

USB for Linux

A study by

Bhanu Nagendra Pisupati

under the guidance of

Prof. Steven D.Johnson

Contents

1	High Level Description	3
1.1	Acronym	3
1.2	Structure of Bus	3
1.3	Heirarchical Description	4
1.4	Transfer Types	5
1.5	Device Enumeration	5
1.6	Master-slave setup	5
2	Operating System Aspects	5
2.1	USB Stack	5
2.2	Responsibilities of the operating system	7
2.3	Filesystem for USB	8
2.3.1	Virtual Filesystems	8
2.3.2	USBFS	9
2.4	Example directory structure	9
2.5	Selection of driver	10
2.6	Programming for USB	10
3	Host Controller	11
3.1	Funcationality	12
3.2	Transfer Schedule	12
3.3	Format of a frame	13
3.4	Control Transfers	14
3.4.1	Device Discovery & Address Assignment	15
4	Description of Project	15
4.1	Objective	15
4.2	Algorithm	16
4.3	Aspects of the Module	16
4.3.1	Kernel Modules	16
4.3.2	Memory Allocation	17
4.3.3	PCI	17
4.3.4	Interrupts	18

1 High Level Description

Its quite likely that you have worked with a USB device before, for linking up your camera to download pictures, or maybe even hook up your mouse/keyboard. But USB for many of us is a fancy new technology, which seems to provide lot of features - it is fast (your pictures download much faster from your camera on the USB link compared to, say, a serial link), does not seem to have those annoying driver problems. All that there is, is a small connector which fits snugly into the slot at the back of your machine.

1.1 Acronym

To understand USB better it may be instructive to understand the acronym - which stands for *Universal Serial Bus*. The words *Serial* and *Bus* are crucial.

- * *Bus* reminds you that, though we seem to use USB for data transfer between a computer and an external device much like a serial cable, it is much more than that. To understand the difference, consider what is required to get a serial connection to work on a computer. All that is needed, is a driver for the serial port installed and that would make our serial port functional.

We are not concerned about the device that is hooked up at the other end of the serial cable. With the driver installed, the only other utility that maybe required is a serial client application like Hyperterm (if you are using Windows) or Minicom (if you running some flavor of Unix/Linux). The point is that you are only concerned about your end of things. But USB is a *bus*, like a PCI bus. Since the bus is part of the computer's datapath, it follows that all devices connected the USB cable, by virtue of being 'on the bus', are part of the computer. So, apart from being interested in just putting the bits on the wire - as was the case with the serial link - now we are also concerned about the devices on the other end.

- * *Serial* tells us that data transmission on the wire is serial (as opposed to parallel)- one bit at a time
- * *Universal* clarifies that the standard is portable across devices and platforms

1.2 Structure of Bus

In spite of the fact that most USB connections seem to run from computer to device (point to point) this hardly needs to be the case. The internal structure of the bus is really a tree

rather than a link. Every node in the tree represents a hub. The *root hub* is present inside the computer with the two USB sockets present on the computer being nodes connected to the root hub. Thus a compound USB device which consists of a keyboard and the mouse with one USB link going to the host may be represented as follows:

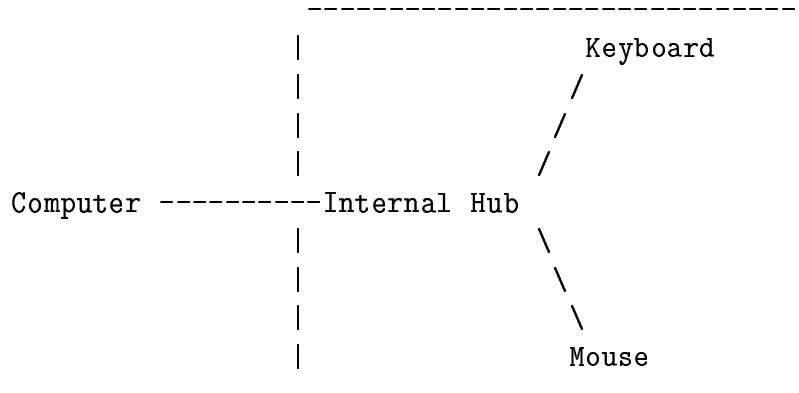


Figure 1: Layout of Bus

where the boxed region represents the device. The internal hub is invisible to the user. This raises a couple of points. First, obviously the computer needs to be aware and able to address either of the devices. This is done by way of assigning device addresses to each of the devices on the bus. Also, in spite of the fact that the function and the location of the internal hub is different from the devices, as far as the computer is concerned, it treats the hub in the very same way as it treats the other devices.

