Design and Development of a USB Device, Firmware Stack, and Linux Kernel Driver

Frank Jenner
Rochester Institute of Technology

May 16, 2011

Abstract

The purpose of this independent study was to gain a thorough understanding of the Universal Serial Bus (USB) protocol, as well as the hardware and software interfaces used to communicate between a USB host and device. As such, this study included the design and development of several components spanning the entire communication chain between the device hardware platform and the host application software. Specifically, this chain consists of a USB-enabled device platform, the USB stack firmware for the device, a Linux kernel module driver for the host PC, a user-space library to interface application code with the driver, and a demonstration software application that utilizes these underlying components to implement a useful system resource monitor.

The hardware platform consists of a custom designed circuit based around a Microchip PIC18F4550 8-bit microcontroller. This microcontroller contains built-in hardware peripherals for handling the hardware level (PHY and some data link) USB communication. The platform also contains a monochromatic graphical LCD display and a set of pushbutton switches. This design is intended to provide very generic and extensible functionality while demonstrating bidirectional communication with the host PC.

The firmware consists of the code that is programmed into the microcontroller in order for it to properly interface with the hardware components. The firmware comprises the majority of the effort of the project, as it includes a scratch implementation of a USB communication stack, in addition to code for controlling the LCD display and UART peripheral.

On the host PC, a kernel module is developed to abstract the USB communication into a char driver. This effectively creates a filesystem entry in the `/dev/` directory. By using `udev` rules to configure the permissions, this device node can be accessed in user space in order to read from and write to the device.

A software library which sits on top of the driver is also created. The purpose of this library is to abstract the device file interaction by wrapping a set of high-level functions around the communication. These functions provide operations such as getting the button state from the device and drawing pixels, lines, or text to the LCD display.

Lastly, at the top level, a simple application utilizes this device library (among other libraries) to demonstrate the successful bidirectional communication through all of the layers developed in this study. The application displays system resource usage information to the device's LCD screen, and responds to the device's pushbuttons by switching which sets of resource usage data are displayed.

Contents

Chapter 1

Introduction

The system developed in this study consists of several components, both hardware and software, that span the entire communication chain between host PC application and USB device. An outline of the system architecture and its layers is illustrated in Figure 1.1. The modular architecture presents a logical separation between the roles of the components in order to reduce coupling and promote cohesion.

Throughout the report, several code listings are used to elaborate upon certain concepts presented throughout the document. In many cases, these listings have been edited for brevity and serve primarily as a reference point to locate the region of source code that presents the concept implementation in its entirety. Thus, the actual source code should be referenced for full implementation details. To aid in navigating the source code, the `README` file on the accompanying disc contains a description of the source tree and the role of each source file.

This report will first provide an overview of the USB protocol that is central to the operation of the system. Then each of the layers in Figure 1.1 will be explained in detail, starting from the hardware and working upwards toward the application layer. The report will conclude with a summary of concepts learned and provide suggestions for possible future improvements to the system.

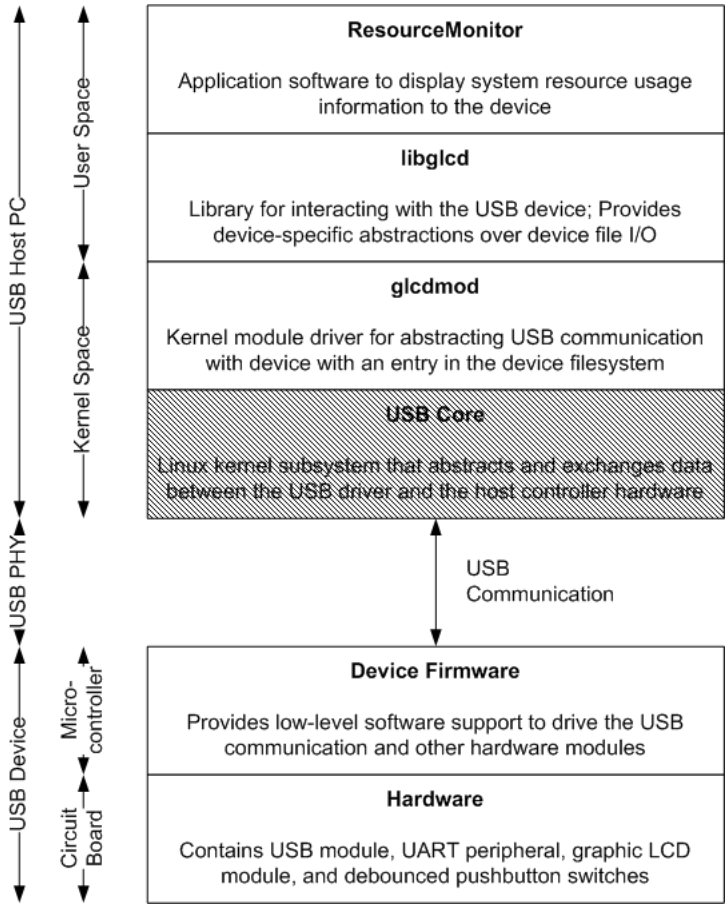

Figure 1.1: The layered architecture of the system clearly illustrates the roles of the components developed in this study. The USB core is grayed out because it was not designed as part of the study (it already exists as part of the kernel).

Chapter 2

Universal Serial Bus

2.1 Overview

The Universal Serial Bus (USB) has become an important communication bus on modern computer systems. It is well-liked for its compact footprint, high speed capabilities, and versatility. USB can support virtually any type of device, and devices may be added or removed ("hotplugged") from the system at any time. In addition, many operating systems now include driver support for many classes of USB devices out-of-the-box. These characteristics make USB a popular replacement for several older communications ports such as the RS-232 serial port, parallel printer port, and PS/2 interfaces. Despite these advantages, the Universal Serial Bus is a much more complex interface with a strict protocol, and therefore typically requires a much greater design effort to utilize effectively.

USB has a bus-based logical topology with a tree-like physical topology. The host PC's USB controller hub is the root of the tree, and all directly connected devices form the immediate leaves. However, the tree may be augmented with additional branches through the use of external USB hubs (up to five of which may be connected in series) to expand the number of ports available for connecting devices [1]. Despite this tiered architecture, the use of hubs is often transparent to the application software layer, and the entire network appears as a single bus. Each device (root hub and external hubs included) is assigned a 7-bit address, so there may be a total of 127 peripherals attached to a USB host controller (address 0 is reserved). The USB trident logo seeks to represent the topology and diversity of devices that might be found on a typical USB bus.

Since its initial release in 1996, the Universal Serial Bus has gradually evolved. The USB specification is developed by a non-profit organization called the USB Implementers Forum (USB-IF), which is a collaboration of several important companies including Hewlett-Packard, Intel, and Microsoft, among others [6]. In the USB 1.0 specification release, two speeds were supported: Low-Speed devices operated using a 1.5 Mbps raw bus speed, and Full-Speed devices operated at a 12 Mbps speed. The USB 1.1 release provided only very slight changes by adding support for interrupt transfers (transfers will be discussed in the next section) from host to device. The USB 2.0 specification, finalized in 2000, made a

significant impact on the Universal Serial Bus by providing a new speed, High-Speed. High-Speed communication is 40 times faster than Full-Speed communication (note the misnomer that Full-Speed is slower than High-Speed), using a 480 Mbps bus speed. Along with the new speed came some subtle protocol changes to support the new High-Speed devices. More importantly to the success of the protocol, however, is that the specification remained backwards compatible with USB 1.x devices. Thus, older devices would still be continued to be supported by newer host controllers. Note that USB 2.0 devices do not necessarily have to be High-Speed devices – the introduction of this new speed class simply coincides with the USB 2.0 specification. Most recently, the first revision of the USB 3.0 specification was completed in 2008. This version introduced another speed mode, SuperSpeed, which uses a 5 Gbps bus speed. Again, backwards compatibility was maintained to support 2.0 and 1.x devices [1]. While USB 2.0 has proved to be very popular, USB 3.0 has been slow to be adopted. This is likely to be because few devices require or can operate at such a high data rate. Furthermore, USB 3.0 cabling and connectors are slightly different than those used in previous versions. As a result, a USB 3.0 cable will not fit in a USB 2.0 socket, although the converse is true.

The raw data rates presented are theoretical maxima for the bus hardware, and are not indicative of the actual data throughput in the end application. The protocol itself imposes data overhead and may also enforce bandwidth restriction policies. As the bus is shared, the actual achieved bandwidth depends highly on the number of devices on the bus and which types of transfers are taking place at any given time. Nonetheless, the remaining bandwidth is more than sufficient for the operation of most devices. In fact, even most new USB 2.0 keyboards and mice opt to use Low-Speed communication because it is more than sufficient for transferring the small amounts of data associated with key presses or mouse movement.

Throughout the remainder of this document, unless specified otherwise, the USB protocol will be discussed with regard to USB 2.0 Full-Speed communication. While there are subtle protocol differences and exceptions between the variants, the bulk of the concepts remain consistent among all of the USB versions and speed modes. Most of the discrepancies are differences in parameter values such as frame times and packet sizes, which ultimately determine the throughput of the connection.

2.2 Transfers

USB is a bus-based master/slave protocol. In general, a host PC is the master, and any connected USB devices are slaves. In this scheme, all communication must be initiated by the master. Thus, in order for a device to send data to a host, the host must first request that information from the device. Furthermore, a device cannot asynchronously signal to the host that it has data it needs to send. It is the responsibility of the device and driver developers to agree on a policy for synchronizing the exchange of data in an appropriate manner.

A data read or write to or from a USB device is known as a USB “transfer”. A transfer is an abstract notion defined by the end application, without any specific mapping to the

underlying communication. For example, a transfer could be something as small as retrieving a status byte from a mouse, or as large as sending a DVD image to a flash drive. Four types of transfers are currently defined in the USB specification: bulk, interrupt, isochronous, and control [6]. Each of these transfer types has a particular set of characteristics which make them suitable for different types of operations. A summary of the the transfer types is presented in Table 2.1 and discussed in more detail below.

Transfer Type	**Control**	**Bulk**	**Interrupt**	**Isochronous**
Typical Use	Identification and configuration	Printer, scanner, drive	Mouse, keyboard	Streaming audio, video
Support required?	yes	no	no	no
Low speed allowed?	yes	no	yes	no
Maximum packet size; maximum guaranteed packets/interval (SuperSpeed).	512; none	1024; none	1024; 3 / 125 µs	1024; 48 / 125 µs
Maximum packet size; maximum guaranteed packets/interval (high speed).	64; none	512; none	1024; 3 / 125 µs	1024; 3 / 125 µs
Maximum packet size; maximum guaranteed packets/interval (full speed).	64; none	64; none	64: 1 / ms	1023; 1 / ms
Maximum packet size; maximum guaranteed packets/interval (low speed).	8; none	not allowed	8; 1 / 10 ms	not allowed
Direction of data flow	IN and OUT	IN or OUT	IN or OUT (IN only for USB 1.0)	IN or OUT
Reserved bandwidth for all transfers of the type	10% at low/full speed, 20% at high speed & SuperSpeed	none	90% at low/full speed, 80% at high speed and SuperSpeed (isochronous and interrupt combined, maximum)	
Message or Stream data?	message	stream	stream	stream
Error correction?	yes	yes	yes	no
Guaranteed delivery rate?	no	no	no	yes
Guaranteed latency (maximum time between transfers attempts)?	no	no	yes	yes

Table 2.1: The characteristics of each of USB's four transfer types for different speeds are detailed in this table. [1].

Bulk transfers are arguably the simplest type of transfer. Bulk transfers are used for applications where bandwidth and latency are not important. No bus bandwidth is reserved for this type of transfer. Instead, bulk transfers simply utilize whatever bandwidth is available on the bus after bandwidth has been allocated for any other concurrent transfers of other types. Most of the time, however, the USB bus has very little activity. In fact, when the

bus is otherwise idle, bulk transfers can use the entire bus bandwidth. In this respect, bulk transfers are actually potentially the fastest transfer type, although there is no bandwidth guarantee. Bulk transfers do, however, guarantee the integrity of the data by using error detection and handshaking mechanisms. Thus, bulk transfers are used where realtime response is not a requirement, but data integrity is, such as transferring a file from an external hard drive or sending a document to a printer [1].

In contrast, interrupt transfers offer guaranteed maximum latency for applications that require realtime response. As noted previously, devices do not have any mechanism to signal the host that there is data ready to be read. However, interrupt transfers guarantee that the host will query the device periodically (with a poll rate specified by the device) to check if the device needs to send data. It is the quick response time that gives rise to the term "interrupt" transfer, even though it is actually achieved by using high frequency polling. Like bulk transfers, data integrity is guaranteed. Despite latency and data integrity guarantees, interrupt transfers do not have bandwidth guarantees. For these reasons, interrupt transfers are used in applications where only small amounts of time-sensitive data need to be transferred. For example, mice and keyboards typically use interrupt transfers so that there is no noticeable delay between the user action and the software response [1].

Isochronous transfers are used for applications where both low latency and high bandwidth are required, but where data integrity may be sacrificed. For example, streaming media devices such as webcams or audio interfaces require high speed data communication without considerable delay, but are able to recover quickly and gracefully from an occasional error. The USB specification reserves a portion of the bus bandwidth for these types of transfers to guarantee that devices can attain their requested bandwidth and latency characteristics [1].

Lastly, but perhaps most importantly, are control transfers. These transfers differ considerably from the other types of transfers because they have a very specific, structured format that is used predominantly to communicate USB information to and from the devices. As a result, every USB device must support control transfers (the other types need to be supported only if their use is required for the intended device functionality). A more detailed discussion of control transfers follows in the section on enumeration. Control transfers typically carry only a small amount of data, but do have a modest bandwidth and latency guarantee, as well as enforced data integrity [1].

Despite the abstract meaning of a transfer, all USB transfers are fragmented into one or more "transactions" which do have a meaningful underlying manifestation. A transaction can be considered to be the basic unit for communicating data as part of a transfer. However, transactions, too, may be broken down into multiple smaller packets which, in turn, are comprised of several fields. This hierarchy is depicted in Figure 2.1. Although it is important to understand the role of all of these layers, the driver software on both the host and device sides is concerned primarily with the transaction level.

