USB SUSBSYSTEM FOR MINIX 3

A PROJECT REPORT

Submitted By

ALTHAF K BACKER

In partial fulfillment of the requirements for the Degree

of

Bachelor of Technology (B.Tech)

in

COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF ENGINEERING

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

KOCHI-682022

MARCH 2010

Division of Computer Engineering
School of Engineering
Cochin University of Science & Technology
Kochi-682022

CERTIFICATE

Certified that this is a bonafide record of the project work titled

USB Subsystem for Minix 3

Done by

Althaf K Backer

of VIII semester, Computer Science & Engineering, in the year 2010 in partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science & Engineering of Cochin University of Science & Technology

Dr. David Peter S.	*Vinod Kumar P.P*
Head of the Division	*Project Guide*

ACKNOWLEDGEMENT

At the outset, I thank God almighty for making my endeavor a success. I also express my gratitude to **Dr. David Peter S.**, Head of Division of Computer Engineering for providing me with adequate facilities, ways and means by which I was able to complete this project.
.

I also express my sincere gratitude to my Project Guide **Mr. Vinod Kumar PP,** for his constant support and valuable suggestions without which the successful completion of this project would not have been possible.

I also express my immense pleasure and thankfulness to all the teachers and staff of the Division of Computer Engineering, CUSAT for their cooperation and support. Last but not the least, I thank all others, and especially my classmates and my family members who in one way or another helped me in the successful completion of this work.

Althaf K Backer

ABSTRACT

MINIX 3 a microkernel based, Open Source UNIX clone which is POSIX compliant, MINIX is a good platform to learn and experiment, the present implementation that is MINIX 3 lags USB subsystem,which could ease programming USB device drivers , which other wise could be a tedious task , understanding the importance the project aims at implementing it on MINIX 3.

Project focus on three essential parts of a USB Stack

- USB driver (USBD), which is an abstraction of USB and provides uniform interfaces namely USB driver interface (USBDI) and Host Controller Driver interface (HCDI). USB class drivers can be written on top of USBDI which is not part of project.
- Host Controller driver Interface (HCDI) , which is another layer that abstracts host controller drivers and provides a uniform abstraction to USBD as well as host controller drivers.
- Implementing host controller driver for well known host controller Universal Host Controller Interface (UHCI).
- Human Interface Device (HID) Class specific request at USBDI and dummy USB HID keyboard and mouse drivers.

Entire worked is biased on USB 2.0 specifiation and UHCI revision 1.1.

List of Tables

List of Figures

Table of Contents

CHAPTER 1

Introduction

In this and the next chapter, I give an overview towards important foundations for my project. First i give a brief idea about Minix 3 and then a small introduction to driver framework ,further I explain why Minix 3 was chosen.

1.1 Introduction to Minix 3

Minix 3 is an open source ,POSIX complaint full multiuser, multiprogramming and highly fault tolerant microkernel operating system designed to be reliable and flexible see Figure 1.0, following the microkernel approach entire operating system is divided into separate user mode process well insulated from each other ,unlike monolithic kernel the device drivers runs as separate process at user space ,which contributes towards the reliable and robust nature of minix. When a driver crashes its is gracefully replaced by a new one without any user intervention.

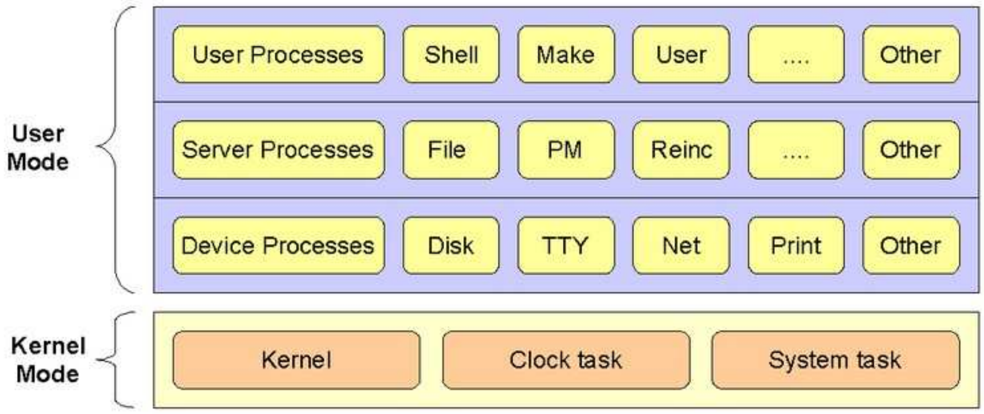
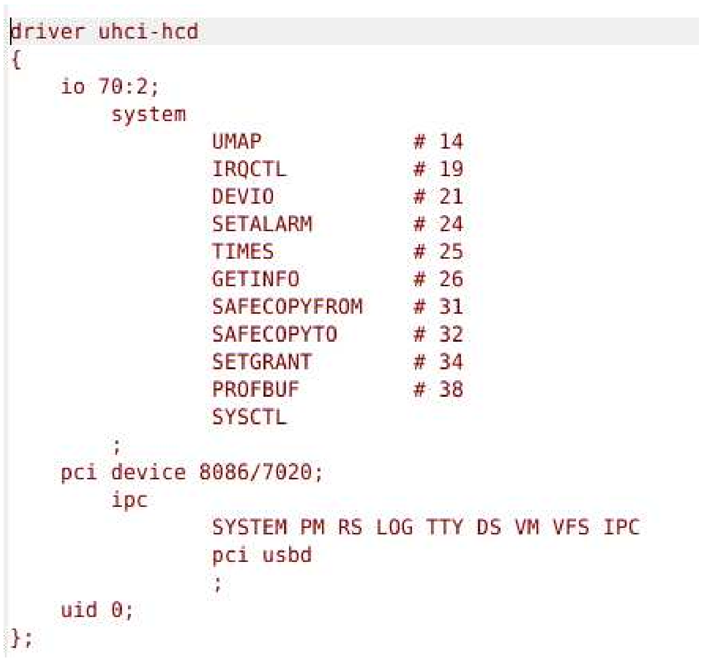


Figure 1.0 Microkernel Architecture of Minix 3

1.2 Device driver framework of Minix 3

Device drivers for minix are programmed in C language , so one should be good enough with C and should have good understand of device they are going to deal with ,there is a special process called Reincarnation Server (RS) which keeps the driver alive and check its status periodically. In Minix device drivers are separate programs which send and receive message to communicate with the other operating system components.. Device drivers, like any other program, may contain bugs and could crash at any point in time. The Reincarnation server will attempt to restart device drivers when it notices they are abruptly killed by the kernel due to a crash. The Reincarnation Server sends keep-a-live messages to each running device driver on the system periodically, to ensure they are still responsible and not i.e. stuck in an infinite loop.

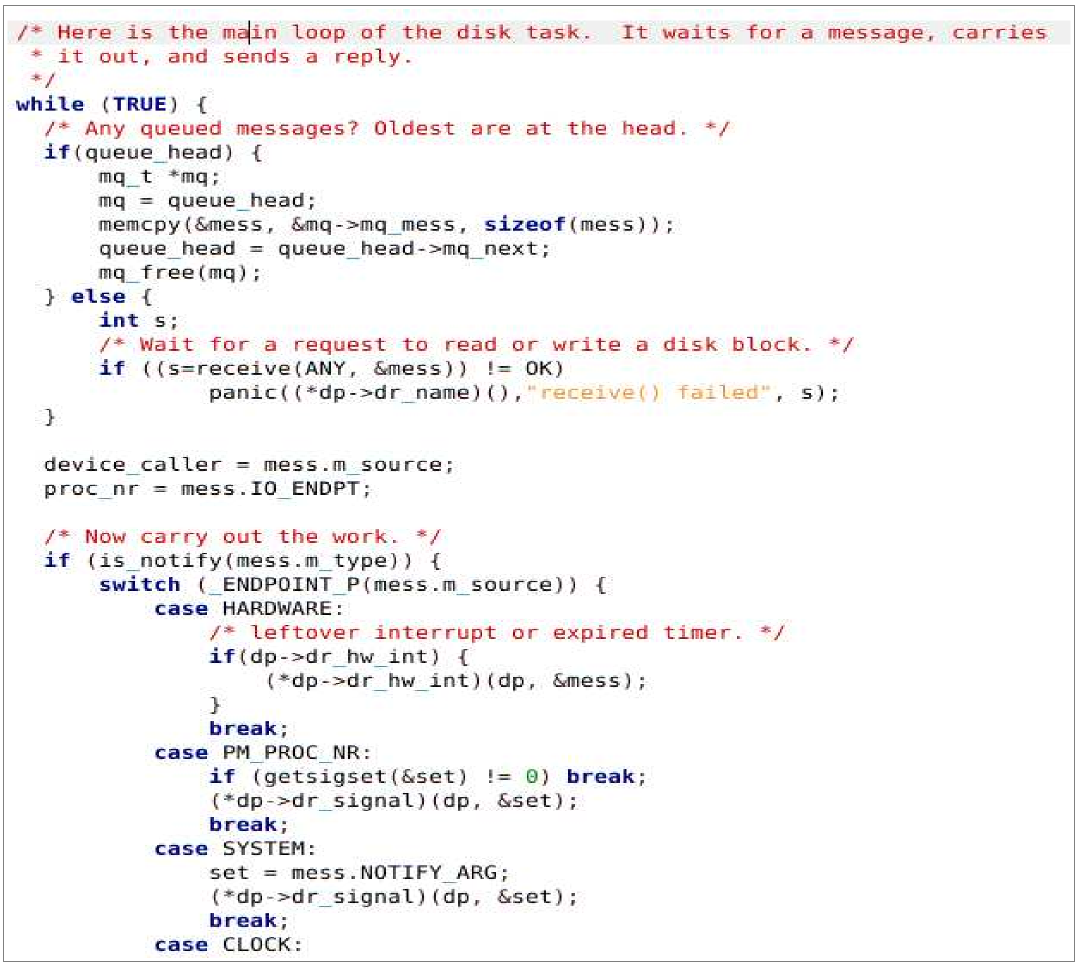
Each device driver typically only needs to access one real hardware device, and uses a few functions provided by Minix, these details are put up into */etc/system.conf* one such example is given in fig 1.1.



```
driver uhci-hcd
{
    io 70:2;
        system
                UMAP            # 14
                IRQCTL          # 19
                DEVIO           # 21
                SETALARM        # 24
                TIMES           # 25
                GETINFO         # 26
                SAFECOPYFROM    # 31
                SAFECOPYTO      # 32
                SETGRANT        # 34
                PROFBUF         # 38
                SYSCTL
        ;
    pci device 8086/7020;
        ipc
                SYSTEM PM RS LOG TTY DS VM VFS IPC
                pci usbd
                ;
    uid 0;
};
```

Figure 1.1 Sample of /etc/system.conf for uhci-hcd

The main() of the driver is an infinite loop waiting for events ,one such event is the ping from RS ,when a driver receive this message it is suppose to notify RS that it is alive , other event for which the driver is waiting for is context specific see Figure 1.2 sample from libdriver.



```
/* Here is the main loop of the disk task.  It waits for a message, carries
 * it out, and sends a reply.
 */
while (TRUE) {
  /* Any queued messages? Oldest are at the head. */
  if(queue_head) {
      mq_t *mq;
      mq = queue_head;
      memcpy(&mess, &mq->mq_mess, sizeof(mess));
      queue_head = queue_head->mq_next;
      mq_free(mq);
  } else {
      int s;
      /* Wait for a request to read or write a disk block. */
      if ((s=receive(ANY, &mess)) != OK)
              panic((*dp->dr_name)(),"receive() failed", s);
  }

  device_caller = mess.m_source;
  proc_nr = mess.IO_ENDPT;

  /* Now carry out the work. */
  if (is_notify(mess.m_type)) {
      switch (_ENDPOINT_P(mess.m_source)) {
          case HARDWARE:
              /* leftover interrupt or expired timer. */
              if(dp->dr_hw_int) {
                  (*dp->dr_hw_int)(dp, &mess);
              }
              break;
          case PM_PROC_NR:
              if (getsigset(&set) != 0) break;
              (*dp->dr_signal)(dp, &set);
              break;
          case SYSTEM:
              set = mess.NOTIFY_ARG;
              (*dp->dr_signal)(dp, &set);
              break;
          case CLOCK:
```

Figure 1.2 Code snippet of a generic driver interface libdriver

There is utility called '*service*' meant for handling the drivers and servers with service utility we can bring 'up' ,'down', 'refresh', 'restart', 'rescue' and 'shutdown' drivers there are much more advanced features like setting period at which the RS should check the status. This is just a brief idea of how drivers in minix work, explaining entire driver framework is beyond the scope.

1.3 Minix 3

One solid reason is educational purpose, Minix is a research operating system ,however its is quite useful enough for a computer science student. Minix goal, to teach operating system theory in a practical way is still continued to date, further the microkernel architecture can be quite interesting to play with and coding device drivers for them can help understand the microkernel architecture in much depth. I found USB protocol to be quite interesting and on understanding that Minix lags one it was a perfect combination for a major project. More than anything else the open source nature of Minix is another choosing factor.

CHAPTER 2
Universal Serial Bus

In this chapter I present the basics of the universal serial bus (usb) and how the usb protocol is defined,it covers the chapter 4 and 5 of [1], this excellent introduction for USB is taken from [4].

2.1 The Universal Serial Bus

The interfaces used in the original ibm pc designs of the early 1980s had a number of problems. For example, there existed a wide diversity of connectors. Furthermore, most of these interfaces were not hot pluggable. Limited system resources, which had to be shared by more and more devices, appeared to be another problem. One approach to handle these problems is usb, whose specification was first published in early 1996 by a consortium of it companies including IBM and Microsoft .

To overcome the shortcomings of traditional peripheral interfaces, the designers of usb were striving for the following design goals:

- A single connector type for all pc peripherals,
- Hot plug support,
- Preventing system resource conflicts,
- Low cost for system and peripheral implementations,
- Automatic detection and configuration of peripherals,
- Support for legacy hardware and software, and
- Low-power implementation.

usb can be used to connect a wide variety of peripheral devices ranging from input devices like keyboards and mice to more complex hardware like video frame grabbers and mass storage devices. The first revision of usb introduced two transfer speeds, namely low speed using a transfer rate of 1.5 Mb/s and full speed using 12 Mb/s. In the year 2000, a second revision of usb was published, which included high speed transfer at a data rate of 480 Mb/s. Nevertheless, usb 2.0 is fully backward compatible to usb 1.x, see table 2.0.

It is to be expected that the third major revision of usb ie usb 3.0 will also be fully backward compatible to previous revisions. Additionally, it will offer a yet higher transfer speed called super speed. Devices supporting this transfer speed will be able to transfer data at rates of up to of 4.8 Gb/s. USB uses twisted-pair data wires for data transmission. Additionally, there are two more wires in a usb cable providing a supply voltage of 5 V to a usb device. A good overview of usb can be found [2] and [3].

Name	Speed	Available since
Low Speed	1.5 Mb/s	USB 1.x
Full Speed	12 Mb/s	
High Speed	480 Mb/s	USB 2.0
Super Speed	*4.8 Gb/s*	*USB 3.0 (planned)*

Table 2.0 USB revision and speed

2.2 USB Hardware

The hardware part of the usb mainly consists of three elements. These include the host controller with its root hub, usb hubs, and usb devices and are explained in the following.

2.2.1 Host Controllers

USB is a single master bus, which is managed by a single host controller (hc).Thus, all communication on the bus is controlled by the hc. The hc's task is to perform transactions that have been scheduled by the hc driver. For that,the hc's counter part, the hc driver, generates transfer descriptors (td) and enqueues them for execution by the hc into the hc's transfer queue. A hc always comes with a root hub to provide usb ports for one or more usb devices. The amount of usb ports can be increased by using usb hubs.

There are three important hc designs:

Universal Host Controller Interface (UHCI) This design was developed by Intel [Int96] and fulfills the usb 1.1 specification. It is mainly used in products by Intel and via Technologies. In comparison to other hc designs, more work is done in software, in order to reduce hardware complexity.

Open Host Controller Interface (OHCI) This design, developed by Compaq, Microsoft, and National Semiconductors [CMN99], also supports the usb specification up to revision 1.1. In contrast to the uhci design, ohci-type hcs do more work in hardware, which allows to provide a more abstract interface for driver developers.

Enhanced Host Controller Interface (EHCI) The ehci design, which supports the usb 2.0 Specification was created by Intel [Int02]. Usually, an ehci hc is equipped with one or more companion hcs for backward compatibility. Thus, if a usb 1.x Device is connected to a port of the root hub of an ehci hc, this port will be forwarded to the corresponding companion hc. However, it is also possible to ensure backward compatibility without a companion hc. For that the hc's root hub has to support so called split transactions.

2.2.2 USB Hubs

Hubs can be used, to extend the number of available usb ports. They can be integrated into devices like keyboards and monitors or can be implemented as standalone devices. Furthermore, usb hubs can be bus-powered or self- powered. If a usb device is bus-powered, it does not need an external power source and will be powered by the bus. Because a usb port only provides a limited amount of power (500 mA at 5 V), a bus-powered hub may only provide up to four usb ports. Otherwise, the hub has to be self-powered. usb hubs play an important role in usb's hot-plugging mechanism, because they are also used to detect connection changes on the bus.

2.2.3 USB Devices

USB devices provide the actual functionality to the user. Their attributes are stored in descriptors. Whenever a new device is attached to the bus, the hc driver first reads these descriptors. Afterward, the hc driver uses the information contained in the descriptors to find a corresponding usb device driver for this device. This device driver can also use the device's descriptors to obtain more information on the device. usb devices can provide one or more configurations, each of which can contain one or more interfaces. This interfaces have a default setting and may contain one or more alternate settings. These settings in turn can contain one or more numbered endpoints, which can be understood as source or sink of data. Each of these endpoints can either be ingoing, which means information flows from the device to the hc or, outgoing. Again, like hubs, which are actually normal usb devices, usb devices can be bus-powered or self-powered, whereas, if a device is bus-powered, it may not consume more current than 500 mA.

2.3 USB Protocol elements

In this section I give an overview of how communication works on usb. This overview includes the existing transfer types and their purpose, how usb transactions are executed by the hc, and how the usb time-base, called frames, is generated. Afterward, I describe how usb devices present themselves to the usb software stack with the help of descriptors.

2.3.1 Transfer Types

USB provides the following transfer types, not all of which need to be implemented by usb devices.

2.3.1.1 Control Transfer

Control transfer is used to configure a usb device and to control aspects of its operation. This transfer type is mandatory for usb devices. Every usb device must implement at least one control endpoint, which is endpoint zero. hc drivers and usb device drivers use special request to endpoint zero for getting information on a device an managing it.

2.3.1.2 Interrupt Transfer

This transfer type is used to periodically poll usb devices for data that needs to be transmitted. For that, the usb device driver specifies an interval, in which the usb device should be polled. This polling is done by the hc, and the usb device driver is not informed until data is available. The minimal polling interval for low speed devices is 10 ms, for full speed devices it is 1 ms, and for high speed devices it is 125 µs. Interrupt transfer is often used for input devices, like keyboards and mice.

2.3.1.3 Isochronous Transfer

Isochronous transfer is used whenever guaranteed bandwidth is needed, like in video frame grabbers or audio interfaces. Only full and high speed devices support this transfer type. The default setting of an interface may not include an isochronous endpoint. Furthermore, when an alternate setting of an interface is activated that includes an isochronous endpoint, the hc driver reserves the bandwidth needed, according to the USB specification.1 . If the required bandwidth is not available the activation of the alternate interface setting fails. Up to 90 percent of the bus bandwidth are reserved for periodical transfer types, like isochronous and interrupt transfer.

2.3.1.4 Bulk Transfer

Bulk transfer is used when there is no need of guaranteed bandwidth. It uses the remaining bandwidth of the bus, and is, for example, used for printers, scanners, and usb storage devices.

2.3.2 Transactions

A usb device driver's request to send or receive data on the bus is represented by an input–output request package (irp). Whenever a device driver submits an irp to the usb subsystem, the hc driver translates the irp into one or more usb transactions. These usb transactions, in turn, are represented by transfer descriptors (tds) and are scheduled by the hc driver.

A td contains all the information needed by the hc to perform a transaction, including:

- The address for this transaction, consisting of the device's address on the bus, and the endpoint id,
- The transaction type (e.g., ingoing/outgoing),
- The transfer speed,
- The number of bytes to be transfered, and
- The memory location of the transfer buffer, containing the actual data.

The hc driver enqueues the tds into a linked list, which is called the frame list. During a certain interval (usb 1.x: 1 ms, usb 2.0: 125 µs), called a frame, the hc fetches the tds belonging to the current frame and executes them. Fig 2.1 illustrates this process: On the left side, we can see the system's memory containing several transfer descriptors, and a memory location where data of the usb mouse should be stored. On the right side, there is a simple usb setup, including a hc and two usb devices attached to it, a mouse and printer. As the first step of this usb transaction, the hc grabs the transfer descriptor. Second, the hc generates the ingoing (i.e. reading) transaction, described in this td. This transactions is addressed to the usb mouse. Although the transaction is received by all devices attached to the same usb port as the mouse, only the mouse is allowed to send a response, which it does in the third step. In the end, the host controller transfers the data sent by the mouse to memory.