1.3 Heirarchical Description

USB was designed having in mind devices with multiple capabilities. To infor the host of its multiple capabilities, a device describes itself hierarchically . A device may have multiple configurations, each of which may in turn have multiple interfaces, which in turn may have multiple endpoints. The interfaces also have classes (and sub-classes) associated with them. Common classes include those for human interface devices such as keyboard, mice etc. and those for storage devices such as hard disks. This allows a device to present the relevant interfaces/configurations for different purposes.

1.4 Transfer Types

All transfers to USB devices fall into one of four types, namely: control, isochronous, bulk and interrupt. The names of each of the types are indicative of the purpose of each of the transfers. Control and interrupt transfers are used for sending small, fixed-format messages, for purposes such as setting a configuration of a device or sending small sets of data across. Bulk and isochronous transfers are used for transferring larger sets of data. While isochronous (as the name suggests) gives some time guarantees with regard to the transfers, bulk transfers are highly flexible in their transfer mechanisms. Every endpoint that a device may possess has a specific transfer type associated with itself. Hence a device which wants to make transfers of multiple types, must have one corresponding endpoint for each of the types.

1.5 Device Enumeration

The host learns of the various devices on the bus through a process known as enumeration. All communication between the host and a device takes place through pipes. The device is controlled using pipe 0, which is the standard control pipe for any device. The host can ask the device to enumerate itself using the control pipe, on which device sends back details regarding itself - its product and vendor id, information regarding configurations, interfaces and the different endpoints that it has.

1.6 Master-slave setup

USB link is a resource that is shared across all the devices. This means that contention can result in terms of more than one device wanting to send data at the same time (as is the case with ethernet). USB resolves this issue by following a master-slave protocol. The host is the master and controls the bus, and all the devices attached to the bus are slaves. Any device can transfer on the bus only at the request of the master.

2 Operating System Aspects

2.1 USB Stack

All information described thus far is operating system independent. Any operating system will have to provide means for the client applications to perform operations such as open pipes to particular endpoints of devices, make transfers of desired types etc. Typically on booting up, the operating system would enumerate all the devices on the bus to get an idea

of the layout of the entire bus. The actual layout of USB on the host is as shown in the figure:

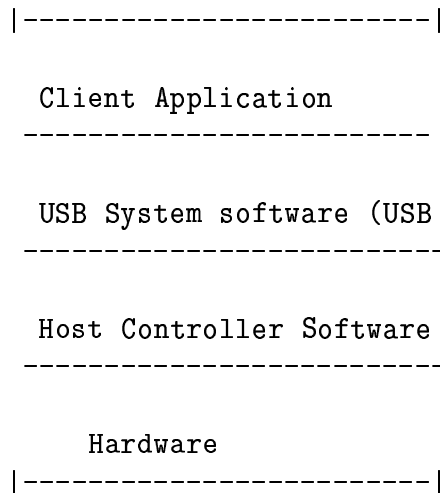


Figure 2: USB Stack

As the figure shows, USB has a layered structure on the host side. At the bottom of the stack is the actual hardware which throws signals on the wire and most importantly consists of the *host controller*. The host-controller is the hardware component inside the host which is responsible for implementing all host-aspects of USB. It is protocol-aware with a knowledge of the types of transfers (control/bulk etc.) for of its transactions.

2.2 Responsibilities of the operating system

First we will look at what it takes to get a USB device to work under Linux Consider a typical scenario of a USB device being used in Linux. Whenever a new device is plugged in, the host controller detects the device and conveys the information to the operating system. Once the operating system gets this information, the device is enumerated. Enumeration takes place by way of the operating system issuing commands to the host controller, more of which is discussed later. Enumeration of devices on the bus also occurs at boot time. Based on the information that the host learns at the time of enumeration, the operating system assigns a driver to the device. It is possible that none of the drivers known to the OS supports the device.