Each transaction is directed at a specific endpoint on a particular device. An endpoint may be thought of as a port on the device, where each port may only communicate in a single direction. A USB device may have up to 32 endpoints: 16 input endpoints and 16 output

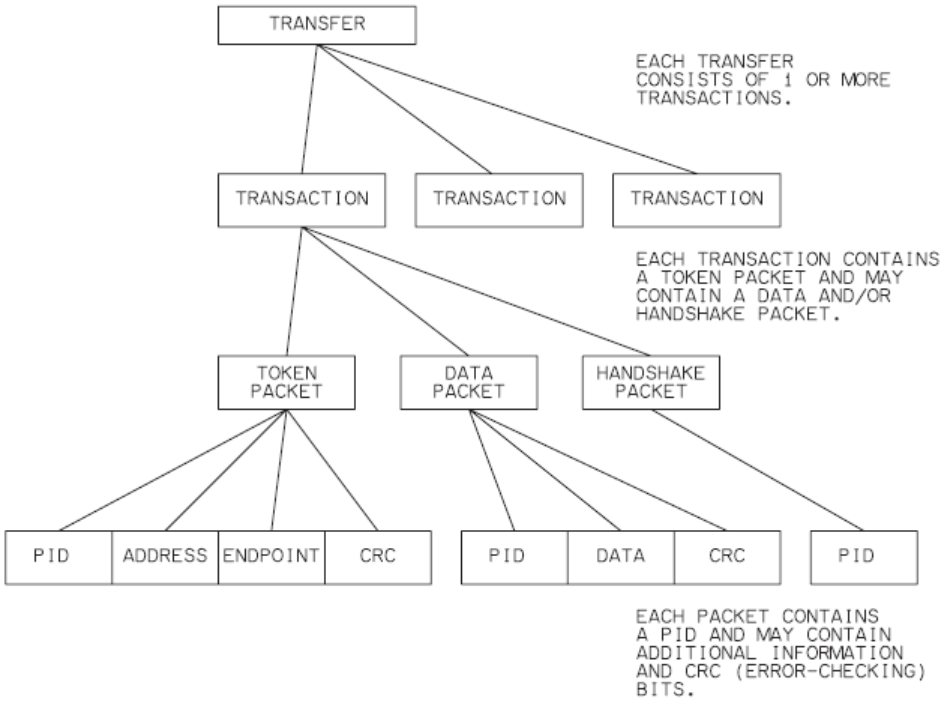

Figure 2.1: This diagram demonstrates the hierarchy of data transfer from the transfer level down to the packet field level [1].

endpoints. To avoid ambiguity when discussing data directions in USB, the terms "input" and "output" are usually denoted by IN and OUT, respectively, which always describe the direction from the host's perspective. The endpoint 0 IN/OUT pair is special in that it is used for USB protocol-specific control transfers (although it may sometimes support additional, vendor-defined control transfers). The remaining 30 endpoints may be used for any transfer type to fulfill the needs of the device.

Each transaction consists of either two or three packets. The first packet is called the token packet. As USB is a master-initiated protocol, this packet is always sent from the host to the device. The token packet contains fields that specify the target device address and endpoint number for the transaction. The token packet also contains a packet identifier (PID) field which specifies the direction in which the following packet, the data packet, will be transferred, if present. The possible PID's for the token packet are IN, to request data from the device, OUT, to send data to the device, or SETUP, which is specific to control transfers and is described later in the section on enumeration. The PID of the token packet is often used to describe the whole transaction. For instance, if a transaction's token packet contains an OUT PID, then it is said to be an OUT transaction. Lastly, a CRC field is used for error detection.

The second packet is the data packet. The data packet is transferred in the direction specified by the token packet's PID and carries the actual data payload for the transaction.

The payload is variable in length, up to a certain maximum packet size, and may even be a zero-length packet (ZLP). The PID of a data packet in a Full-Speed device is a field whose value toggles between DATA0 and DATA1 for successive transactions to that endpoint. Each time a device endpoint or a host sends or receives a data packet, it toggles a local variable of which PID it expects to get or send next. If the following data packet's PID does not match the locally toggled value, then it is concluded that a duplicate packet was received, and it will be discarded. Additionally, a CRC field is also included to ensure data integrity within the data packet.

The handshake packet, if present, consists of only one field: a PID. This packet is sent by the intended recipient of the data. For Full-Speed devices, the PID may be either ACK, NAK, or STALL. An ACK is sent as an acknowledgment that the data payload was received correctly without error. NAKs and STALLs may only be issued by devices. A NAK indicates that there was an error, that the device endpoint was busy and could not accept the data, or that the endpoint does not yet have any data available to send, and that the host should retry. A device may instead issue a STALL PID if the host sends a request that the device does not support. In this case, the host should not retry the request. A lack of a handshake packet when one is expected by the sender indicates that there was a data error. Isochronous transfers never use the handshake packet because they do not require reliable data transfer.

2.3 Enumeration

In order for the host PC to communicate with a device, it must first learn about the device's capabilities through a process known as enumeration. Among other things, the information obtained during enumeration is used to help the operating system to select an appropriate driver, or possibly drivers, to control the device. The enumeration process is very complicated, as it involves numerous control transfers and a firm understanding of USB descriptors.

2.3.1 Control Transfers

Until now, the details of control transfers have been neglected. However, since the enumeration process uses control transfers exclusively and liberally, this is an appropriate point of discourse to explain them. Unlike the other transfer types, which simply transfer an unstructured buffer of payload data, control transfers are conducted in a specific sequence with meaningful, tightly structured payload data. A single control transfer consists of up to three stages: setup, data (optional), and status. Be careful to note the unfortunate duplicate of the term "data", which has now been used to describe both a packet type and a control transfer stage, as well as the term "setup", which has been used to describe both a PID value (SETUP) and a control transfer stage. Each of the three stages, if present, consists of one or more transactions.

A control transfer starts with the setup stage, which in turn consists of a single SETUP transaction. The token packet for this transaction contains the SETUP PID that indicates to the device that it will have to take special action to handle this transfer. The data packet

for a setup transaction is always an eight byte request that specifies a particular action that the device should take. The structure of the request is shown in Table 2.2. The fields specify the direction of the data stage, the amount of data that will be transferred during the data stage, a request code, and some parameters whose meaning varies based on the particular request. The requests used during enumeration are called standard requests, although setup transactions could alternatively contain non-standard requests that are specific to a particular device class or vendor. A table of standard requests is shown in Table 2.3. In addition to setting up the request, the setup stage of the transfer should also be used by the device to initialize the data toggle values for the remainder of the transaction. Since a SETUP transaction's data packet is always sent from host to device, the handshake packet is sent from device to host, and will contain an ACK PID as long as the data was received without error (otherwise no handshake packet will be sent).

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
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

Table 2.2: This table describes all of the fields used in the data packet of a SETUP transaction for a control transfer request. [6].

The presence or absence of a data stage is determined by the request type, as indicated in Table 2.3. If the data stage is present, it consists of one or more transactions. The direction of data transfer remains the same for all transactions in the data stage and is foretold by the `bmRequestType` field in the preceding setup stage. Hence, if the direction in the setup request was "device-to-host", then all transactions in the data stage will be IN transactions,

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor 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	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor 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 2.3: This table lists the types of standard USB requests that can be performed in a control transfer, and the meanings of the data packet fields for each type of request [6].

and if the direction in the setup request was "host-to-device", then all transactions in the data stage will be OUT transactions. Each transaction serves to communicate part of the requested data. The data packet PID's will alternate between DATA0 and DATA1, starting with DATA1. The handshake packet can return ACK, NAK, or STALL as appropriate, depending on whether the device is ready and willing to honor the request. If the requested data size (as specified by the `wLength` field) is larger than the maximum packet size for the target endpoint, the data must be fragmented into multiple transactions. The data stage is complete when all of the requested data has been transferred, as indicated by the receipt of a data packet whose length is less than the maximum packet size for the endpoint, or by the receipt of a ZLP if the length of the data is a multiple of the maximum packet size.

The last stage of a control transfer is the status stage, whose purpose is simply to indicate

the end of the transfer. The reason for having an explicit stage to denote the end of the transfer is because the preceding data stage does not always run to completion. In fact, it is normal for the host to abort a control transfer prematurely – a situation that must be accounted for in the device firmware. The status stage consists only of a single transaction whose token packet PID is the opposite of those in the data stage. Thus, if the data stage consisted of IN transactions, the status stage will consist of an OUT transaction. The data packet in the status stage always contains a ZLP and has a PID of DATA1, regardless of the previous data toggle state. These characteristics can be used to unambiguously detect that the transaction represents the status stage. The handshake packet normally returns ACK, although this packet could instead return NAK or STALL as appropriate if there was no data stage in which to issue such a response.

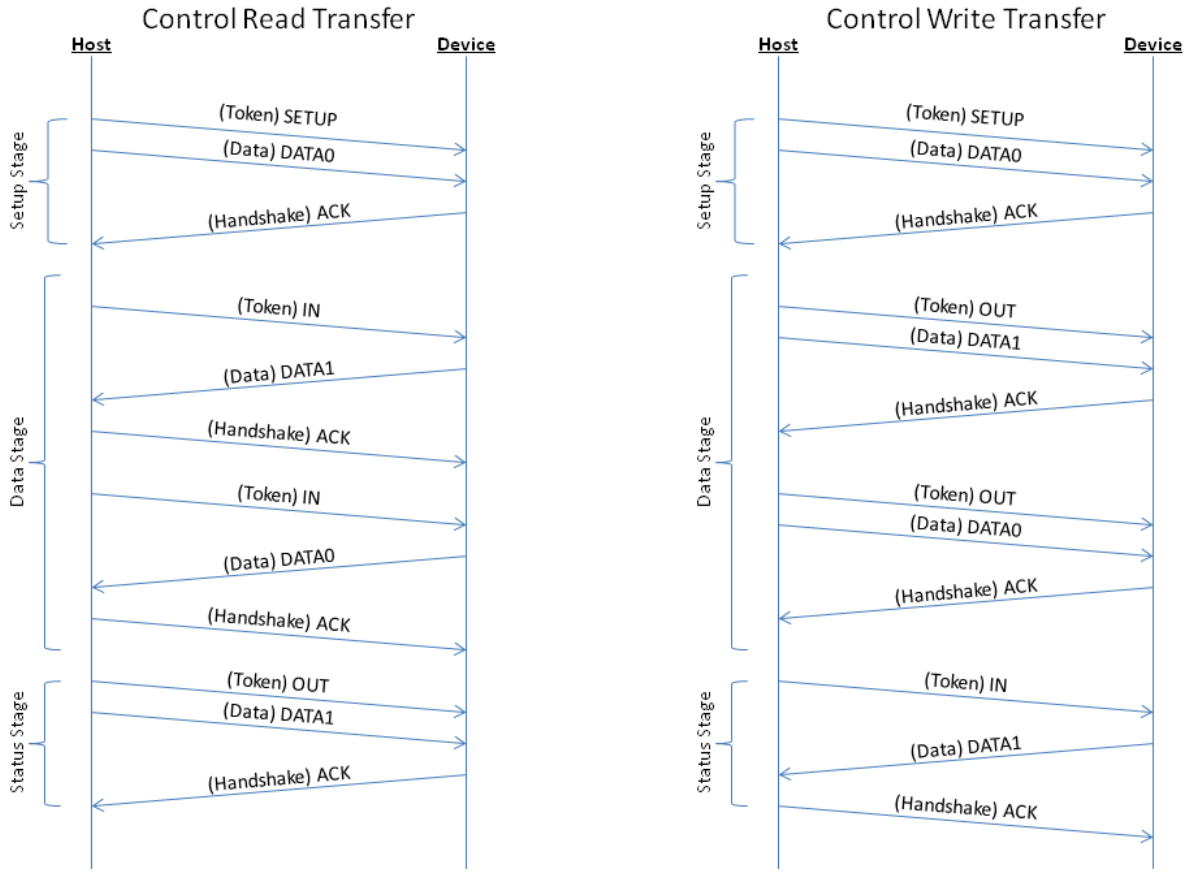

Figure 2.2: This diagram depicts the sequence of packet exchanges for a typical control transfer read and control transfer write. In both of these examples, the data stage consists of two transactions, and all data is received without error.

To clarify the operation of control transfers, two control transfer sequences have been

illustrated in Figure 2.2. In these diagrams, all three stages are present, although this may not always be the case. Notice that, regardless of whether the host is reading from or writing to the device, the token packet in every transaction always originates from the host, and its PID indicates the direction of the following data packet (recall that the data packet for SETUP transactions always travels from host to device). The data packets in the data stage always travel in the expected direction of the overall transfer (i.e., from host to device for write transfers and from device to host for read transfers). Lastly, the handshake packets always travel in the opposite direction as the preceding data packet in the transaction.

2.3.2 Descriptors

As can be seen from the list of standard requests in Table 2.3, one of the possible `bRequest` values is `GET_DESCRIPTOR`. This is one of the most important standard requests, as it provides the host with a wealth of information about the device through data structures known as descriptors. The USB specification defines numerous types of descriptors, and the particular type requested by the control transfer is specified by the `wValue` field of the SETUP transaction. Although other descriptors exist, the device must contain at minimum a device descriptor, a configuration descriptor, and an interface descriptor. Almost always, endpoint descriptors are also required. String descriptors are also very common, but not required.

To understand the significance of these descriptors, it is important to understand how a USB device is organized. The organizational hierarchy of a USB device is laid out in Figure 2.3. A device must consist of at least one configuration. Broadly, a configuration describes the current set of functionality and associated characteristics of the device. Configurations are mutually exclusive, as only one configuration of a USB device is active at a given time. Many devices support only a single configuration. Each configuration, in turn, consists of at least one interface. An interface describes a particular function of the device, and all interfaces are active concurrently. A device with multiple interfaces is sometimes referred to as a composite device. For example, a multifunction printer may have an interface for printing, one for scanning, and one for sending faxes. Usually each interface is handled by a separate driver on the host. Finally, each interface is associated with zero or more of the device's endpoints for communication (endpoint zero is not included in the association because it is always required for any USB device). Each of these entities in the device's structure is associated with a descriptor [4].