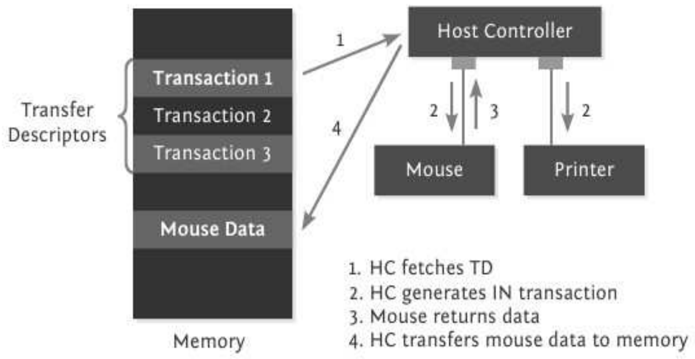


Figure 2.0 Conceptual View of USB transaction

2.3.3 Frame Generation

The hc is responsible to partition time into usb frames.To generate the usb frames, a clock and a counter are used. Each clock tick increments the counter. When this counter reaches its limit, the frame number counter increments. Because in usb 1.x systems a frame lasts 1ms, while the transfer rate is 12 Mb/s, a 12 MHz clock and a counter that counts up to 12 000 are used. According to the higher transfer rate of 480 Mb/s, and the frame interval of 125 µs, in usb 2.0, a 480 MHz clock and a counter with a limit of 60 000 is used for frame generation. This setup is illustrated in Figure 2.1. On the left side we can see a 12 MHz clock incrementing a counter, whose carry output increments the frame number counter.

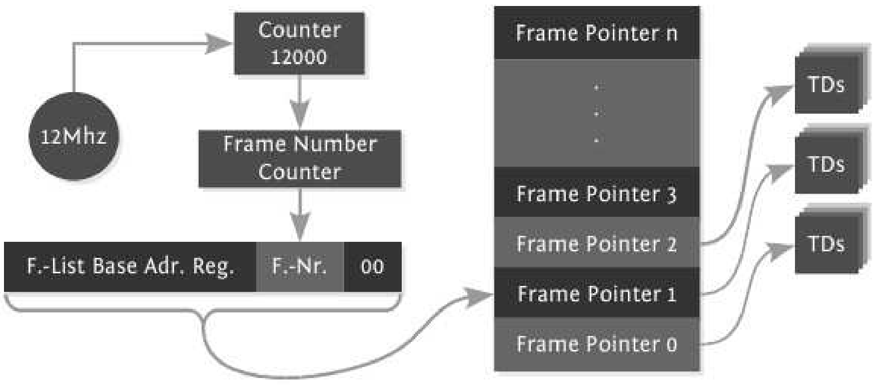


Figure 2.1 Conceptual view of frame generation in usb 1.x systems

As we can further see, the frame number is used as offset, to generate the address of the next frame pointer, which points to a list, containing the transfer descriptors for the current time frame. The base for this address is to be set in
the hc’s frame list base register by the hc driver.

2.3.4 Configuration and Interfaces

usb devices have two levels of configuration: configurations and interfaces. Configurations can, for example, be used to support a low power mode in a high power device. So, if no external power supply is attached to the device, the low power configuration can be used, only supporting a subset of the functionality. Otherwise the high power configuration can be used, then offering full functionality. Interfaces, on the other hand, are used to access different functionality of a device, for instance, the video function of a web camera and the built-in microphone. Furthermore, the different interfaces of a device can be driven by different drivers.

2.3.5 Descriptors

As mentioned before, usb devices use descriptors to present their features to software. These descriptors include:

2.3.5.1 Device Descriptor

Every device provides one descriptor containing information on the device such as the manufacturer, the usb device id and whether it is a full or a low-speed device. Furthermore, it contains information on the number of and references to the configuration descriptors this device contains.

2.3.5.2 Configuration Descriptor

A usb device provides one configuration descriptor per configuration it supports. These descriptors contain information on the number of interfaces provided by the corresponding configuration. Further, they include references to the corresponding interface descriptors.

2.3.5.3 Interface Descriptor

The interface descriptors hold general information on the corresponding interface, and on the number of endpoints included in an interface. An interface may include up to 15 endpoints, and further may include alternative settings. Alternative settings can for example be used to switch between different transfer bandwidths or to enable and disable endpoints.

2.3.5.4 Endpoint Descriptor

An interface may contain multiple endpoint descriptors, each of which describes one endpoint, and this endpoint's attributes. These descriptors contain information on the transfer types supported by this endpoint (e.g., isochronous, control, .. .), and on the maximum transfer rate.

2.3.5.5 String Descriptor

String descriptors may be defined for the whole device, for certain configurations, and interfaces. They contain human-readable information about the corresponding device, configuration, or interface.

2.3.5.6 Class Descriptor

If a device is implementing a usb device class, it may contain class descriptors containing class-specific information used by the usb class device driver.

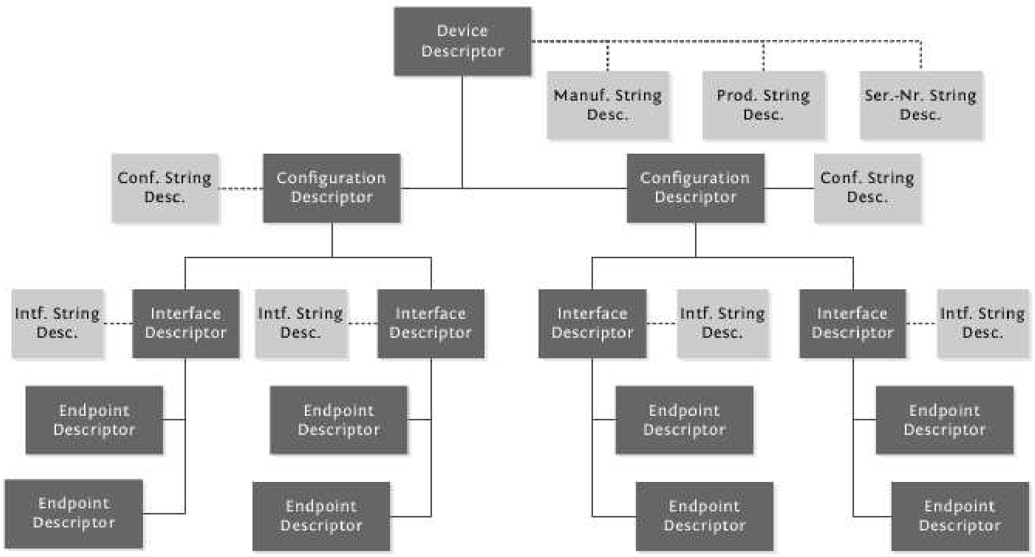


Figure 2.2 Descriptor hierarchy for a USB device with two configurations

2.4 USB communication flow

The USB provides a communication service between software on the host and its USB Functions (Devices) . Functions can have different communication flow requirements for different client-to-function interactions. Each communication flow makes use of some bus access to accomplish communication between client and function. Each communication flow is terminated at an endpoint on a device. Device endpoints are used to identify aspects of each communication flow. Figure 5.3 is detailed. Figure 2.4 gives the summery of usb communication flow.

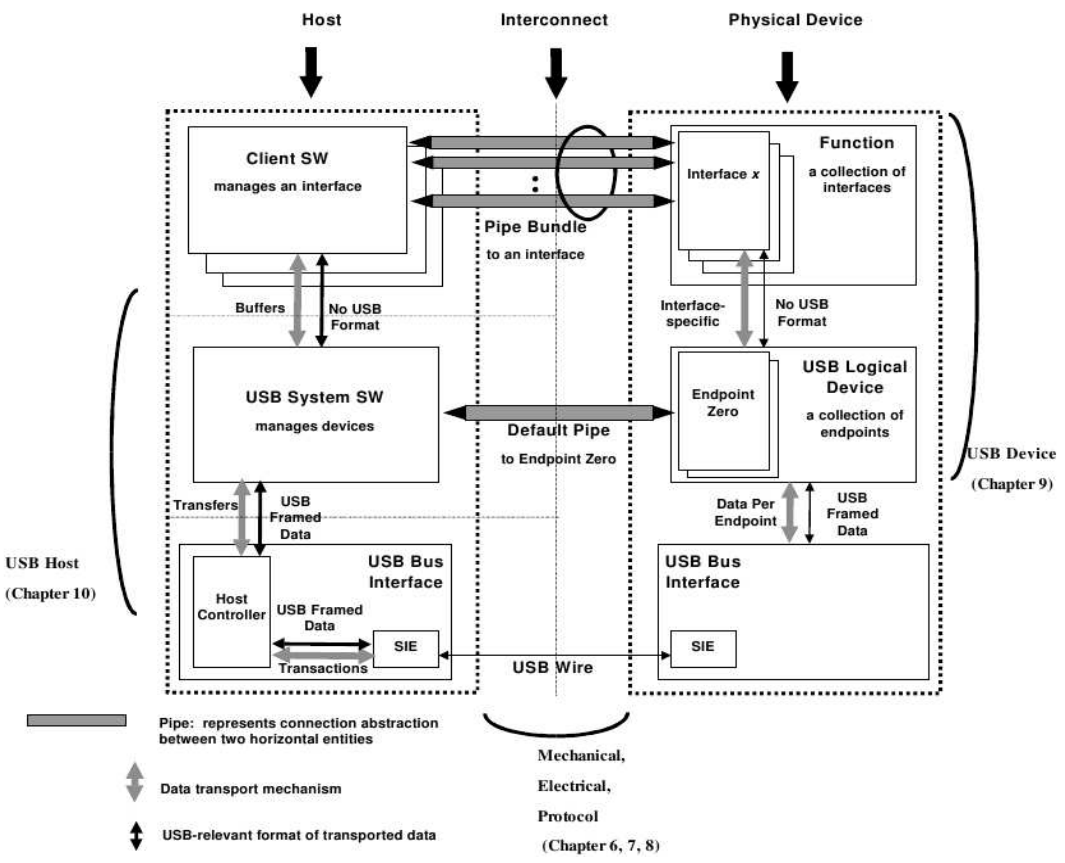


Figure 2.3 Detailed USB communication flow (chapters are of USB 2.0)

Host Controller Driver (HCD): The software interface between the USB Host Controller and USB System Software. This interface allows a range of Host Controller implementations without requiring all host software to be dependent on any particular implementation. One USB Driver can support different Host Controllers without requiring specific knowledge of a Host Controller implementation.

USB Driver (USBD): The interface between the USB System Software and the client software. This interface provides clients with convenient functions for manipulating USB devices. A USB logical device appears to the USB system as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function. The USB System Software manages the device using the Default Control Pipe. Client software manages an interface using pipe bundles (associated with an endpoint set). Client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The Host Controller (or USB device, depending on transfer direction) packet sizes the data to move it over the USB.

2.4.1 Device Endpoints

An endpoint is a uniquely identifiable portion of a USB device that is the terminus of communication flow between the host and device. Each USB logical device is composed of a collection of independent endpoints. Each logical device has a unique address assigned by the system at device attachment time. Each endpoint on a device is given at design time a unique device-determined identifier called the endpoint number. Each endpoint has a device-determined direction of data flow. The combination of the device address, endpoint number, and direction allows each endpoint to be uniquely referenced.

Each endpoint is a simplex connection that supports data flow in one direction: either input (from device to host) or output (from host to device). An endpoint has characteristics that determine the type of transfer service required between the endpoint and the client software. An endpoint describes itself by:

- Bus access frequency/latency requirement
- Bandwidth requirement
- Endpoint number
- Error handling behavior requirements
- Maximum packet size that the endpoint is capable of sending or receiving
- The transfer type for the endpoint
- The direction in which data is transferred between the endpoint and the host

Endpoints other than those with endpoint number zero are in an unknown state before being configured and may not be accessed by the host before being configured.

2.4.2 Pipes

A USB pipe is an association between an endpoint on a device and software on the host. Pipes represent the ability to move data between software on the host via a memory buffer and an endpoint on a device. There are two mutually exclusive pipe communication modes:

- **Stream**: Data moving through a pipe has no USB-defined structure
- **Message**: Data moving through a pipe has some USB-defined structure

The USB does not interpret the content of data it delivers through a pipe. Even though a message pipe requires that data be structured according to USB definitions, the content of the data is not interpreted by the USB.

Additionally, pipes have the following associated with them:

- A claim on USB bus access and bandwidth usage.
- A transfer type.
- The associated endpoint’s characteristics, such as directionality and maximum data payload sizes. The data payload is the data that is carried in the data field of a data packet within a bus transaction .

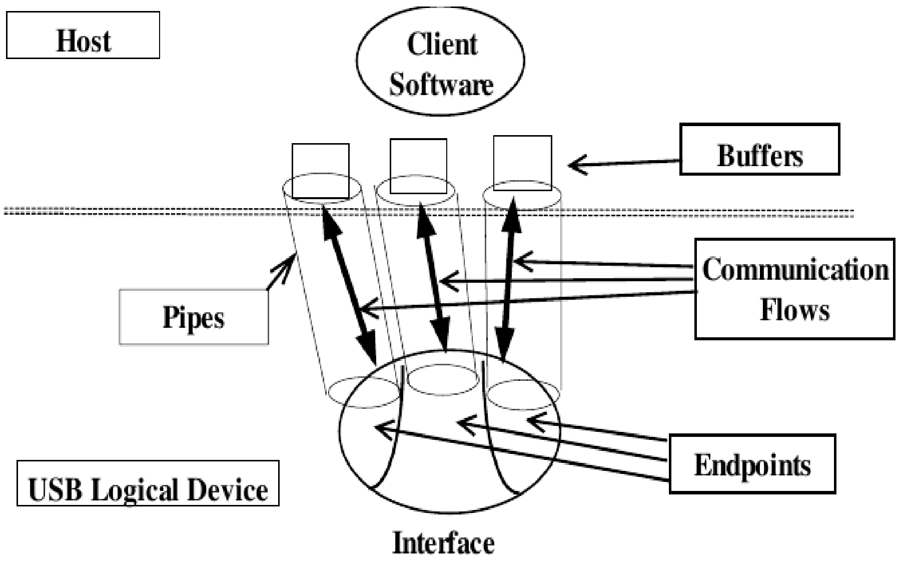


Figure 2.4 Summery of USB communication flow

CHAPTER 3
Universal Host Controller Interface

This chapter introduces all data structures needed for a successful communication between the UHCI host controller and a function as far as they can be controlled by the host controller driver. All data structures generated and controlled by the hardware are not object of this chapter,see [5] for more information about UHCI , this brief introduction about UHCI is taken from [6].

3.1 Overview

As mentioned in chapter 2 the USB specification separates the USB bus into frames, of length one millisecond (USB 1.x) each and are started by a start of frame SOF-packet. The structure of different transfer types in such frames (see Figure 3.0) is given by the UHCI specification [5].After the SOF-packet the isochronous transfers are sent followed by the interrupt transfers. According to the USB specification , these two transfer types achieve at most 90 percent of the total bandwidth. After the interrupt transfers the control transfers follows. At least 10 percent of the bandwidth is reserved for the control transfers to guarantee that configuration of the attached USB devices can be assured. The rest of the bandwidth is used for bulk transfers. If there are not enough transfers the remaining time is spent idle. It is in the responsibility of the host controller driver to achieve the correct order of the different transfer types. Therefor a data structure (see Figure 3.1) must be created in main memory.

Interrupt, control and bulk transfers begin with a Queue Header. This data structure stores control data needed for the correct processing by the hardware. A linked list consisting of Queue Headers is build, whereas each Queue Header represents a single transaction. The list of Transfer Descriptors which is linked by each Queue Header represents the transactions itself. For the isochronous transfer no Queue Headers are used. Instead all the Transfer Descriptors of all isochronous transfers in this frame are linked together

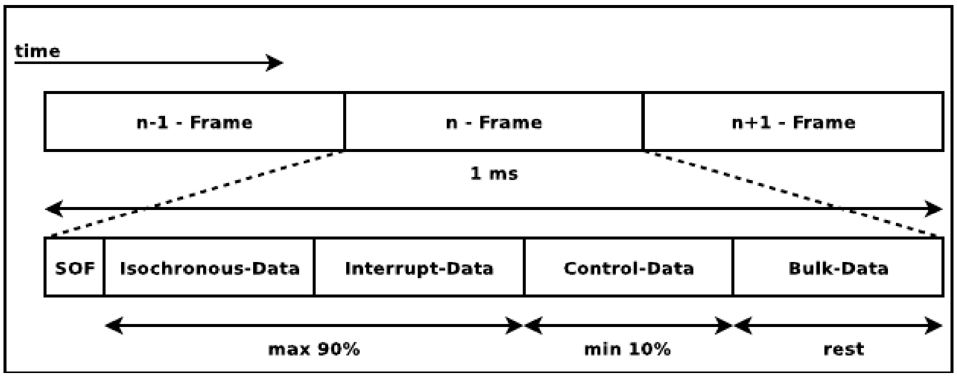


Figure 3.0 Showing the bandwidth allocation of each type to transfers

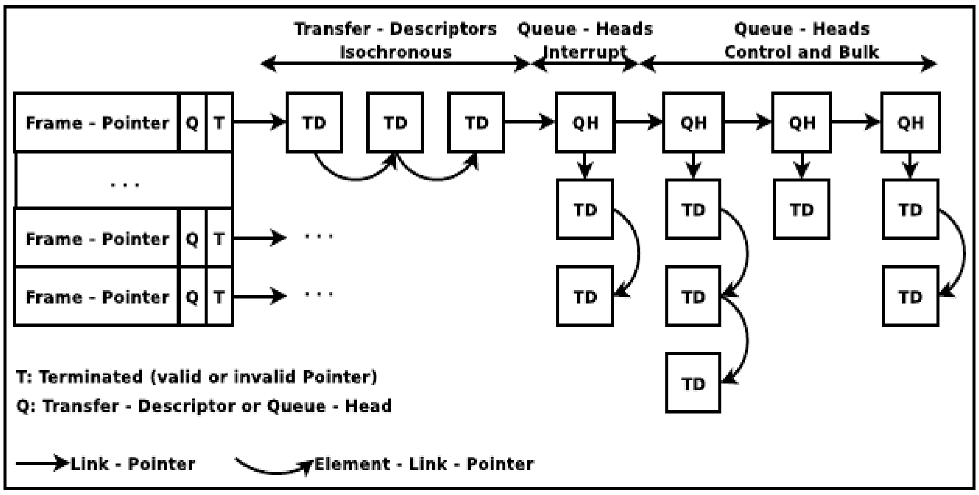

Transfer - Descriptors Isochronous
Queue - Heads Interrupt
Queue - Heads Control and Bulk
Frame - Pointer
Q
T
TD
QH
T: Terminated (valid or invalid Pointer)
Q: Transfer - Descriptor or Queue - Head
Link - Pointer
Element - Link - Pointer

Figure 3.1 A typical transfer arrangement of the UHCI transfers

3.2 Data Structures

3.2.1 Frame List and SOF

During initialization, a list of 1024 Frame List Pointers has to be allocated in main memory. Each Frame List Pointer stores the start address of the data structures that represent the data which will be transferred during a frame (see Figure 3.1). The current entry in the frame list is determined by the Start-Of-Frame counter (see Figure 3.2) which is responsible for generating the Start-Of-Frame packet each millisecond (signals the beginning of a new frame). It gets decremented by a 12 MHz clock until it reaches zero, taking into account the Start-Of-Frame modify register which provides an adjusted starting value for the Start-Of-Frame counter. Every time the Start-Of-Frame counter is decremented to zero a new frame is generated and there for the frame counter gets incremented. The frame list base address register contains the base address of the whole frame list. The combination of the frame counter and the frame list base register results in the address of the current processed frame list pointer.

