**DOCUMENTATION**

**Group members:**

Abubakar Sarwar K13-2212

Elsa Zehra K13-2164

Fahad Tanweer K13-2068

**PROJECT TITLE:**

**“USB DEVICE DRIVER”**

**DESCRIPTION:**

Firstly, we should be aware of the fact that whether a driver for a USB device is there or not on a Linux system, a valid USB device will always be detected at the hardware and kernel spaces of a USB-enabled Linux system, since it is designed (and detected) as per the USB protocol specifications. At the hardware level, the Hardware-space detection is done by the USB host controller which is typically a native bus device The corresponding host controller driver would pick and translate the low-level physical layer information into higher-level USB protocol-specific information. The USB protocol formatted information about the USB device is then populated into the generic USB core layer (the usbcore driver) in kernel-space, thus enabling the detection of a USB device in kernel-space, even without having its specific driver available.

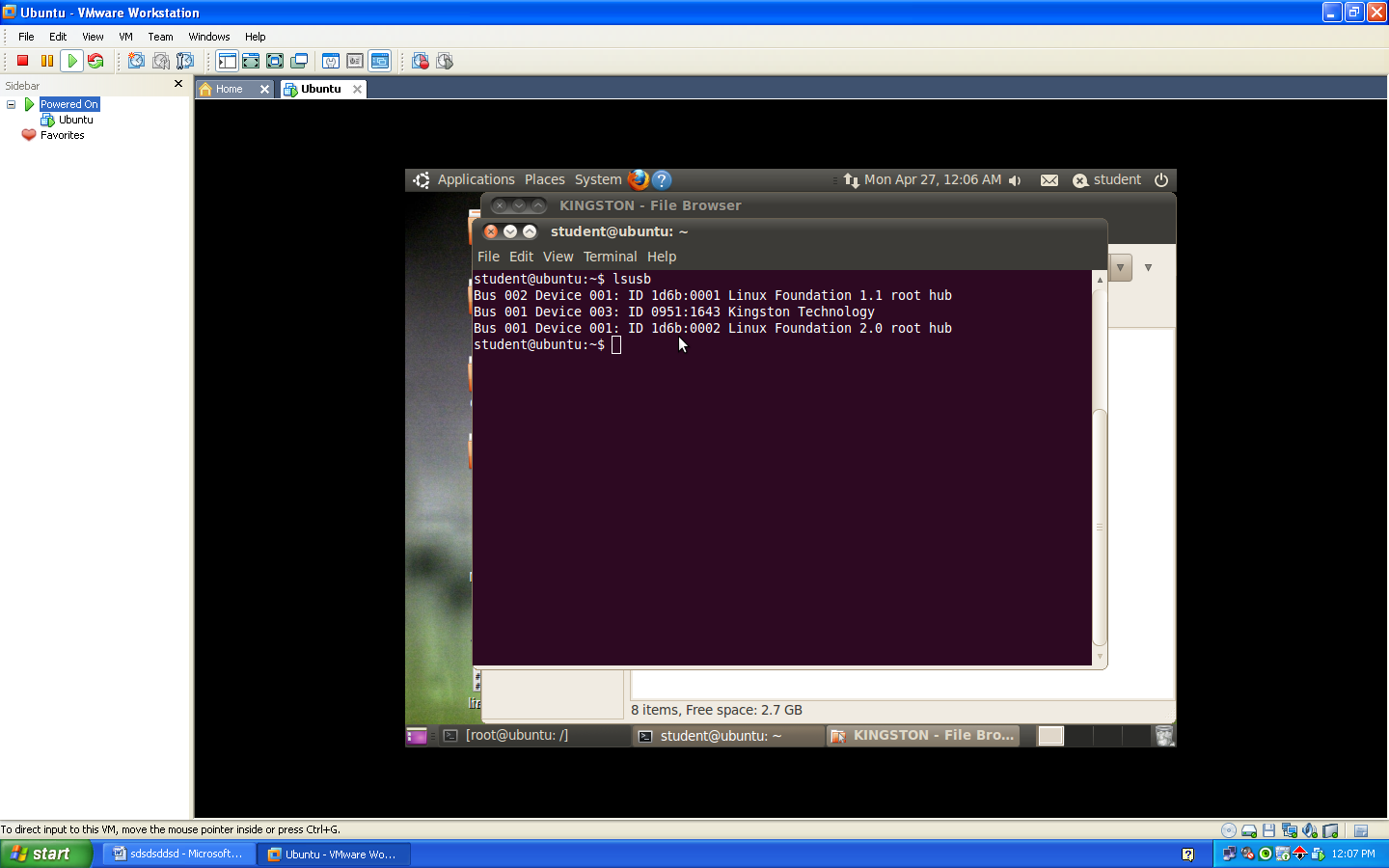
A basic listing of all detected USB devices can be obtained using the lsusb command, as root.