To provide an understanding of the structure of a typical descriptor, the layout of a device descriptor is shown in Table 2.4. For brevity, the structures of the other types of descriptors will not be presented explicitly. All descriptors begin with fields that present the length and type of the descriptor. Although the device descriptor has a fixed length, many other descriptors are variable in length. The device descriptor contains fields that describe the vendor and product identifiers (which are often used as part of the process of matching a driver to the device on the host), the number of configurations supported by the device, and the indices of string descriptors that contain textual descriptions of the manufacturer and product. The device also contains fields related to device classes. The USB-IF has specifications for several types of standard device classes: for instance, mass

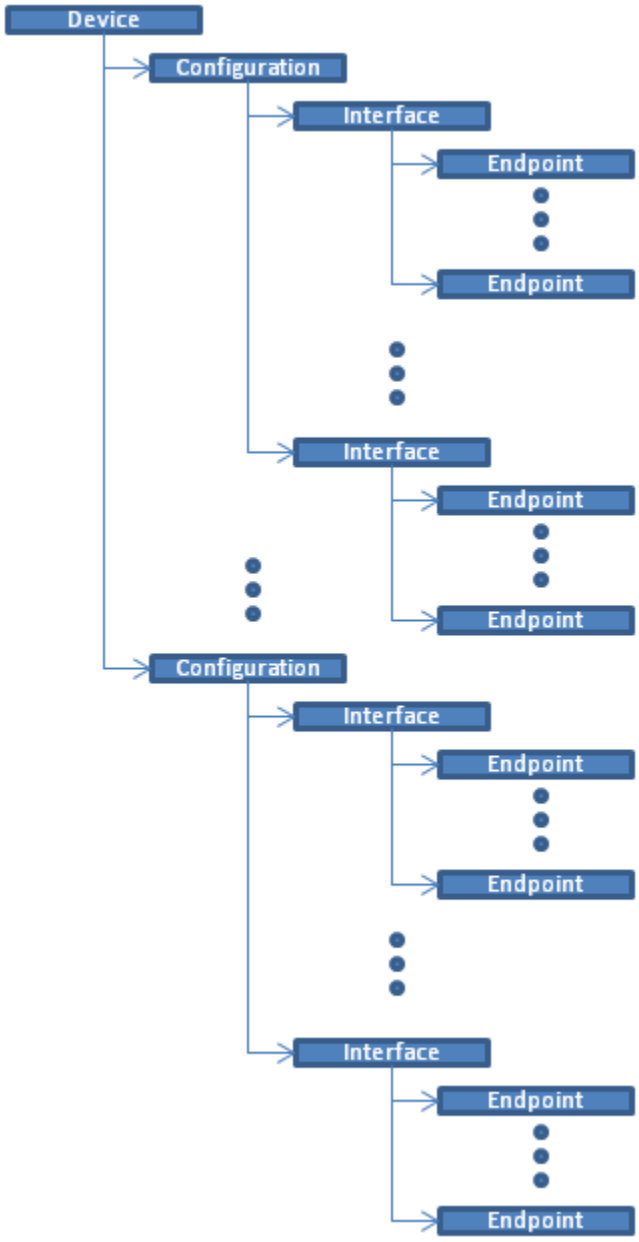

Figure 2.3: A device consists of one or more configurations, which in turn consist of at least one interface. Each interface uses zero or more endpoints.

storage devices, human interface devices, and printers. Host operating systems usually have built-in support for many of these pre-defined device classes. Devices with recognized USB device classes conform their payload data using a well-defined structure mandated in the device class specification. A device which does not implement and existing device class types is said to be vendor-defined.

Offset (decimal)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes (12h)
1	bDescriptorType	1	The constant DEVICE (01h)
2	bcdUSB	2	USB specification release number (BCD)
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol Code
7	bMaxPacketSize0	1	Maximum packet size for endpoint zero
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number (BCD)
14	iManufacturer	1	Index of string descriptor for the manufacturer
15	iProduct	1	Index of string descriptor for the product
16	iSerialNumber	1	Index of string descriptor for the serial number
17	bNumConfigurations	1	Number of possible configurations

Table 2.4: This presents the data structure layout for a USB device descriptor [1].

The configuration descriptor contains a few interesting fields. For example, one field indicates whether the device is externally powered or bus powered. Another field specifies the maximum power that this device may draw from the bus. By default, the USB bus provides 100 mA to an unconfigured device. However, USB 2.0 device may request up to 500 mA from the 5V bus for any of its configurations. Violating this limit will typically disconnect the device from the bus by removing power to the port. The configuration descriptor also specifies the number of interfaces that are associated with this configuration.

An interface descriptor provides information about the interface's function. As such, it contains a field for a string descriptor that describes the interface and the number of endpoints required to support the interface. Like the device descriptor, the interface descriptor also contains fields to specify a particular device class. In this way, a device can support multiple functions from different standard device classes simultaneously.

Endpoint descriptors provide information about the device communication endpoints. In particular, each endpoint descriptor contains fields specifying the number and direction of the endpoint, the transfer type that will be used on the endpoint, and the maximum packet size that can be transferred in a single transaction. Each endpoint is locked to a single transfer type for a given configuration.

String descriptors specify two things: the languages in which the strings have been pro-

vided, and the string data themselves. These descriptors are simply used to provide a textual counterpart to many of the fields that are encountered in the other descriptor types. String descriptors are not required to be used, but they can be very helpful in helping users to identify a particular device attached to their host machine.

Numerous other types of descriptors exist. Some of these allow for irregularities in the device structure, for example by assigning multiple interfaces to a single function. High-Speed and SuperSpeed devices have descriptors that optionally allow them to operate with reduced performance on slower bus controllers. Additionally, standard device classes usually have their own sets of descriptors that must be included as part of the device class specification.

2.3.3 The Enumeration Sequence

In order for a USB device to be recognized by the host, it must first exchange important information about itself with the host through USB descriptors and other standard requests. This process is known as enumeration. The USB specification defines several terms that describe the state of the device at any given time, and their state diagram is presented in Figure 2.4. During enumeration, the device state progresses in the following sequence: Attached, Powered, Default, Address, Configured. Once it reaches the configured state, it may be accessed by the host software.

When a device is first attached, it starts off in the Attached state. Assuming that the hub through which it is connected is configured, it proceeds immediately to the powered state. The host is able to detect the device when the device hardware pulls one of the differential data lines high. Once the device is detected, the host indicates that the device should reset itself to an known state. This indication is accomplished by the host pulling both of the differential data lines low. Following a reset, the device should respond to communication at address 0x00. This address is reserved for uninitialized devices. The device is now in the Default sate.

Since the host now knows that it can communicate with the device at address 0x00, it can send requests to the default control endpoint (always endpoint 0). The first request that the host sends is a `GET_DESCRIPTOR` in order to obtain the device descriptor. However, because the host does not yet know anything about the device, it does not know what the maximum packet size is for endpoint 0. Thus, the host simply requests the maximum possible number of bytes for any control endpoint, and aborts the control transfer as soon as it receives the first data transaction from the device. The USB specification mandates that the maximum packet size for endpoint 0 must be at least 8 bytes. Since the field for endpoint 0's maximum packet size is contained within these first 8 bytes (referring back to Table 2.4), the host is guaranteed to have received this field during the truncated transfer. The host is now aware of the maximum packet size that it may use to communicate with the default control endpoint. The host can then reset the device again to ensure that it is in a known state, although this step is not required.

Next, the host sends a `SET_ADDRESS` request to the device in order to assign it an address on the bus. Once this control transfer has successfully completed, the device is considered to be in the Addressed state. Subsequent transfers must use this new address in order to

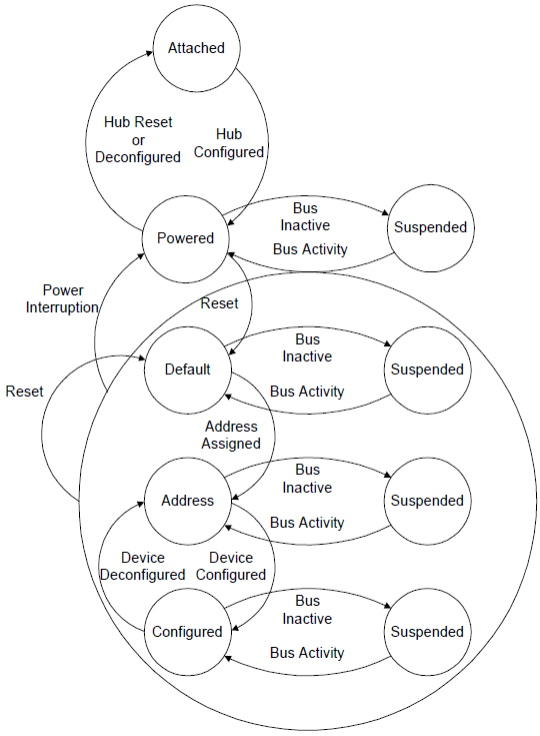

Figure 2.4: This state diagram describes how the device state changes with respect to bus events and standard requests [6].

communicate with the device.

Recall that the host never actually received the full device descriptor during the first request (instead, it terminated the transfer prematurely after the first data transaction). Now that the host is aware of the maximum packet size, however, the host performs another `GET_DESCRIPTOR` request, this time receiving the entire device descriptor. Using the information from the device descriptor (specifically, the `bNumConfigurations` field), the host has sufficient information to request the configuration descriptor(s) and, subsequently, any other remaining descriptors.

Once the host has obtained all of the descriptors from the device, the operating system will issue a `SET_CONFIGURATION` request to specify which configuration to activate. The completion of this control transfer brings the device into the Configured state. Software on the host system will now be able to successfully access the device.

Although the sequence outlined above is typical, different systems may perform the sequence differently, and the device must be prepared to handle these requests in any order. Figure 2.5 shows the actual Linux enumeration sequence for the device developed in this study (captured using a USB protocol analyzer). First, the device is reset. The host queries the device for 64 bytes of the device descriptor at endpoint 0 of device address 0. The host aborts the transfer after 8 bytes are returned and then resets the device again. The host

then sets the address of the device to 4. The host queries for the device descriptor again, this time targeting address 4 and retrieving all 18 bytes. The host then queries the device for some descriptors that the device does not support, so the device returns STALLs. Then, the host requests the configuration descriptor. This requires two requests because the configuration descriptor is actually grouped together with all of its dependents (interface and endpoint descriptors), and is therefore variable in length. The information received from the first request is sufficient to request the remainder of the data during the second request. The host continues by requesting all of the available string descriptors for this device (as denoted in the other descriptors). Finally, the host issues the `SET_CONFIGURATION` request, and the device becomes fully configured.

Dev	Ep	Record	Summary
		<Reset> / <Chirp J> / <Tiny J>	
		<Full-speed>	
00	00	Get Device Descriptor	Index=0 Length=64
		<Reset> / <Chirp J> / <Tiny J>	
		<Full-speed>	
00	00	Set Address	Address=05
05	00	Get Device Descriptor	Index=0 Length=18
05	00	SETUP txn	80 06 00 01 00 00 12 00
05	00	SETUP packet	2D 05 D0
05	00	DATA0 packet	C3 80 06 00 01 00 00 12 00 E0 F4
05	00	ACK packet	D2
05	00	IN txn	12 01 00 02 00 00 00 08
05	00	IN packet	69 05 D0
05	00	DATA1 packet	4B 12 01 00 02 00 00 00 08 57 E7
05	00	ACK packet	D2
05	00	IN txn	D8 04 04 02 34 12 01 02
05	00	IN packet	69 05 D0
05	00	DATA0 packet	C3 D8 04 04 02 34 12 01 02 A0 EE
05	00	ACK packet	D2
05	00	IN txn	00 01
05	00	IN packet	69 05 D0
05	00	DATA1 packet	4B 00 01 3F 8F
05	00	ACK packet	D2
05	00	OUT txn	
05	00	OUT packet	E1 05 D0
05	00	DATA1 packet	4B 00 00
05	00	ACK packet	D2
05	00	Control Transfer (STALL)	Index=0 Length=10
05	00	Control Transfer (STALL)	Index=0 Length=10
05	00	Control Transfer (STALL)	Index=0 Length=10
05	00	Get Configuration Descriptor	Index=0 Length=9
05	00	Get Configuration Descriptor	Index=0 Length=39
05	00	Get String Descriptor	Index=0 Length=255
05	00	Get String Descriptor	Index=2 Length=255
05	00	Get String Descriptor	Index=1 Length=255
05	00	Set Configuration	Configuration=1

Figure 2.5: A USB protocol analyzer was used to capture this communication during the enumeration of a device in Linux. The second device descriptor request has been fully expanded in order to emphasize the underlying transactions and packets that comprise a control transfer.

Chapter 3

Device Hardware

One of the primary components of this independent study is the development of a USB device. Although semiconductor manufacturers typically offer development boards to aid in the rapid development of products based on their microcontrollers, it was desired to design a more permanent, custom solution. This choice was made in an effort to gain further experience with hardware design and to impart a greater greater sense of accomplishment in the overall project.

The hardware design revolves around a Microchip PIC18F4550 microcontroller. This is a low cost 8-bit microcontroller that contains internal program memory and data memory (it utilizes a modified Harvard architecture), numerous digital I/O pins, a USART (universal synchronous/asynchronous receiver/transmitter) peripheral, support for in-circuit debugging, and, most importantly for this project, an internal USB transceiver module. A more detailed table of device characteristics is provided in Table 3.1. Another interesting feature of the microcontroller is that it contains phase-locked loop (PLL) circuitry that allows the microcontroller core clock to potentially run at a different rate than the peripherals or the external clock. This is useful because the USB transceiver specifically requires a 48 MHz clock source. A much lower speed 4 MHz crystal oscillator is used on the board in order to prevent high frequency interference on the printed circuit board (PCB) traces, and the microcontroller clocking system internally steps this up to the required frequency for the USB peripheral. Therefore, in this device, both the peripherals and the core operate at the full 48 MHz, while requiring only a 4 MHz external clock source.

As this project is intended to clearly demonstrate that bidirectional USB communication between the device and host is achieved, the hardware platform must contain some input and output interfaces. In order to make the hardware extensible and reusable for other purposes, the design includes an array of six pushbutton switches as well as a graphical, monochromatic, 128×64 pixel LCD display. These user interface components are generic enough to be used in any number of applications, either in conjunction with a USB connection or as a standalone device.

In general, mechanical buttons suffer from a problem known as bounce. When a button is pressed, it is common for the electrical connection made inside the button to have an unstable transient response due to the minute mechanical perturbations at the point of contact. This

Feature	Specification
Architecture	8-bit RISC
SRAM (data memory)	2 KB
Flash ROM (program memory)	32 KB
Max. Clock Rate	48 MHz
Timers	4
Capture/Compare/PWM Modules	2
Analog-to-Digital Converter	10-bit, 13 channels
USART Peripherals	1
USB Module	Low/Full speed support
I/O Pins	35
Operating Voltage	2.0 to 5.5V
Package	40 pin PDIP
Volume Pricing	<$4 each

Table 3.1: The PIC18F4550 microcontroller has many useful features.

"bouncing" can cause logic to be falsely triggered multiple times, as shown in Figure 3.1. Fortunately, however, there are several ways to counter this problem in either hardware or software. For this device, a hardware approach is used. The output of each switch is fed into a resistor/capacitor network which causes a slow, time-dependent rise in the voltage at the output of the switch when the switch is opened. When the switch is closed, the capacitor is shorted, causing the voltage to quickly drop back to zero. Thus, rapid fluctuations in the input are effectively filtered out since the voltage cannot rise beyond the logic threshold in such a short amount of time. The filtered output is then routed through Schmitt-trigger inverters, which change the output to active-high signaling and also provide hysteresis to counter bounces that might occur near the logic threshold.

The device also exposes two header connectors. One header is connected to the microcontroller's USART pins, and may be used to output debug information to a PC. An external TTL-to-RS232 converter is required to facilitate the voltage level shifting between the two interfaces. The other header provides access to the microcontroller's programming pins. These are used in conjunction with a programmer device in order to upload firmware to the microcontroller's flash memory. In addition to providing programming access, this header can also support in-circuit debugging, which enables the developer to set breakpoints or step through firmware code on the microcontroller. The Microchip Pickit2 was used as the programmer/debugger throughout the development of this project. Thus, the device is designed with debugging abilities in mind. The complete circuit schematic for the device is shown in Figure 3.2.