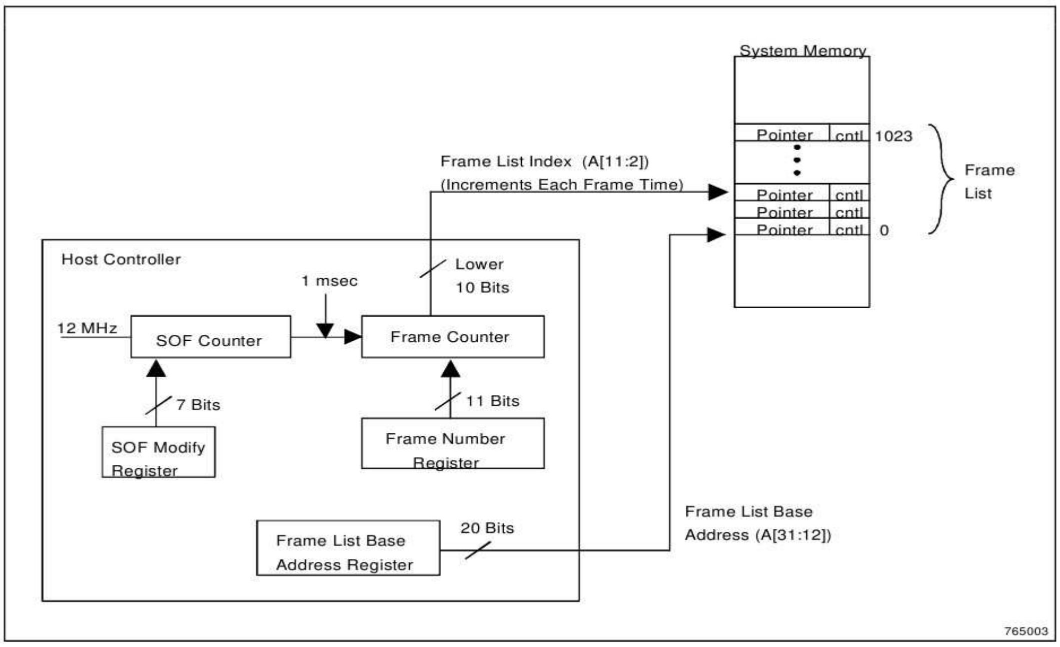
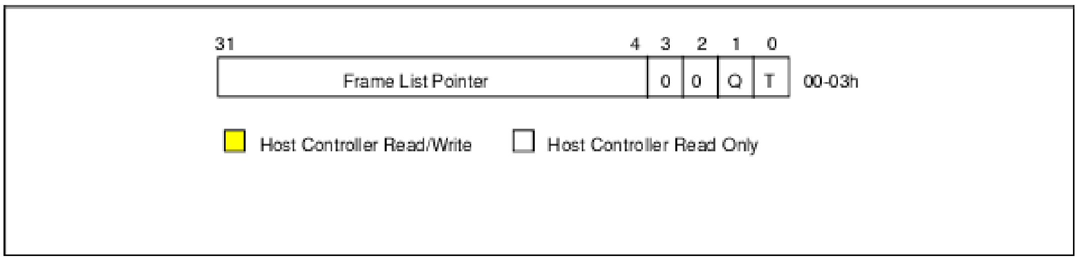


Figure 3.2 UHCI Start of Frame and Frame list pointer relation

3.2.2 Frame List Pointer

A Frame List Pointer (see Figure 3.3) consists of three fields. The address field stores the memory address of the first data object that will be processed by the hardware during this frame. This must be a Transfer Descriptor or a Queue Head. To distinguish this two kinds of data structures the Q-field of the Frame List Pointer is used. It is set if the following data structure is the Queue Head. If the host controller driver was not able to generate a correct frame data structure before it is processed by the hardware or if there are no transactions left, the frame gets marked as invalid, by the host controller driver by setting the T- field to zero. This causes the hardware to skip this frame and to immediately process the next frame. All transactions, beside the isochronous transfers, are realized as a linked list of Queue Heads (each Queue Head symbolizes one transaction) in the main memory. The individually parts of each transaction are attached as a linked list of Transfer Descriptors to each Queue Head. This is true for interrupt, control and bulk transfers. If there are no isochronous transfers during a frame the Frame List Pointer would have to point to next Queue Head of one of the other transfer-types.



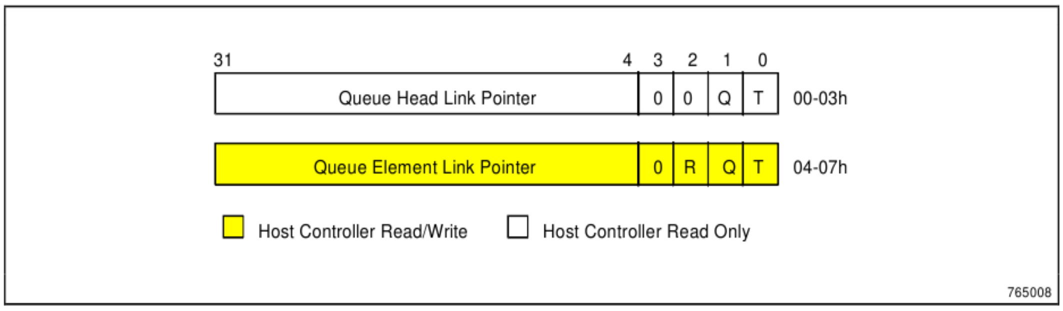
Bit	Description
31:4	**Frame List Pointer (FLP).** This field contains the address of the first data object to be processed in the frame and corresponds to memory address signals [31:4], respectively.
3:2	**Reserved.** These bits must be written as 0s.
1	**QH/TD Select (Q).** 1=QH. 0=TD. This bit indicates to the hardware whether the item referenced by the link pointer is a TD or a QH. This allows the Host Controller to perform the proper type of processing on the item after it is fetched.
0	**Terminate (T).** 1=Empty Frame (pointer is invalid). 0=Pointer is valid (points to a QH or TD). This bit indicates to the Host Controller whether the schedule for this frame has valid entries in it.

Figure 3.3 Frame list pointer and bit roles

3.2.3 Queue Head

Queue Heads as shown in Figure 3.4, are used to separate the individual transactions from each other. They consist of two main parts a queue head part and a queue element part. First, the queue head field is used to generate the linked list of the Queue Headers. The end of this list is signaled to the hardware by marking the memory address in the address field invalid through the T- field. This corresponds to the way a Frame List Pointer gets marked inactive. The Q- field which is also available in the queue head region must always be set to one, because after a control, interrupt or bulk transfer is reached transfer types that make use of Queue Head will follow. The only transfer type which does not make use of Queue Headers is the isochronous transfer and these transfers have to be always at the beginning of a frame.

The second part of a Queue Head is used to point to the next Transfer Descriptor, that belongs to this transfer. After processing this Transfer Descriptor, the hardware updates this part of the Queue Header to point to the next Transfer Descriptor of this transfer. This part of the Queue Head is structured the same way as the queue head part, but since the queue element part only points to next Transfer Descriptor its Q- field must always be zero


31
4
3
2
1
0
Queue Head Link Pointer
0
0
Q
T
00-03h
Queue Element Link Pointer
0
R
Q
T
04-07h
Host Controller Read/Write
Host Controller Read Only
765008

Bit	Description
31:4	**Queue Head Link Pointer (QHLP).** This field contains the address of the next data object to be processed in the horizontal list and corresponds to memory address signals [31:4], respectively.
3:2	**Reserved.** These bits must be written as 0s.
1	**QH/TD Select (Q).** 1=QH. 0=TD. This bit indicates to the hardware whether the item referenced by the link pointer is another TD or a QH. This allows the Host Controller to perform the proper type of processing on the item after it is fetched.
0	**Terminate (T).** 1=Last QH (pointer is invalid). 0=Pointer is valid (points to a QH or TD). This bit indicates to the Host Controller that this is the last QH in the schedule. If there are active TDs in this queue, they are the last to be executed in this frame.

Bit	Description
31:4	**Queue Element Link Pointer (QELP).** This field contains the address of the next TD or QH to be processed in this queue and corresponds to memory address signals [31:4], respectively.
3	**Reserved.** This bit must be 0.
2	**Reserved.** This bit has no impact on operation. It may vary simply as a side effect of the Queue Element pointer update.
1	**QH/TD Select (Q).** 1=QH. 0=TD. This bit indicates to the hardware whether the item referenced by the link pointer is another TD or a QH. This allows the Host Controller to do the proper type of processing on the item after it is fetched. For entries in a queue, this bit is typically set to 0.
0	**Terminate (T).** 1=Terminate (No valid queue entries). This bit indicates to the Host Controller that there are no valid TDs in this queue. When HCD has new queue entries it overwrites this value with a new TD pointer to the queue entry.

Figure 3.4 Queue Head and bit roles

3.2.4 Transfer Descriptor

Transfer Descriptors (Figure 3.5) represent a transaction between the host controller and a function. They can be separated into four parts. The first part, the so called link pointer, is used to generate a linked list of several Transfer Descriptors which belong together. Therefor it consists of an address field and a Q-field that determines the type of the element the address-field points to. If a Transfer Descriptor is the last one in a interrupt, control or bulk transfer transaction the address- field gets marked invalid by setting the T- field. Beside these fields there is the Vf- field. This field determines the next

Transfer Descriptor, after processing the current Transfer Descriptor and updating the corresponding Queue Header, that will be processed whether of the current transaction (depth-first). Or whether the next available Queue Header is processed instead (breath-first).

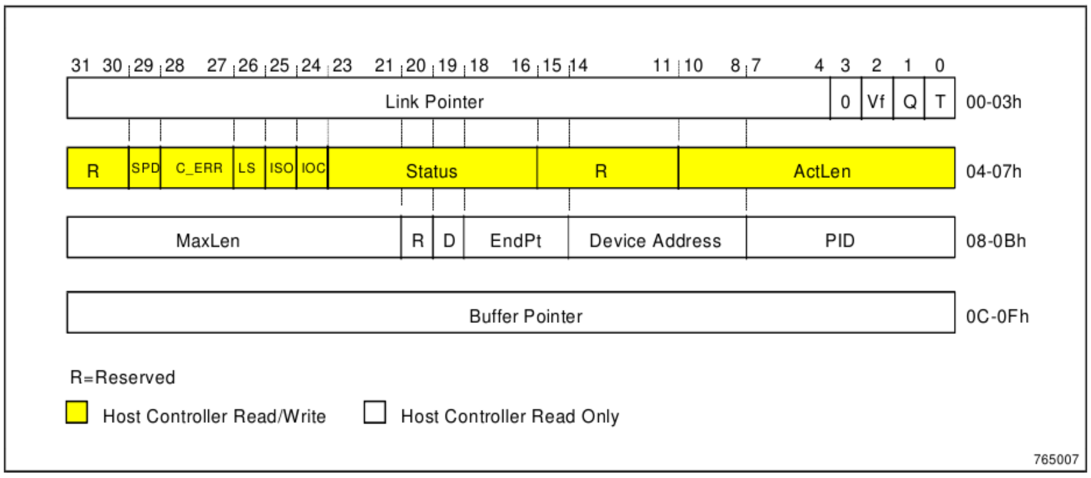


Fig 3.5 Transfer Descriptor , roles of bit fields are explained below

The next part is the so called control section. It gives the hardware additional information about the function, with whom the communication will take place (if the function is a low-speed device the LS-field has to be set), and about how to react on certain events during the communication. The host controller driver can be told after how many errors the transaction gets marked as invalid through the ERR- field. Via the IOC- field, whether there should be an interrupt at the end of the whole frame in which this Transfer Descriptor was processed or not, can be controlled. The hardware can determine through the ISO-field if the current Transfer Descriptor is part of an isochronous transfer or not. During processing the host controller gives the host controller driver feedback about the status of the transfer through the STATUS-field. This field tells the host controller driver if the host controller has already started processing the current Transfer Descriptor, or if the processing was stopped or canceled because of errors that occurred. Beside the STATUS-field the ACTLEN- field tells the host controller driver how many bytes of the Transfer Descriptor have already been processed by the hardware.

The third part is called token section which stores the total number of bytes that must be processed by the hardware in the MAXLEN-field. The D- field determines if the so called toggle bit must be set or not. The toggle mechanism is used as an alternating bit sequence of one bit to synchronize the data transfer between host and a pipe on function. The pipe is clearly identified through the function id (which is stored in the DEVICE ADDRESS- field) and the endpoint number (ENDPT- field) on this function. The PID- field specification the type of transfer that must be generated by the controller for this Transfer Descriptor. Possible values are IN (this is a transaction which moves data from a function to the host controller), OUT (moves data from the host controller to the function) and SETUP (with a SETUP-transfer the configuration of a function can be changed by the host controller or by other higher layers).

The last part of a Transfer Descriptor only contains the starting-address of the memory block that must be transfer to the function or that stores the data, sent from the function to the host, after the transfer has been successfully determined. Beside these four parts the USB specification reserves another 32 bytes in the Transfer Descriptor. This part is not used by the host controller and can therefor be used by the host controller driver

3.3 Transfer queuing and Traversal state diagram

The hardware begins the execution of a frame by determining the current Frame List Pointer through the Start-Of-Frame Register and broadcasting a Start-Of-Frame packet to all connected functions signaling that the previous frame has ended. Mean while the hardware analyzes the Frame List Pointer. If it is not valid (T-field of the Frame List Pointer is set) the hardware idles one millisecond and starts all over again.But, if the Frame List Pointer is valid the hardware checks whether the referenced data structure is a Transfer Descriptor or a Queue Header and begins processing. If the referenced data structure is a Queue Header the hardware determines whether to follow the Transfer Descriptors striping through all transfers of this frame (in the UHCI specification this is called horizontal context) or along the current transfer (the so called vertical context) by analyzing the Vf- field of the Transfer Descriptor referenced by the Queue Header. After processing this Transfer Descriptor and updating the Queue Head (the element link pointer points to the next not jet processed Transfer Descriptor of the transfer) the hardware fetches the next Transfer Descriptor.

In vertical context this is the next Transfer Descriptor of the current transfer of the current transfer. If the processing context is horizontal, the next Transfer Descriptor of the following Queue Header is fetched and processed. The vertical context offers a successive way of processing all the transfers in a frame whereas the horizontal context offers a concurrent processing model of the frame data. During the processing of a Transfer Descriptor the hardware transmits the transfer to the function. The ACTLEN-field of the Transfer Descriptor gets updated during processing the memory. In case of errors the transfer may be canceled (depending on the ERR-field, defining the number of errors that may occur without canceling the transfer) and the STATUS-field gets updated with an error code. If the transfer is not canceled and the end of the Transfer Descriptor is reached the STATUS-field gets marked as inactive. If the Transfer Descriptor is part of a isochronous transfer (not Queue Head controlled) the next Transfer Descriptor gets fetched and processed,because isochronous transfers do not make use of Queue Headers, until the frame is terminated or a Queue Header is reached the state diagram in Figure 3.6 explains all this really well.

3.4 Choice of UHCI

When considering a system level programming project like this one we have to consider factors like debugging, previous work on this domain, development environment, testing etc, after my extensive searching I found out that Qemu should be use as my hardware Virtualizer, as minix is well supported and tested by developers on it, at the time of development of this project UHCI emulation was really well supported by Qemu than other host controllers. Driver for UHCI is bit complex, as major part is done in software than hardware, more than any thing else it is the Qemu's debugging support that keep me biased to UHCI.

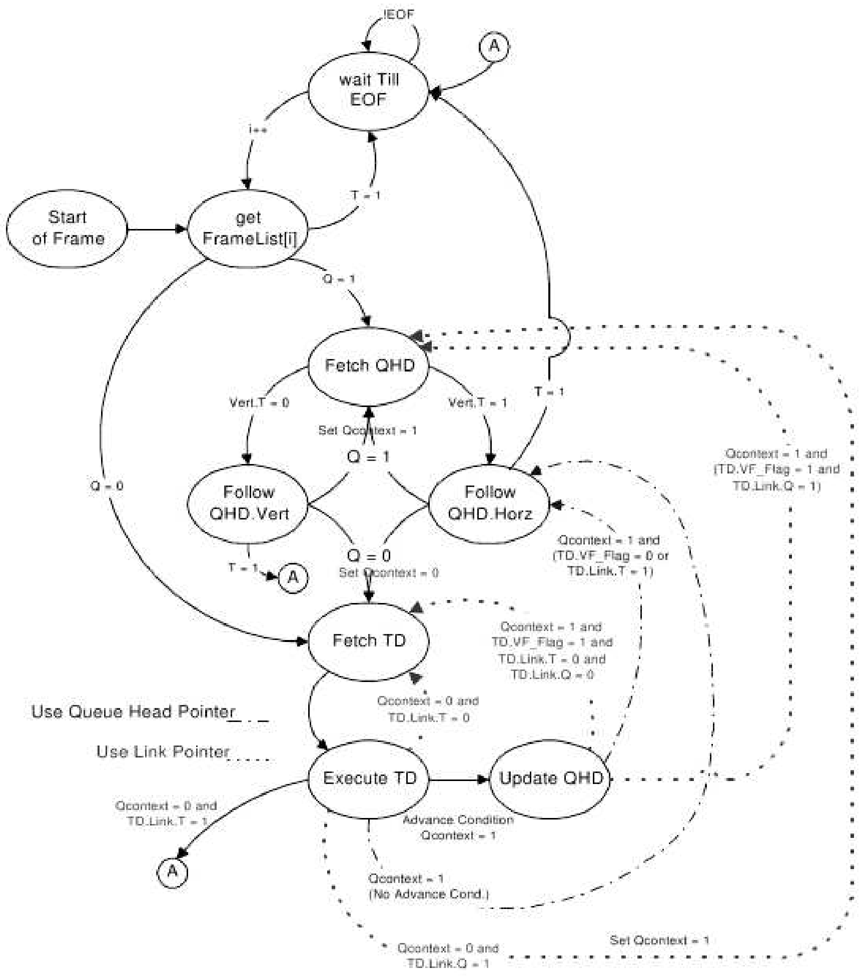

!EOF
A
wait Till EOF
i++
T = 1
Start of Frame
get FrameList[i]
Q = 1
Fetch QHD
T = 1
Vert.T = 0
Vert.T = 1
Set Qcontext = 1
Q = 1
Q = 0
Follow QHD.Vert
Follow QHD.Horz
Qcontext = 1 and (TD.VF_Flag = 1 and TD.Link.Q = 1)
Qcontext = 1 and (TD.VF_Flag = 0 or TD.Link.T = 1)
T = 1
A
Q = 0
Set Qcontext = 0
Fetch TD
Qcontext = 1 and TD.VF_Flag = 1 and TD.Link.T = 0 and TD.Link.Q = 0
Use Queue Head Pointer
Qcontext = 0 and TD.Link.T = 0
Use Link Pointer
Execute TD
Update QHD
Qcontext = 0 and TD.Link.T = 1
A
Advance Condition Qcontext = 1
Qcontext = 1 (No Advance Cond.)
Set Qcontext = 1
Qcontext = 0 and TD.Link.Q = 1

Figure 3.6 Transfer queuing and traversal state diagram

CHAPTER 4
Project Scope and Limitations

This chapter explains about the exact scope of this project and does explain the limitations it would be having once completed , this is just elaboration of the project abstract.

4.1 Scope

When we verbally say 'USB stack' or 'USB subsystem' it is completely vague, in other words it is the actual scope that defines the USB stack, USB specification is really complex and complying to the USB standard is really a challenging task. Within the project time frame allocated and what I researched on USB ,along with a single programmer working on it ,I have defined the scope in the following points :

- Low / Full speed support
- Control transfers
- Interrupt transfers
- USB driver interface (USBDI)
- Host controller driver interface (HCDI)
- Multiple host controller support
- UHCI host controller driver
- USBDI for Human Interface Device (HID) class driver
- Dummy HID Keyboard and Mouse Driver using boot protocol.

4.2 Limitations

- Bulk transfer
- Isochronous transfer
- Limited to 2 ports of UHCI root hub

- Synchronous Control transfer (usual is asynchronous)
- Single driver Single device (usually multiple driver can handle endpoints)
- No Direct Memory Access (DMA support)
- Performance and efficiency is not considered
- Power management

As we have defined the scope and limitation the following chapters would explain the design , development environment, implementation and testing.

CHAPTER 5
Design of the Minix USB Subsystem

This chapter explains the software design of the Minix USB stack, I explain why have I chosen such a design along with the advantages and drawbacks , further I brief how to achieve that in Minix.

5.1 Design

Original inspiration for the design came from the Minix 3 operating system, Minix 3 has a Microkernel design (see chapter 1) , with this design as a base ,i have divided the USB stack into 3 independent components that communicate each other with Minix Message IPC (Inter Process Communication).The three components namely USBD , USBDI , HCDI run in separate address space isolated from each other , which makes the design much more reliable and fault tolerant ie. If one of them crash others will keep on running. In simple terms I have followed the client - server architecture.

Main design principles are given below

- Modularity
- Reliability
- Scalability
- Fault tolerant

5.1.1 Modularity

A modular approach is an approach that subdivides a system into smaller parts (modules) that can be independently created and then used to drive multiple functionality ,further the advantages are flexibility in designs, augmentation (adding new solution by merely plugging in a new module), and exclusion. In our design this is achieved by dividing the entire USB stack into separate user mode process that are independent of each other, further they communicate each other through message passing , even if other end is not alive it doesn't affect the sender. The disadvantage of modularity is performance in our case USBD would have run time over head in logarithmic time ie in Big O , O(log(n)).

5.1.2 Reliability

As a result of modular design what we gain is reliability, so what exactly is reliability with respect to our design ?

- It does what it is suppose to do as expected in time
- Resisting the failure infecting on other modules

Thus we achieve reliability through modularity as we separate them into individual processes failure in one doesn't affect others, it does as designed with help of rendezvous message IPC the minix provides.

5.1.3 Scalability

Scalability is a desirable property of a system, which indicates its ability to either handle growing amounts of work in a graceful manner, we attain scalability as the entire design is made as generic as possible , in our case any number of host controller drivers could be hooked to USBD , along with the any number of device drivers using USBDI , scalability is only limited to amount of memory available.