**COMMANDS USED:**

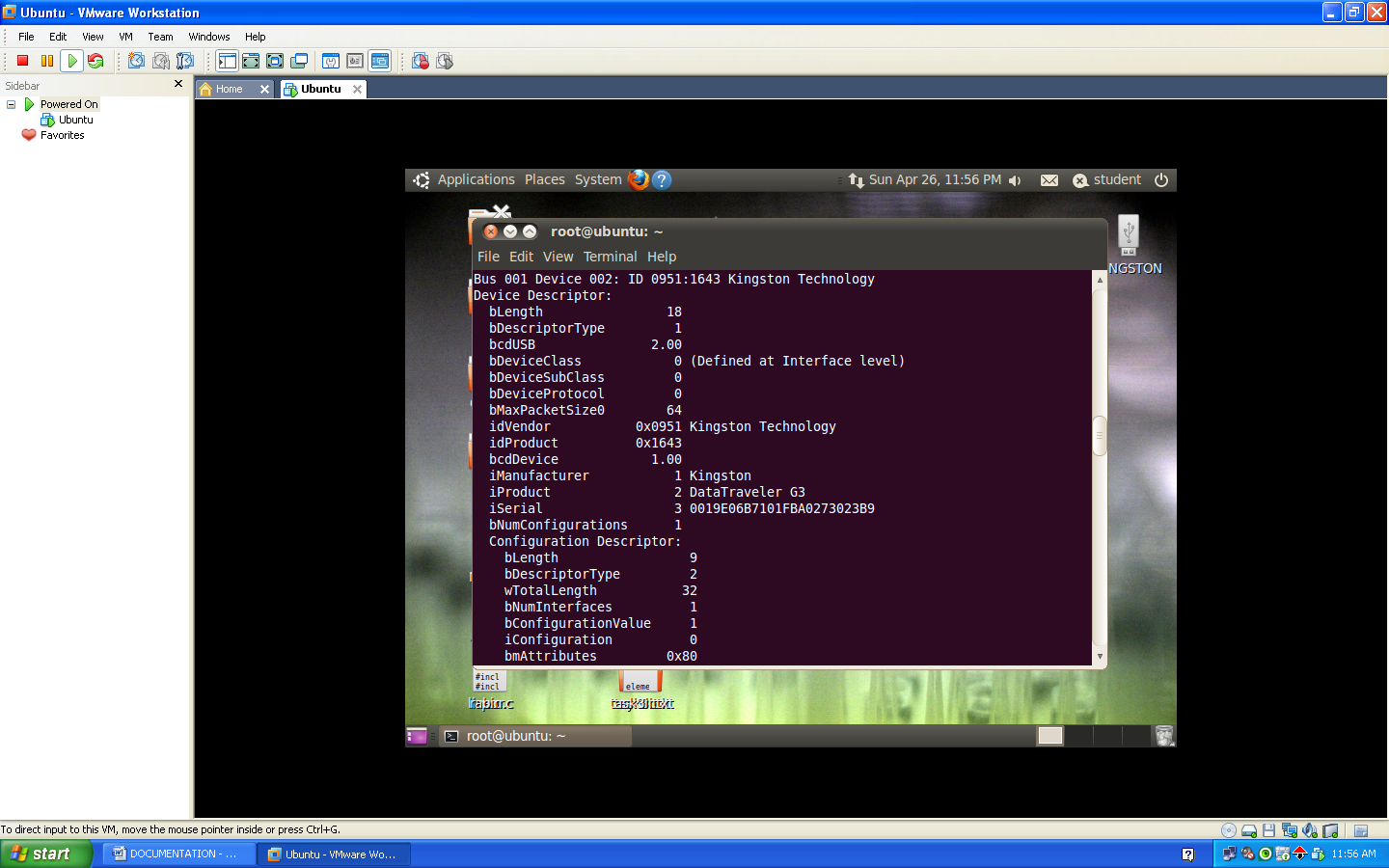
1. lsusb (It is used to view all the usb devices connected )
2. lsusb –v (Is is used to give a detailed view of the usb devices connected)
3. cat /sys/kernel/debug/usb/devices (It also gives detail info of the connected devices including the details of interfaces and endpoints)
4. sudo –i (in order to log in from root)
5. make (to compile a module)
6. insmod (inserting a module)
7. rmmod (removing a module)
8. modprobe (inserting back a module)
9. mount (lists all mounted devices)
10. fdisk –l (Gives detailed info of the mounted disks)
11. mkdir (create new directory)

**A few screenshots of the important commands**

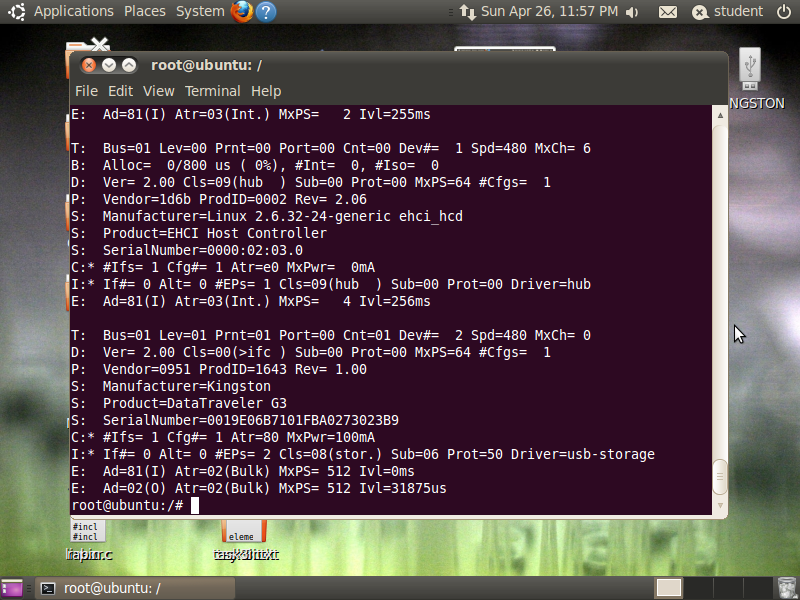
1. **lsusb**



1. **lsusb -v**



1. **cat /sys/kernel/debug/usb/devices**



The USB input device must connect the USB input device driver to two different infrastructures: The USB core device driver that handles hardware event on the USB port . And the Input module that collects and dispatches input events.