Instead of using a breadboard or prototyping board on which to assemble the circuit, a more permanent and sturdy solution was desired. Thus, the Cadsoft Eagle software was used to create a schematic capture of the circuit and to develop a PCB layout for the device.

The layout diagram is shown in Figure 3.3. Although the CAD designs from this software are suitable for submission for professional fabrication, such services are prohibitively expensive for low quantity production. Thus, the PCB fabrication was instead performed using hobbyist techniques. First, the PCB layer containing the traces is printed onto specially designed toner transfer paper using a laser printer. Then, using a household iron, the toner is transferred from the paper onto a blank copper clad circuit board. This effectively masks the only the copper directly underneath the toner traces. The board is then saturated in a ferric chloride acid bath to etch away the unmasked regions of copper, leaving behind only the copper traces underneath the toner. The board can then be rinsed, and the underlying copper exposed by sanding away the toner. Once this process is complete, a drill press is used to create holes for all of the device leads. Finally, all of the components are placed onto the board and soldered. The entire process is inexpensive, but extremely labor-intensive.

All of the digital components on the device (the microcontroller, the LCD, and the Schmitt-trigger inverters) are compatible with 5 V logic levels, which means that the device can be operated from a single voltage source without requiring additional power supply circuitry. Conveniently, 5 V power is provided over the USB cabling, as is used to power the device. USB devices that operate solely from USB power are known as bus-powered devices. Devices that have a dedicated power supply are known as self-powered devices. It is also possible for a device to support multiple USB configurations (recall that the configuration descriptor contains power consumption information) to handle both modes. This USB device has been measured to draw approximately 50 mA with the LCD backlight off, and 450 mA with the backlight on. Thus, the device may be safely bus-powered, so long as the backlight is not switched on until the device is configured (its configuration descriptor requests a 500 mA maximum current from the bus).

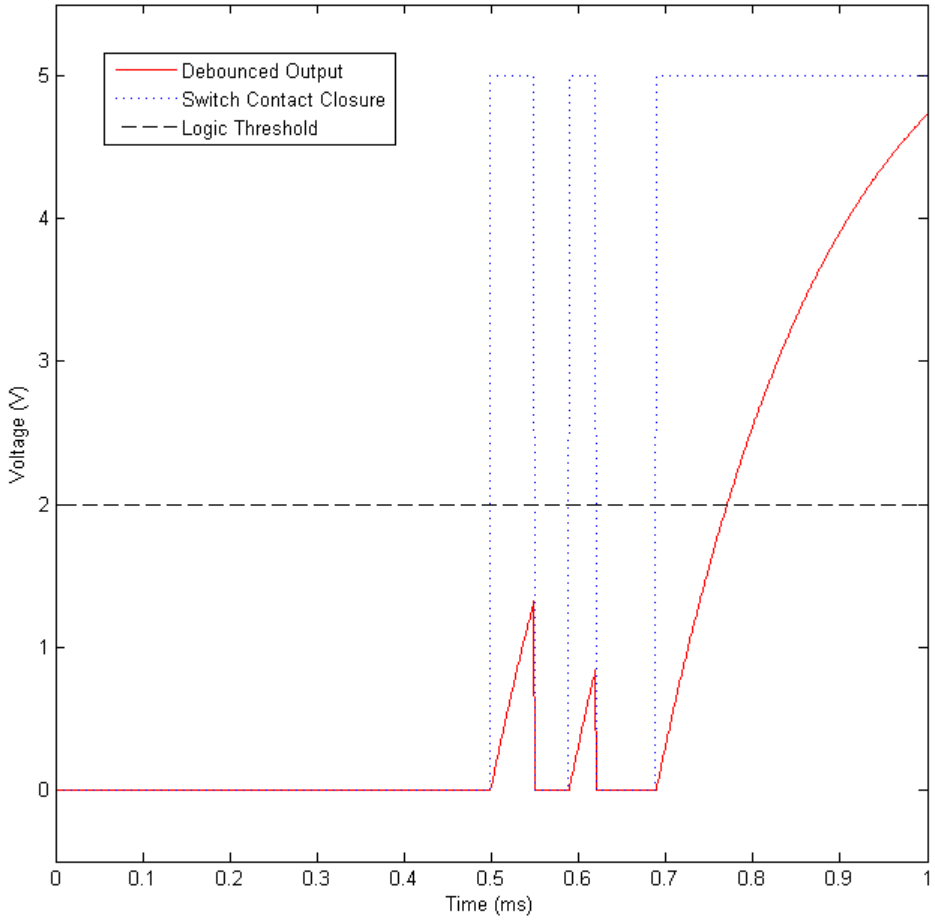

Figure 3.1: This plot illustrates the effect of switch bouncing and filtered output using an RC network.

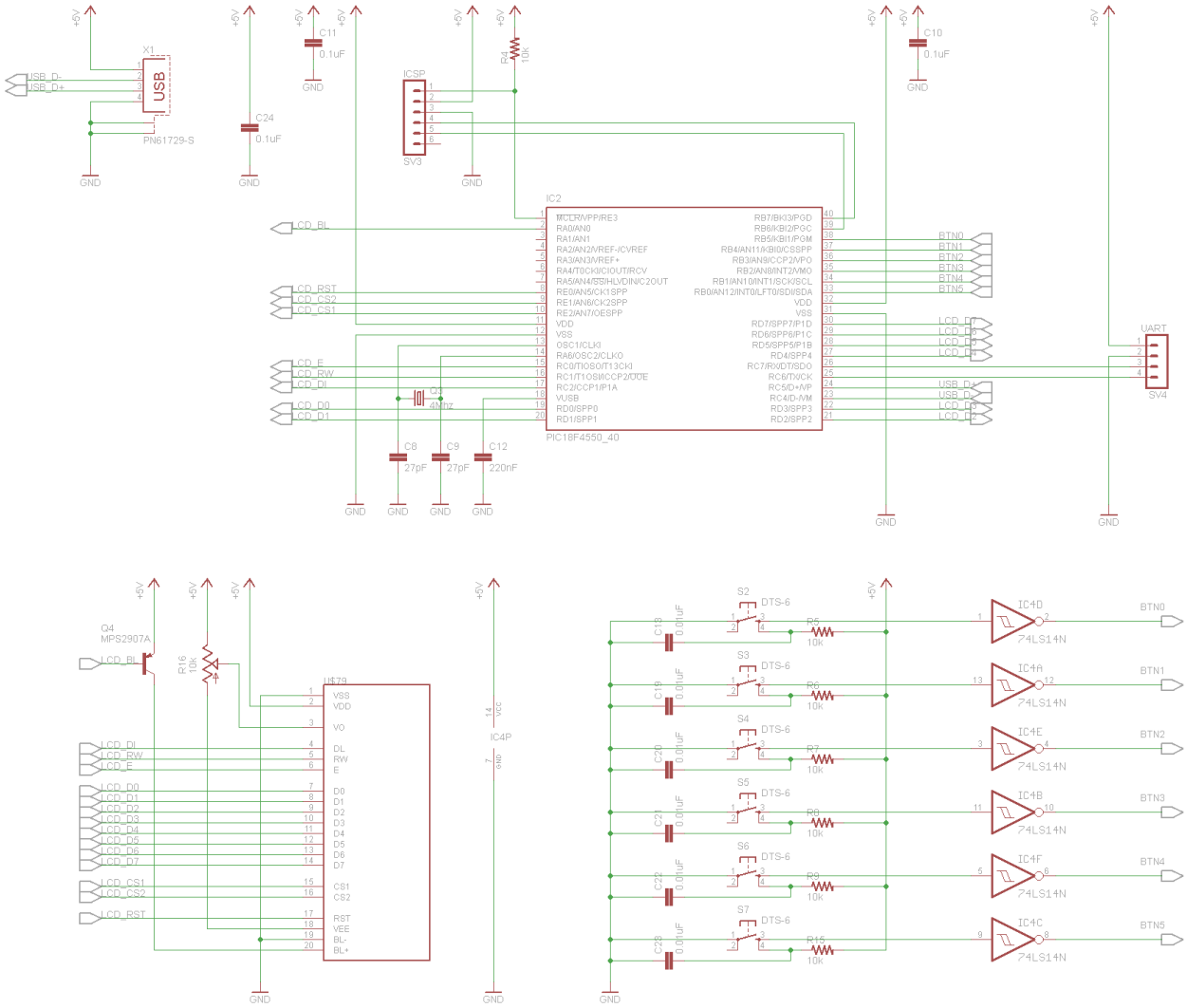

Figure 3.2: The circuit diagram depicting all of the hardware and connections used to implement the device.

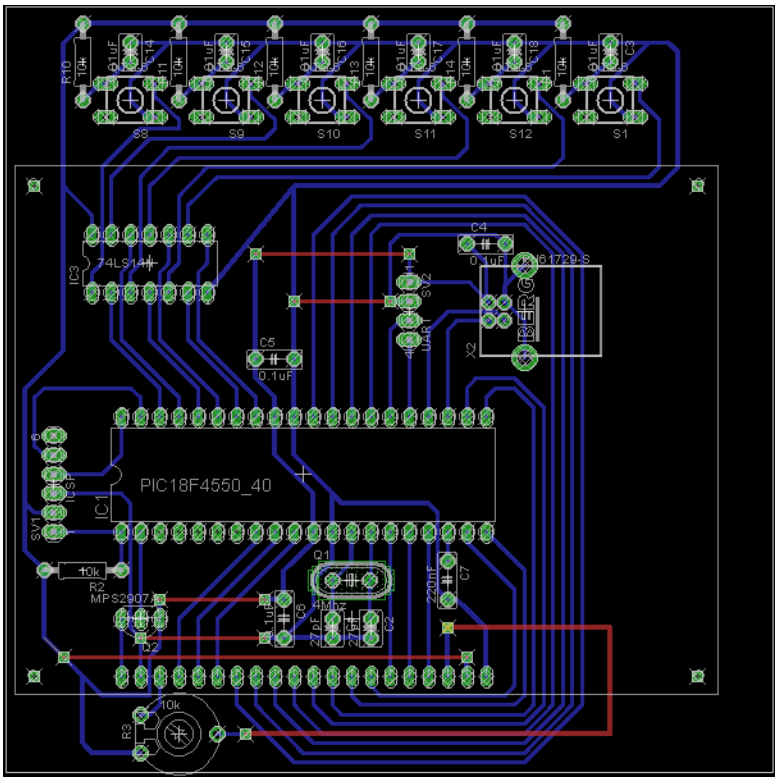

Figure 3.3: This screenshot illustrates the component placement and traces for the PCB layout.

Chapter 4

Device Firmware

The majority of the effort of this project has been focused on the device firmware. The firmware is the software that gets uploaded into the microcontroller's flash ROM and is designed to coordinate interfacing with the hardware components. For this particular device, the firmware is responsible for driving the LCD display, managing USB communication by implementing a minimal USB stack, operating the USART peripheral for debugging output, and coordinating these activities in a sensible manner to achieve the desired end operation.

4.1 LCD Driving

One of the major responsibilities of the firmware is to control the LCD display. The LCD module contains an 8-bit parallel data bus and several control pins that are used to interface with the display by transferring commands and data. Internally, the LCD module actually contains two controller chips that each control a 64×64 pixel region of the LCD. Specifically, chip 1 controls the left half of the screen and chip 2 controls the right half. Both of these chips share the same control and data pins on the LCD module, so if a specific chip is to be targeted for a particular operation, its respective chip select pin (denoted `CS1` or `CS2` in the code) must be asserted. It is possible to improve performance by asserting both chip select lines when writing the same data to both chips simultaneously, as when clearing the screen. The LCD module contains an internal frame buffer that contains the contents of the display. The `RW` pin, selects whether data is being read from or written to this frame buffer. Alternatively, instead of sending display data to the LCD, the module can also interpret a limited number of commands that instruct the LCD to execute a particular operation. The `DI` pin is asserted appropriately to indicate whether the data is to be interpreted as display data or instruction data. Lastly, an enable pin (simply denoted as `E`) is used to actually latch the data from the bus into the LCD module.

To perform a transfer to the LCD, the appropriate chip select and control lines must be asserted. The appropriate data byte is then written to the parallel data port. Finally, the enable pin is asserted momentarily, and then deasserted. Unfortunately, the timing characteristics for the LCD require that the enable pin is toggled at a much slower rate than

the microcontroller instruction clock, so busy waits must be used to allow for ample time between pin transitions. An example of writing a command to the LCD is shown in Listing 4.1. Note that a precondition for using this function is that the targeted chip select lines are already asserted.

```
1  void glcd_write_command(uint8_t command)
2  {
3      // Set the data direction for writing
4      GLCD_DATA_TRIS = TRIS_ALL_OUTPUT;
5      GLCD_RW_PIN = GLCD_RW_WRITE;
6
7      // Set the data type to instruction
8      GLCD_DI_PIN = GLCD_DI_INSTRUCTION;
9
10     // Set the data port value and clock it in
11     GLCD_DATA_LAT = command;
12
13     // Pulse the enable pin
14     GLCD_E_PIN = 0;
15     glcd_delay_us(ENABLE_PULSE_US);
16     GLCD_E_PIN = 1;
17     glcd_delay_us(ENABLE_PULSE_US);
18     GLCD_E_PIN = 0;
19     glcd_delay_us(ENABLE_PULSE_US);
20 }
```

Listing 4.1: This code performs the appropriate sequence of operations to write a provided command byte to the LCD screen.

The use of commands is very important to the operation of the LCD. The commands perform such operations as turning the screen on and off and specifying the screen location to which to write data. The format for these commands is shown in Table 4.1. Understanding the geometry of how display data is mapped to a particular location on the screen is not straightforward from this table. Thus, a diagram that explains the significance of the location fields is shown in Figure 4.1. The screen is divided into vertical strips of 8 pixels, each of which is represented by a data byte in the LCD's frame buffer. The location of this strip is determined by a page value which specifies the vertical offset in steps of 8 pixels, and a column address. These fields can be individually set using the different command messages.

Although the microcontroller contains 2 kB of RAM, the memory region is divided such that there is not a 1 kB contiguous memory region available in which to store a local copy of the frame buffer. This has the unfortunate consequence that read-modify-write operations (i.e., operations that require require reading a byte, setting or clearing individual bits, and writing the byte back) cannot be supported locally on the microcontroller. Performing such operations would instead require either significant communication with the LCD module to read from it's internal frame buffer, or maintaining the frame buffer at a higher level in the system. The latter approach has been adopted for this device. The firmware therefore

INSTRUCTION	D/I	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	DESCRIPTION
Display ON/OFF	0	0	0	0	1	1	1	1	1	1	Controls the display on or off. Display RAM data and internal status is not affected. 0: OFF. 1:ON
Set Address	0	0	0	1	Y address (0~63)						Sets the Y address at the Y address counter.
Set Page (X address)	0	0	1	0	1	1	1	Page (0~7)			Sets the X address at the X address register.
Display Start Line	0	0	1	1	Display start line (0~63)						Indicates the display data RAM displayed at the top of the screen.
Status Read	0	1	BUSY	0	ON/OFF	RESET	0	0	0	0	Read status: BUSY 0:Ready 1:In operation ON/OFF 0:Display ON 1:Display OFF RESET 0:Normal 1:Reset
Write Display Data	1	0	Write Data								Writes data DB0~DB7 into display data RAM. After writing instruction, Y address is increased by 1 automatically.
Read Display Data	1	1	Read Data								Reads data DB0~DB7 from display data RAM to the data bus.