A device driver for a USB device implements all the functions implemented in a usual device driver, namely `open`, `read`, `write`, `ioctl` etc. Among these `ioctl()` performs a specially important role.

`ioctl()` is a system call in unix used for I/O functions related to devices. Since unix handles

all hardware devices as files, it is necessary to be able to device specific operations on file descriptors corresponding to these devices. `ioctl()` takes the file descriptor corresponding to the device as the first argument. The second argument is the request type, which is device specific and the third optional argument contains any arguments that need to be passed with the request. Operations concerning the USB device(such as writing) have several attributes associated with them, such as the transfer-type and specific attributes associated with the transfer type. It would be convenient to implement a write inside an `ioctl` so that the necessary attributes may be passed to the `ioctl` call as parameters. An example `ioctl` call may be as follows:

```
ret = ioctl(dev->fd, IOCTL_USB_BULK, &bulk);
```

This makes a bulk write by calling `ioctl` function associated with the file of USB device `dev` having a corresponding file-descriptor `fd`. The second parameter `IOCTL_USB_BULK` identifies the call as a bulk write, with the structure `bulk` containing the actual data to be written.

2.3 Filesystem for USB

In the previous example, it was mentioned that Linux compares the device description that it gets back at the time of enumeration, to check for any suitable drivers for the device. This means that the operating system needs to keep track of attached devices along with a list of registered drivers. Linux accomplishes this by using the *USB filesystem* or *USBFS* in short. *USBFS* is a virtual filesystem much like the */proc* filesystem.

2.3.1 Virtual Filesystems

Modern versions of the Linux kernel use one central filesystem, known as the Virtual Filesystem. All 'real' filesystems register themselves with the VFS, where the filesystem registering itself specifies the 'driver' that the VFS can use to talk to itself. This requires the filesystem to implement a particular interface that the VFS can use to communicate with it. Once a particular filesystem has been registered, directories can be mounted of the different filesystem types in various places inside the VFS. For example, *ext2* and *dos* may register themselves and have respective directories mounted at `/` and `/dos`.

One of the motivations for using the VFS is to accommodate the inter-operability of different types of filesystems. The other advantage of using the VFS is being able to accommodate virtual filesystem i.e. filesystems which have no physical existence. As we saw earlier, the only requirement on part of a filesystem to register itself with the VFS is for it to implement the necessary interface with which the VFS can interact with it.

The proc filesystem for example - mounted at /proc - presents a directory structure containing a whole lot of system information concerning memory, system buses, processor etc. None of this information is physically stored in any particular file on disk. When the VFS accesses any of the files in the /proc directory, the /proc filesystem generates all this information realtime. This makes the implementation of the virtual filesystem completely transparent to the end user and indeed even to the VFS. Another example of a virtual filesystem is devfs.

2.3.2 USBFS

Having seen the idea behind a virtual filesystem, it is straightforward to see the motivation behind using USBFS and making it a virtual filesystem. USBFS tries to make all attached USB devices seem like traditional files to go with the UNIX philosophy. It dynamically keeps track of devices that are attached to, or removed from the bus. The idea behind *USBFS* is an extension of the *devfs* filesystem, which provided devices the capability to dynamically update their status as and when they were attached or removed. This is in contrast to the old unix philosophy of using nodes inside */dev*. This approach suffered from the fact that there never had to be any correspondence between an entry in */dev* and a physical device on the machine. Often huge number of nodes are present in */dev* with node entries being created for every conceivable scenario, but rarely ever used. USBFS being dynamically updated, does not have this problem.

Apart from the aspect of providing a cleaner filesystem, USBFS also makes drivers aspect of things much more transparent. all USB devices have the same major number, making the use of them to determine the drivers impractical. Instead, the filesystem decides the association based on the description the device returns on enumeration and the list of registered drivers. Apart from this aspect, USB drivers work in the same way as conventional drivers, with functions being defined for open,read,write ,ioctl etc.