5.1.4 Fault Tolerant

Fault-tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components. We gain this feature from the minix inherent design of how device drivers work (see chapter 1) further as I have explained above since each part of the subsystem execute as a single independent process ,even if one of them goes down or some fault is trying to spread it doesn't affect others , say in our case if UHCI-HCD goes down , and if any transfer was on progress with USBD and OHCI-HCD it will have zero effect on them , thus we gain reliability along with fault tolerance.

Alternatively I could have designed a monolithic USB stack where by which we compile entire drivers into the stack ,this monolithic design give a good performance boost but under the cost of proposed design principles. Drawback of proposed design is the message passing overhead at USBD which might make USBD non responsive at times . but will continue to work ,next drawback since we are using the message passing IPC which are rendezvous there could be a possibility of hidden deadlock which might pop up in logarithmic time period. The proposed design is showing in the Figure 5.0

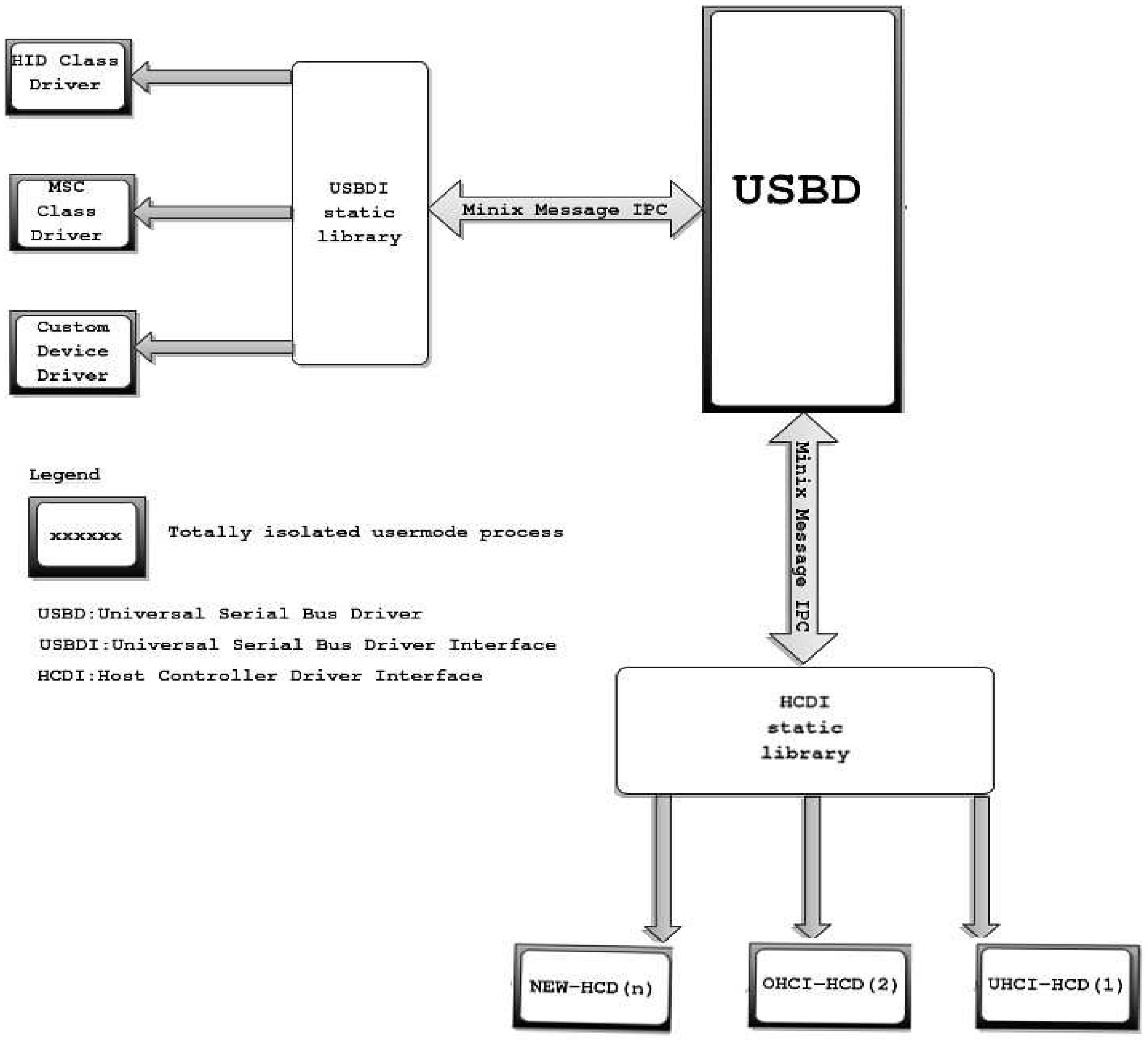


Figure 5.0 Conceptual View of the proposed design

5.2 Brief idea of implementation in minix

Its been explained that entire elements run as user mode process , now from Figure 5.0 we can understand that only USBD is the standalone process that act as a server for the clients USBDI and HCDI ,one thing to keep in mind is that USBDI and HCDI are not process rather a static library interface to which the clients link. The communication protocol from clients to USBD is completely hidden in these interfaces , as far as clients are concerned they are just linking to C API (Application Programming Interface) nothing more. When they start to execute they interface handles the rest , but it is mandatory that clients follow a systematic approach to use of the C API as like to register , deregister etc. One problem with it approach is that there is duplication of the static library in memory for each clients well at the moment we don't deal with shared library which is an obvious solution to it.

5.3 Drawbacks of proposed design

- Dead locking code
- As no threads are used , USBD have high latency response
- Replication of static library in memory for each driver
- If USBD goes down clients are stateless and have to be restarted with USBD

CHAPTER 6
Development Environment

This chapter explains the development environment ,tools and compiler used along with how to set I it up for the purpose.

6.1 Development Environment

Since it is system level programming project , what we mainly need is a good debugging support with this focus in mind I decided to run Minix on a hardware virutualizer which had good debugging support details of tools used are given below

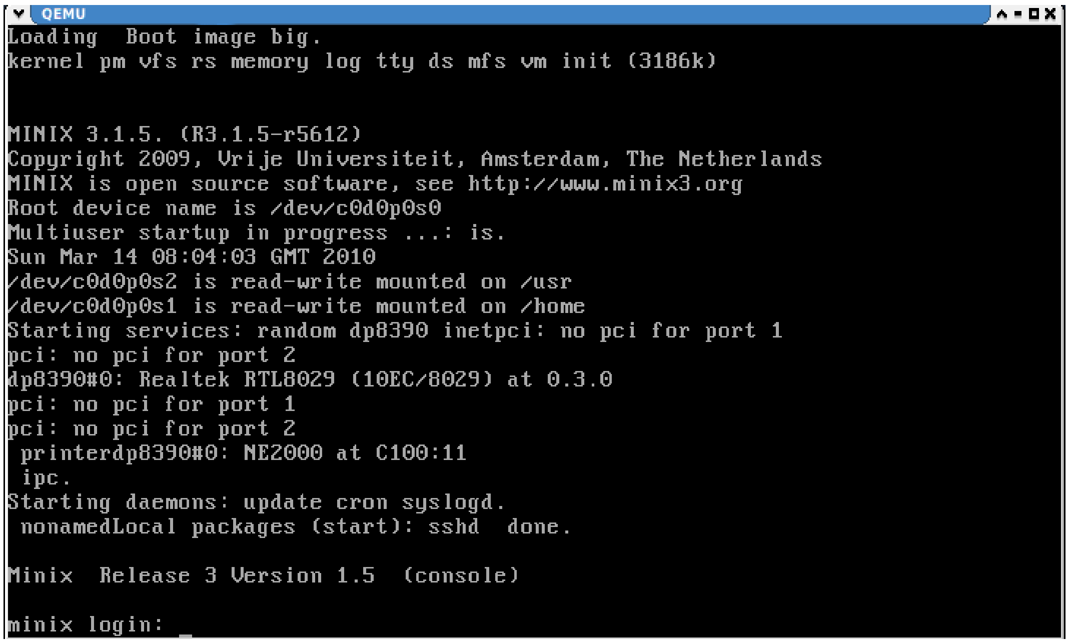
- Qemu 0.11.1 : Hardware virtualizer
- Minix 3.1.5 svn5612
- Linux 2.6.33 (Slackware 13) : Host platform
- ACK C compiler (Minix default)
- SSH server on minix
- SSH FS on Linux
- cscope : to browse / grep minix code
- Editor : Vim

6.2 Setting Up

Main tool that helped for fast development is the sshfs (linux) along with sshd on minix without which development would be really slow. This is how entire setting up goes. I have minix 3.1.5 installed in qemu along with the ssh server ,once minix boots up(Figure 6.0) I will have a qemu monitor (Figure 6.1) which I can use it to control qemu, such that I would be able to virtually add new usb device , detach them etc, another window I have a ssh login session (Figure 6.2) where by which I compile the code, one advantage is the scrolling support which is absent in Minix console, next for debugging I redirect the out put from minix's *\/var/log/messages* (Figure 6.3) using ssh. Main factor is the ability to mount a virtual minix file system to minix using sshfs ,this has helped me lot, advantage of this approach is that we could use local tools in Linux to prepare a IDE say I used Vim for it (Figure 6.4).

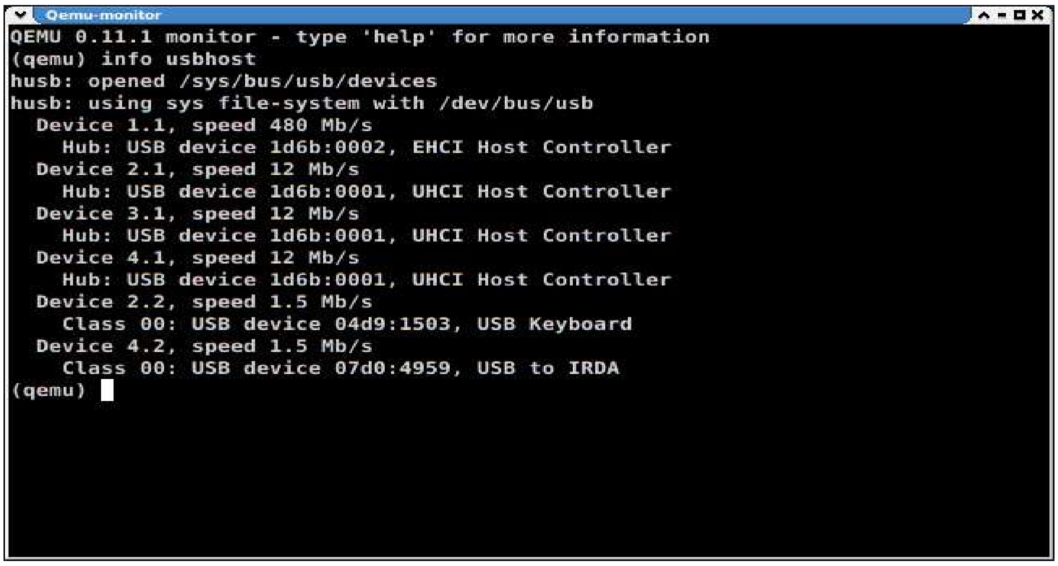
6.2.1 Qemu 0.11.1 configuration

```
./qemu -name Minix3.1.4svn5547 -monitor stdio -usb -redir tcp:2002::22 -m 300M \
                                            -hda ./vm-hd/Minix-3.1.5r.img       \
                                            -net user -net nic,model=ne2k_pci
```



```
Loading  Boot image big.
kernel pm vfs rs memory log tty ds mfs vm init (3186k)


MINIX 3.1.5. (R3.1.5-r5612)
Copyright 2009, Vrije Universiteit, Amsterdam, The Netherlands
MINIX is open source software, see http://www.minix3.org
Root device name is /dev/c0d0p0s0
Multiuser startup in progress ...: is.
Sun Mar 14 08:04:03 GMT 2010
/dev/c0d0p0s2 is read-write mounted on /usr
/dev/c0d0p0s1 is read-write mounted on /home
Starting services: random dp8390 inetpci: no pci for port 1
pci: no pci for port 2
dp8390#0: Realtek RTL8029 (10EC/8029) at 0.3.0
pci: no pci for port 1
pci: no pci for port 2
 printerdp8390#0: NE2000 at C100:11
 ipc.
Starting daemons: update cron syslogd.
 nonamedLocal packages (start): sshd  done.

Minix  Release 3 Version 1.5  (console)

minix login: _
```

Figure 6.0 Minix 3.1.5 on qemu 0.11.1



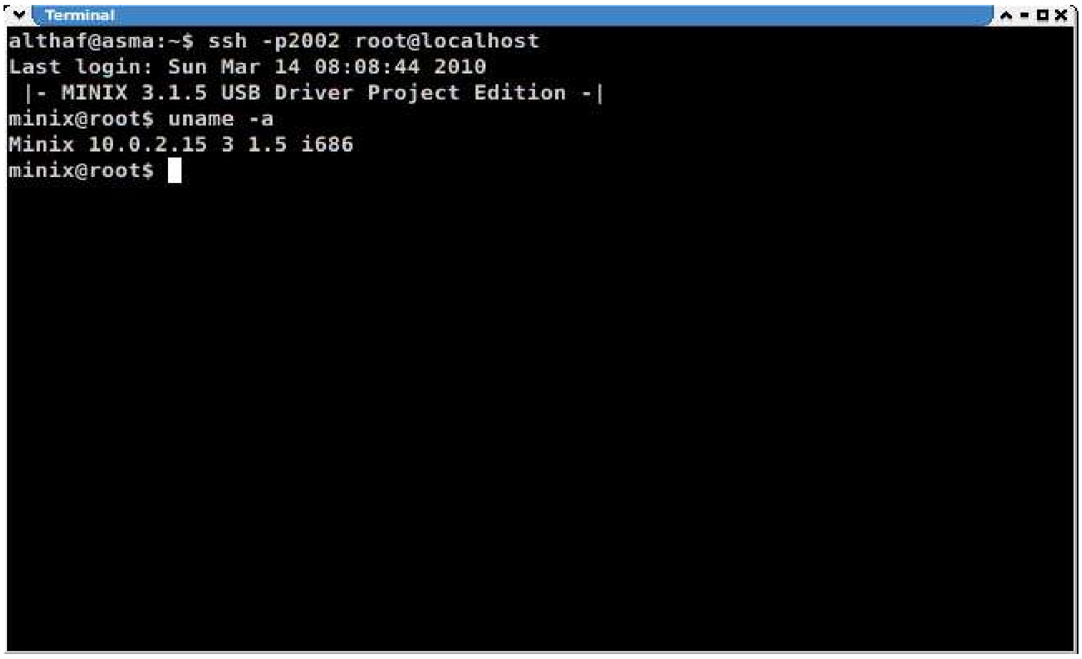
```
QEMU 0.11.1 monitor - type 'help' for more information
(qemu) info usbhost
husb: opened /sys/bus/usb/devices
husb: using sys file-system with /dev/bus/usb
  Device 1.1, speed 480 Mb/s
    Hub: USB device 1d6b:0002, EHCI Host Controller
  Device 2.1, speed 12 Mb/s
    Hub: USB device 1d6b:0001, UHCI Host Controller
  Device 3.1, speed 12 Mb/s
    Hub: USB device 1d6b:0001, UHCI Host Controller
  Device 4.1, speed 12 Mb/s
    Hub: USB device 1d6b:0001, UHCI Host Controller
  Device 2.2, speed 1.5 Mb/s
    Class 00: USB device 04d9:1503, USB Keyboard
  Device 4.2, speed 1.5 Mb/s
    Class 00: USB device 07d0:4959, USB to IRDA
(qemu)
```

Figure 6.1 Qemu Monitor

6.2.2 Mounting Minix file system using sshfs

sshfs -p2002 root@localhost:/usr/src/drivers/usb ~/minixfs

ssh -p2002 root@localhost

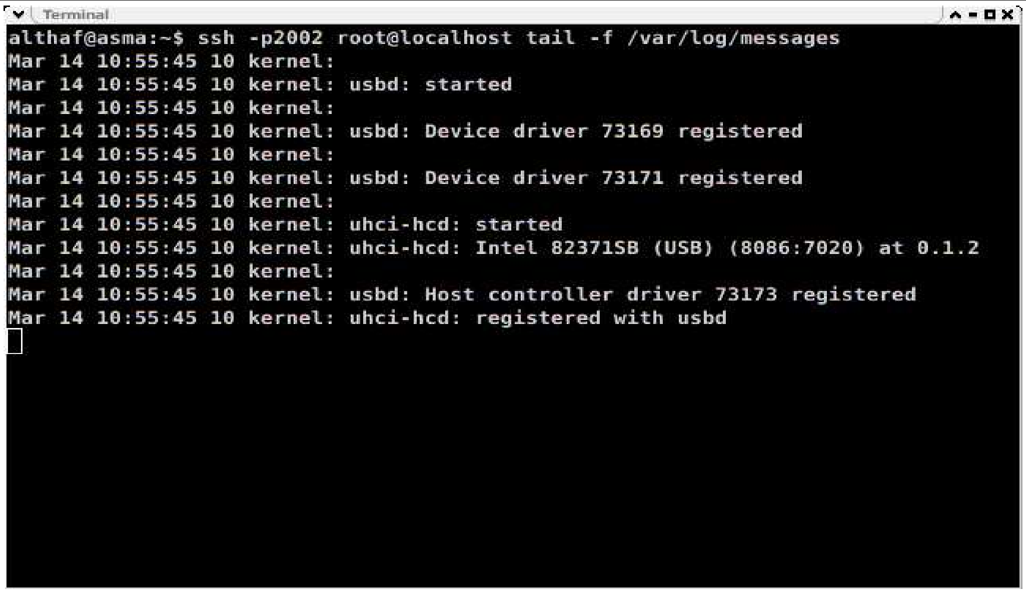


```
althaf@asma:~$ ssh -p2002 root@localhost
Last login: Sun Mar 14 08:08:44 2010
 |- MINIX 3.1.5 USB Driver Project Edition -|
minix@root$ uname -a
Minix 10.0.2.15 3 1.5 i686
minix@root$
```

Figure 6.2 local ssh session for compilation purpose

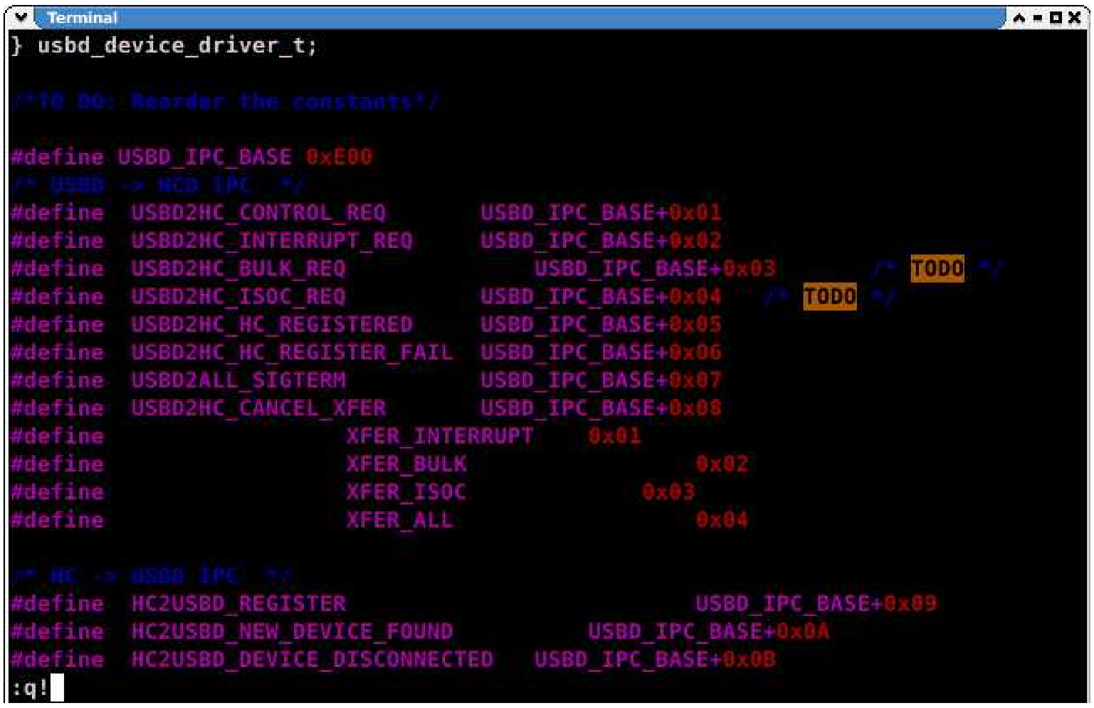
6.2.3 Redirecting kernel messages from /var/log/messsages

ssh -p2002 root@localhost tail -f /var/log/messages



```
althaf@asma:~$ ssh -p2002 root@localhost tail -f /var/log/messages
Mar 14 10:55:45 10 kernel:
Mar 14 10:55:45 10 kernel: usbd: started
Mar 14 10:55:45 10 kernel:
Mar 14 10:55:45 10 kernel: usbd: Device driver 73169 registered
Mar 14 10:55:45 10 kernel:
Mar 14 10:55:45 10 kernel: usbd: Device driver 73171 registered
Mar 14 10:55:45 10 kernel:
Mar 14 10:55:45 10 kernel: uhci-hcd: started
Mar 14 10:55:45 10 kernel: uhci-hcd: Intel 82371SB (USB) (8086:7020) at 0.1.2
Mar 14 10:55:45 10 kernel:
Mar 14 10:55:45 10 kernel: usbd: Host controller driver 73173 registered
Mar 14 10:55:45 10 kernel: uhci-hcd: registered with usbd
```

Figure 6.3 Redirecting kernel messages



```
} usbd_device_driver_t;

/*TO DO: Reorder the constants*/

#define USBD_IPC_BASE 0xE00
/* USBD -> HCD IPC  */
#define  USBD2HC_CONTROL_REQ       USBD_IPC_BASE+0x01
#define  USBD2HC_INTERRUPT_REQ     USBD_IPC_BASE+0x02
#define  USBD2HC_BULK_REQ              USBD_IPC_BASE+0x03          /* TODO */
#define  USBD2HC_ISOC_REQ          USBD_IPC_BASE+0x04    /* TODO */
#define  USBD2HC_HC_REGISTERED     USBD_IPC_BASE+0x05
#define  USBD2HC_HC_REGISTER_FAIL  USBD_IPC_BASE+0x06
#define  USBD2ALL_SIGTERM          USBD_IPC_BASE+0x07
#define  USBD2HC_CANCEL_XFER       USBD_IPC_BASE+0x08
#define                 XFER_INTERRUPT    0x01
#define                 XFER_BULK                 0x02
#define                 XFER_ISOC             0x03
#define                 XFER_ALL                  0x04

/* HC -> USBD IPC  */
#define  HC2USBD_REGISTER                         USBD_IPC_BASE+0x09
#define  HC2USBD_NEW_DEVICE_FOUND         USBD_IPC_BASE+0x0A
#define  HC2USBD_DEVICE_DISCONNECTED   USBD_IPC_BASE+0x0B
:q!
```

Figure 6.4 Editing file from minix locally in linux using vim

Once this development environment was setup , half of the headache while it were development within minix was revealed, as minix would be really show while running on a emulator it will not be fit for an IDE, the next chapter explains the implementation of the USB subsystem.