Table 4.1: This table describes the format of the data and the appropriate control line settings for reading and writing to the LCD display module. [2]

only exposes low level functionality for reading and writing command and data bytes, and cannot efficiently support higher-level operations such as drawing lines (which would require read-modify-write operations for a packed frame buffer).

4.2 USB Stack

The most challenging aspect of this project was the development of the USB stack used to support the device end of the USB communication. This task was particularly difficult because it required a very thorough understanding of both the USB protocol and the operation of the USB hardware built into the microcontroller. As the relevant aspects of the USB protocol have already been covered, this section will concentrate on the firmware logic and the USB module interface.

It is worthwhile to note that Microchip does provide a firmware library to support USB communication on their microcontrollers. While this library is very robust and full-featured, it is also very bulky. The library is intended to work with all of Microchip's microcontrollers spanning different architectures, compilers, and feature sets. The library also includes support layers for several common USB device classes [8]. Overall, the library spans dozens of files and is fairly difficult to use. On the opposite end of the spectrum, Dr. Bradley Minch of the Franklin W. Olin College of Engineering has released a small collection of minimalist USB device examples for use in academic lab exercises. Though lightweight (consisting of a single .c and .h file), these firmwares are tightly coupled with the end applications, and support only very specific USB transfers [9]. While they provide a good starting point for

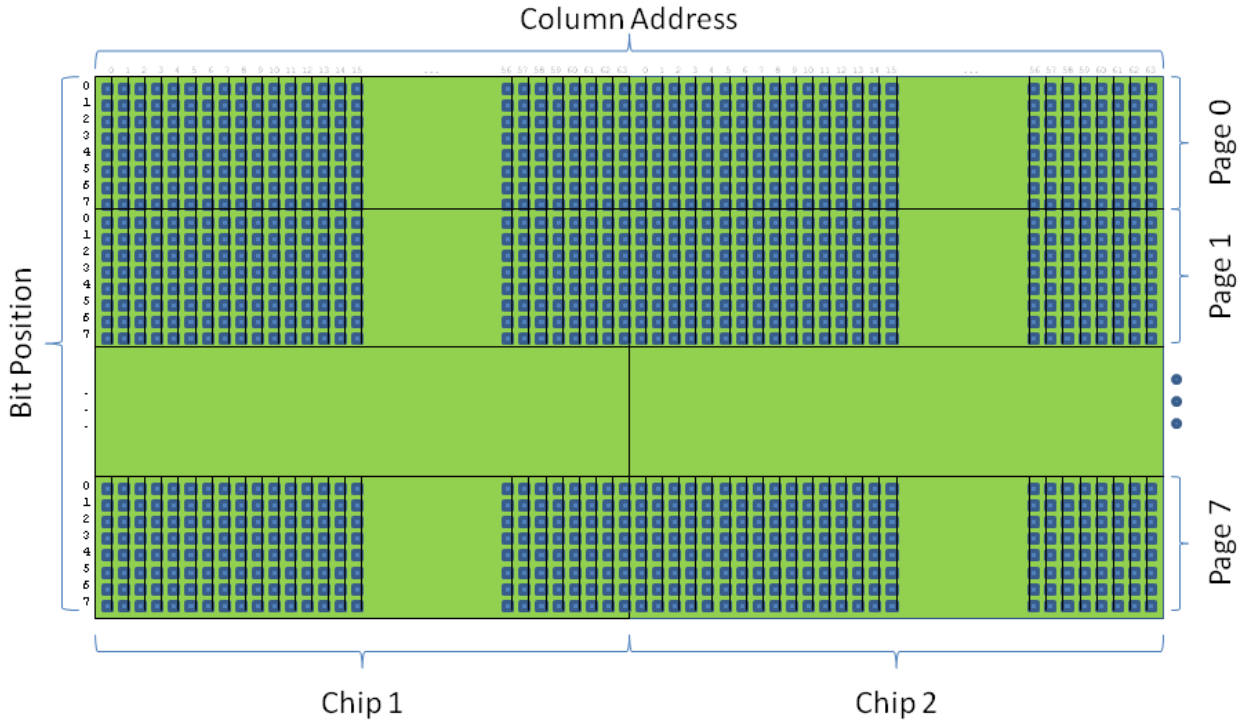

Figure 4.1: This diagram explains the addressing scheme used to read or write display data to particular locations on the LCD screen.

understanding how to use the PIC18F4550's USB hardware, they are clearly not intended to be extensible toward real applications.

The USB stack developed in this firmware is designed to fill the void in the middle of the aforementioned spectrum, and borrows inspiration from both sets of firmware. In particular, the USB stack itself is lightweight, consisting only of `usb.c` and `usb.h` files. These files contain no device-specific code or data structures, and can therefore be used in any number of applications without requiring any changes to the stack code. Because the USB stack does require access to some device-specific data structures and variables (in particular, descriptors), these symbols are implemented in separate files, `usb_params.c` and `usb_params.h`, and `extern`'ed to the USB stack. This effectively decouples device-specific USB paramaters from the USB stack code. Furthermore, the stack supports most transfer types, and can utilize all 32 USB endpoints. The limitations to this stack are that there is no support for isochronous transfers, and no support for vendor-specific or class-specific control transfers on endpoint 0. Furthermore, the stack is specifically targeted at the PIC18F4550 device.

One of the first steps in developing the USB firmware is to create data structures and macros for the different descriptors and requests used in the USB protocol. By carefully mapping these structures to new data types, the individual fields may be accessed and modified very easily. Listing 4.2 shows an example of the `typedef`'s used to represent the

a SETUP data packet. It is important to note that the USB protocol uses little endian byte ordering for multi-byte fields. Fortunately, the Microchip compiler used for this project also uses little endian byte ordering (the PIC hardware itself does not have an associated endianness since the native operand size is a single byte; the compiler is responsible for providing the abstraction of multi-byte types), which makes the mapping process fairly straightforward. For clarity, the field names (including case conventions) used in these data structures are taken directly from the USB specification [6].

```
1  // Represents the bmRequestType field of the setup data packet
2  typedef struct
3  {
4      uint8_t recipient       : 5;
5      uint8_t type            : 2;
6      uint8_t direction       : 1;
7  } setup_request_t;
8
9  // Setup data packet format
10 typedef struct
11 {
12     setup_request_t bmRequestType;
13     uint8_t bRequest;
14     union
15     {
16         uint16_t value;
17         struct
18         {
19             uint8_t descriptor_index;
20             uint8_t descriptor_type;
21         };
22     } wValue;
23     uint16_t wIndex;
24     uint16_t wLength;
25 } setup_data_t;
```

Listing 4.2: All important USB data structures are carefully mapped to `struct`'s to make field access simple. This example shows the layout of a SETUP request, as outlined previously in Table 2.2.

To develop the USB stack firmware, it is important to have a thorough understanding of the microcontroller's built-in USB peripheral hardware. This hardware provides basic physical and data link layer functions. Specifically, it abstracts all bus communication timing, medium access control (MAC), error detection, and packetization of payload data. However, it is still up to the developer to decide how to handle errors, when and how to set the device address, maintain the data toggle state, and determine the appropriate PID for the handshake packet, if used. Of course, more importantly, it is the responsibility of the developer to decide how to process the data in each transaction (which is particularly important for

control transfers) and to "arm" each endpoint as necessary.

The interaction with the USB hardware is not trivial. In fact, the USB peripheral has 22 dedicated registers as well as a bank of virtual registers (4 for each endpoint in use) in the USB RAM region. Furthermore, the USB module has its own interrupt subsystem that contains 12 interrupt sources. Instead of presenting the operation of each of these registers and components up front, they will be explained as encountered during a chronological example of a USB control transfer (because of their complexity, control transfers exercise the majority of the USB stack). A more detailed account of all of these registers may be found in [8].

Before any USB communication can take place, the USB module must be initialized with several configuration settings. The `UCON` register contains a bit, `USBEN`, that enables and disables the USB module hardware. This bit should be cleared prior to performing any configuration in order to ensure that the USB module is not active while attempting change settings. Similarly, the USB interrupts should be masked and all USB interrupt flags cleared so that the execution does not immediately jump to ISRs once the module is enabled.

The initialization of the USB module also involves setting up the `UCFC` USB configuration register. Most importantly, this register contains bits to control the pullup resistors on the bus lines. The pullup state of the bus lines is used to signal to the host which speed mode the device is using. The register also contains bits to configure whether ping-pong buffering is used on the endpoints. Ping-pong buffering is simply a double-buffering scheme whereby the USB hardware is allowed to modify one buffer while the user application operates on the other. The two buffers are alternated after each transaction. This scheme is typically employed in high throughput applications so that the hardware and firmware do not have to wait for one another to pass a shared buffer back and forth. For simplicity, this device does not utilize ping-pong buffering.

The `UADDR` register contains the bus address to which the device is configured to respond. This register is used by the USB hardware to determine which bus traffic to ignore or acknowledge. During initialization, this register should be set to 0 since the device starts in an unconfigured state and has not yet been assigned an address via a `SET_ADDRESS` request.

The initialization routine also clears out a region of USB data memory called the buffer descriptor table (BDT). The BDT is an interesting data structure that will be described later, but for now it is sufficient to note that zeroing out this memory region has the effect of resetting a set of virtual registers to reasonable initialized values.

The `UCON` register mentioned earlier also has bits to put the USB module into a low power suspend mode, to enable or disable packet reception, and to determine whether the bus data lines are in a single-ended zero (SE0) condition. Now that the rest of the module has been initialized, the USB module may be enabled by enabling packet reception, disabling suspend mode, and setting the `USBEN` bit. SE0 is normally an illegal condition (since the bus uses differential signaling) where both data lines are at a low voltage level, but is an unavoidable transient when the module is first powered up. The firmware should wait for this condition to subside after enabling the USB module and before responding to any interrupt sources. Finally, the USB reset interrupt should be unmasked so that the device can respond to a

reset from the host.

Once the USB hardware is enabled and the SE0 condition has cleared, the host should be able to detect the presence of the device on the bus (due to the enabled pullups) and may begin to communicate. USB interrupts should now be enabled in order to respond to any such events from the host. From this point forward, the vast majority of the USB operation is event-driven through interrupts. All of the possible USB interrupt sources are shown in Figure 4.2. At the top level, the USB module actually only has a single interrupt source, denoted by the `USBIF` bit. However this flag may be set by any one of the numerous other sources, provided the corresponding interrupt enable bit is also set. In the PIC architecture, there is only a single interrupt vector (actually, it is possible to enable two interrupt vectors for dual priority interrupts, but this scheme is not used for this project), so the ISR must scan all of the relevant flags and call appropriate functions to handle each event.

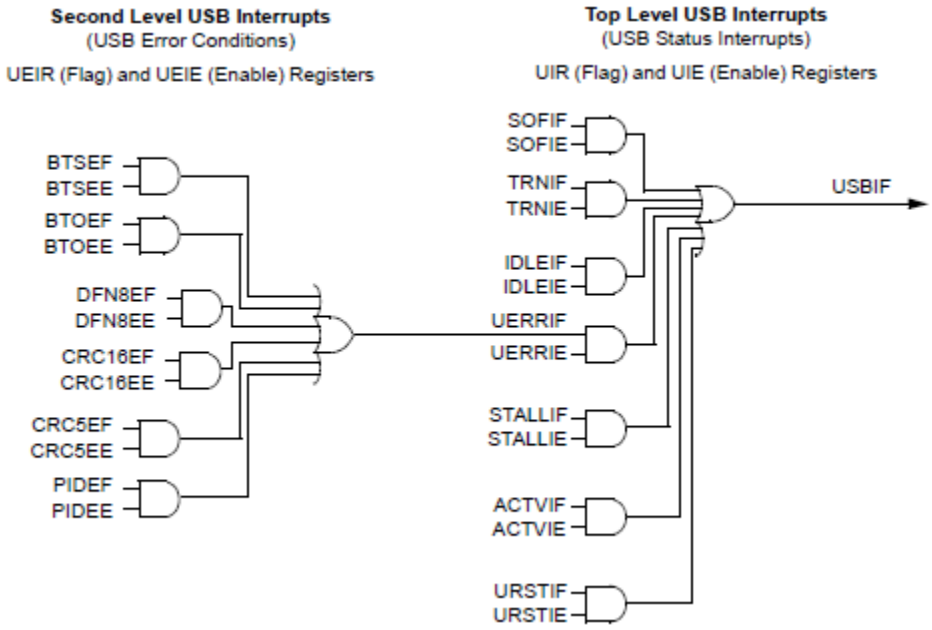

Figure 4.2: This diagram shows each interrupt source flag bit and how those events are funneled into higher-level interrupt flags [8].

Observing Figure 4.2, there are several error interrupts on the left of the diagram, including bit stuffing errors, CRC errors, and data size errors. Fortunately, these error interrupt sources may be ignored in most applications. If an error is encountered, the USB module simply forgoes the handshake packet back to the host, and the host will retry. The USB module does not make the application aware of such erroneous packets, and only passes error-free packets to the firmware. This abstracts the need to explicitly handle these error sources. These error interrupts are therefore only of interest in applications which need to gather statistics about the quality of the underlying link.

The interrupt sources on the right tier are typically of greater interest. The SOFIF flag indicates that a start-of-frame (SOF) token was encountered on the bus. Usually, no action needs to be taken for this event, but some devices might use this as a timing reference (particularly for isochronous transfers), since a correctly-operating host should be sending an SOF token to Full-Speed devices every 1 ms. The TRNIF flag indicates that a transaction has completed, and typically is used to prepare for the next transaction. The IDLEIF flag is set when there is no bus activity (from SOFs or other traffic) for a period of 3 ms or more. The USB specification mandates that, when this happens, any bus-powered devices must enter a low-power state with a maximum current draw of 2.5 mA. Unfortunately, the hardware design of this device violates this power ceiling, and is therefore not fully USB compliant. However, an idle bus is almost never encountered on desktop PCs, and exists mostly for mobile hosts in low-power modes. The STALLIF flag indicates that an endpoint is in a stall condition, and is not particularly useful since the developer determines when an endpoint should stall anyway. The ACTVIF flag indicates when there is bus activity and is primarily used to wake up a suspended device. Lastly, the URSTIF flag indicates that the host has issued a device reset by setting both data lines low. When an interrupt is serviced, its corresponding flag bit should be cleared. The TRNIF and URSTIF interrupts are the most important and will be examined in further detail.