2.4 Example directory structure

The usb filesystem is mounted traditionally at */proc/bus/usb*. Inside this directory there is one entry for each bus. The the contents of the directory typically reads as follows

```
/proc/bus/usb$ ls -l
total 0
dr-xr-xr-x    1 root    root          0 Oct 18 16:30 001
dr-xr-xr-x    1 root    root          0 Oct 18 16:30 002
-r--r--r--    1 root    root          0 Oct 29 12:10 devices
-r--r--r--    1 root    root          0 Oct 29 12:10 drivers
```

showing that there are two busses (designated by 001 and 002). The directories corresponding to the hubs have one file per device connected directly or indirectly to the hub. The first hub has one device connected to it. So its directory contents read:

```
/proc/bus/usb/001$ ls -l
total 1
-rw-r--r--  1 root    root      18 Oct 29 12:11 001
-rw-r--r--  1 root    root      18 Oct 29 12:11 003
```

The 001 device is always present and it represents the root hub.

003 is not characterized as a hardware device file(such as a char or a block) as is the case with the serial port, for instance. Once the file is opened, VFS forwards this request to USBFS, which then refers to its list of registered drivers and picks the appropriate driver to complete the open operation.

2.5 Selection of driver

In addition to the usual functions defined in a driver, those for USB need to have a function named `probe` defined. Once a device is connected to the USB bus, USBFS executes this `probe` method in each of the drivers, to decide on the driver corresponding to the device. The signature of the `probe` function is as follows:

```
static void *probe(struct usb_device *dev, unsigned int i,
                  const struct usb_device_id *id)
```

Based on the information retrieved from the device, the function determines as to whether it is compatible with the device and, accordingly, returns a pointer to a driver-specific value or `NULL`. For example, a driver for a hub may check to see whether the interface subclass has the value 0, and also whether the device has one single interrupt endpoint. Devices typically use the product id, vendor id and class/sub-class values while probing the device.

2.6 Programming for USB

Before transferring data to or from a device, a handle needs to be obtained to the device. *libusb* is a user-level library used to implement client-side functionality for USB on Linux. *libusb* requires that the function `usb_init()` be called before any other function is called. To get to access the various devices on the bus, two functions need be called in sequence: `usb_find_busses()`, `usb_find_devices()`. Calling these functions leads to initialization of a global

list named `usb_busses` of type `usb_bus`. The number of elements in this list is equal to the number of busses on the system. The `usb_bus` structure has a field named `devices`, which in itself is a list of all devices on the bus. A brief description on the internals of the functions follows.

`usb_find_busses` iterates through the `/proc/bus/usb/` directory to generate the list of busses and accordingly initialized the `usb_busses` list. `usb_find_devices` then iterates through the directories corresponding to the busses on the system, and creates a `usb_device` entry on the bus for each file(device) inside the directory.

As mentioned earlier, each device on the bus is represented as a file in the usb filesystem (at `/proc/bus/usb/001` directory). The contents of the file represent the device descriptor for the associated device. The descriptor may be read from the file to initialize a pointer of type `dev`, to point to the device as follows:

```
ret = read(fd, (void *)&dev->descriptor, sizeof(dev->descriptor));
```

`fd` is a file descriptor corresponding to the device file inside the usb filesystem.

After calling the two functions (`usb_find_busses()` and `usb_find_devices()`), the devices list in the appropriate bus may be iterated along to get access to each of the devices. On reaching the desired device, a handle to the device may be obtained by calling the `usb_open()` method, passing a pointer to the `usb_device` structure as a parameter. The handle may then be used to read to, write from and configure the device.

The range of functions broadly classify into two classes:

- * those for configuring the device: `usb_set_configuration`, `usb_claim_interface`
- * those for doing data transfer: `usb_bulk_write`, `usb_bulk_read`, `usb_control_msg`

3 Host Controller

This section deals with the bottom-up implementation of a USB system under Linux. At the time of configuring the Linux kernel, one has the option of enabling support for USB. When this is enabled the kernel provides support for the bottom two software layers in the USB stack, namely the host controller software layer and the USB system software layer. Inclusion of support for USB means that kernel provides a broad-based support for a variety of devices in terms of drivers. As noted in the previous sections, Linux uses the concept of a virtual filesystem named USBFS to support USB. Use of this virtual filesystem further increases the size of the USB component of the kernel.

