.27: HcITLBufferPort Register

Bit	Symbol	Description
15. . . 0	DataWord(15:0)	Read/Write ITL buffer RAM's two data bytes

C.6.7 HcATLBufferPort

This is the ATL buffer RAM read/write port. The bits 15:8 contain the data byte that comes from the ATL buffer RAM's odd address. The bits 7:0 contain the data byte that comes from the ATL buffer RAM's even address.

Table C.28: HcATLBufferPort Register

Bit	Symbol	Description
15. . . 0	DataWord(15:0)	Read/Write ATL buffer RAM's two data bytes

Bibliography

[1] Linux devices. http://www.linuxdevices.com.

[2] *Open Host Controller Interface Specification for USB*, 1.0a edition, September 1999. http://www.compaq.com.

[3] *PLX PCI9054, Data Book*, 2.1 edition, January 2000. http://www.plxtech.com.

[4] *Universal Serial Bus Language Identifiers*, 1.0 edition, March 2000. http://www.usb.org.

[5] *Universal Serial Bus Specification*, 2.0 edition, April 2000. http://www.usb.org.

[6] *ISP1160 Embedded Programming Guide*, 1.0 edition, March 2002. http://www.semiconductors.philips.com.

[7] *PCI Local Bus Specification*, 2.3 edition, March 2002. http://www.pcisig.com.

[8] *ISP1160 Embedded Universal Serial Bus Controller, Product Data*, 05 edition, December 2004. http://www.semiconductors.philips.com.

[9] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2nd edition, June 2001.