We have two broad categories Of USB Device driver here:

* **USB host:** It has all the information of USB. It knows about the endpoints, versions, etc.

The USB host forwards the information to the USB core.

* **USB core**: And the basic handling is done through the USB core.

A configuration of a USB device is like a profile, where the default one is the commonly used one. As such, Linux supports only one configuration per device — the default one. For every configuration, the device may have one or more interfaces. An interface corresponds to a function provided by the device.

So, unlike other device drivers, a USB device driver is typically associated per interface, rather than the device as a whole which means that one USB device may have multiple device drivers, and different device interfaces may have the same driver. But of course, one interface can have a maximum of one driver only. For every interface, there would be one or more end-points. An end-point is like a pipe for transferring information either into or from the interface of the device, depending on the functionality. A device has multiple interfaces and each interface has multiple endpoints.Depending on the type and attributes of information to be transferred, a USB device may have one or more endpoints, each belonging to one of the following **four** categories:

* **Control** — to transfer control information. Examples include resetting the device, querying information about the device, etc. All USB devices always have as their default control endpoint point as zero.
* **Interrupt** — for very small and very fast data transfers, typically of up to 8 bytes not more than that. Examples include data transfer for serial ports, keyboards, mouse, etc.
* **Bulk** — for big but comparatively very slow data transfers. A typical example is data transfers for mass-storage devices such as mobile phones.
* **Isochronous** — for big data transfers but with a bandwidth guarantee, though data integrity may not be guaranteed in this case. Typical practical usage examples include transfers of time-sensitive data such as audio, video, etc.

However, the main work in our USB device driver is of bulk and interrupt.

**Endpoint can be classified in two portions:**

1. **(IN) Input**: it means that the data is being transferred from the device to the CPU.
2. (**OUT) Output**: It means that the data is being transferred from the CPU to the device.

It means that all the four endpoints will be either IN or OUT. This way we can say that there are a total of 8 types of endpoints. However, the main four that are being used in in the project are given below:

1. **Bulk input:** When large amount of data is being transferred from the device to the CPU.
2. **Bulk output:** When large amount of data is being transferred from the CPU to the device
3. **Interrupt input:** When small amount of data is being transferred from the device to the CPU.
4. **Interrupt output:** When small amount of data is being transferred from the CPU to the device

Technically, an endpoint is identified by using an 1 byte number, the most significant bit or you can say that the (MSB) of which indicates the direction — 0 means “out”, and 1 means “in”. Control endpoints are bi-directional, and the MSB is ignored. As per the USB protocol specification, all valid USB devices have an implicit special control end-point zero, the only bi-directional end-point.

Coming back to the USB device sections (Figure 3), the first letter on each line represents the various parts of the USB device specification just explained. For example, D for device description, C for configuration, I for interface, E for endpoint, etc..

In order to design a USB device driver, there are so many things to know in hand. For example: device configuration, interfaces, transfer pipes, their four types, and so many other symbols like T, S ,etc..

To be specific, the E: lines in the figure show examples of two bulk endpoints of the USB Kingston device under consideration. Also, the endpoint numbers (in hex) are, respectively, 0x81 and 0x02. The MSB of the first being 1, indicates that it is the part of ‘in’ endpoint, represented by (I) in the figure. The second is an (O) or ‘out’ endpoint.

MxPS specifies the maximum packet size or the data size that can be transferred in a single go. Now lets talk about “Ivl”, it specifies the interval in milliseconds to be given between two consecutive data packet transfers for proper transfer.

Like any other Linux device module, here, too, the constructor and the destructor are required basically the same driver template that has been used for all the drivers. However, the content would vary, as this is a hardware protocol layer driver, i.e., a horizontal driver. The difference is that here this would be done with the corresponding protocol layer — the USB core in this case; instead of providing a user-space interface like a device file, it would get connected with the actual device in hardware-space.

I speak to the interface descriptor with its file, interchange number, the usefulness class/classification of this interface, the driver connected with this interface, and the quantity of endpoints under this interface. . What's more, contingent upon the quantity of endpoints, there would be the same number of E lines, points of interest of which have as of now been talked about before. Bus here specifies the number of buses currently associated. It depends on the number of buses your CPU has. We are mainly concerned here with the bus and the port here.

The \* after the C and I represents the currently active configuration and interface, respectively. The P line provides the vendor ID and product ID. S lines are string descriptors showing up some vendor-specific descriptive information about the device.

Furthermore, we also incorporated the mounting concept in our project. Mounting is the act of making a storage device accessible as part of a file system. Mount basically attaches a device to a pre-specified sub-directory (folder) in the very own file system (all attached devices) and it makes the devices online or you can say accessable.