In cases of implementing USB support for smaller kernels, there may not be kernel support for virtual filesystems. In such a case, there is a need for lightweight system software support for USB, providing a low-level basic interface for using the bus. This is also the case when implementing USB support for a realtime operating system like RTLinux, where the driver supporting USB has realtime constraints. In such a case, it would greatly help if the implementation of the driver was as explicit as possible. The lack of generality is typically not a concern if the problem being addressed does not demand a generic solution. The simpler design of the driver makes it easier to follow realtime constraints. The rest of the report describes an implementation of a driver at the host-controller level for USB.

3.1 Funcationality

The host controller is responsible for USB capabilities on the host side of the bus. It encapsulates a root hub along with a set of ports, onto which devices may be attached externally. Two types of host controllers exist. The first is Intel's Universal Host Controller Interface(UHCI) and the second is the Open Host Controller Interface, backed by Compaq. Each type has its own hardware specification and interface. This project targets the UHCI controller.

The host controller has a set of registers by which it interacts with the outside world.

- * COMMAND register: Its role is to act essentially as the control register for USB. It may be used to reset the entire bus, reset the host controller and also to start or stop the execution of the host controller.
- * STATUS register: It stores both the status of the host controller and the results of last transfer.
- * PORTC & PORTD: The host controller has one register corresponding to each of its ports named PORTC, PORTD etc. These registers store configuration information regarding the respective ports. They may also be used to perform operations on the port. For instance, specified bits on the port store information as to whether a device is attached on the particular port. Other bits may control suspension and activation of the port.
- * Apart from this the controller has other registers for configuring the setup and execution of the transfer schedule.

3.2 Transfer Schedule

Perhaps the most important aspect of transferring data on the USB is generation of a suitable schedule. A schedule is a sequential representation of the transfers that need to take place on

the bus. The idea of a transfer is encapsulated by what is called a frame. The host controller executes each of these frames for a duration of 1 millisecond. At the completion of the 1 ms interval, it moves onto the next frame irrespective of the present frame being fully executed. These frames are stored in a contiguous section in memory, known as the Frame List. Each frame is of size 4 bytes, and the frame list has a total of 1024 frames, giving the list a size of 4K (4096 bytes). The host controller treats the frame list as a cyclical list, wrapping around on reaching the end.

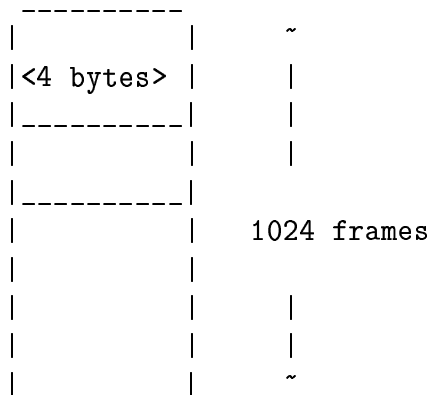


Figure 3: Frame List

3.3 Format of a frame

During each frame, a set of isochronous, bulk, control and interrupt transfers may be performed. Each of these transfers is encapsulated by a transfer descriptor. The way the transfer descriptors are laid out is different for isochronous transfers as compared to other types for transfers. For isochronous transfers, the TDs are laid out sequentially in a list as follows:

IsoTD1 -> IsoTD2 ->-> IsoTDn

But for other types of transfers, these are pointed to by separate elements called queue heads. Putting all these together, the transfer schedule looks as follows: The order of execution of various types of transfers is implicit in the layout. It is evident that the initial set of transfers to take place is those relating to the isochronous type. After completing all the isochronous ones, there are two ways in which the rest of the TDs in the queue heads may be executed. First is depth-wise, in which case all the interrupt TDs are executed after which the control

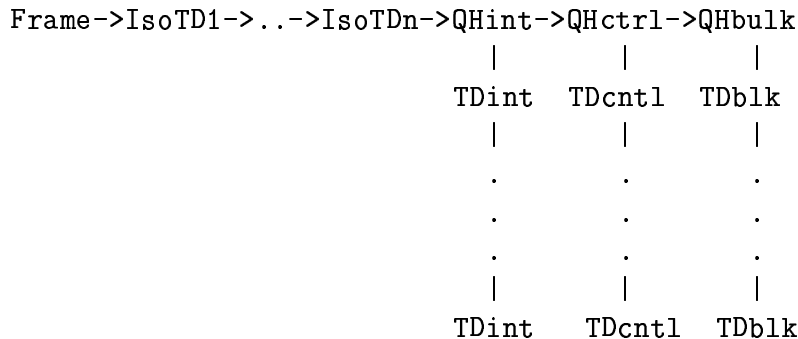


Figure 4: Transfer Schedule

and finally the bulk. The other option is breadth-wise, in which case the first TD in each of the interrupt ,control , and bulk is executed after which the second TD in each is taken up and so on.