CHAPTER 7
Implementation

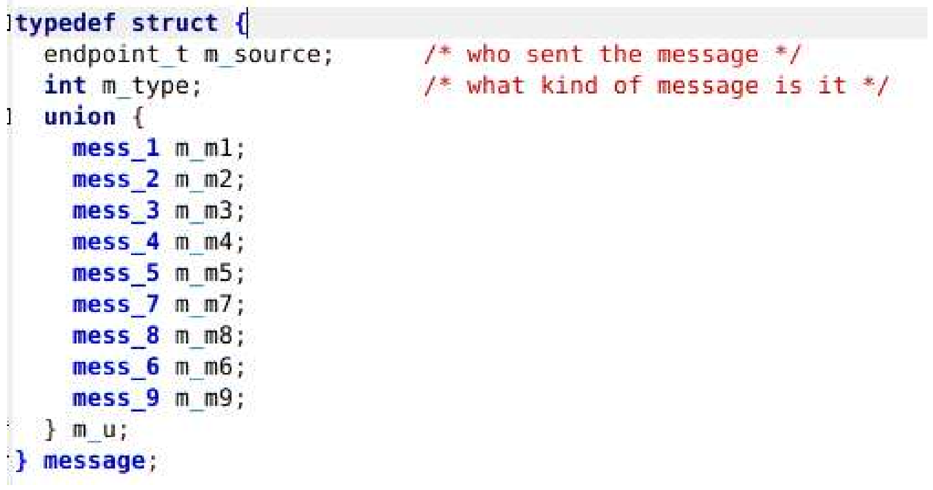
Explains of how the proposed designed was implemented ,to be exact this chapter will elaborate the various APIs the stack provides , this chapter doesn't explaind any language specific implementation , for that purpose check the Appendix 1.

7.1 Implementation of USB subsystem

It started from bottom to top ,ie starting from host controller driver. After understanding the UHCI design guide [5] ,the work was started from HCD for UHCI, even though HCDI never existed at time of development ,i had good idea of how it should be implemented after understanding Linux and Net BSD usb stack, further during development I alternated my self in understanding the USB 2.0 specification.

7.2 Minix 3 services

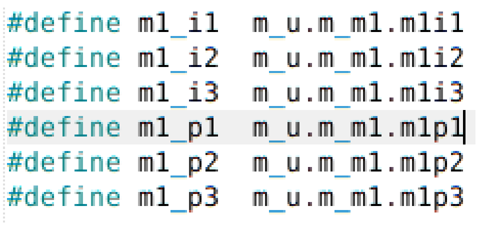
Entire USB stack is based on minix 3 message passing IPC,which has support for synchronous and asynchronous message passing , each component communicate each other through a well defined custom protocol for this purpose which will be discussed below.



```
typedef struct {
  endpoint_t m_source;      /* who sent the message */
  int m_type;               /* what kind of message is it */
  union {
    mess_1 m_m1;
    mess_2 m_m2;
    mess_3 m_m3;
    mess_4 m_m4;
    mess_5 m_m5;
    mess_7 m_m7;
    mess_8 m_m8;
    mess_6 m_m6;
    mess_9 m_m9;
  } m_u;
} message;
```

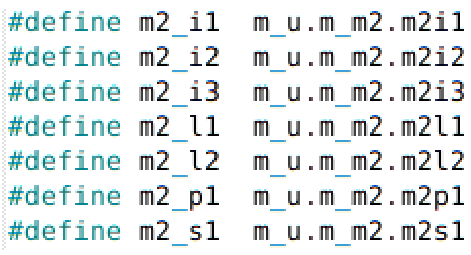
Figure 7.0 structure that defines message data type

We basically use 3 types of messages (Figure 7.1,7.2,7.3) from the 9 provided by minix, the message types used are shown below.



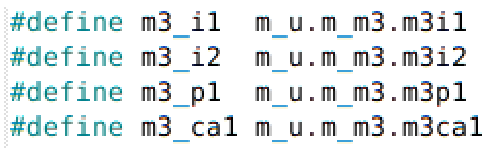
```
#define m1_i1  m_u.m_m1.m1i1
#define m1_i2  m_u.m_m1.m1i2
#define m1_i3  m_u.m_m1.m1i3
#define m1_p1  m_u.m_m1.m1p1
#define m1_p2  m_u.m_m1.m1p2
#define m1_p3  m_u.m_m1.m1p3
```

Figure 7.1 Message format 1



```
#define m2_i1  m_u.m_m2.m2i1
#define m2_i2  m_u.m_m2.m2i2
#define m2_i3  m_u.m_m2.m2i3
#define m2_l1  m_u.m_m2.m2l1
#define m2_l2  m_u.m_m2.m2l2
#define m2_p1  m_u.m_m2.m2p1
#define m2_s1  m_u.m_m2.m2s1
```

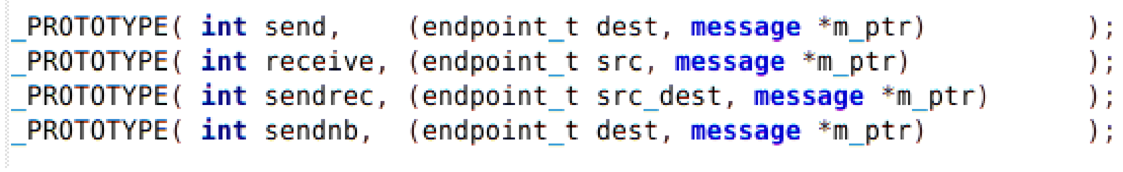
Figure 7.2 Message format 2



```
#define m3_i1  m_u.m_m3.m3i1
#define m3_i2  m_u.m_m3.m3i2
#define m3_p1  m_u.m_m3.m3p1
#define m3_ca1 m_u.m_m3.m3ca1
```

Figure 7.3 Message format 3

These are the core message formats used for our IPC , message passing primitives that are used from minix is shown in Figure 7.4



```
_PROTOTYPE( int send,    (endpoint_t dest, message *m_ptr)          );
_PROTOTYPE( int receive, (endpoint_t src, message *m_ptr)           );
_PROTOTYPE( int sendrec, (endpoint_t src_dest, message *m_ptr)      );
_PROTOTYPE( int sendnb,  (endpoint_t dest, message *m_ptr)          );
```

Figure 7.4 Minix message passing primitives used for our purpose

All the calls are synchronous and rendezvous except *sendnb()* which is a non blocking call .

7.3 Subsystem components

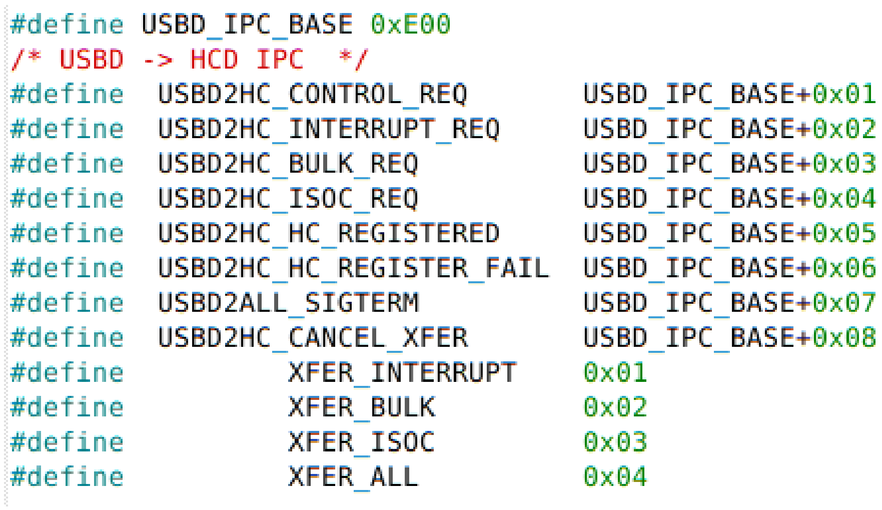
As explained in chapter 5 there are 3 components in the implementation

- USBD : USB driver which comply usb specification
- USBDI : a static library meant for device drivers
- HCDI : a static library to which HCDs link to

7.3.1 USBD

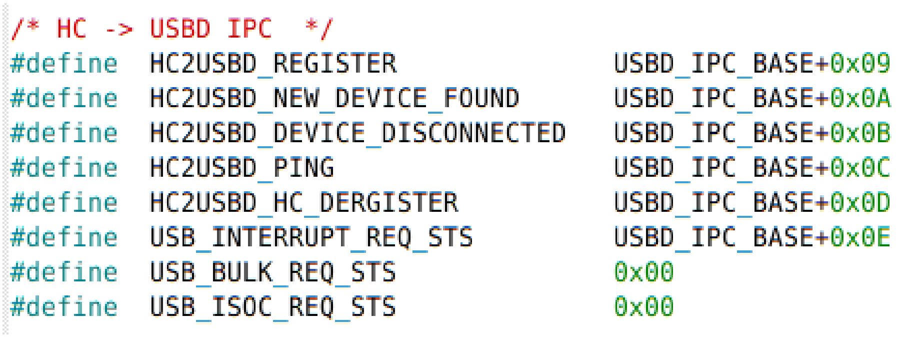
USBD is the core of the stack , it can be considered as the glue between host controllers and the device drivers , it is USBD that would be sticking on to and implementing USB specification request and data structures , in our implementation we have defined set of protocols that USBDI and HCDI uses to communicate with USBD.

These are shown below



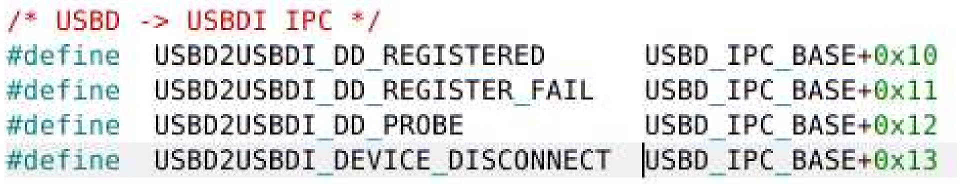
```
#define USBD_IPC_BASE 0xE00
/* USBD -> HCD IPC  */
#define  USBD2HC_CONTROL_REQ       USBD_IPC_BASE+0x01
#define  USBD2HC_INTERRUPT_REQ     USBD_IPC_BASE+0x02
#define  USBD2HC_BULK_REQ          USBD_IPC_BASE+0x03
#define  USBD2HC_ISOC_REQ          USBD_IPC_BASE+0x04
#define  USBD2HC_HC_REGISTERED     USBD_IPC_BASE+0x05
#define  USBD2HC_HC_REGISTER_FAIL  USBD_IPC_BASE+0x06
#define  USBD2ALL_SIGTERM          USBD_IPC_BASE+0x07
#define  USBD2HC_CANCEL_XFER       USBD_IPC_BASE+0x08
#define          XFER_INTERRUPT    0x01
#define          XFER_BULK         0x02
#define          XFER_ISOC         0x03
#define          XFER_ALL          0x04
```

Figure 7.5 USBD to HCDI



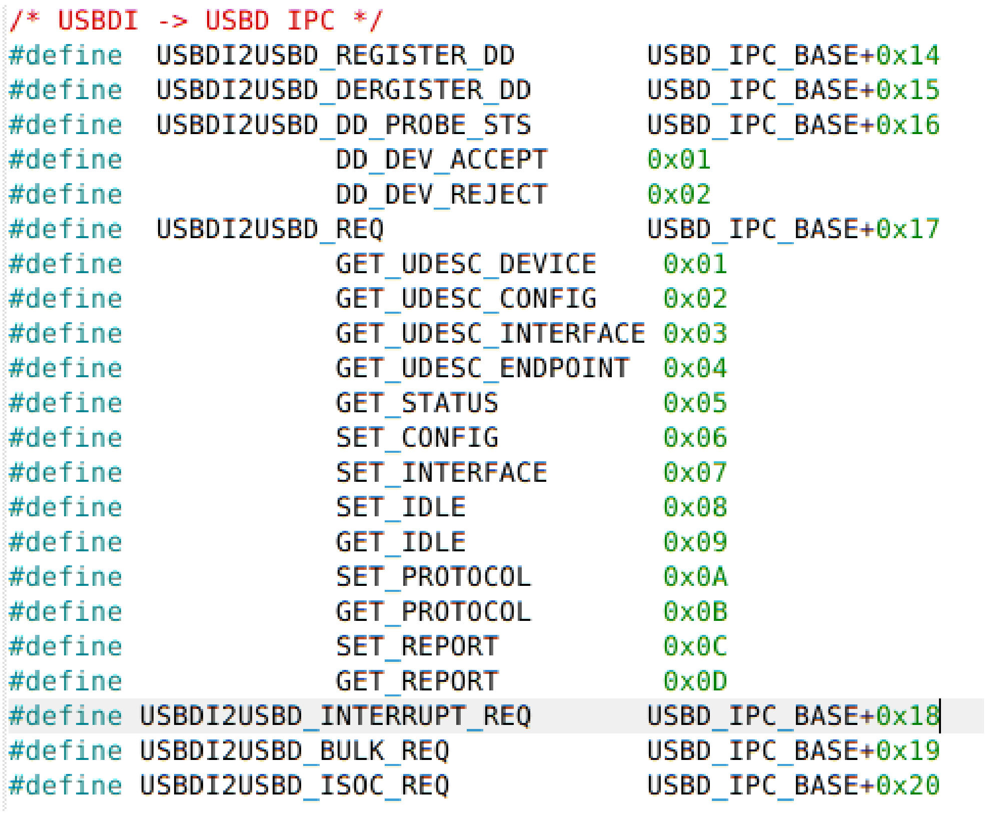
```
/* HC -> USBD IPC  */
#define  HC2USBD_REGISTER            USBD_IPC_BASE+0x09
#define  HC2USBD_NEW_DEVICE_FOUND    USBD_IPC_BASE+0x0A
#define  HC2USBD_DEVICE_DISCONNECTED USBD_IPC_BASE+0x0B
#define  HC2USBD_PING                USBD_IPC_BASE+0x0C
#define  HC2USBD_HC_DERGISTER        USBD_IPC_BASE+0x0D
#define  USB_INTERRUPT_REQ_STS       USBD_IPC_BASE+0x0E
#define  USB_BULK_REQ_STS            0x00
#define  USB_ISOC_REQ_STS            0x00
```

Figure 7.6 HCDI to USBD



```
/* USBD -> USBDI IPC */
#define  USBD2USBDI_DD_REGISTERED      USBD_IPC_BASE+0x10
#define  USBD2USBDI_DD_REGISTER_FAIL   USBD_IPC_BASE+0x11
#define  USBD2USBDI_DD_PROBE           USBD_IPC_BASE+0x12
#define  USBD2USBDI_DEVICE_DISCONNECT  USBD_IPC_BASE+0x13
```

Figure 7.7 USBD to USBDI

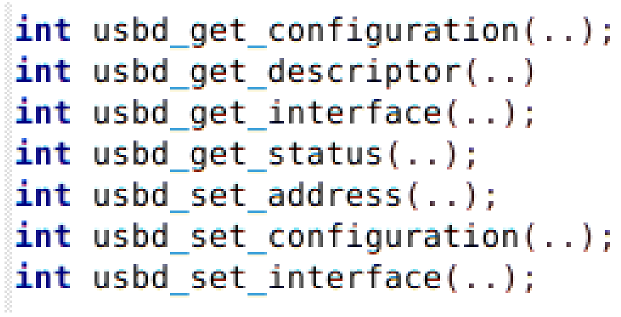


```
/* USBDI -> USBD IPC */
#define  USBDI2USBD_REGISTER_DD        USBD_IPC_BASE+0x14
#define  USBDI2USBD_DERGISTER_DD       USBD_IPC_BASE+0x15
#define  USBDI2USBD_DD_PROBE_STS       USBD_IPC_BASE+0x16
#define             DD_DEV_ACCEPT      0x01
#define             DD_DEV_REJECT      0x02
#define  USBDI2USBD_REQ                USBD_IPC_BASE+0x17
#define             GET_UDESC_DEVICE    0x01
#define             GET_UDESC_CONFIG    0x02
#define             GET_UDESC_INTERFACE 0x03
#define             GET_UDESC_ENDPOINT  0x04
#define             GET_STATUS          0x05
#define             SET_CONFIG          0x06
#define             SET_INTERFACE       0x07
#define             SET_IDLE            0x08
#define             GET_IDLE            0x09
#define             SET_PROTOCOL        0x0A
#define             GET_PROTOCOL        0x0B
#define             SET_REPORT          0x0C
#define             GET_REPORT          0x0D
#define USBDI2USBD_INTERRUPT_REQ       USBD_IPC_BASE+0x18
#define USBDI2USBD_BULK_REQ            USBD_IPC_BASE+0x19
#define USBDI2USBD_ISOC_REQ            USBD_IPC_BASE+0x20
```

Figure 7.7 USBDI to USBD

7.3.1.1 Standard USB requests

Chapter 9 of USB specification [1] specify set of standard requests USBD implementations the following ,



```
int usbd_get_configuration(..);
int usbd_get_descriptor(..)
int usbd_get_interface(..);
int usbd_get_status(..);
int usbd_set_address(..);
int usbd_set_configuration(..);
int usbd_set_interface(..);
```

Figure 7.8 Standard USB request that USBD implements

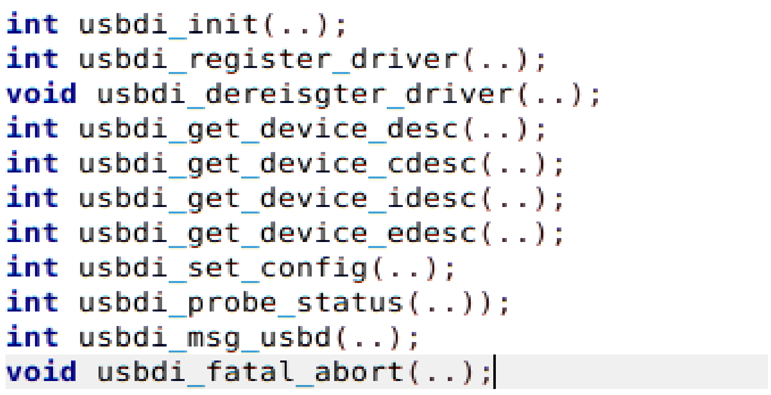
These calls are not directly accessible ,rather these calls are provided as services to USBDI , USBDI as has to make request using the minix message IPC with respect to the protocols defined above.