It is detected by GNU/Linux as /dev/sdax, where x stands for the number of the USB port. And once our USB device is connected to the USB port of the machine in your CPU, we can mount that manually.

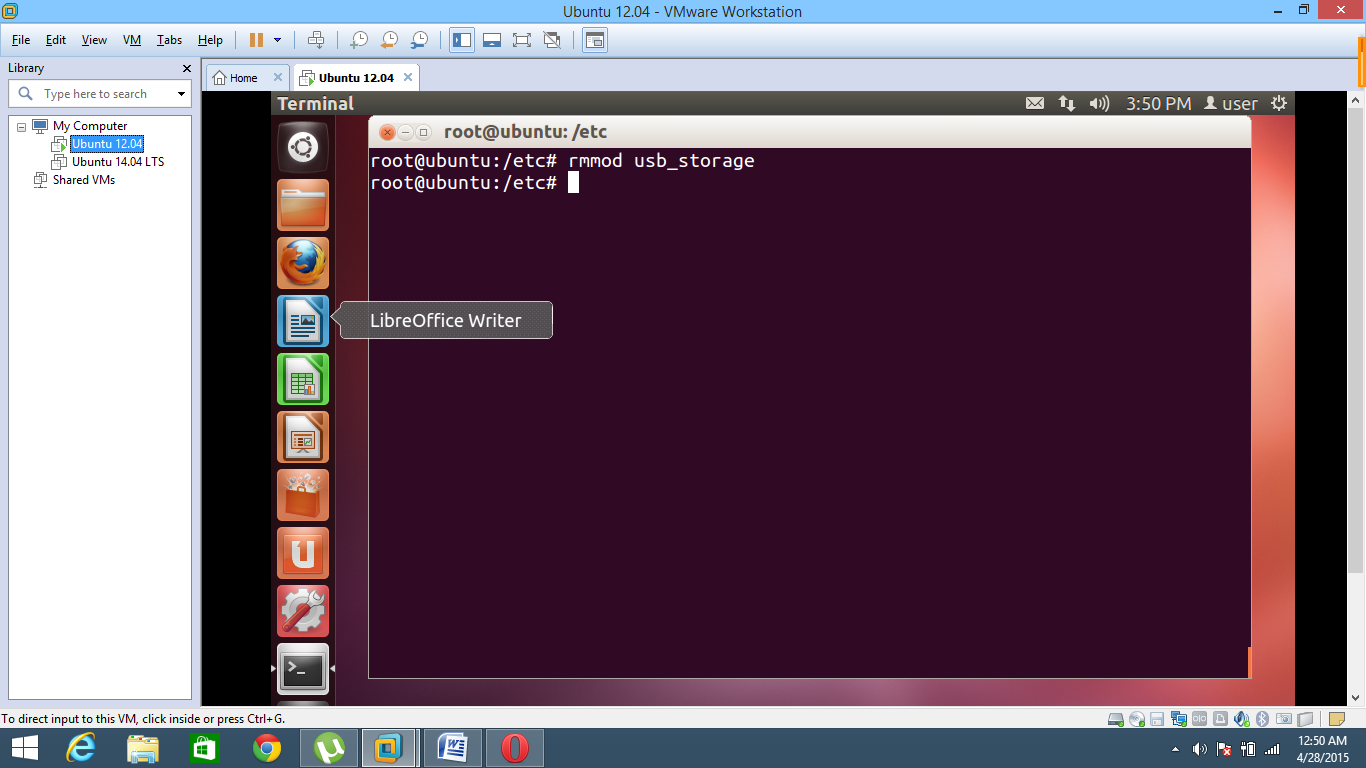
**FROM WHERE WE INITIATED OUR PROJECT?**

Initially, we searched for our topic but since our topic is difficult hence we were not able to find anything useful. We wasted most of our time as we were confused from where to initiate our project. But at last, we were able to find some useful links related to our topics which are mentioned in the last of our documentation.