By changing the relative numbers of the TD's of various types in each frame, the system software can alter the bandwidth allocated to each of the different transfer types. The specification states that isochronous needs to be allocated at least 80% and control at least 5% of the bandwidth. The software also needs to ensure that it does not try to schedule more TD's in one frame than is possible to transfer. It needs to take into account the actual speed of the transfer to estimate the time required for the transfers.

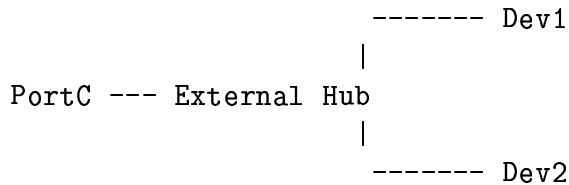
In course of normal execution, the host controller executes frame sequentially in periods of 1 millisecond. However the COMMAND register has a DEBUG bit, which on being set makes the host controller stop after execution of each frame, which allows examining the values of the different registers at the end of each frame.

3.4 Control Transfers

Control transfers are crucial to the host controller as they allows the host to configure the bus and also the devices on it. Some of the functions that may be performed with control transfers include - setting address of a device, getting the device configuration and getting/setting device descriptor.

3.4.1 Device Discovery & Address Assignment

One of the uses for control transfers is in setting the address of a device. One of the most basic functions that the USB driver needs to accomplish on starting up, is to generate a tree structure of all the devices attached to the bus. In the process of doing so it also assigns a unique address to each of the devices. All future references of the device are done using the assigned address.



Consider the layout shown above. Once the device has been attached to the USB port with a hub and two devices, the port recognizes the change in its status and that a new device has been attached. Initially, each of the three external devices (hub ,dev2 ,dev2) is disabled. The port first enables the external hub, with the two other devices still disabled. Once this happens, the external hub becomes active with an address of the default value of 0. The host controller uses this zero address to address it and assign it a unique address (lets say 1). Then, the host enables the port on the hub on which dev1 is attached to the external hub, after which the dev1 becoms active with a default address of 0. Just as before, the host assigns dev1 a unique address (lets say 2). The same is done for dev2. Thus, the host controller successfully generates a map of the structure of the bus, also assigning a unique address to every device on the bus in the process.

4 Description of Project

4.1 Objective

Device address assignment represents a very typical control transfer and is representative of most other control transfers. The objective of the project was the development of a kernel module to address this problem of address assignment. The kernel module developed, attempts to configure and set the address for a device on the bus. Even though the present problem targets address assignment for a single device, solving the problem involving a heirarchy of devices is a minor extension as it essentially involves using the solution for the single-device problem repeatedly.

4.2 Algorithm

The algorithm for solving the problem may be stated as:

1. The USB protocol specification requires global resetting of the bus before the bus can be used. So assert global reset for 10 ms
2. Reset the host controller
3. Setup interrupt management
4. With the host controller having been reset in step 2, assert reset on the port where the device is attached for 50 ms
5. Enable the port
6. Setup the Frame list, QH and TD for the control transfer
7. Set the host controller in debug mode
8. Run the schedule

4.3 Aspects of the Module

4.3.1 Kernel Modules

Any application that needs to run as part of the kernel has two options of doing so. First, the code can be hard-coded into the kernel, or it can be loaded in as a module. The latter is the approach adopted in this application. Kernel modules, as the name suggests, have the advantage of retaining the modularity of the kernel. Only the necessary features need to be loaded into the kernel. Also newer versions of the module can easily be integrated without having to recompile the entire kernel. A module being loaded into the kernel is similar to object code being linked with libraries. All functions used in the object code, need to have been defined either in the code itself or in libraries that are linked to it. Similarly, all functions used in the module need to have been defined in the module itself, or it needs to have been defined elsewhere in the kernel. So, for instance a kernel module cannot use `printf`, as this is defined in the `libc` library and not inside the kernel. However, the kernel does provide the `printk` function for logging purposes.