7.3.1.2 Responsibilities of USBD

- Servicing USBDI for standard USB request.
- Allocating / Deallocating resources for new usb device when attached / detached.
- Mapping Device driver request to appropriate host controller.
- Mapping Device drivers to devices.
- Managing multiple host controller drivers.
- Allocating resources for host controller driver.
- Gracefully terminating itself during panic and informing the clients.
- Bus enumaration see [1] 9.1.2

7.3.2 USBDI

USBDI is a static library (Figure 7.9) that should be linked to and usb device driver so that it could avail service from USBD , USBDI doesn't implement any USB specific standards rather it map (Figure 7.10) the the calls or hides it from the driver actual message passing involved with USBD.



```
int usbdi_init(..);
int usbdi_register_driver(..);
void usbdi_dereisgter_driver(..);
int usbdi_get_device_desc(..);
int usbdi_get_device_cdesc(..);
int usbdi_get_device_idesc(..);
int usbdi_get_device_edesc(..);
int usbdi_set_config(..);
int usbdi_probe_status(..));
int usbdi_msg_usbd(..);
void usbdi_fatal_abort(..);
```

Figure 7.9 USBDI interface for device drivers

usbdi_init(..)

This should be called before the driver try to use any of the usbdi interface, this call actually gets the USBD process endpoint number and stores it in a global variable, further communication / other usbi functions use this value for IPC with USBD.

usbdi_register_driver(..)

When this call is made USBD register the driver process endpoint number in the usbd driver data structure, this information is needed while probing.

usbdi_deregister_driver(..)

This call should be made when the driver wants to exit , so that usbd could release all the resources.

usbdi_get_desc(..)

get the device descriptor see usb specification [1] section 9.4.3

usbdi_get_cdesc(..)

get the configuration descriptor see usb specification [1] section 9.4.2

usbdi_get_idesc(..)

get the interface descriptor see usb specification [1] section 9.4.4

usbdi_get_edesc(..)

get the endpoint descriptor for given config number , interface number and enpoint index.

usbdi_set_config(..)

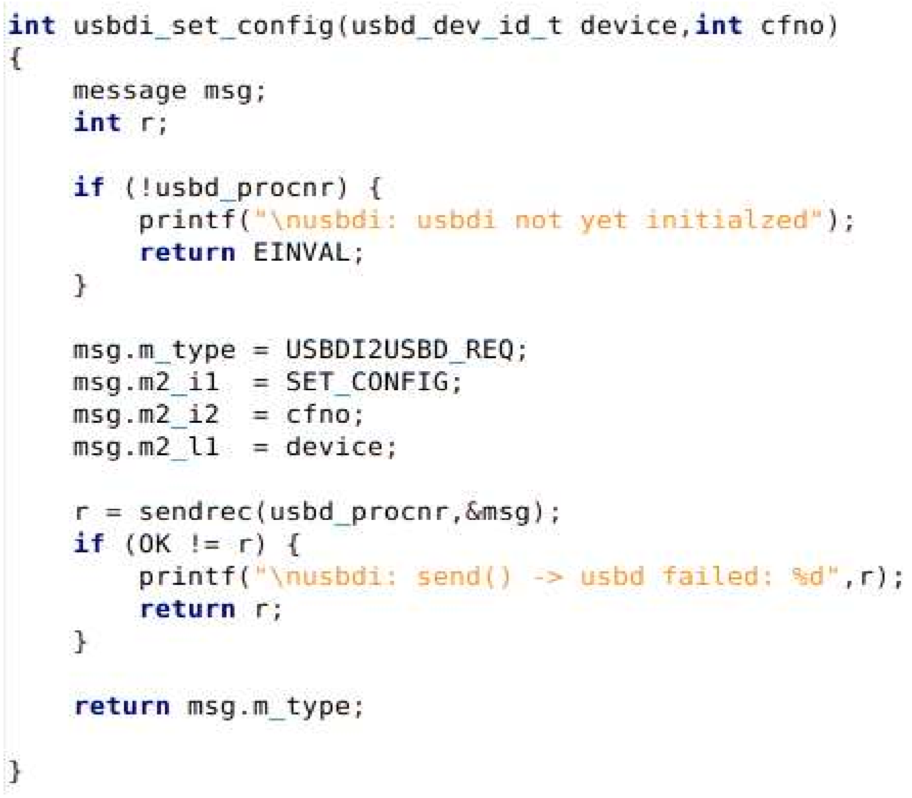
set given configuration number see usb specification [1] section 9.4.7.

usbdi_probe_status(..)

used to reply DD_DEV_ACCEPT / DD_DEV_REJECT as probe satus , if the driver is ready to claim the device or not claiming it.

usbdi_msg_usbd(..)

send custom message to usbd , not used by drivers but for future use.

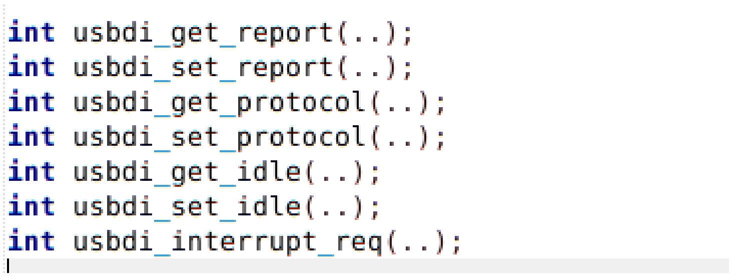


```
int usbdi_set_config(usbd_dev_id_t device,int cfno)
{
    message msg;
    int r;

    if (!usbd_procnr) {
        printf("\nusbdi: usbdi not yet initialzed");
        return EINVAL;
    }

    msg.m_type = USBDI2USBD_REQ;
    msg.m2_i1  = SET_CONFIG;
    msg.m2_i2  = cfno;
    msg.m2_l1  = device;

    r = sendrec(usbd_procnr,&msg);
    if (OK != r) {
        printf("\nusbdi: send() -> usbd failed: %d",r);
        return r;
    }

    return msg.m_type;

}
```

Figure 7.10 Shows how USBDI hides message passing in usbdi_set_config

7.3.2.1 Extending USBDI for HID Request

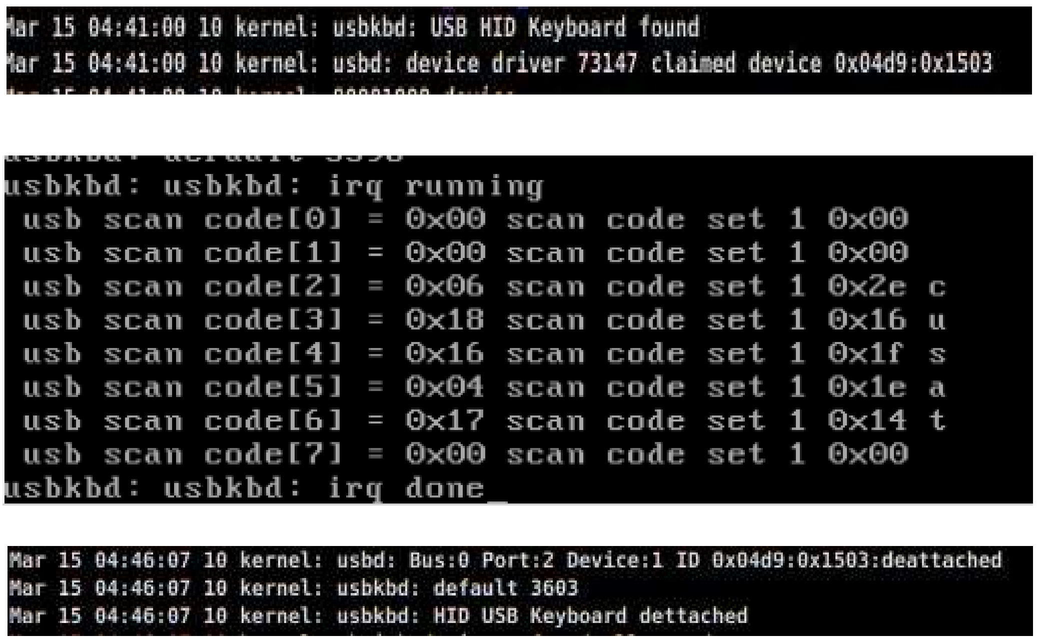
In order show that implementation is working ,I have extended the USBDI for all the HID class requests see [7] chapter 7, further I explain how to write device drivers for HID class devices see Figure 7.11 for USBDI HID request implementation. Apart from the control transfers ,USBD implements interrupt transfers which is usually used by the HID class device for asynchronous interrupt events, USBDI does the mapping for interrupt as shown in Figure 7.11.



```
int usbdi_get_report(..);
int usbdi_set_report(..);
int usbdi_get_protocol(..);
int usbdi_set_protocol(..);
int usbdi_get_idle(..);
int usbdi_set_idle(..);
int usbdi_interrupt_req(..);
```

Figure 7.11 HID request implementation and interrupt request

HID request based dummy USB keyboard and mouse driver code is given in Appendix 2 , Figure 7.12 , 7.13, 7.14, 7.15 shows drivers in action along with the standard descriptors read ,this driver doesn't do much other than getting the scan codes and coordinates,in actual implementation the device driver developer is suppose to handle the rest like integrating it to tty etc. USBI just provides mechanism not policy.

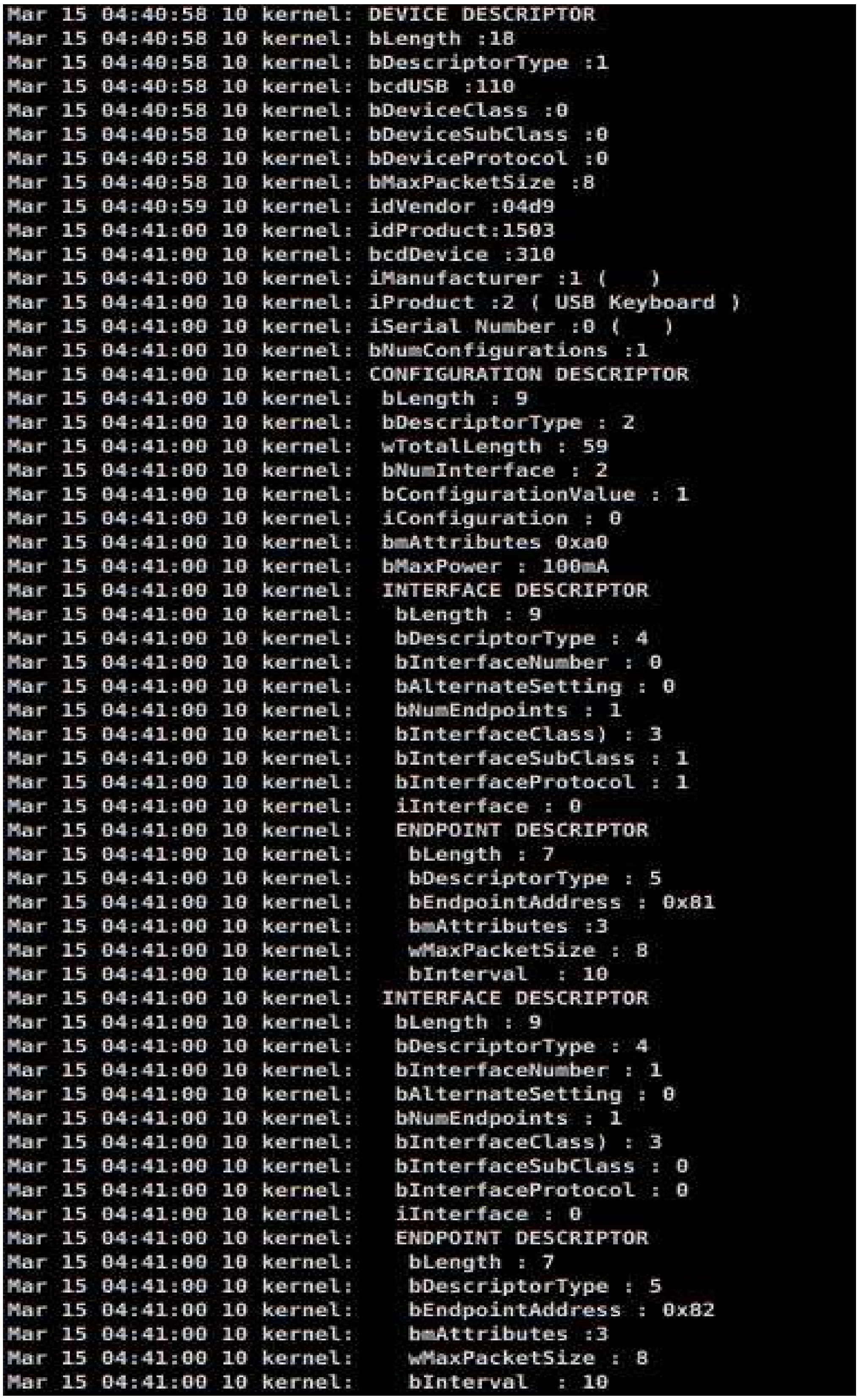


```
Mar 15 04:41:00 10 kernel: usbkbd: USB HID Keyboard found
Mar 15 04:41:00 10 kernel: usbd: device driver 73147 claimed device 0x04d9:0x1503
```

```
usbkbd: usbkbd: irq running
 usb scan code[0] = 0x00 scan code set 1 0x00
 usb scan code[1] = 0x00 scan code set 1 0x00
 usb scan code[2] = 0x06 scan code set 1 0x2e c
 usb scan code[3] = 0x18 scan code set 1 0x16 u
 usb scan code[4] = 0x16 scan code set 1 0x1f s
 usb scan code[5] = 0x04 scan code set 1 0x1e a
 usb scan code[6] = 0x17 scan code set 1 0x14 t
 usb scan code[7] = 0x00 scan code set 1 0x00
usbkbd: usbkbd: irq done_
```

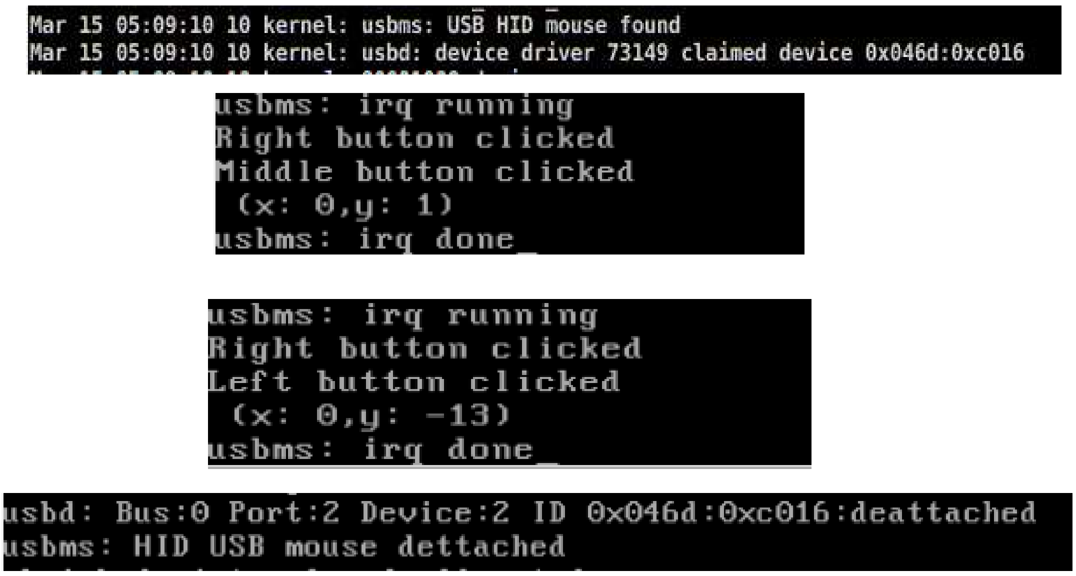
```
Mar 15 04:46:07 10 kernel: usbd: Bus:0 Port:2 Device:1 ID 0x04d9:0x1503:deattached
Mar 15 04:46:07 10 kernel: usbkbd: default 3603
Mar 15 04:46:07 10 kernel: usbkbd: HID USB Keyboard dettached
```

Figure 7.12 Showing attach / detach of HID keyboard device along scans codes



```
Mar 15 04:40:58 10 kernel: DEVICE DESCRIPTOR
Mar 15 04:40:58 10 kernel: bLength :18
Mar 15 04:40:58 10 kernel: bDescriptorType :1
Mar 15 04:40:58 10 kernel: bcdUSB :110
Mar 15 04:40:58 10 kernel: bDeviceClass :0
Mar 15 04:40:58 10 kernel: bDeviceSubClass :0
Mar 15 04:40:58 10 kernel: bDeviceProtocol :0
Mar 15 04:40:58 10 kernel: bMaxPacketSize :8
Mar 15 04:40:59 10 kernel: idVendor :04d9
Mar 15 04:41:00 10 kernel: idProduct:1503
Mar 15 04:41:00 10 kernel: bcdDevice :310
Mar 15 04:41:00 10 kernel: iManufacturer :1 (    )
Mar 15 04:41:00 10 kernel: iProduct :2 ( USB Keyboard )
Mar 15 04:41:00 10 kernel: iSerial Number :0 (    )
Mar 15 04:41:00 10 kernel: bNumConfigurations :1
Mar 15 04:41:00 10 kernel: CONFIGURATION DESCRIPTOR
Mar 15 04:41:00 10 kernel:  bLength : 9
Mar 15 04:41:00 10 kernel:  bDescriptorType : 2
Mar 15 04:41:00 10 kernel:  wTotalLength : 59
Mar 15 04:41:00 10 kernel:  bNumInterface : 2
Mar 15 04:41:00 10 kernel:  bConfigurationValue : 1
Mar 15 04:41:00 10 kernel:  iConfiguration : 0
Mar 15 04:41:00 10 kernel:  bmAttributes 0xa0
Mar 15 04:41:00 10 kernel:  bMaxPower : 100mA
Mar 15 04:41:00 10 kernel:  INTERFACE DESCRIPTOR
Mar 15 04:41:00 10 kernel:   bLength : 9
Mar 15 04:41:00 10 kernel:   bDescriptorType : 4
Mar 15 04:41:00 10 kernel:   bInterfaceNumber : 0
Mar 15 04:41:00 10 kernel:   bAlternateSetting : 0
Mar 15 04:41:00 10 kernel:   bNumEndpoints : 1
Mar 15 04:41:00 10 kernel:   bInterfaceClass) : 3
Mar 15 04:41:00 10 kernel:   bInterfaceSubClass : 1
Mar 15 04:41:00 10 kernel:   bInterfaceProtocol : 1
Mar 15 04:41:00 10 kernel:   iInterface : 0
Mar 15 04:41:00 10 kernel:   ENDPOINT DESCRIPTOR
Mar 15 04:41:00 10 kernel:    bLength : 7
Mar 15 04:41:00 10 kernel:    bDescriptorType : 5
Mar 15 04:41:00 10 kernel:    bEndpointAddress : 0x81
Mar 15 04:41:00 10 kernel:    bmAttributes :3
Mar 15 04:41:00 10 kernel:    wMaxPacketSize : 8
Mar 15 04:41:00 10 kernel:    bInterval  : 10
Mar 15 04:41:00 10 kernel:  INTERFACE DESCRIPTOR
Mar 15 04:41:00 10 kernel:   bLength : 9
Mar 15 04:41:00 10 kernel:   bDescriptorType : 4
Mar 15 04:41:00 10 kernel:   bInterfaceNumber : 1
Mar 15 04:41:00 10 kernel:   bAlternateSetting : 0
Mar 15 04:41:00 10 kernel:   bNumEndpoints : 1
Mar 15 04:41:00 10 kernel:   bInterfaceClass) : 3
Mar 15 04:41:00 10 kernel:   bInterfaceSubClass : 0
Mar 15 04:41:00 10 kernel:   bInterfaceProtocol : 0
Mar 15 04:41:00 10 kernel:   iInterface : 0
Mar 15 04:41:00 10 kernel:   ENDPOINT DESCRIPTOR
Mar 15 04:41:00 10 kernel:    bLength : 7
Mar 15 04:41:00 10 kernel:    bDescriptorType : 5
Mar 15 04:41:00 10 kernel:    bEndpointAddress : 0x82
Mar 15 04:41:00 10 kernel:    bmAttributes :3
Mar 15 04:41:00 10 kernel:    wMaxPacketSize : 8
Mar 15 04:41:00 10 kernel:    bInterval  : 10
```