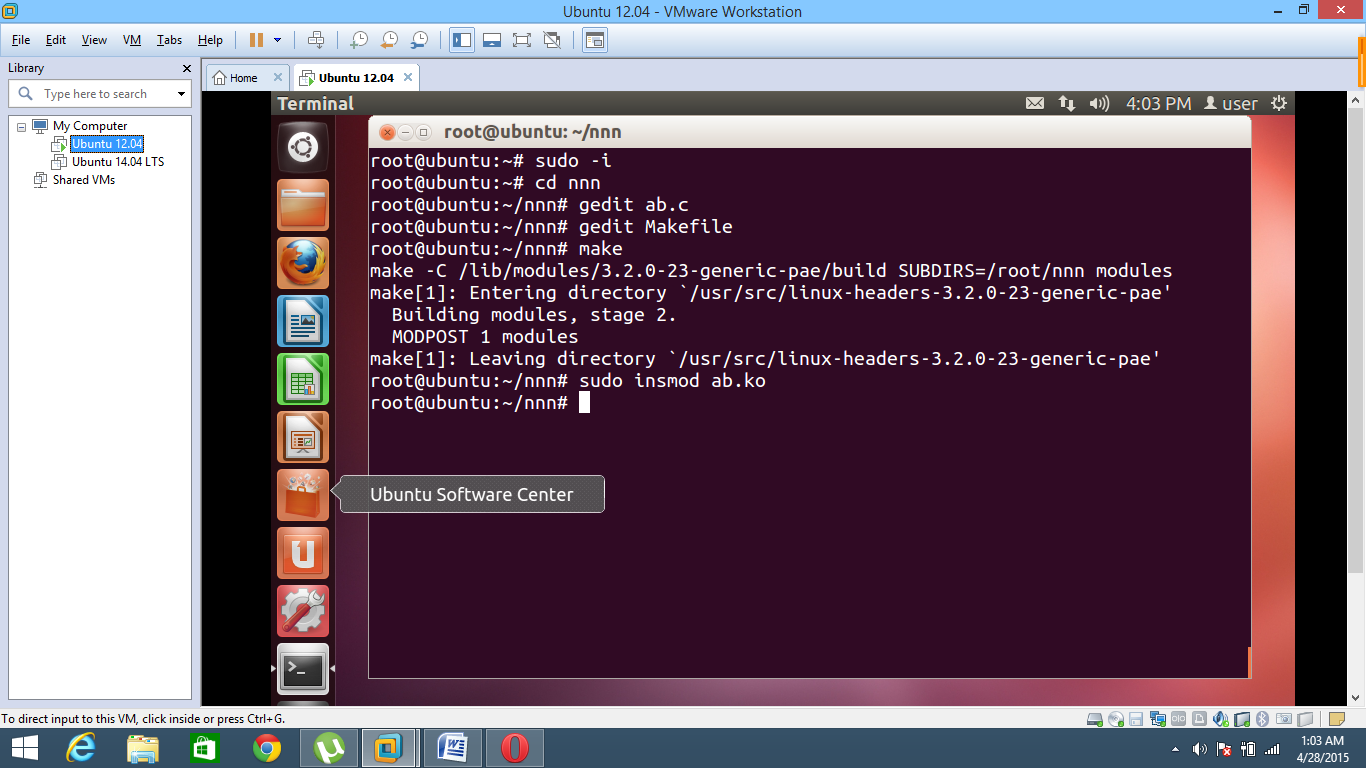
* Firstly, we checked all the functions which are a part of the device driver. After that we implemented that in the structure form of the program.
* Secondly, we got to know about the vendor and product ID. These ID’s are an important part for implementing the USB device driver. Hence we identified the Id’s and incorporated those in our code.
* Thirdly, we solved all the below mentioned problems step by step in order to reach the conclusion.

**THE ACTUAL WORKING OF THE PROJECT:**

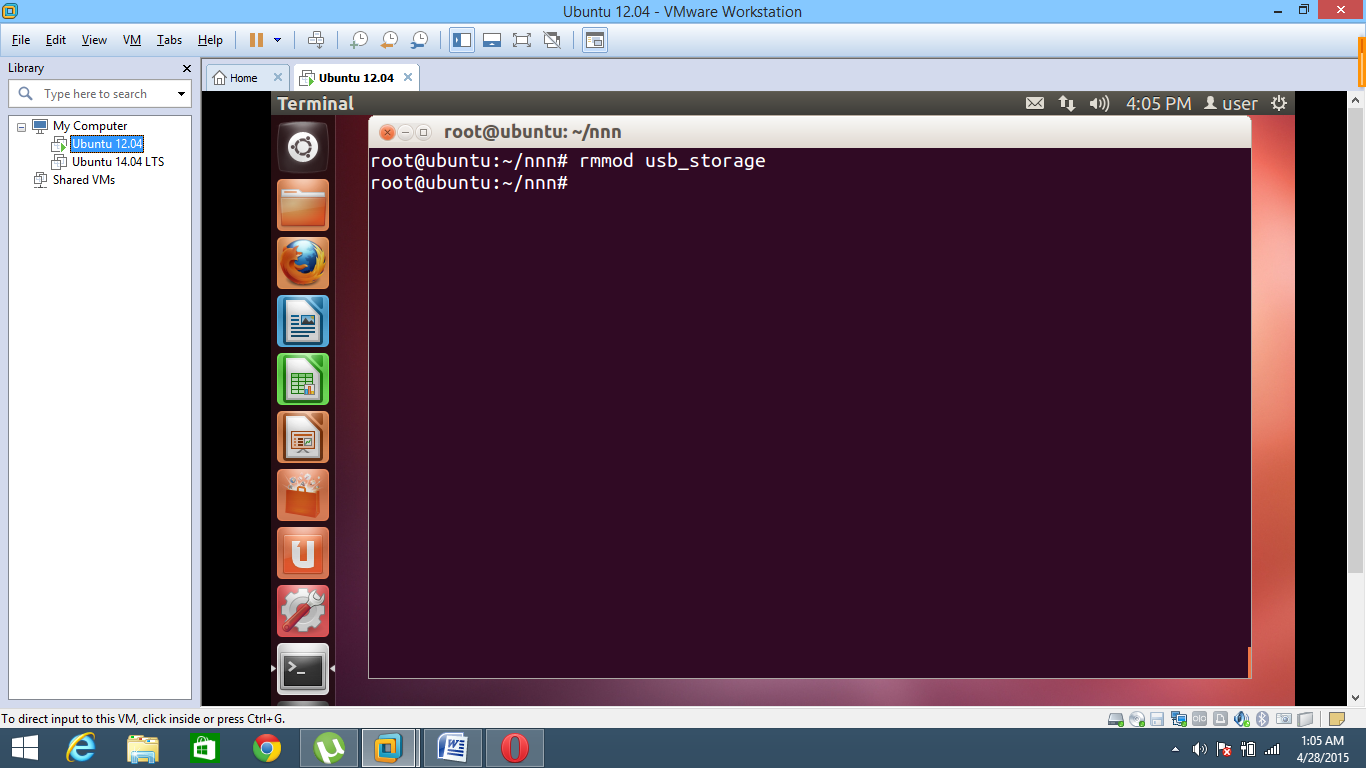
The actual working of the project is basically that it first of all removes the initial device driver installed in ubuntu using the command “rmmod usb\_storage”.