After having initialized the USB module and enabling interrupts, the first event that occurs is usually a reset from the host, thereby triggering the URSTIF interrupt. Note that, while the host is known to reset the device during the enumeration process, hosts are actually permitted to reset a device at any time, so the reset handler should make no assumptions about the state of the device. The reset handler must do a few things. First, all endpoints must be disabled except for endpoint 0. The 16 endpoint pairs can be enabled, disabled, and configured via registers UEP0 - UEP15. The device address must also be reset to 0 through the UADDR register. Since a reset condition is basically designed to reinitialize the USB communication, all pending interrupt flags should also be cleared. Most importantly, endpoint 0 OUT must be "armed" to handle OUT or SETUP transactions from the host. An explanation of what it means to "arm" an endpoint will follow shortly. Now that the device has been reset, the state variable can be advanced to the Default state, and all USB interrupts can be unmasked.

The next interrupt that it likely to follow a reset is a TRNIF, which denotes that a transaction has completed. In order to determine which endpoint number and direction the transaction occurred on, the USTAT register must be read. Actually, the USTAT register reads the head of a FIFO which queues the status of up to four transactions. The FIFO is automatically advanced when the TRNIF interrupt flag is cleared [3]. In the case of the completion of a transaction on an IN endpoint, the TRNIF handler should arm the endpoint with additional data to send to the host in order to prepare for the next IN transaction (if one is expected). If the completed transaction was on an OUT endpoint, then data was received from the host. The data should be read and processed appropriately, and the endpoint should be rearmed to prepare for future OUT or SETUP transactions. The details of reading to, writing from, and arming endpoints requires a digression that explains the use

of the buffer descriptor table (BDT).

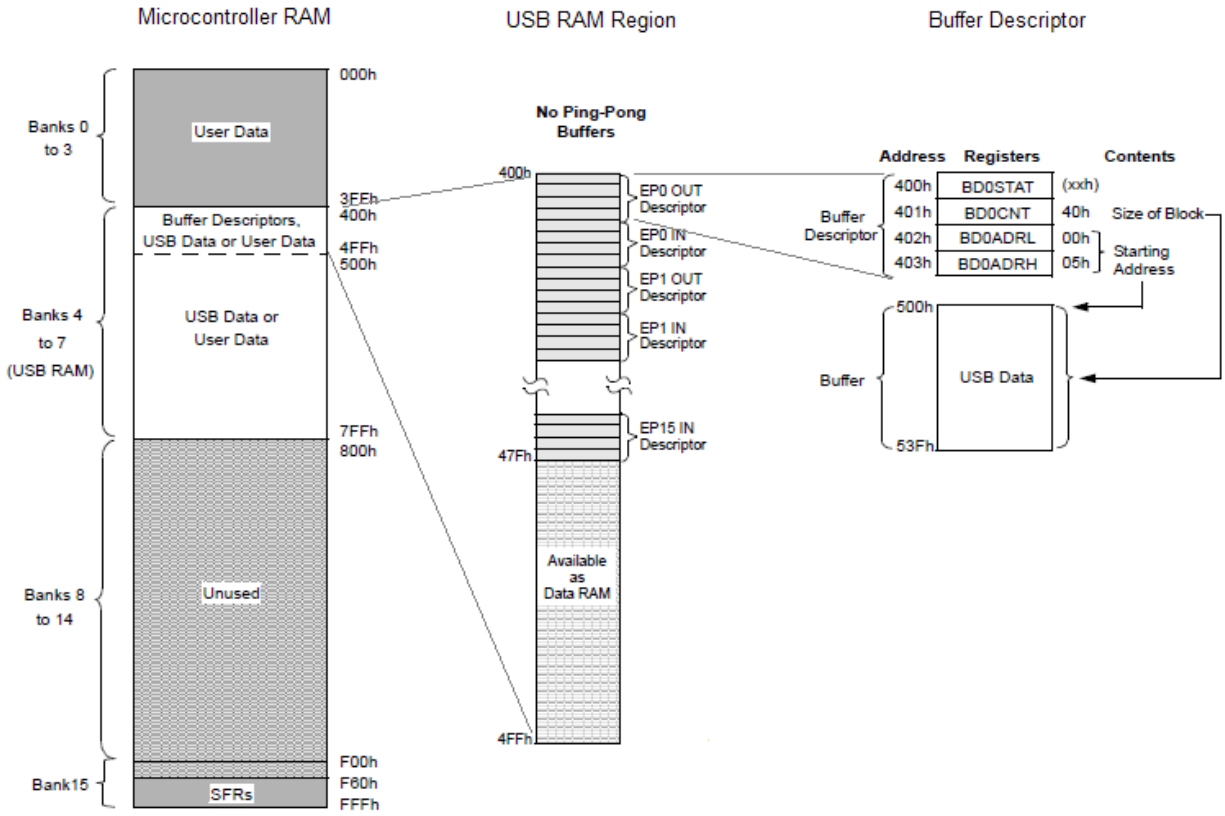

Figure 4.3: This diagram shows how the PIC18F4550 data memory is segmented and how buffer descriptors are laid out in the USB memory region [8].

The BDT is a table of virtual registers at the beginning of the USB RAM region that provide important information about transactions for each endpoint. The BDT is a facet of this particular USB hardware and is not part of the USB specification. Figure 4.3 shows the memory layout of the microcontroller. The center portion of the figure shows an expanded view of the USB RAM region. At the start of this memory region are groups of four 8-bit registers, with one such group allocated for each endpoint. Each group is called a buffer descriptor, and the collection of all of the buffer descriptors is called the BDT. The rightmost portion of the figure shows a single buffer descriptor in detail. The buffer descriptor has four registers: `BDnSTAT`, `BDnCNT`, `BDnADRL`, and `BDnADRH` (where n is the number of the endpoint).

Each buffer descriptor is shared between the USB hardware and the firmware. Consider, for example, an IN endpoint, whose purpose is to send data from the device to the host. For this to happen, the firmware must first fill a buffer with the data to be sent and then indicate to the USB hardware that the buffer is ready. The USB hardware then takes control over the buffer and sends the data to the host upon receiving a token packet with an IN PID.

Once the data has been sent, the USB hardware returns control of the buffer to the firmware, and the cycle can repeat. In the case of an OUT endpoint, the firmware is responsible for providing a buffer location in which to place received data from the host. Once this location has been set, the firmware must signal the hardware to take control over the buffer. When a token packet with an OUT (or SETUP) PID is received, the USB hardware is ready to place the data from the following data packet into the provided buffer. Once the transaction is complete, the hardware returns control of the buffer to the firmware, and the cycle can repeat.

In order for this system to operate correctly, it is essential that the firmware does not try to access the buffer when the hardware has ownership of the buffer, and that the hardware does not try to access the buffer when the firmware has ownership of the buffer. In order to enforce this, the `UOWN` bit in each buffer descriptor's `BDnSTAT` register is used to coordinate access to the buffer descriptor. When the `UOWN` bit is 0, the buffer descriptor is owned by the firmware, and the firmware can safely access any buffer descriptor registers as well as the buffer itself. When the `UOWN` bit is 1, the buffer descriptor is owned by the hardware, and the endpoint is considered to be "armed" for sending or receiving. The hardware can only clear the `UOWN` bit, and can therefore never steal a buffer descriptor from the firmware. Similarly, the developer should take care only to set the `UOWN` bit so as not to steal a buffer descriptor from the USB hardware (although the firmware could potentially clear `UOWN` in order to cancel a pending transaction). Note that the assertion of the `TRNIF` flag occurs when the USB hardware relinquishes control of the buffer after a successful transaction, so the firmware does not need to explicitly check the `UOWN` bit before accessing the buffer in the `TRNIF` handler.

In order to prepare an endpoint for a transaction, several steps must be taken. First, the `BDnADRL` and `BDnADRH` register pair must be set to point to the memory address of the buffer containing the payload data. It is essential that this buffer be completely contained within the USB RAM region and not overlap with any other active buffers or the buffer descriptor table (certain `#pragma` directives can be used with the compiler to explicitly map these buffers to a specific memory location). The `BDnCNT` register should be loaded with the number of bytes to send to the host (for IN endpoints) or the maximum number of bytes to retrieve from the host (for OUT endpoints) during a transaction. Finally, the `BDnSTAT` register must be configured. This register contains flags that allow the firmware to stall the endpoint, provide the expected data toggle value, and enable or disable handshaking on the endpoint. If the `BDnSTAT` register is not assigned to atomically (i.e., if specific bits are masked and assigned independently), then it is important that the `UOWN` bit be set last so that the hardware does not take control of the buffer descriptor before it has been fully configured. The code used to arm an endpoint is presented in Listing 4.3. The register names used in the code are slightly different than those used in the datasheet and presented throughout this section, but the mapping is obvious.

```
1 void arm_endpoint(buffer_descriptor_t *bd, uint8_t *data, uint16_t size,
2         uint8_t stall)
```

```
3  {
4      // Specify the size and source of the data
5      bd->data = data;
6      bd->status.count_h = HIGH_BYTE(size);
7      bd->count = LOW_BYTE(size);
8
9      // Set up other endpoint options
10     bd->status.bstall = stall;
11     bd->status.dts = !(bd->status.dts);
12     bd->status.ken = 0;
13     bd->status.incdis = 0;
14     bd->status.dtsen = 1;
15
16     // Turn control of the buffer descriptor back over to the USB module
17     bd->status.uown = 1;
18 }
```

Listing 4.3: This code demonstrates the use of all of the buffer descriptor registers in order to arm the endpoint.

The `BDnSTAT` register has different meaning when being read from, and provides information about the last transaction that occurred on that endpoint. Specifically, it has a field containing the PID of the token packet, which must be used to determine whether the data packet was IN, OUT, or SETUP. Listing 4.4 demonstrates how the `USTAT` register, the BDT, and the buffer descriptor's corresponding `BDnSTAT` register are used to make a high-level decision about how the last transaction should be processed. First, the USTAT FIFO is read to determine which endpoint the transaction occurred on. The endpoint information is used as an index into the BDT, where further information is available. Since there is now a local variable containing endpoint information, the `USTAT` register is no longer needed, and FIFO may be advanced by clearing the `TRNIF` flag (it is a good idea to do this as soon as possible to prevent FIFO overrun). If the endpoint number was determined to be 0, then the transaction is a control transfer on the default endpoint, and must be handled by the stack firmware. The token packet's PID is then read from the buffer descriptor's `BDnSTAT` to determine whether to process the transaction as IN, OUT, or SETUP.

```
1  void handle_transaction(void)
2  {
3      uint8_t endpoint;
4      buffer_descriptor_t *bd;
5
6      // Locate the appropriate buffer descriptor
7      endpoint = (USTAT & USTAT_ENDPOINT_MASK) >> USTAT_ENDPOINT_SHIFT;
8      bd = &BDT[endpoint].direction[USTATbits.DIR];
9
10     // Clearing the interrupt flag advances the USTAT FIFO
11     UIRbits.TRNIF = 0;
12
```

```
13      // If this transaction was directed at the default control endpoint
14      if(endpoint == 0)
15      {
16          // Determine what to do based on the PID
17          switch(bd->status.pid)
18          {
19              case PID_SETUP:
20                  process_setup_transaction();
21                  break;
22
23              case PID_IN:
24                  process_in_transaction();
25                  break;
26
27              case PID_OUT:
28                  process_out_transaction();
29                  break;
30
31              default:
32                  break;
33          }
34      }
35  }
```

Listing 4.4: This code demonstrates the simplicity and modular design of the `TRNIF` handler. The bulk of the processing is performed in the other functions that this calls.

SETUP transactions are the most difficult to deal with, as they require many special considerations. Because SETUP transactions denote the start of a new transfer, any variables that are used to keep track of an ongoing control transfer should be cleared. Also, since many of the fields in the data packet are important throughout the entire transfer, the packet should be copied into memory so that it may be referenced while preparing subsequent transactions. Once the packet has been copied, the endpoint can be immediately rearmed so that the USB hardware can accept additional transactions. The USB hardware automatically disables packet processing upon receiving a SETUP packet, so the firmware must explicitly re-enable it. Next, the fields of the request structure must be observed to determine what to do.

As an example, assume that this SETUP transaction initiates a transfer to get a descriptor from the device. The `bmRequestType` field must be parsed to determine whether this is a standard request, class request, or vendor request. In this example, the request is a standard request (this firmware stack was designed to handle only standard requests anyway). The `bRequest` field is checked to determine which type of standard request was issued: in this case, `GET_DESCRIPTOR`. The `wValue` field of the copied request packet is then used to determine which type of descriptor and which index was requested. This information is used to retrieve the appropriate descriptor from ROM and set up some persistent variables that will be used to keep track of the control transfer. In particular, it is important to keep track of a pointer to the data to be transferred, whether the pointer is to ROM or RAM, the

number of bytes remaining in the transfer, and whether an error is encountered. If the host requests fewer bytes than the actual size of the data, the firmware must adjust the "bytes remaining" counter to only send as much data as the host requests (it is a violation to try to send more). A helper function is called to copy up to the maximum packet size worth of data into the USB endpoint buffer memory, update the transfer pointer and byte counter to reflect this packet, and arm the endpoint.

Once the endpoint is armed, the data will be sent to the host during the IN transaction in the data stage that follows. The IN transaction also triggers the handler for IN transactions, which will in turn check the state variables to see if the transfer is complete and call the helper function to prepare the next packet of data if necessary. In the special case that the IN handler was called due to entering the Status stage of a control transfer for a `SET_ADDRESS` request, the `UADDR` register is updated to reflect the newly assigned address.

For this stack implementation, processing an OUT transaction on the default endpoint is trivial because none of the supported standard requests actually use an OUT data stage in their control transfers. Instead, all of the host-to-device standard requests supported by this firmware actually send any necessary data in the request packet during the Setup stage of the transfer. The processing for OUT transactions therefore consists of nothing more than re-arming the endpoint so that it may receive further OUT or SETUP transactions.

The internal operations of the firmware stack are complicated, but the external interface to the developer is surprisingly simple. Only five functions (excluding the initialization function) are required to perform USB communication. The `usb_get_state()` function simply returns the state (Reset, Address, Configured, etc.) of the device, and is necessary in order to ensure that the device is Configured before enabling user endpoints. The `usb_use_endpoint()` function zeros out the specified endpoint's region of the BDT, initializes the data toggle value in the `BDnSTAT` register, and configures the `UEPn` register to set up the endpoint according to the user's options (endpoint direction(s) and handshake options). This is sufficient to set the endpoint up for sending and/or receiving data. Before actually performing a read or write on an endpoint, the application should call the `usb_busy()` function and ensure that the endpoint is not busy. Internally, this function merely returns the endpoint's `UOWN` bit to determine whether the underlying buffer is available to the firmware. Finally, the `usb_read()` and `usb_write()` functions arm the requested endpoint using the provided buffer information.