Figure 7.13 Descriptors of HID keyboard that is read by USBD



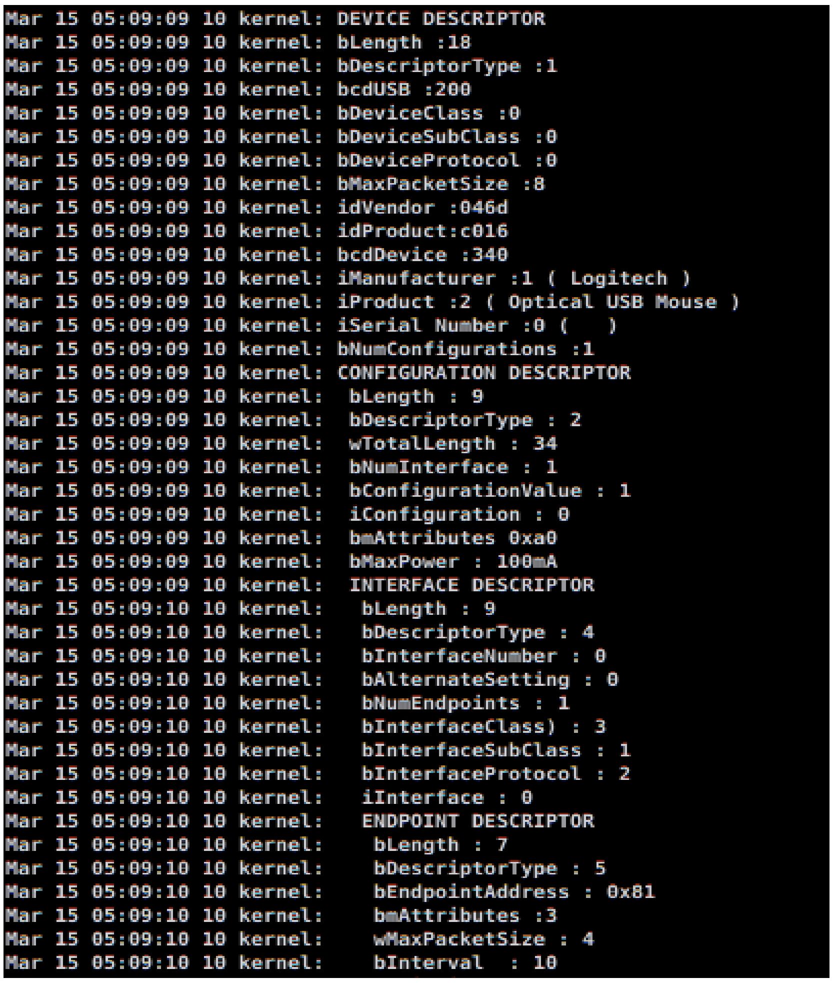
```
Mar 15 05:09:10 10 kernel: usbms: USB HID mouse found
Mar 15 05:09:10 10 kernel: usbd: device driver 73149 claimed device 0x046d:0xc016
```

```
usbms: irq running
Right button clicked
Middle button clicked
 (x: 0,y: 1)
usbms: irq done_
```

```
usbms: irq running
Right button clicked
Left button clicked
 (x: 0,y: -13)
usbms: irq done_
```

```
usbd: Bus:0 Port:2 Device:2 ID 0x046d:0xc016:deattached
usbms: HID USB mouse dettached
```

Figure 7.14 Showing attach / detach of mouse device along with events



```
Mar 15 05:09:09 10 kernel: DEVICE DESCRIPTOR
Mar 15 05:09:09 10 kernel: bLength :18
Mar 15 05:09:09 10 kernel: bDescriptorType :1
Mar 15 05:09:09 10 kernel: bcdUSB :200
Mar 15 05:09:09 10 kernel: bDeviceClass :0
Mar 15 05:09:09 10 kernel: bDeviceSubClass :0
Mar 15 05:09:09 10 kernel: bDeviceProtocol :0
Mar 15 05:09:09 10 kernel: bMaxPacketSize :8
Mar 15 05:09:09 10 kernel: idVendor :046d
Mar 15 05:09:09 10 kernel: idProduct:c016
Mar 15 05:09:09 10 kernel: bcdDevice :340
Mar 15 05:09:09 10 kernel: iManufacturer :1 ( Logitech )
Mar 15 05:09:09 10 kernel: iProduct :2 ( Optical USB Mouse )
Mar 15 05:09:09 10 kernel: iSerial Number :0 (   )
Mar 15 05:09:09 10 kernel: bNumConfigurations :1
Mar 15 05:09:09 10 kernel: CONFIGURATION DESCRIPTOR
Mar 15 05:09:09 10 kernel:  bLength : 9
Mar 15 05:09:09 10 kernel:  bDescriptorType : 2
Mar 15 05:09:09 10 kernel:  wTotalLength : 34
Mar 15 05:09:09 10 kernel:  bNumInterface : 1
Mar 15 05:09:09 10 kernel:  bConfigurationValue : 1
Mar 15 05:09:09 10 kernel:  iConfiguration : 0
Mar 15 05:09:09 10 kernel:  bmAttributes 0xa0
Mar 15 05:09:09 10 kernel:  bMaxPower : 100mA
Mar 15 05:09:09 10 kernel:  INTERFACE DESCRIPTOR
Mar 15 05:09:10 10 kernel:   bLength : 9
Mar 15 05:09:10 10 kernel:   bDescriptorType : 4
Mar 15 05:09:10 10 kernel:   bInterfaceNumber : 0
Mar 15 05:09:10 10 kernel:   bAlternateSetting : 0
Mar 15 05:09:10 10 kernel:   bNumEndpoints : 1
Mar 15 05:09:10 10 kernel:   bInterfaceClass) : 3
Mar 15 05:09:10 10 kernel:   bInterfaceSubClass : 1
Mar 15 05:09:10 10 kernel:   bInterfaceProtocol : 2
Mar 15 05:09:10 10 kernel:   iInterface : 0
Mar 15 05:09:10 10 kernel:   ENDPOINT DESCRIPTOR
Mar 15 05:09:10 10 kernel:    bLength : 7
Mar 15 05:09:10 10 kernel:    bDescriptorType : 5
Mar 15 05:09:10 10 kernel:    bEndpointAddress : 0x81
Mar 15 05:09:10 10 kernel:    bmAttributes :3
Mar 15 05:09:10 10 kernel:    wMaxPacketSize : 4
Mar 15 05:09:10 10 kernel:    bInterval  : 10
```

Figure 7.15 Descriptors of HID keyboard that is read by USBD

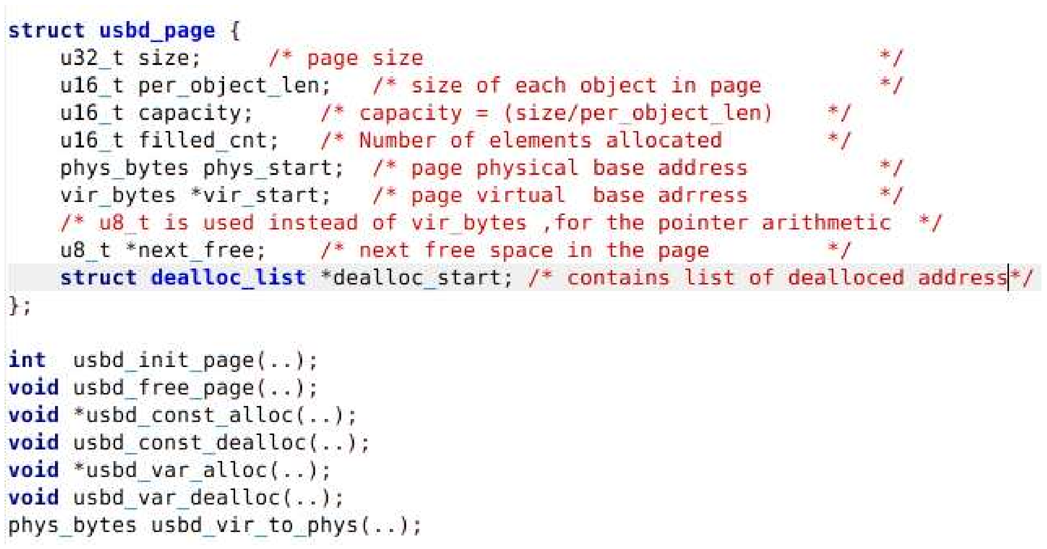
7.3.3 HCDI and UHCI HCD

UHCI is well introduced in chapter 3, one of the main hurdles during the UHCI driver development was the requirement that Tds and Qhs should be aligned in 16byte boundary,as I couldn't verify that minix implementation of *malloc()* will always return address of such nature ,i had to design a memory manager for this purpose without which transfer queuing would be impossible. According to design in chapter 5 HCDI is a static library linked to HCDs , since we don't have multiple HCDs, implementation of the library was a low priority task right now HCDI is embedded in the UHCI HCD , HCDI basically consist of the protocols specified in Figure 7.5 and Figure 7.6, UHCI-HCD implementation is shown in Appendix 2.

7.3.3.1 Implementation of USBD memory Manager

We have to get a 16byte aligned address for each instances of td and qh , for that purpose I implemented a tiny slab allocator , which divides a 4K or 64K page by fixed memory size during initialization of the slab , if the given memory size is not divisible by 16 its is rounded off to a size divisible by 16.

These functionality along with other memory custom memory management routine is combined into USBMEM , see the Figure 7.17 below for the core data structures and functions in usbdmem.



```
struct usbd_page {
    u32_t size;      /* page size                              */
    u16_t per_object_len;   /* size of each object in page         */
    u16_t capacity;     /* capacity = (size/per_object_len)    */
    u16_t filled_cnt;   /* Number of elements allocated        */
    phys_bytes phys_start;  /* page physical base address          */
    vir_bytes *vir_start;   /* page virtual  base adrress          */
    /* u8_t is used instead of vir_bytes ,for the pointer arithmetic  */
    u8_t *next_free;    /* next free space in the page         */
    struct dealloc_list *dealloc_start; /* contains list of dealloced address*/
};

int  usbd_init_page(..);
void usbd_free_page(..);
void *usbd_const_alloc(..);
void usbd_const_dealloc(..);
void *usbd_var_alloc(..);
void usbd_var_dealloc(..);
phys_bytes usbd_vir_to_phys(..);
```

Figure 7.17 Methods and data structures of USBD

See appendix 3 for actual code of usbd memory management , though it was aimed for UHCI HCD, it was made generic enough to be used with USBD. Appendix 4 shows a simplified event trace diagram of USBD and UHCD.

CHAPTER 8
Conclusion

Universal Serial Bus is a vast topic and covers many domains in engineering such as electronics, electrical and computer science, thus proper understanding of the USB requires narrowing the domain. This project focus was on the software approach to the USB which requires familiarizing the chapters 5, 8, 9 of USB specification 2.0.

This project implemented usb subsystem for Minix 3 as per the specified scope. Further, evaluated the usb subsystem using HID requests and implemented a HID keyboard and mouse driver that makes use of the driver API to communicate with the device , result of the evaluation and testing were really encouraging.

The stack is limited in functionality with major features lagging as seen in the limitations, However within the given time frame a working prototype of proposed design were implemented. This implementation need further improvement to be made useful enough for daily use, it requires rigorous testing to make sure it is functioning as per the specification.

Future scope of this stack is plenty. As it was designed in a modular independent way new modules that give functionality such as USB mass storage, USB Audio etc could be implemented. This project has good scope if open sourced, so it has been releasing under GNU General Public License.

References

[1] Universal Serial Bus Revision 2.0 Compaq Computer Corporation,Hewlet Packard Company,Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V, April 27,2000

[2] USB Complete: Everything You Need to Develop Custom USB Peripherals , Lakeview Research August 31, 2005

[3] Universal Serial Bus Architecture by Anderson and Dzatko, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 2001 ISBN:0201309750

[4] USB for the L4 Environment ,Dirk Vogt September 2008 ,Technische Universität Dresden ,Fakultät Informatik Institut für Systemarchitektur

[5] UHCI Design Guide ,Intel Corporation ,Revision 1.1, March 1999

[6] Design and Implementation of a USB Stack for the Java-based JX Operating System,vorgelegt von Dreweke Alexander Bachelor Thesis in Computational Engineering,03 November 2003

[7] Device Class Definition for Human Interface Devices , USB implementers Forum ,Firmware Specification,Version 1.11 ,06/27/2001

Appendix 1

Source code Organization

```
|-- Makefile        :gnu makefile
|-- drivers.conf    :usb specific driver configuration
|-- uhci-hcd.c      :uhci host controller driver
|-- uhci.h          :uhci driver meta data
|-- usb.h           :usb specification chapter 9
|-- usbd.c          :usb driver
|-- usbd.h          :usbd meta data
|-- usbdi.c         :usbdi implementation
|-- usbdi.h         :usbdi meta data
|-- usbdmem.c       :usbdmen implementation
|-- usbdmem.h       :usbdmem meta data
|-- usbkbd.c        :dummy keyboard driver
`-- usbms.c         :dummy mouse driver
```

Appendix 2

Dummy Keyboard and mouse driver

HID keyboard driver using boot protocol

```
/*
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation; either version 2 of the License, or (at your
 * option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * February 2010
 *
 * (C) Copyright 2009,2010 Althaf K Backer <althafkbacker@gmail.com>
 * (C) Copyright  1998 The NetBSD Foundation, Inc.
 *          for the usb_2_set1[] mapping
 *
 * Simple USB HID keyboard driver that works on Boot protocol as per HID
 * specification version 1.11 (27/6/2001)
 */

/*NOTE: This is not exactly a driver rather just an implementation to show
 *      that the usb stack work from top to bottom , what you see here is
 *      just a dummy driver that reads what ever key is being pressed
 *      and display them along with scan code
 *
 *      It would be nice if some one try to integrate this with tty
 */

#include "../drivers.h"
#include "../libdriver/driver.h"

#include <minix/ds.h>
#include <minix/vm.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <ibm/pci.h>

#include <sys/mman.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "usbd.h"
#include "usbdi.h"

#undef  DPRINT_PREFIX
#define DPRINT_PREFIX "\nusbkbd: "

/* Testing */
#define SET_BOOT_P 1
#define SET_REPORT_P 0

usbd_dev_id_t kbd_device;

char numon  = 1 << 0;
char capson = 1 << 1;
char scrlon = 1 << 2;
char allon  = 1 << 0 | 1 << 1 | 1 << 2;
char alloff = 0;

char dat[10] =
{          0, 0,
           0, 0,
     0, 0,
     0, 0,
     0, 0
};

char buf[2] = {0, 0};
#define NN 0                            /* no translation */
short keymap[256] = {
          NN, NN, NN, NN, 'a', 'b', 'c', 'd',
          'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',

          'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
          'u', 'v', 'w', 'x', 'y', 'z', '1', '2',

          '3', '4', '5', '6', '7', '8', '9', '0',
          '\n', '\e', '\b', '\t', ' ', '-', '=', '[',

          ']', '\\', NN, ';', '\'', '`', ',', '.',
          '/', NN, F1, F2, F3, F4, F5, F6,
          F7, F8, F9, F10, F11, F12, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, '*', '-', '+',
          NN, END, DOWN, PGUP, LEFT, NN, RIGHT, HOME,
          UP, PGDN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN,

          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
          NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN, NN,
};

/*
 * Translate USB keycodes to US keyboard XT scancodes.
 * Scancodes >= 0x80 represent EXTENDED keycodes.
 *
 * See http://www.microsoft.com/whdc/device/input/Scancode.mspx
 */
const u8_t usb_2_set1[256] = {
      NN,   NN,   NN,   NN, 0x1e, 0x30, 0x2e, 0x20, /* 00 - 07 */
    0x12, 0x21, 0x22, 0x23, 0x17, 0x24, 0x25, 0x26, /* 08 - 0f */
    0x32, 0x31, 0x18, 0x19, 0x10, 0x13, 0x1f, 0x14, /* 10 - 17 */
    0x16, 0x2f, 0x11, 0x2d, 0x15, 0x2c, 0x02, 0x03, /* 18 - 1f */
    0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, /* 20 - 27 */
    0x1c, 0x01, 0x0e, 0x0f, 0x39, 0x0c, 0x0d, 0x1a, /* 28 - 2f */
    0x1b, 0x2b, 0x2b, 0x27, 0x28, 0x29, 0x33, 0x34, /* 30 - 37 */
    0x35, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40, /* 38 - 3f */
    0x41, 0x42, 0x43, 0x44, 0x57, 0x58, 0xaa, 0x46, /* 40 - 47 */
    0x7f, 0xd2, 0xc7, 0xc9, 0xd3, 0xcf, 0xd1, 0xcd, /* 48 - 4f */
    0xcb, 0xd0, 0xc8, 0x45, 0xb5, 0x37, 0x4a, 0x4e, /* 50 - 57 */
    0x9c, 0x4f, 0x50, 0x51, 0x4b, 0x4c, 0x4d, 0x47, /* 58 - 5f */
    0x48, 0x49, 0x52, 0x53, 0x56, 0xdd, 0x84, 0x59, /* 60 - 67 */
    0x5d, 0x5e, 0x5f,   NN,   NN,   NN,   NN,   NN, /* 68 - 6f */
      NN,   NN,   NN,   NN, 0x97,   NN, 0x93, 0x95, /* 70 - 77 */
    0x91, 0x92, 0x94, 0x9a, 0x96, 0x98, 0x99, 0xa0, /* 78 - 7f */
    0xb0, 0xae,   NN,   NN,   NN, 0x7e,   NN, 0x73, /* 80 - 87 */
    0x70, 0x7d, 0x79, 0x7b, 0x5c,   NN,   NN,   NN, /* 88 - 8f */
      NN,   NN, 0x78, 0x77, 0x76,   NN,   NN,   NN, /* 90 - 97 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* 98 - 9f */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* a0 - a7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* a8 - af */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* b0 - b7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* b8 - bf */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* c0 - c7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* c8 - cf */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* d0 - d7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* d8 - df */
    0x1d, 0x2a, 0x38, 0xdb, 0x9d, 0x36, 0xb8, 0xdc, /* e0 - e7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* e8 - ef */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* f0 - f7 */
      NN,   NN,   NN,   NN,   NN,   NN,   NN,   NN, /* f8 - ff */
};

_PROTOTYPE(void kbd_probe, (message m2kbd));
_PROTOTYPE(void kbd_device_dettached, (void));
_PROTOTYPE(void kbd_irq, (void));

int main(void)
{
          U32_t self_proc;
          message m2kbd;
          int r;

          system_hz = sys_hz();
          r = ds_retrieve_u32("usbkbd",&self_proc);
          if (OK != r) {
                  printf("ds_retrive_32: failed");
                  return EXIT_FAILURE;
          }

          if(OK != usbdi_init())
             usbdi_fatal_abort("usbkbd","failed to register");
          if(OK != usbdi_register_driver())
             usbdi_fatal_abort("usbkbd","failed to register");

          while (TRUE) {
                    if ((r = receive(ANY, &m2kbd)) != OK)
                            panic("uhci-hcd:", "receive failed", r);
```

```
                      switch (m2kbd.m_source) {
                      case RS_PROC_NR:
                               notify(m2kbd.m_source);
                      break;
                      case PM_PROC_NR: {
                                      sigset_t set;
                                      if (getsigset(&set) != 0)
                                          break;
                                      if (sigismember(&set, SIGTERM)) {
                                                usbdi_dereisgter_driver();
                                                goto aborted;
                                      }
                    }
                      break;
                      default:
                               DPRINTF(1, ("default %d ", m2kbd.m_type));
                               goto usbd2usbdi_msg;
                     }
                    continue;

 usbd2usbdi_msg:
                     switch (m2kbd.m_type) {
                     case USBD2USBDI_DD_PROBE:
                              kbd_probe(m2kbd);
                     break;
                     case USBD2USBDI_DEVICE_DISCONNECT:
                              /* Message info from usbd
                               *
                               * m2kbd.m2_l1 : device id
                               * this a valid case if this driver
                               * handle multiple devices
                               */
                              kbd_device_dettached();
                     break;
                     case USB_INTERRUPT_REQ_STS:
                              if (OK == m2kbd.m2_i1)
                                        kbd_irq();
                     break;
                     case USBD2ALL_SIGTERM:
                              DPRINTF(1,("SIGTERM received from usbd , driver unstable"));
                     break;
                     default:
                            DPRINTF(1, ("unknown type %d from source %d", m2kbd.m_type,

                                      m2kbd.m_source));
                    }
          }
 aborted:
          return (OK);
}


void kbd_probe(message m2kbd)
{
          usb_interface_descriptor_t idesc;
          usb_endpoint_descriptor_t edesc;
          usb_config_descriptor_t cdesc;
          message reply;
          int r;

          DPRINTF(1,("inside kbd_probe"));

          r = usbdi_get_device_cdesc(m2kbd.m2_l1,&cdesc,1);
          if (OK != r) {
                    printf("\nusbkbd: failed to GET_UDESC_CONFIG: %d",r);
                    return;
          }

    r = usbdi_get_device_idesc(m2kbd.m2_l1,&idesc,1,0);
    if (OK != r) {
                    printf("\nusbkbd: failed to GET_UDESC_INTERFACE: %d",r);
                    return;
          }

          if(3 != idesc.bInterfaceClass || 1 != idesc.bInterfaceSubClass ||
             1 != idesc.bInterfaceProtocol)  {
       usbdi_probe_status(m2kbd.m2_l1,DD_DEV_REJECT);
       return;
    }

    kbd_device = m2kbd.m2_l1;
    /* Inform usbd driver can claim the device */
    usbdi_probe_status(kbd_device,DD_DEV_ACCEPT);
    printf("\nusbkbd: USB HID Keyboard found");
    r = usbdi_get_device_edesc(kbd_device,&edesc,1,0,0);
          if (OK != r) {
                    printf("\nusbkbd: failed to GET_UDESC_ENDPOINT: %d",r);
                    return;
          }

    r = usbdi_set_config(kbd_device,1);
    if (OK != r) {
```

```
                    printf("\nusbkbd: failed to SET_CONFIG: %d",r);
                    return;
    }

#if SET_BOOT_P
          printf("\nusbdkbd: setting boot protocol");
          /* Set boot protocol */
          r = usbdi_get_protocol(kbd_device, idesc.bInterfaceNumber, buf);
          if (OK != r) {
                    printf("\n get protocolas");
          }
          printf("\n get protocol 0x%x 0x%x",buf[0],buf[1]);
    r = usbdi_set_protocol(kbd_device, 0, 0);
    if (OK != r)
                    return;
#endif

#if SET_REPORT_P
          printf("\nusbdkbd: setting report protocol");
          /* Set boot protocol */
    r = usbdi_set_protocol(kbd_device, 1, idesc.bInterfaceNumber);
    if (OK != r)
                    return;
          r = usbdi_get_report(kbd_device, RT_IN, 0, 8, idesc.bInterfaceNumber, dat);
          if (OK != r) {
                    printf("\n failed");
                    return;
          }
           kbd_irq();
#endif

#if SET_BOOT_P
    r = usbdi_set_idle(kbd_device, 0, 0, idesc.bInterfaceNumber);
          if (OK != r)
                    return;
          r = usbdi_get_idle(kbd_device, 0, idesc.bInterfaceNumber, buf);
          if (OK != r)
                    return;
          printf("\n get idle 0x%x 0x%x",buf[0],buf[1]);
    /* Just for fun show up some light show :D */
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&numon);   USBD_MSLEEP(2);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&capson);  USBD_MSLEEP(50);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&scrlon);  USBD_MSLEEP(2);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&alloff);  USBD_MSLEEP(100);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&numon);   USBD_MSLEEP(2);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&alloff);  USBD_MSLEEP(10);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&scrlon);  USBD_MSLEEP(60);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&capson);  USBD_MSLEEP(2);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&numon);   USBD_MSLEEP(10);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&alloff);  USBD_MSLEEP(10);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&allon);       USBD_MSLEEP(2);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&alloff);
    usbdi_set_report(kbd_device,RT_OUT,0,1,idesc.bInterfaceNumber,&numon);
    /* Start the interrupt polling for keyboard events */
    usbdi_interrupt_req(kbd_device,edesc.bEndpointAddress,dat);
#endif

}

void kbd_irq( void )
{
   int i = 0;
   char buf[10];
   DPRINTF(1, ("usbkbd: irq running"));
   memcpy(buf,dat,8);
   for (i = 0;i < 8;i++)
            printf("\n usb scan code[%d] = 0x%02x scan code set 1 0x%02x %c",i,buf[i],

usb_2_set1[buf[i]],keymap[buf[i]]);
   DPRINTF(1, ("usbkbd: irq done"));
}

void kbd_device_dettached( void )
{
          /* Handle dettached case */
          DPRINTF(1,("HID USB Keyboard dettached"));
}
```