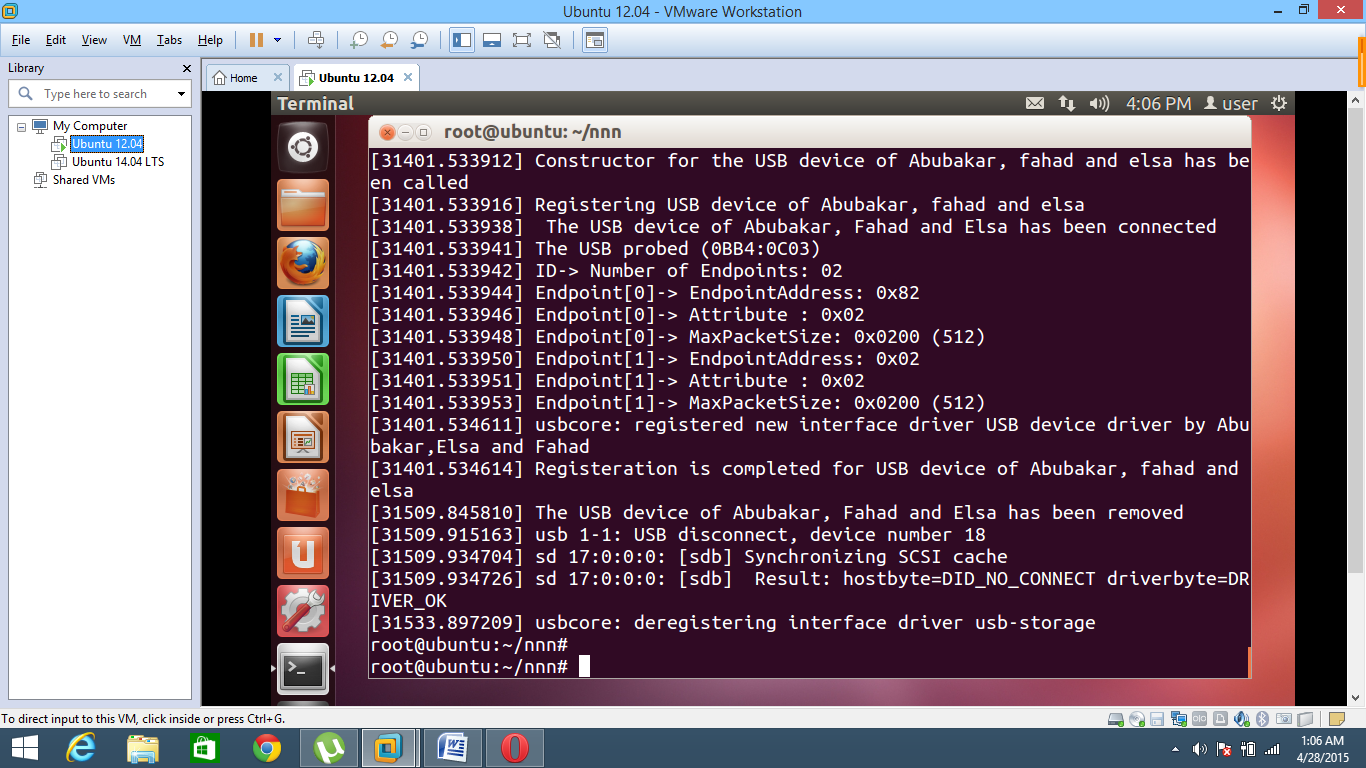
Then we will go to root and create a new directory. Inside the directory we will create our code and Makefile. Then we will make/compile our module. Then the module is inserted to the kernel.



Note that if we wouldn’t have deleted the usb\_storage driver then our prob function would have never been called. Prob function is basically the function that is called when the device is connected. So we will first remove the initial usb driver in the ubuntu.

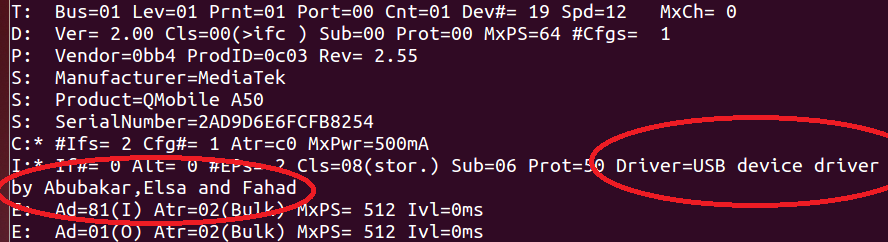


Meanwhile, when the module is inserted our init function will register the device driver by taking license of GPL. Without license of GPL the Ubuntu kernel doesn’t allow us to access the registeration function from the <usb.h> library. Once we have registered the our module as a device driver for USB. Now we will connect the usb, and when we connect the USB our prob function is called and it also prints the endpoints and it’s details. To check this we will call the “Dmesg” command and verify.