To demonstrate the simplicity of the interface, Listing 4.5 displays the pertinent segments of code responsible for handling sending the button status to the host PC. The code has been edited to illustrate only the button press endpoint, although the device actually uses three endpoints (not including endpoint 0). Endpoint 1 IN is an interrupt endpoint configured for a packet size of just 1 byte, and is responsible for sending the button state (the state of the six buttons is represented by the least significant six bits in the byte) to the host any time that there is a change. Endpoint 1 OUT is configured as a bulk endpoint with a packet size of 64 bytes, and is responsible for receiving one LCD "page" (see Figure 4.1) worth of screen data from the host during each transaction. Lastly, endpoint 2 OUT is configured as an interrupt endpoint with a packet size of a single byte, and is responsible for receiving

meta commands from the host to clear the display or turn the backlight on and off.

```
1     // Initialize the USB module
2     usb_init();
3
4     // Enable unmasked interrupts
5     ENABLE_PERIPHERAL_INTERRUPTS();
6     ENABLE_GLOBAL_INTERRUPTS();
7
8     // Device must reach configured state before enabling endpoints
9     while(usb_get_state() < STATE_CONFIGURED);
10
11    usb_use_endpoint(USB_ENDPOINT_BUTTON, EP_ENABLE_IN | EP_ENABLE_OUT |
12            EP_ENABLE_HANDSHAKE | EP_DISABLE_SETUP);
13
14    // Arm the first reads/writes
15    button_handle = usb_write(USB_ENDPOINT_BUTTONS,
16            button_packet, USB_BUTTON_EP_SIZE);
17
18    // Main loop - loops forever servicing tasks
19    for(;;)
20    {
21        *button_packet = PORTB & BUTTON_MASK;
22
23        // Report changes in button state
24        if(*button_packet != prev_buttons)
25        {
26            if(!usb_busy(button_handle))
27            {
28                prev_buttons = *button_packet;
29
30                // Re-arm the endpoint
31                button_handle = usb_write(USB_ENDPOINT_BUTTONS,
32                        button_packet, USB_BUTTON_EP_SIZE);
33            }
34        }
35    }
```

Listing 4.5: This code shows the entirety of the interaction required from the application perspective to setting up and use USB communication for sending button press data to the host PC.

Chapter 5

Kernel Module Driver

In order to facilitate the low-level interaction with the USB device from the host PC, a Linux kernel module is developed. A kernel module is an extension of the kernel that is loaded at runtime and enables the developer to perform privileged operations such as accessing hardware layers. Thus, kernel modules are frequently used for implementing drivers (although user-space drivers do exist for many devices). The kernel module developed in this project makes gratuitous use of the abstractions provided by the USB subsystem of the Linux kernel to implement a char driver for the device.

5.1 USB Subsystem

In general, kernel development has significant differences compared to application development. As the kernel executes at a privileged level and has access to critical system hardware, it is very important that the developer be constantly aware of their environment. It is much more costly to have a memory leak or synchronization issue in the kernel, as this could hang or crash the entire system. Furthermore, debugging in the kernel is non-trivial, leaving many developers to fall back to primitive `printk()` (the kernel equivalent of `printf()`) statements for debugging. On that note, familiar libraries such as `stdio` and `stdlib` exist only in user space, and a completely different (and constantly evolving) set of APIs must be used [5]. Thus, kernel development is arguably more challenging than typical application development.

Kernel modules have all of the same challenges as mainline kernel development, except that they are not compiled directly into the kernel. Because of the diversity of hardware on which the Linux kernel runs, it is not sensible to build all possible drivers and other features directly into the kernel. For this reason, kernel modules are used. Kernel modules can be dynamically loaded and unloaded from the kernel at any time. This feature of kernel modules is especially useful for hotpluggable devices, including USB devices, as it means that the kernel module driver for the device only needs to be loaded when the device is plugged in.

In the case of developing kernel drivers for USB devices, many of the challenges of kernel

development are already handled by existing layers of the kernel. Specifically, the Linux kernel has its own USB subsystem that handles the majority of low-level USB interactions. Figure 5.1 shows a diagram of the Linux USB subsystem. The USB subsystem is fairly complex and handles interactions between several different components.

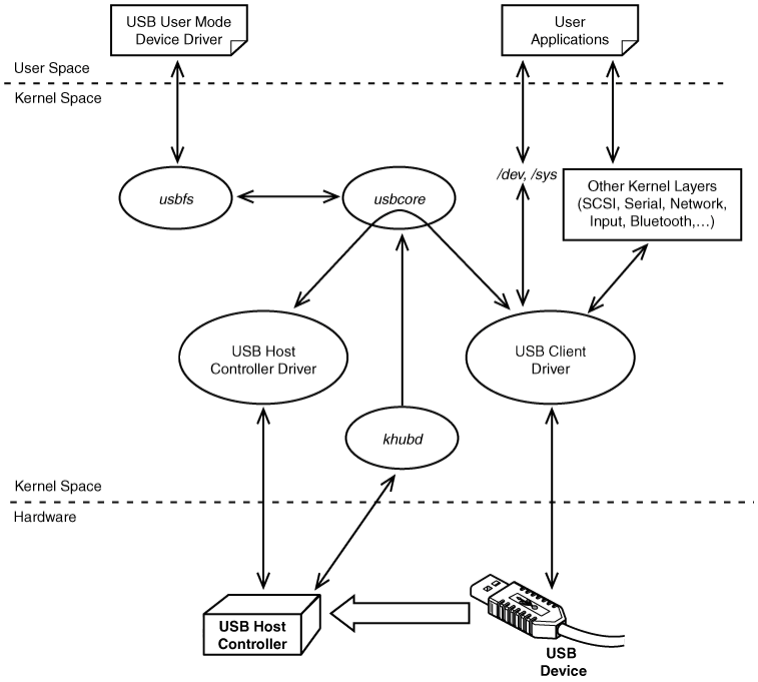

Figure 5.1: This diagram shows the interactions between the components of the Linux USB subsystem [10].

At the lowest level of the USB subsystem is the actual USB host controller device (HCD) hardware. There are three mainstream HCD interfaces currently in use: Universal Host Controller Interface (UHCI), Open Host Controller Interface (OHCI), and Enhanced Host Controller Interface (EHCI). Only the latter of these supports high-speed USB 2.0 transfers. The USB host controller driver is therefore necessary in order to abstract these hardware interface differences and provide a unified software interface for the other components. The khubd helper is a kernel thread that monitors the state of the USB ports on the host controller. The thread remains asleep until there is a change on one of the ports, at which point it notifies the USB core of the hotplug event. The USB core is central to the USB subsystem, as it provides high-level abstractions that USB device driver developers may use to communicate with devices connected to the host controller. The USB core also exports certain functionality to user space through the usbfs virtual filesystem. In fact, it is possible to write many device drivers entirely in user space through this facility [10]. However, for educational purposes, the kernel module approach is used in this project.

5.2 USB Driver

The USB core provides many data structures, macros, and functions crucial to the development of a USB device driver. First, in order for the driver to identify which device(s) the driver is attempting to control, the `struct usb_device_id` must be used. This structure has fields for vendor and product ID's, device class information, and device versions. It also contains a field that specifies which of the other fields will be used in matching the device to a driver. Fortunately, there are also macros available to create and populate these data structures with the most common fields. For example, the `USB_DEVICE(vendor, product)` creates a `usb_device_id` that matches based solely on vendor ID and product ID. In order to make the system aware of this association, the `MODULE_DEVICE_TABLE(usb,ids)` macro must be called, where `usb` denotes that it will match devices on the USB bus, and `ids` is an array of any number of `usb_device_id` structures that the driver would like to match on [5]. This macro should appear in the kernel module outside of any function.

Another important data structure is the `usb_driver`. This structure contains the name of the driver as it will be displayed by the sysfs virtual filesystem (and must be unique among the USB kernel drivers) as well as a pointer to the table of `usb_device_id`'s to be associated with the driver. The data structure also contains function pointers that will be called in response to various events. The `probe` member points to a function that should be called when a matching device is connected to the system, and the `disconnect` member points to a function to be called when the device is disconnected [5]. The structure also contains some other fields, but the ones presented here are the most important.

Kernel modules are highly event-driven. There is no single entry point into a kernel module, and in fact multiple instances of the same kernel module code may be executing concurrently. Two entry points that are always present in a kernel module are the initialization and exit routines. These are called when the kernel module is installed (as through `insmod` or `modprobe`, or via module autoloading) or removed (as through `rmmod` or `modprobe`) from the kernel, respectively. The `module_init()` and `module_exit()` macros are used to designate which kernel module functions should be used for these routines.

Listing 5.1 shows how these routines are used in conjunction with the data structures that have been presented in the driver code. The implementation here is trivial, as only a single function call is present in either routine. The `usb_register()` function takes an initialized `usb_driver` structure and registers it with the USB core so that its `probe()` and `disconnect()` functions may be called when a candidate device is detected on the system. The `usb_deregister()` function removes the driver's association with the USB core so that USB devices will no longer invoke its functions.

```
1 // Set up a device identifier list to register with the system
2 static struct usb_device_id glcd_ids[] =
3 {
4     { USB_DEVICE(GLCD_USB_VENDOR_ID, GLCD_USB_PRODUCT_ID) },
5     { } // List terminated by empty entry
6 };
```

```
7
8  // Register the device identifier with the system
9  MODULE_DEVICE_TABLE(usb, glcd_ids);
10
11 // Set up the probe and disconnect function handlers for this device driver
12 static struct usb_driver glcd_driver =
13 {
14     .name       = GLCD_DEVICE_NAME,
15     .probe      = glcd_probe,
16     .disconnect = glcd_disconnect,
17     .id_table   = glcd_ids
18 };
19
20 static int __init glcd_usb_init(void)
21 {
22     // Register this USB driver with the USB core
23     return usb_register(&glcd_driver);
24 }
25
26 static void __exit glcd_usb_exit(void)
27 {
28     // Unregister this USB driver from the USB core
29     usb_deregister(&glcd_driver);
30 }
31
32 // Set the initialization and cleanup handlers
33 module_init(glcd_usb_init);
34 module_exit(glcd_usb_exit);
```

Listing 5.1: This code associates the driver with the USB core when the module is loaded into the kernel and unregisters the association with the module is unloaded.

When a device is plugged into the system, khubd detects the new device and initiates the necessary communication to perform enumeration and retrieve the device's descriptors. This information is handed off to the USB core in order to find a suitable driver from among the registered `usb_driver` structures. When a potential candidate driver is found, the USB core invokes the driver module's `probe()` function. The purpose of this function is to provide the driver with the detected device's enumeration information so that the driver can be sure it is capable of handling the device. This decision is typically made by examining the descriptors to be sure the device contains all of the expected types of endpoints that the driver will handle. Assuming that the driver determines it will handle the device, it may then perform any one-time initialization operations that the device may require; for example, allocating memory for the device or sending initialization commands to the device. Listing 5.2 shows the implementation of the `probe()` function for the LCD device. It searches through the configuration descriptor hierarchy for all of the endpoints that it requires. If it finds all of the endpoints that it expected and no errors occurred, the function returns 0 to indicate that it will control the device. If an error occurs or the driver determines that it cannot support

the device, its `probe()` function returns a negative error code.

```
1  static int glcd_probe(struct usb_interface *interface,
2          const struct usb_device_id *id)
3  {
4      struct usb_host_interface *iface_desc;
5      struct usb_endpoint_descriptor *endpoint;
6      glcd_usb_t *glcd_device;
7      int status;
8      int index;
9
10     // Initialize the return status to an erroneous state
11     status = -ENOMEM;
12
13     // Allocate and zero out kernel space for the GLCD device structure
14     glcd_device = kzalloc(sizeof(glcd_usb_t), GFP_KERNEL);
15
16     if(glcd_device == NULL)
17     {
18         return status;
19     }
20
21     // Fill in the fields provided from enumeration
22     glcd_device->usb_dev = usb_get_dev(interface_to_usbdev(interface));
23     glcd_device->interface = interface;
24
25     // Iterate through all endpoints discovered during enumeration
26     iface_desc = interface->cur_altsetting;
27     for(index = 0; index < iface_desc->desc.bNumEndpoints; index++)
28     {
29         // Extract the endpoint information
30         endpoint = &iface_desc->endpoint[index].desc;
31
32         // If this is an interrupt IN endpoint (e.g., for button status)
33         // and one has not already been found
34         if(!glcd_device->buttons_in_addr && usb_endpoint_is_int_in(endpoint))
35         {
36             // Extract endpoint information and create appropriate buffer
37             glcd_device->buttons_in_length =
38                     le16_to_cpu(endpoint->wMaxPacketSize);
39             glcd_device->buttons_in_addr = endpoint->bEndpointAddress;
40             glcd_device->buttons_in_buffer =
41                     kmalloc(glcd_device->buttons_in_length, GFP_KERNEL);
42         }
43
44         // If this is a bulk OUT endpoint (e.g., for screen data)
45         // and one has not already been found
46         if(!glcd_device->screen_out_addr && usb_endpoint_is_bulk_out(endpoint))
47         {
48             glcd_device->screen_out_addr = endpoint->bEndpointAddress;
```

```
49          }
50
51          // If this is an interrupt OUT endpoint (e.g., for command messages)
52          // and not the default control OUT endpoint
53          if(!glcd_device->command_out_addr && usb_endpoint_is_int_out(endpoint))
54          {
55              glcd_device->command_out_addr = endpoint->bEndpointAddress;
56          }
57      }
58
59      // If we couldn't locate all three expected endpoints, return an error
60      if(!glcd_device->buttons_in_addr ||
61              !glcd_device->screen_out_addr ||
62              !glcd_device->command_out_addr)
63      {
64          return status;
65      }
66
67      // Save the device pointer in a private data field for recollection later
68      usb_set_intfdata(interface, glcd_device);
69
70      // Register the char device in the filesystem; clear reference if error
71      status = usb_register_dev(interface, &glcd_class);
72      if(status)
73      {
74          usb_set_intfdata(interface, NULL);
75          return status;
76      }
77
78      return 0;
79  }
```

Listing 5.2: This code searches through USB enumeration information to determine whether the device contains all of the required endpoints that this driver.

This code demonstrates a few interesting aspects. Notice the `kzalloc()` function that allocates memory for a device-specific structure that later gets saved in the `usb_interface` structure. Due to the event-driven nature of the kernel module, and the fact that it may be handling multiple devices simultaneously, it is not appropriate to have a set of global variables for keeping track of a single device. Even a dynamic data structure that stores information about all of the devices that the driver is handling would be cumbersome to use and difficult to maintain. As an alternative, Linux driver developers often use "floating" data structures that are allocated in the `probe()` method each time a new device is handled by the driver. The pointers to these device-specific data structures are then bound to private data fields within other structures that are passed to the module during future event callbacks. In this way, the driver can handle any number of devices without explicitly having to maintain a data structure of all associated devices .