HID Mouse driver using boot protocol

```
* February 2010
 *
 * (C) Copyright 2009,2010 Althaf K Backer <althafkbacker@gmail.com>
 *
 * Simple USB HID mouse driver that works on Boot protocol as per HID
 * specification version 1.11 (27/6/2001)
 */

/* NOTE: You are better off looking into usbkbd.c ,both are of similar
 * nature.
 */
#include "../drivers.h"
```

```
#include "../libdriver/driver.h"

#include <minix/ds.h>
#include <minix/vm.h>
#include <minix/sysutil.h>
#include <minix/keymap.h>
#include <ibm/pci.h>

#include <sys/mman.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "usbd.h"
#include "usbdi.h"

#undef  DPRINT_PREFIX
#define DPRINT_PREFIX "\nusbms: "

_PROTOTYPE(void ms_probe, (message m2ms));
_PROTOTYPE(void ms_irq, (void));
_PROTOTYPE(void ms_device_dettached, ( void ));

usbd_dev_id_t ms_device;

char dat[10] =
{          0, 0,
           0, 0,
     0, 0,
     0, 0,
     0, 0
};

int main(void)
{
          U32_t self_proc;
          message m2ms;
          int r;

          system_hz = sys_hz();
          r = ds_retrieve_u32("usbms",&self_proc);
          if (OK != r) {
                    printf("ds_retrive_32: failed");
                    return EXIT_FAILURE;
          }

          if(OK != usbdi_init())
             usbdi_fatal_abort("usbms","failed to register");
          if(OK != usbdi_register_driver())
             usbdi_fatal_abort("usbms","failed to register");

          while (TRUE) {
                      if ((r = receive(ANY, &m2ms)) != OK)
                                panic("uhci-hcd:", "receive failed", r);

                      switch (m2ms.m_source) {
                      case RS_PROC_NR:
                                 notify(m2ms.m_source);
                      break;
                      case PM_PROC_NR: {
                                        sigset_t set;
                                        if (getsigset(&set) != 0)
                                            break;
                                        if (sigismember(&set, SIGTERM)) {
                                                  usbdi_dereisgter_driver();
                                                  goto aborted;
                                        }
                    }
                      break;
                      default:
                                 DPRINTF(0, ("default %d ", m2ms.m_type));
                                 goto usbd2usbdi_msg;
                     }
                    continue;

 usbd2usbdi_msg:
                     switch (m2ms.m_type) {
               case USBD2USBDI_DEVICE_DISCONNECT:
                                /* Message info from usbd
                                 *
                                 * m2kbd.m2_l1 : device id
                                 * this a valid case if this driver
                                 * handle multiple devices
                                 */
                                ms_device_dettached();
                     break;
                     case USB_INTERRUPT_REQ_STS:
                                if (OK == m2ms.m2_i1)
                                          ms_irq();
                     break;
                     case USBD2USBDI_DD_PROBE:
```

```
                                ms_probe(m2ms);
                     break;
                     case USBD2ALL_SIGTERM:
                                DPRINTF(1,("SIGTERM received from usbd , driver unstable"));
                     break;
                     default:
                              DPRINTF(0, ("unknown type %d from source %d", m2ms.m_type,
                                     m2ms.m_source));
                    }
          }
 aborted:
          return (OK);
}
void ms_probe(message m2ms)
{
          usb_interface_descriptor_t idesc;
          usb_endpoint_descriptor_t edesc;
          usb_config_descriptor_t cdesc;
          message reply;
          int r;

          DPRINTF(0,("inside ms_probe"));

          r = usbdi_get_device_cdesc(m2ms.m2_l1,&cdesc,1);
          if (OK != r) {
                    DPRINTF(1, ("failed to GET_UDESC_CONFIG: %d",r));
                    return;
          }

    r = usbdi_get_device_idesc(m2ms.m2_l1,&idesc,1,0);
    if (OK != r) {
                    DPRINTF(1, ("failed to GET_UDESC_INTERFACE: %d",r));
                    return;
          }

          if(3 != idesc.bInterfaceClass || 1 != idesc.bInterfaceSubClass ||
             2 != idesc.bInterfaceProtocol)  {
       usbdi_probe_status(m2ms.m2_l1,DD_DEV_REJECT);
       return;
    }

    ms_device = m2ms.m2_l1;
    /* Inform usbd driver can claim the device */
    usbdi_probe_status(ms_device,DD_DEV_ACCEPT);
    DPRINTF(1, ("USB HID mouse found"));

    r = usbdi_get_device_edesc(ms_device,&edesc,1,0,0);
          if (OK != r) {
                    DPRINTF(1, ("failed to GET_UDESC_ENDPOINT: %d",r));
                    return;
          }

    r = usbdi_set_config(ms_device,1);
    if (OK != r) {
                    DPRINTF(1, ("failed to SET_CONFIG: %d",r));
                    return;
    }
    DPRINTF(1,("setting boot protocol"));
          /* Set boot protocol */
    usbdi_set_protocol(ms_device, 0,0);
    usbdi_set_idle(ms_device,0,0,0);
    usbdi_interrupt_req(ms_device,edesc.bEndpointAddress,dat);
}
void ms_irq( void )
{
   int i = 0;
   char buf[10];
   DPRINTF(1, ("irq running"));
   memcpy(buf,dat,8);

              if (buf[0] & 1 << 0)
                              printf("\nRight button clicked");
                    if (buf[0] & 1 << 1)
                              printf("\nLeft button clicked");
                    if (buf[0] & 1 << 2)
                              printf("\nMiddle button clicked");

                    printf("\n (x: %d,y: %d)",buf[1],buf[2]);


   DPRINTF(1, ("irq done"));
}

void ms_device_dettached( void )
{
          /* Handle dettached case */
          DPRINTF(1,("HID USB mouse dettached"));
}
```

Appendix 3

USBD memory management

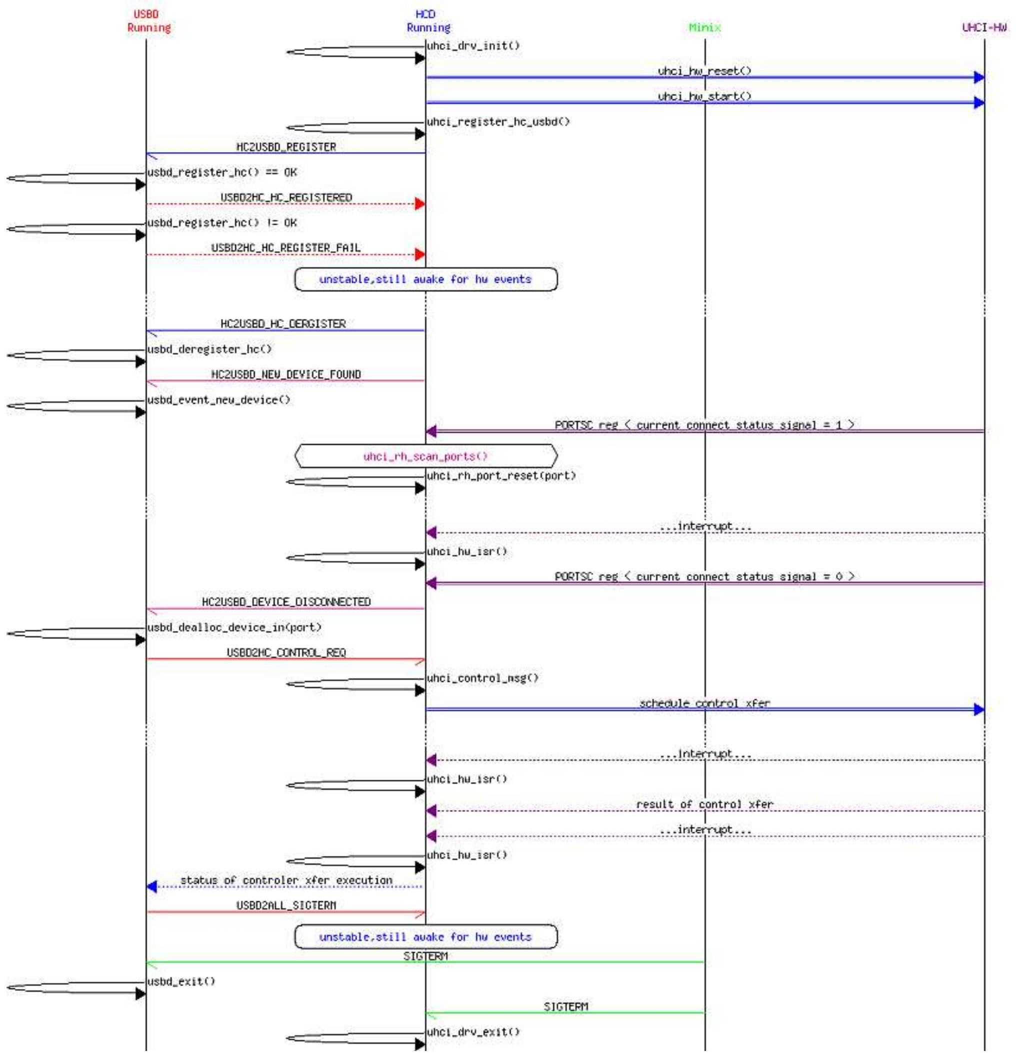
```
 *
 * USB driver memory management routines.
 *
 * These should be used instead of the standard memory management routines,
 * by the drivers that is suppose to communicate with USBD and layer below,
 * these guarantee that we get a contigious memory chunks.
 *
 * usbdmem.o could be linked to other drivers for the purpose.
 *
 */
#include "usbdmem.h"

/* Allocates 4K / 64K page and fille up usbd_page information  */
int usbd_init_page(size_t struct_size, int pageflag, struct usbd_page *page)
{
          int cnt;

          if (page == NULL) {
                    DPRINTF(1, ("NULL reference caught"));
                    return EINVAL;
          }

          switch (pageflag) {
          case A4K:
                    page->size = I386_PAGE_SIZE;
                    break;
          case A64K:
                    page->size = 64 * 1024;
                    break;
          case A4K_ALIGN16:
                    page->per_object_len = ALIGN16;
                    page->size = I386_PAGE_SIZE;
                    pageflag = A4K;
                    break;
          case A64K_ALIGN16:
                    page->per_object_len = ALIGN16;
                    page->size = 64 * 1024;
                    pageflag = A64K;
                    break;
          default:
                    DPRINTF(1, ("unknown page flag"));
                    return ENOMEM;
          }

          if (page->per_object_len != HC_DEV_FLAG)
                    if (!
                        (page->vir_start =
                         (vir_bytes *) alloc_contig(page->size, pageflag,
                                                    &page->phys_start)))
                              return ENOMEM;

          if (page->per_object_len == ALIGN16) {
                    cnt = (struct_size / 16);
                    struct_size = (cnt * 16) >= struct_size ? (cnt * 16) : ((cnt + 1) * 16);
          }

          DPRINTF(0, ("page->per_object_len %d", struct_size));
          page->per_object_len = struct_size;
          page->capacity = (page->size / page->per_object_len);
          page->filled_cnt = 0;
          page->next_free = (u8_t *) page->vir_start;
          page->dealloc_start = NULL;

          DPRINTF(0, ("\n Page info\n Page size:%d\n Object len: %d\n"
                      "\n Capacity:%d\n Phys: 0x%08x\n Vir: 0x%08x\n",
                      page->size, page->per_object_len, page->capacity,
                      page->phys_start, page->vir_start));

          return OK;
}

/* Free the page allocated by the init_this_page() */
void usbd_free_page(struct usbd_page *this_page)
{
          int r = 0;
          if (this_page == NULL) {
                    DPRINTF(1, ("NULL reference caught"));
                    return;
          }

          if ((r = munmap(this_page->vir_start, this_page->size)) != OK)
                    DPRINTF(1, ("usbdmem: munmap failed %d\n", r));

          DPRINTF(0,
                    ("freed page vir %08x phys %08x", this_page->vir_start,
                     this_page->phys_start));
```

```
}

/* Once we have the page allocated we virtually allocate
 * within the page with usbd_page->per_object_len chunks
 * as per request ,its called const beacause it allocate
 * predefined size.
 */
void *usbd_const_alloc(struct usbd_page *this_page)
{
         vir_bytes *allocated;
         u8_t *phys_addr;

         if (this_page == NULL) {
                 DPRINTF(1, ("NULL reference caught"));
                 return NULL;
         }

         /* we use u8_t instead of phys_bytes for the pointer arithmetic. */
         phys_addr = (u8_t *) this_page->phys_start;
         DPRINTF(0, ("usbd_const_alloc()"))
             DPRINTF(0, ("Page index %d", this_page->filled_cnt));

         if (this_page->capacity == this_page->filled_cnt) {
                 DPRINTF(1, ("usbd_const_alloc() NOMEM"));
                 return NULL;
         }

         /* if we have any deallocted addresses then use them */
         if (this_page->dealloc_start != NULL) {
                 allocated = (vir_bytes *) this_page->dealloc_start;
                 this_page->dealloc_start = this_page->dealloc_start->next;

         } else {
                 allocated = (vir_bytes *) this_page->next_free;
                 this_page->next_free += this_page->per_object_len;
         }

         this_page->filled_cnt++;

         DPRINTF(0, ("allocated  0x%08x", allocated));
         DPRINTF(0, ("next free 0x%08x", this_page->next_free));
         DPRINTF(0, ("Space left %d", this_page->capacity - this_page->filled_cnt));

         memset(allocated, 0, this_page->per_object_len);

         return allocated;
}

/* This procedure is tricky it virtually deallocates the instances allocated
 * by usbd_const_alloc() keep the list of the deallocated addresses from the
 * page,trick part is the each instance of dealloc_list use the currently
 * deallocated space for its own data structure.
 */
void usbd_const_dealloc(struct usbd_page *this_page, void *page_chunk)
{
         struct dealloc_list *start = this_page->dealloc_start;
         struct dealloc_list *dealloc_new = page_chunk;

         DPRINTF(0, ("usbd_const_dealloc"));

         if (this_page == NULL || page_chunk == NULL) {
                 DPRINTF(1, ("NULL reference caught"));
                 return;
         }
         this_page->filled_cnt--;
         if (this_page->dealloc_start == NULL) {
                 this_page->dealloc_start = dealloc_new;
                 this_page->dealloc_start->next = NULL;
         } else {
                 dealloc_new->next = this_page->dealloc_start;
                 this_page->dealloc_start = dealloc_new;
         }

         if (0)
                 while (start != NULL) {
                         DPRINTF(1, ("\n Dealoced %08x", start));
                         start = start->next;
                 }

         DPRINTF(0, ("deallocated 0x%08x", page_chunk));
}

/*
 * Allocate arbitary size chunks to a 4K page this is sort
 * of improper use of memory so this must not be used
 * extensively any address from drivers communicating with
 * USBD must use this as contigious memory chunk, this
 * is a prerequisite for certain controllers below.
 */
void *usbd_var_alloc(int size)
{
         vir_bytes *allocated;
```

```
        phys_bytes null;

        if (size > I386_PAGE_SIZE)
                return NULL;
        allocated = (vir_bytes *) alloc_contig(size, AC_ALIGN4K, &null);
        if (allocated == NULL)
                return NULL;
        DPRINTF(0, ("allocated vir %08x", allocated));
        return allocated;
}

void usbd_var_dealloc(void *addr)
{
        int r = 0;
        vir_bytes *vaddr;

        if (addr == NULL) {
                DPRINTF(1, ("NULL reference caught"));
                return;
        }

        vaddr = (vir_bytes *) addr;
        if ((r = munmap(vaddr, I386_PAGE_SIZE)) != OK) {
                DPRINTF(1, ("munmap failed %d\n", r));
                return;
        }
        DPRINTF(0, ("deallocated vir %08x", vaddr));
}

phys_bytes usbd_vir_to_phys(void *vir_addr)
{
        phys_bytes phys_addr;
        int r;
        if (vir_addr == NULL) {
                DPRINTF(1, ("NULL reference caught"));
                return EINVAL;
        }
        /* Find the physical address from the virtual one
         * NOTE: since SELF is given as endpoint number usbdmem.o
         * should be linked to any driver trying to use these. Any
         * sort of shared approach may not give required result.
         */
        r = sys_umap(SELF, VM_D, (vir_bytes) vir_addr, sizeof(phys_bytes),
                        &phys_addr);
        if (r != OK) {
                DPRINTF(1,("sys_umap failed for proc %d vir addr 0x%x", SELF,vir_addr));
                return EINVAL;
        }
        return phys_addr;
}

/* Return the free bit index/offset in the given map */
int next_free_bit(bitmap_t map)
{
        int offset = 0;
        while (offset < 128) {
                if (!(MAP_ISSET_BIT(map, offset)))
                        break;
                offset++;
        }

        return offset;
}
```

Appendix 4

A simplified event trace diagram of the USB stack


USBD Running
HCD Running
Minix
UHCI-HW
uhci_drv_init()
uhci_hw_reset()
uhci_hw_start()
uhci_register_hc_usbd()
HC2USBD_REGISTER
usbd_register_hc() == OK
USBD2HC_HC_REGISTERED
usbd_register_hc() != OK
USBD2HC_HC_REGISTER_FAIL
unstable,still awake for hw events
HC2USBD_HC_DERGISTER
usbd_deregister_hc()
HC2USBD_NEW_DEVICE_FOUND
usbd_event_new_device()
PORTSC reg < current connect status signal = 1 >
uhci_rh_scan_ports()
uhci_rh_port_reset(port)
...interrupt...
uhci_hw_isr()
PORTSC reg < current connect status signal = 0 >
HC2USBD_DEVICE_DISCONNECTED
usbd_dealloc_device_in(port)
USBD2HC_CONTROL_REQ
uhci_control_msg()
schedule control xfer
...interrupt...
uhci_hw_isr()
result of control xfer
...interrupt...
uhci_hw_isr()
status of controler xfer execution
USBD2ALL_SIGTERM
unstable,still awake for hw events
SIGTERM
usbd_exit()
SIGTERM
uhci_drv_exit()