4.3.2 Memory Allocation

A lot of times in the application contiguous chunks of memory need to be allocated. This is true for example in the case of frame list, where 4096 sequential bytes of memory are required. Since userspace application use virtual memory, when a chunk of memory is requested by the application, though the memory returned has contiguous virtual address, it need not (in most cases does not) correspond to one big contiguous space in physical memory. However, since the application in this case is a kernel module, and hence runs in kernel space, the only memory being dealt with is physical memory. There is simply no concept of virtual memory space within the kernel. `kmalloc` is a kernel function which may be used to request memory space from within the kernel. Any chunk of memory returned by `kmalloc` is guaranteed to be contiguous.

Another aspect of memory allocation is with respect to alignment. The frame list, as mentioned earlier is of size 4096 bytes. The host controller specification requires that the start address of this frame list be alligned on a 4K boundary. This has the obvious advantage that then the 32 bit address has only 20 significant bits, with the last 12 bits being 0. So the register storing the base address of the frame list needs to set aside only 20 bits for storing the address. Fortunately, all memory requests of power of 2 size are alligned on the boundary corresponding to that power of 2. In other words, `kmalloc(4096)` would return a chunk of memory alligned on a 4K boundary.

4.3.3 PCI

The host controller specification for UHCI specifies the location of each register as being at a particular offset inside the configuration space for the device. Since the host controller is a PCI device, the base address for the configuration space needs to be queried from the PCI configuration.

PCI is a bus standard used in most modern day computers as a means to connect peripherals to the computer's datapath. Among the entire address space in the computer's memory, specific regions are set aside to be used for purposes other than addressing conventional memory. Specifically, PCI and ISA are allocated regions which may be used by devices on these busses, known as I/O memory. Each peripheral device generally has a set of registers and on board memory of its own. These registers when mapped to locations in I/O memory are called I/O ports. The on-board memory may be mapped to regions in the I/O memory too, in which case it is called memory mapped I/O. The host controller, being a PCI device in itself, has its registers mapped to a specific location in the I/O space. To retrieve PCI-related information about the device, one needs to use the set of PCI-related functions provided by the kernel. First, the location of the device needs to be determined. This is fully specified

by (a) bus value and (b) function value.

To determine these two values, the function:

```
pcibios_find_device(vendor,id,0,&bus,&function)
```

may be used. As the prototype suggests, the vendor and the id values of the device need to be passed to the function. The function iterates through the list of the PCI devices and accordingly the bus and function values are assigned for the specified device. The vendor value for Intel (maker of UHCI host controllers) is 0x8086 and the id value for the host controller is 0x2442. These values are generally available with the documentation for the product. Having obtained the values of the function and bus of the device, any offset into the configuration space for the device may be referenced with the functions *pcibios_read_config_dword* or *pcibios_read_config_byte*.

4.3.4 Interrupts

Interrupts may be used for two different purposes. First, the host controller has the option of interrupting once it completes executing a transfer descriptor. The Interrupt enable register may be used to configure the interrupt behaviour of the host controller. The other case when interrupts are used is to report error conditions. Since two different scenarios cause interrupts to fire, polling between the various status bits needs to be done to establish the cause for the interrupt.

Similar to the address of the configuration space, the IRQ number for the host controller needs to be extracted from the information in its configuration space. The offset for the IRQ number is given by the macro `PCI_INTERRUPT_LINE`. Hence the value can be retrieved as:

```
pcibios_read_config_byte(bus,function,PCI_INTERRUPT_LINE,&HCirq);
```

on which `HCirq` gets assigned the value of the IRQ

References

- [1] *Universal Serial Bus Specification*, Revision 2 (2000)
- [2] *Universal Host Controller Interface(UHCI) Design Guide*, Revision 1.1 (1996)
- [3] Alessandro Rubini, Jonathan Corbet (2001). *Linux Device Drivers* O'Reilly, Sebastopol, California
- [4] Jan Axelson (2001). *USB Complete*, 2nd edition, Lakeview Research