Constructor calling shows the insertion of the module. Then Registering USB device line is showing that our prob function is called. Then we have state the number of endpoints and given a small detail about them.

After that I have removed the device so it says that the device has been removed. Afterwards the deregistration is done. The details can be verified by using the “cat /sys/kernel/debug/usb/devices” command too. After that the ID of the device is matched with the ID’s available in our ID table. If the vendor ID + product ID of the device is exactly same as in our ID table then our driver will take control of the device. Once it has the control you can again use the “cat /sys/kernel/debug/usb/devices” command and check the driver controlling our device. The driver will show that the driver of the USB is now “USB Device driver by Abubakar, Ilsa and Fahad.”



The above picture shows that the controller of the device attached is USB device driver by Abubakar, Elsa and Fahad which was the name of our driver. Now we can simply remove our mod again using the rmmod if we wish too.

**EXPLANATION OF SOURCE CODE:**

The source code is very simple. The header files used are linux/module.h, linux/kernel.h and linux/usb.h. The must important structure is our usb\_driver structure. It is the core of our code. It includes 4 main branches. Name, id , prob and disconnect. There are three functions defined inside the structure of usb\_driver. Name is simply the name of the device driver.

First of all lets talk about our ID table which is taken in the form of an array. It is because we want to detect as many devices we want. The ID table is fully dependent upon two things. The Vendor ID and the Product ID. Whenever a device is connected this ID\_Table functions just matches that whether the vendor ID and product ID matches to any of the stored ID’s. If yes, then it takes control of the device. However , if it doesn’t match then it lets go. The vendor ID and Product ID are saved in the USB\_DEVICE(vendor,product) function. At the end we have a terminating condition for a situation in which our driver receives null.

Moving on towards the prob function. It is called whenever a device is connected to any of the serial ports in our system. In this function we have two arguments one is the interface pointer while other is for the USB ID(Yes it is same as the ID I mentioned about above). Other than the arguments I have initialized two different structures. One of type usb\_host\_interface while other is of type usb\_endpoint\_descriptor. The host interface has all the information about the interface of the device you have connected. Meanwhile the endpoint descriptor will have the detailed information about the endpoints available in our device. These both are just used inorder to print the number of endpoints and to print out the attributes and details of the endpoints. After that we have stored the information of the usb\_host into a pointer and then printed the number of endpoints. Then we had a loop from 1 to the number of endpoints available. Then one by one we have printed the addresses, MXPS and the type of endpoint for all the endpoints inside the device.

After that we have a function named disconnect. The name suggests it very well. Yes, the disconnect function is only called when the device is disconnected. We don’t have to do anything special in this function. Therefore, I have just printed a statement on the console instead.

Now talking about the init function. We all know that init function is the function that is called when you are installing a mod. Therefore, we have registered the module as a usb device driver in the init function. However, if the usb registration is not done properly then the return value will be -1 else than that it will be greater than 0.

To conclude, we can see 5 different lines. The first two lines of module\_init(usb\_init) is just telling the kernel that the init function for this module is usb\_init. Same goes for our favourite exit function. After that we have taken license from GPL. Without this license I cannot call the functions such as usb\_register or usb\_deregister. It is because the ubuntu has a permission set not letting everyone use those special function. But when you give license of GPL then no one can stop you from taking any function from the usb.h library. Next line just shows the author and after then that the module\_description is showing just a small description of your project.

**DIFFICULTIES FACED:**

1. **PROBLEM#1:**

We were confused that how a driver selectively registers or skip a particular interface of a USB device. We need to verify that the device is actually one that it can accept. If not, or if any error occurs during initialization then we should have some idea about that.

1. **PROBLEM#2:**

Initially, we didn’t have any idea how to remove the default driver since we wanted to incorporate our own driver in that.

1. **PROBLEM#3:**

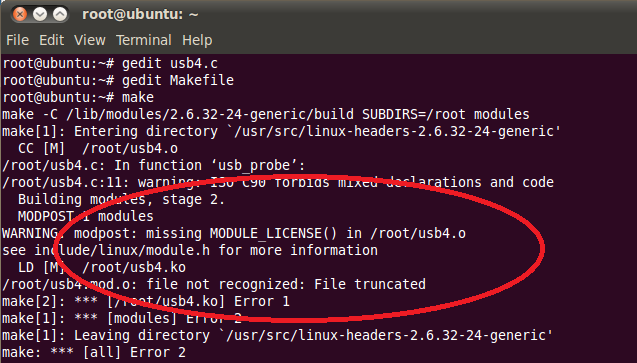
Our third problem was dependent on the second problem. We identified the solution of our second problem. But then we were left with another problem. We had no idea how to incorporate the original driver after we are done with the work. It means that once we have deleted the initial driver, how are we supposed to get it back? The insmod function was failing over here for some reason.