Another important part of this function is a call to `usb_register_dev()`. This function is

used to create a char device in `/dev/` for this device. This filesystem entry is the key interface component for performing communication with the USB device through user space. The code for the `disconnect()` function simply undoes this char device registration through a call to `usb_deregister_dev()` and frees the memory that was allocated to the device-specific data structure.

The fact that the USB device will be presented as a char device means that it must implement the backend for selected file operations such as `read()`, `write()`, `open()`, and `close()` functions. Other functions, such as `ioctl()` may also be implemented to expand the functionality of the device, but this project does not utilize them. All of these operations present new entry points into the kernel module. The implementation for `write()` will be observed in detail, as it demonstrates most of the same principles as the other file operations that interact with the USB device. The code is shown in Listing 5.3.

```
1  ssize_t glcd_write(struct file *filp, const char __user *buf, size_t count,
2          loff_t *f_pos)
3  {
4      glcd_usb_t *glcd_device;
5      __u8 *out_buffer;
6      int result;
7      int bytes_written;
8
9      // Extract the device-specific structure from the file pointer
10     glcd_device = (glcd_usb_t*) filp->private_data;
11
12     // Allocate a kernel-space buffer for the output data
13     out_buffer = kmalloc(count, GFP_KERNEL);
14
15     if(!out_buffer)
16     {
17         return -ENOMEM;
18     }
19     else
20     {
21         // Try to copy the buffer from user space into the new kernel buffer
22         if(copy_from_user(out_buffer, buf, count))
23         {
24             // Free the buffer on error
25             kfree(out_buffer);
26
27             return -EFAULT;
28         }
29     }
30
31     // Send the data to the USB device
32     do
33     {
34         result = usb_bulk_msg(
35                 glcd_device->usb_dev,
```

```
36                      usb_sndbulkpipe(glcd_device->usb_dev, glcd_device->screen_out_addr),
37                      out_buffer,
38                      count,
39                      &bytes_written,
40                      GLCD_USB_IO_TIMEOUT_MS);
41      }
42      while(result == -ETIMEDOUT && !signal_pending(current));
43
44      // Free the buffer
45      kfree(out_buffer);
46
47      // If the read was interrupted by a signal
48      if(result == -ETIMEDOUT && signal_pending(current))
49      {
50          result = bytes_written ? bytes_written : -EINTR;
51      }
52
53      // If there was an error
54      else if(result)
55      {
56          return -EFAULT;
57      }
58
59      return bytes_written;
60  }
```

Listing 5.3: This code illustrates the implementation behind the `write()` operation used to send frame buffer data to the device LCD.

Observe that the first thing that the function does is to retrieve the device-specific data structure, whose pointer was copied into the `file` structure argument during the `open()` routine (not shown). Thus, any entry point into the kernel module for file operations will be able to access the device-specific structure for the device that initiated the operation. Next, a kernel buffer is allocated to store the data requested to be written. For security reasons, kernel space is not allowed to access user space memory directly and vice versa. Hence, the `copy_from_user()` function must be used as an intermediary to transfer the data from the user buffer into the newly allocated kernel buffer. Once the write buffer has been populated, the `usb_bulk_msg()` function of the USB core is used to actually send the data over the USB bus to the device's bulk OUT endpoint. This function requires information about which USB device, and endpoint, and direction to target. The function is contained in a loop that periodically (via timeouts) checks whether the issuing process encountered an interruption by a signal. Using timeouts and checking for signals is crucial because the user process is otherwise uninterruptible (and therefore unkillable) when the kernel code is blocking on an I/O operation. Finally, the buffer is freed and the number of bytes that was actually written to the USB device is returned.

The `usb_bulk_msg()` function merits a closer look. This function is actually a synchronous facade provided as an abstraction over several asynchronous functions that deal

with USB request blocks (URBs). An URB is a fundamental data unit that the USB subsystem uses in USB transfers, and is the USB equivalent to an skbuff in networking code [10]. An URB contains numerous fields that describe the device, endpoint, and direction, the type of the transfer, the buffer type, size, and properties for the data payload, callback pointers for certain transfer events, and several other fields. URBs also use an internal reference counting scheme and must be allocated dynamically through the `usb_alloc_urb()` function. Raw URBs are therefore fairly complicated to work with.

Fortunately, there are helper functions that assist in populating the URB fields depending which type of transfer is required. The `usb_fill_int_urb()`, `usb_fill_bulk_urb()`, and `usb_fill_control_urb()` can be used to populate URBs for interrupt, bulk, and control transfers, respectively. Isochronous transfers are more complicated and do not have a helper function since almost all of the URB fields have particular significance in isochronous transfers and therefore need to be manually assigned.

Once an URB has been created and populated, it may be submitted to the USB core to be sent to the hardware via a call to `usb_submit_urb()`. This function is asynchronous and returns immediately. Submitting a URB places it in a queue for the corresponding endpoint.. Once each URB has completed (either successfully or with error), it's completion callback into the kernel module is called. This callback typically reuses the URB and submits it back to the core with the next set of data. Because all URB operations are asynchronous and there is no waiting, it is possible to attain very high throughput by handling URBs manually. Once an URB is no longer needed, it should be destroyed with a call to `usb_free_urb()`.

Chapter 6

User-Space Software

The USB device driver discussed in the previous chapter took the form of a char driver. Char drivers present themselves as virtual file nodes in `/dev/` that are accessible from user space. By applying appropriate access policies (namely by setting up `udev` rules in `/etc/udev/rules.d/`), user-level applications may access the device nodes to communicate with the underlying hardware (the specific Linux commands that must be run to load the kernel module and set permissions are provided in the `README` file). The user-space layers were the simplest part of this project, as they exercised only a few new concepts.

6.1 Interface Library

Although any application with the correct permissions may access the device nodes directly, it is often undesirable to perform such low-level file manipulation in a high-level application. Thus, an intermediate layer is used to abstract the device node file operations from the end application. This intermediate layer takes the form of a statically compiled library, `libglcd`.

The `libglcd` library provides a few amenities. Most importantly, it houses the frame buffer for the entire system. Since the library contains the frame buffer, it can efficiently and locally perform drawing operations such as plotting lines or setting pixels. The library also contains bitwise pixel mappings for a 96-character font set that can be used to draw text to the frame buffer. Since the device has an array of six buttons located immediately beneath the screen, a very simple six-tabbed menu interface is also made available for use in the end application. All drawing operations are reflected only in the local frame buffer until the buffer is explicitly flushed to the device's LCD module. This policy reduces unnecessary communication overhead, and gives the application developer more control in determining when the display is actually updated.

The library abstracts all device file operations by providing meaningful high-level wrapper functions. The `glcd_init()` function performs the `open()` call, which in turn invokes the appropriate handler in the kernel module. The kernel module's `open()` handler then sends a packet to the USB device's "command" OUT endpoint that instructs it to reset the display. This ensures that the device and software are synchronized. The `glcd_get_buttons()` func-

tion performs the `read()` call on the device node, which in turn invokes the kernel module to read from the USB device's "button" IN endpoint. The `glcd_flush_screen()` function packs the local frame buffer data into a byte stream and calls `write()` to send the packed data to the USB device's "screen" OUT endpoint (it was determined that `stdio.h`'s `fread()` and `fwrite()` do not work for this library because they preclude concurrent reading and writing to the same filesystem entry). Thus, `libglcd` provides a nice abstraction layer that any application may use to generically interface with the LCD device.

6.2 ResourceMonitor

The ResourceMonitor application sits at the very top of the system stack that has been developed throughout this study. Its sole purpose is to provide a demonstration of the successful bidirectional communication with the LCD device. The program attempts to provide a reasonably useful example of a real-world application that outputs information to the LCD and responds to the device's buttons by changing which information is displayed on the screen. Specifically, the application gathers and displays system resource usage statistics to the device. Pressing the buttons on the device toggles between screens that show CPU usage information, memory usage information, and network interface packet statistics.

In order to obtain information about the system's resources, the `libgtop` library is used. This library provides a simple interface to query system information. Of particular use to this application, the library provides access to the number of CPU cores, total uptime and idle time of all cores, amount of installed memory, amount of available memory, and a list of network interfaces, their IP addresses, and number of bytes transmitted and received.

The ResourceMonitor software is multithreaded through the use of the `pthreads` API. One thread simply continuously waits on the blocking `glcd_get_buttons()` from `libglcd` for the button state to change, and updates the contents of the screen as necessary when a button is pressed. The other thread continuously sleeps, waking up periodically to update the statistics of the current screen. In order to prevent concurrent modification of the screen's frame buffer between the threads, the semaphore mechanism from `semaphore.h` is used for mutual exclusion around critical sections that write to the buffer.

To ensure that the program exits cleanly, POSIX signal handlers are used to catch the `SIGINT` and `SIGTERM` signals that are commonly used to quit a program. The new disposition for handling these signals simply sets a "quit" flag that will cause the read and write threads to exit their respective loops after the current iteration. The `SIGINT` signal is reraised targeting the read thread in order to ensure that any pending blocking reads are aborted so that the execution may continue. Finally, the threads are joined, cleanup routines are called, and the device node is closed.

Chapter 7

Conclusions

This independent study provided an extremely valuable hands-on experience in learning about several important topics. In particular, the resources on USB and on Linux driver development were very thorough, and provided a wealth of knowledge beyond what was required for the completion of this project. Because the material was also of personal interest, the project was also genuinely engaging and fun. Some of the important learning outcomes from this study are as follows:

- PCB layout and etching experience
- Familiarity with PIC18F4550 hardware
- Thorough knowledge of USB protocol
- Learning about the Linux kernel and its programming interfaces
- Kernel module and device driver development
- Working with pthreads and signals
- Using LaTeX for document typesetting

The total time commitment for this project was very closely aligned with the original estimate of 160 hours outlined in the project proposal. A fairly strict project schedule was adhered to in which 7 hours each on Mondays and Wednesdays were dedicated to this independent study. Additional unstructured time on nights and weekends was also used toward the completion of this project. Table 7.1 shows a rough outline of the project activities throughout the course of the independent study. Note that the majority of the hardware design stage took place during the finals week and break week prior to the start of the academic quarter.

Week Of	Activity
02/20/2011	Set up breadboard prototype and code for testing LCD functionality
	Created hardware schematic and began PCB layout
02/27/2011	Completed PCB layout
	Etched and assembled board
03/06/2011	Tested board hardware and developed LCD interfacing code
03/13/2011	Began development of firmware USB data structures and stack code
	Wrote code modules for interrupts and UART communication
03/20/2011	Continued development of firmware USB stack
	Set up Linux machine for host side development
03/27/2011	Wrote skeleton char driver kernel module
04/03/2011	Created `libglcd` font set and basic drawing functions
	Started development of `ResourceMonitor`
04/10/2011	Started implementation of USB-specific char driver
	Achieved USB communication using Microchip USB firmware library
04/17/2011	Continued development of firmware USB stack
04/24/2011	Finished baseline USB stack firmware and tweaked host software
05/01/2011	Worked on independent study report
05/08/2011	Finished report and packaged up deliverables
05/15/2011	Submitted all materials and demonstrated final project

Table 7.1: This table outlines the actual project timeline for the independent study.

7.1 Improvements and Future Work

Although the project was completed successfully, there are still several opportunities for improvements. At the hardware level, there are several flaws that would be corrected if the board were to be redesigned. As a bus-powered device with a rather high power floor, the hardware does not meet USB specifications for suspend current. This issue could be alleviated by using an external power supply and operating as a self-powered device. This design would also have the advantage of being able to run the backlight continuously instead of waiting until the device is configured through enumeration. Another issue with the hardware is that the LCD is actually mounted upside down (an embarrassing oversight). Although the software compensates for the incorrect orientation, it does so at the expense of unnecessary translation operations. The hardware also has too many buttons. This seemingly trivial issue is a user interface problem because it limits the amount of text on the menu tabs to just three characters each. Having only four buttons would still be more than sufficient for operating such a simple device and would allow each tab to display five characters.

The firmware also presents some opportunities for enhancements. The most important change would be to offload the frame buffer from the PC to the firmware. Although the memory layout of the PIC18F4550 precludes the possibility of allocating a contiguous frame buffer for the entire screen, it would be possible to split the buffer up into two 512 byte buffers – one for each controller chip on the LCD. If the frame buffer existed in the firmware, all drawing routines could be performed on the device. This would significantly reduce host processing and communication overhead because the PC would only need to send small commands that instruct the LCD what to draw, rather than computing and sending the actual screen data.

The firmware USB stack could also benefit from some additions. Support for isochronous transfers on endpoints 1 through 15, and support for vendor/class requests on endpoint 0 would allow for a much greater diversity of applications. Furthermore, there are several USB events for which it would be helpful to offer callback functions, such as when the device changes states, or when a transaction has completed on a nonzero endpoint.

The host driver is designed to be fairly simple, and does not have too many aspects that require improvement. However, one promising alternative to using a kernel module driver would be to use the `libusb` USB interface library. This library allows for the development of cross-platform USB drivers that conveniently run in user space. This removes many of the challenges of working with kernel code and provides simple abstractions for interacting with devices on Linux, BSD, Mac OS X, and Windows.

The `libglcd` library is fairly solid, but could perhaps be augmented to integrate more UI layer operations (for example, by providing greater support for the tabbed menu system or offering additional UI widgets). The `ResourceMonitor` software could also provide more information to the use, as `libgtop` provides a much greater breadth of data than is actually displayed by the LCD. For example, the LCD could display individual CPU core or network interface statistics, process lists, or time plots of resource usage trends.

Bibliography

[1] Axelson, J. *USB Complete*, 4th ed. Lakeview Research LLC, Madison, WI, 2009.

[2] AZ Displays Inc. Specifications for Liquid Crystal Display (AGM1264F Series), 2001.

[3] Bachiochi, J. Application Communication with USB (Part 1): The Enumeration Process Explained. *Circuit Cellar*, 239 (June 2010), 62–68.

[4] Bachiochi, J. Application Communication with USB (Part 2): The Importance of Descriptors. *Circuit Cellar*, 240 (July 2010), 58–64.

[5] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*, 3rd ed. O'Reilly Media, Inc., Sebastopol, CA, 2005.

[6] Intel, Compaq, Hewlett-Packard, Microsoft, Lucent, Philips, and NEC. Universal Serial Bus Specification, 2000.

[7] Microchip Technology Inc. MCHPFSUSB Library Help, 2009.

[8] Microchip Technology Inc. PIC18F2455/2550/4455/4550 Data Sheet, 2009.

[9] Minch, B. A. Principles of Engineering: Eclectronic System Design. `http://pe.ece.olin.edu/ece/projects.html`, 2006.

[10] Venkateswaran, S. *Essential Linux Device Drivers*, 1st ed. Pearson Education, Inc., Boston, MA, 2008.