1. **PROBLEM#4:**

We were not able to identify that whether the detected driver is by default or the one which we incorporated.

1. **PROBLEM#5:**

We were designing a device driver in spite of the fact that our linux already had its own device driver. So the problem was that we were not allowed to do that in a simple manner. As from 2008, Ubuntu does not allow the users to incorporate their own device driver as they have provided the user with the default one. It meant that we were not allowed to access the usb\_register() function from the <linux/usb.h> library. The screenshot for that respective problem is shown below:



**SOLUTIONS TO ABOVE MENTIONED PROBLEMS:**

1. **SOLUTION TO PROBLEM#1:**
2. We registered the usb driver in our init function of the module. The init function is called whenever a module is inserted. Therefore, whenever we will insert our module into the kernel our registration will perform. To register it we basically used the function of usb\_register. Similarly to deregister it we used the function usb\_deregister that is a predefined function.
3. **SOLUTION TO PROBLEM#2:**

For that we used rmmod usb-storage command.It initially removes the default driver.

* rmmod <module> — removes/unloads the module

1. **SOLUTION TO PROBLEM#3:**

For that we used the command of modprobe.

* modprobe <module> — inserts/loads the module, along with any dependencies

1. **SOLUTION TO PROBLEM#4:**

For this we found more than one solution. The first one was that we used the command of “dmesg” whenever we attatched a USB to our PC. This way we could see that our prob function is called and our device driver is in control of the USB.  
Another way was to use the command “cat /sys/kernel/debug/usb/devices”. Over here in the i row and last column we can see the driver name that is handling the device.

1. **SOLUTION TO PROBLEM#5:**

For that problem we used MODULE\_LICENSE (“GPL”)

Here, GPL stands for “General Public License”. Which means the system gives us the license or authority to incorporate our own device driver.

**LINKS USED IN OUR STUDY**

<http://www.linuxjournal.com/article/4786>

From this link we got the idea about **how to register USB driver with the Linux USB subsystem**, giving it some information about which devices the driver supports and which functions to call when a device supported by the driver is inserted or removed from the system.

<http://www.computerhope.com/unix/umount.htm>

This link was useful in the sense that we tried **to figure out our file system from this link as the initial task was to identify that on which specific system our file system runs.**

<http://www.opensourceforu.com/2011/11/usb-drivers-in-linux-2/>

This link was the most important link that helped us a lot in initiating our project. From this link, we got the understanding of our output. Since, in this link **the concept of endpoints, it’s type and interfaces** is defined properly and in detail.

<http://www.opensourceforu.com/2011/10/usb-drivers-in-linux-1/>

This link build up our concepts that what basically a **USB** **device driver is in terms of the basic structure**. We had a clear idea about the different parts of the driver after going through that link. Moreover, the most important concept **of vendor and product ID** was mentioned in that which helped us a lot.

<http://free-electrons.com/doc/linux-usb.pdf>

It was the best link so far as from that link we had the basic idea of the **usb core and usb host** concept. Moreover, It described everything in a precise manner related to our driver.

<https://lwn.net/images/pdf/LDD3/ch13.pdf>

This link had a complete PDF on **USB drivers**. That sums up all the theory related stuff of our project. Hence, it’s a useful link.

<http://www.mulix.org/lectures/intro_to_linux_device_drivers/intro_linux_device_drivers.pdf>

From this link we got the idea about all the Linux device drivers**. There basic needs, requirements and implementation**. From here, we identified the usage of init and exit module.

<https://developer.apple.com/library/mac/documentation/DeviceDrivers/Conceptual/USBBook/WorkingWithUSB.pdf>

This was a useful link as it provided some theory related useful level in a next level approach so that we could have a vast idea of the things going inside.

<https://repo.zenk-security.com/Linux%20et%20systemes%20d.exploitations/Writing%20Linux%20USB%20Device%20Drivers.pdf>

From this link we had the basic idea about **how should we initialize the functions of the device drivers** and work on it in a proper manner.

[https://msdn.microsoft.com/en-us/library/windows/hardware/ff540174(v=vs.85).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff540174%28v=vs.85%29.aspx)

This link was helpful in the sense that since the description in this link was much elaborated so we initially tried understanding those and then implementing things on our kernel level

<http://www.atmel.com/Images/doc8447.pdf>

From this link we got the clear idea about **the USB device interface**. It had a detailed view of things which were much elaborative.

<https://help.ubuntu.com/community/Mount/USB>

This link helped us a lot in building our basic **concepts of the mount** thing. We tried those commands and it really helped us in understanding major things.

<http://www.blackmoreops.com/2014/08/25/identify-pci-usb-wired-wireless-driver-linux-identify-usb-driver/>

From this link we got the basic idea of **how to identify the USB drivers currently on the syste**m. From here we got to know about our lsusb command and it’s understanding.

<http://www.thegeekstuff.com/2013/01/mount-umount-examples/>

From this link we tried to get the idea about the mounting. We got to know about the **basic mounting commands** from there. We got to know that how we mount a specific mount point in linux.

<http://www.howtogeek.com/howto/31726/mount-usb-devices-in-virtualbox-with-ubuntu/>

This didn’t help us much but thanks to this link as from this link at least we had the idea that after mounting in which folder our data should be present.