Namal Institute Mianwali
Computer Science Department
Operating System
Fall 21

Document:	**OS Semester Project**	**Date:**	**09-12-2021**
Prepared by:	**Jawad Ahmed (BSCS 2019-59)**	**Prepared for:**	**Dr. Adnan & Sir Asad**
Document Details:	**This document contains the details regarding how the usb module is implemented in the linux operating system.**		

01. Prerequisites:

a. There are many operating systems and each operating system has their own requirements for making and implementing the module.
b. In my case, I am making the usb module for the linux operating system, therefore it is a prerequisite that we have to set up the linux environment i.e ubuntu or linux mint.
c. I installed Linux Mint on my system.

02. First Linux Device Driver:

a. Before implementing the usb device driver, first of all I will try the “Hello World” module.
b. Following are the module information.
    i. **Licence:Licence:**
        1. GPL, or the GNU General Public Licence, is an open-source licence meant for software.
        2. In my case I use the GPL licence for the module.
        3. The GPL General Public Licence is a series of widely used free software licences that guarantee end users the freedom to run, study, share, and modify the software.
    ii. **Author:**
        1. I am writing the module, therefore I am the author.
        2. MODULE_AUTHOR("Jawad Ahmed");
    iii. **Module Description:**
        1. Using this macro I can give a description of a module.
        2. MODULE_DESCRIPTION("A simple “Hello World” Module For Linux OS");
    iv. **Module Version:**
        1. Using this Macro I can give the version of the module or driver.
        2. The <version> may contain only alphanumerics and the character `.’
        3. As it is the first version of a simple module so I keep it 1.0.
        4. MODULE_VERSION("1.0");
    v. **Data Structure For this module:**
        1. In all programming there is a starting and ending point. In my case these are following:
            a. **Init function:**
                i. This is the function that will execute first when the Linux device driver is loaded into the kernel.
                ii. For example, when we load the driver using **insmod**, this function will execute.
                iii. This function should register itself by using **module_init()** macro.

b. **Exit function:**
    i. This is the function that will execute last when the Linux device driver is unloaded from the kernel.
    ii. For example, when we unload the driver using **rmmod**, this function will execute.
    iii. This function should register itself by using **module_exit()** macro.

2. **Printk():**
    a. In C-Programming we use the function printf() for printing the values. But printf() is a user-space function. Therefore we cannot use this here. There is another function for the kernel having the same purpose named printk().
    b. printk() has some different functionalities as compared to printf() such as it can classify messages according to their severity by associating different log levels, or priorities, with the messages.
    c. There are many macros used for printk() in order to indicate the log level. Some are described below:
        i. **KERN_EMERG:**
            1. Used for emergency messages, usually those that precede a crash.
        ii. **KERN_ALERT:**
            1. A situation requiring immediate action.
        iii. **KERN_ERR:**
            1. Used to report error conditions; device drivers often use KERN_ERR to report hardware difficulties.
        iv. **KERN_WARNING:**
            1. Warnings about problematic situations that do not, in themselves, create serious problems with the system.
        v. **KERN_INFO:**
            1. Informational messages. Many drivers print information about the hardware they find at startup time at this level.
    d. So in my case I used the kern_info because I want to print the information “Hello World”.
        i. printk(KERN_INFO "Hello World,This is Simple Module");

3. **Code:**
    a. Following is the code for the Hello Module.

vi.

hello_module.c

```c
/**************************************************************************//**
*  \file       hello_module.c
*
*  \details    Simple hello world driver
*
*  \author     Jawad Ahmed
*
* ***************************************************************************/
#include <linux/module.h>    /* Needed by all modules */
#include <linux/kernel.h>    /* Needed for KERN_INFO */
#include <linux/init.h>  /* Needed for the macros */

MODULE_LICENSE("GNU");
MODULE_AUTHOR("Jawad Ahmed <jawad2019@namal.edu.pk>");
MODULE_DESCRIPTION("A simple hello world module for linux operating system");
MODULE_VERSION("1.0");
/*
** Module Init function
*/
static int __init hello_world_init(void)
{
    printk(KERN_INFO "Welcome to Namal\n");
    printk(KERN_INFO "Hello World This is the Simple Module\n");
    printk(KERN_INFO "Kernel Module Inserted Successfully...\n");
    return 0;
}

/*
** Module Exit function
*/
static void __exit hello_world_exit(void)
{
    printk(KERN_INFO "Kernel Module Removed Successfully...\n");
}

module_init(hello_world_init);
module_exit(hello_world_exit);
```

1. **Compiling the module:**
    a. As I have a C Code, now it's time to compile it and create the module hello_world.ko.
    b. In order to do so first I create a **Makefile** for the module which shown below:

Open

Makefile
~/Desktop/final_hello_module

```
1 obj-m = hello_module.o
2 all:
3         make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
4 clean:
5         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

c. Now the C Code and the Makefile is ready, all I need to do is to invoke the sudo make command. It will be my first module **named hello_world.ko.**

2. **Loading The Module in Running Kernel:**
   a. To load a Kernel Module, I use the **insmod** command with root privileges.
   b. For example, my module file name is **hello_module.ko** so i write the following command:
      i. **sudo insmod hello_module.ko**
3. **Check out The Module:**
   a. In order to check that the module is inserted or not, I use the following command:
      i. **Lsmod,** it give the below result:
      ii.
```
jawad@jawad-ThinkPad-T430s:~$ lsmod
Module                  Size  Used by
hello_module           16384  0
```
         1. Which means the module is successfully inserted.
      iii. For more details I use the command **dmesg** which give following result:
   b.
```
[11476.104321] hello_module: module license 'GNU' taints kernel.
[11476.104323] Disabling lock debugging due to kernel taint
[11476.104560] Welcome to Namal
[11476.104560] Hello World This is the Simple Module
[11476.104561] Kernel Module Inserted Successfully...
```
4. **Unloading The Module in Running Kernel:**
   a. As we load the module, in the same way we can unload it from the kernel through the following command:
      i. **sudo rmmod hello_module**
         1. It will remove the module from the kernel.
5. **Getting Module Details:**
   a. In order to get information about a Module (author, supported options), I use the **modinfo** command. Which give the following result:

```
jawad@jawad-ThinkPad-T430s:~/Desktop/final_hello_module$ modinfo hello_module.ko
filename:       /home/jawad/Desktop/final_hello_module/hello_module.ko
version:        1.0
description:    A simple hello world module for linux operating system
author:         Jawad Ahmed <jawad2019@namal.edu.pk>
license:        GNU
srcversion:     14F6FCCA1B5F149DE81A495
depends:
retpoline:      Y
name:           hello_module
vermagic:       5.4.0-91-generic SMP mod_unload modversions
jawad@jawad-ThinkPad-T430s:~/Desktop/final_hello_module$
```

---

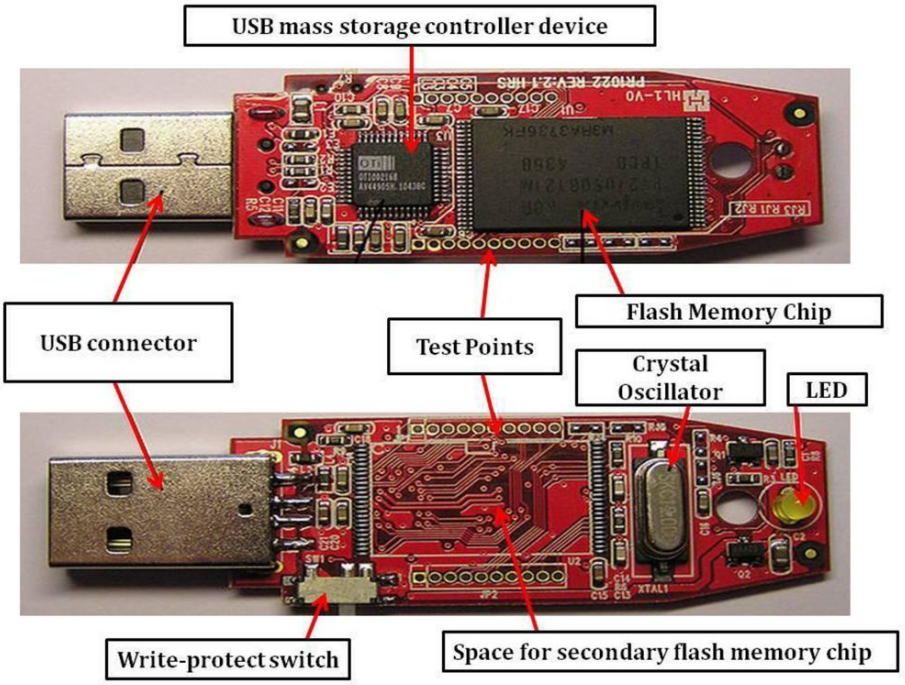
03. USB Device Driver:

a. Before going for the software part let's take a look at the usb hardware part.

b. The overview of USB is following:

i. **Overview:**

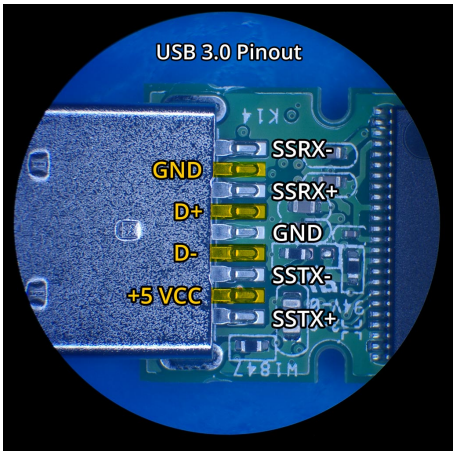
1. A USB flash drive is a device used for data storage that includes a flash memory and an integrated Universal Serial Bus (USB) interface.
2. Most USB flash drives are removable and rewritable.
3. Physically, they are small, durable and reliable.
4. They derive the power to operate from the device to which they are connected (typically a computer) via the USB port.
5. The internal structure of usb flash drive is below:



a.

b. But I am interested in the USB connector part because it is a way to connect the usb device with the computer usb port for which I am writing the driver. The USB connector consists of 4 pins: Ground(GND), D+, D- and +5 VCC, which is shown below:

i.

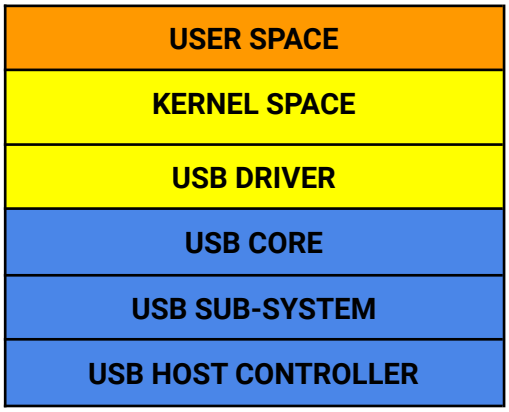


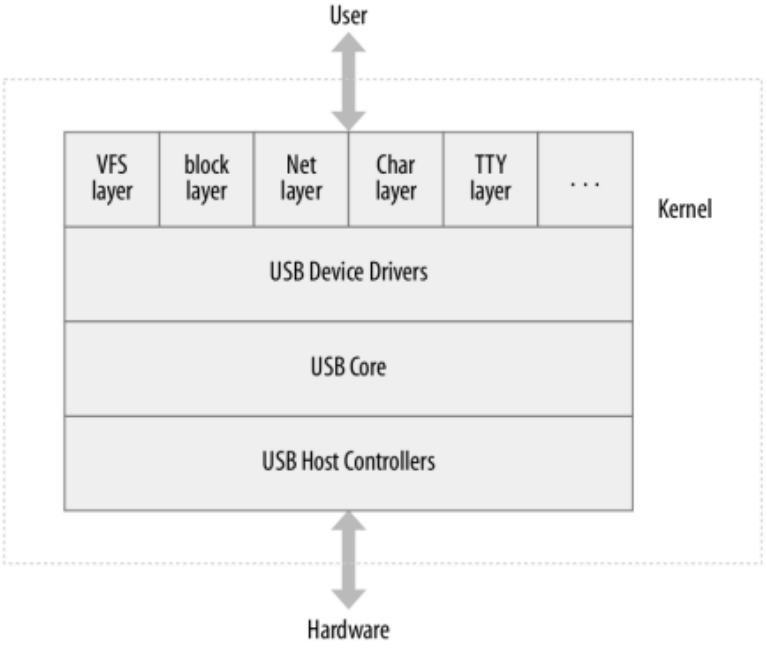
ii. The D+ and D- pins are used to transfer data between host and device. And the other 2 pins are used for the power.

c. USB Device Driver Basics:

i. Introduction:

1. At the early stages of the Linux kernel, the USB subsystem was supporting only mice and keyboards.
2. Later on it continues to grow. Right now it is supporting plenty of USB devices.
3. Before writing the USB device driver, we should discuss how the USB devices are connected to the Linux system and what is happening while plugging into the system. For that, we need to understand the USB Subsystem in Linux.
4. The USB subsystem consists of the **USB core, the USB Host Controller, and the USB host controller's driver.**





5.

ii. USB core:

1. USB core is a codebase consisting of routines and structures available to HCDs (Host Controller Driver) and USB drivers.

iii. USB HOST CONTROLLER:

1. The USB host controller is used to control the communications between the host system and USB devices. This USB host controller contains both hardware and software parts.
    a. **Hardware Part:**
        i. The hardware part is used for detecting the connected or disconnected usb devices.
        ii. Provided power to connected USB devices.

    b. **Software Part:**
        i. The software part is also called a USB host controller driver. This will be used to load the correct USB device drivers, and manage the data transfer between the driver and USB host controller.
        ii. So this USB host controller is doing the initial transaction once the USB device is detected in the system. Here Initial transaction means, get the devices vendor id, get the devices product id, device type, etc. Once that is done, then the USB host controller driver will assign the appropriate USB device driver to the device.
        iii. There are multiple USB host controller interfaces available e.g
            1. Open Host Controller Interface (OHCI) – USB 1.X. eg: USB 1.1
            2. Universal Host Controller Interface (UHCI) – USB 1.X.
            3. Enhanced Host Controller Interface (EHCI) – USB 2.0.
            4. Extended Host Controller Interface (XHCI) – USB 3.0.
        iv. I am interested in the USB 3.0 Host Controller Interface. Because the flash drive supports 3.0 USB ports.
        v. USB Driver:
            1. This is the driver which I am going to write for the USB devices.
        vi. USB Descriptor:
            1. The USB device contains a number of descriptors that help to define what the device is capable of. I am discussing the below descriptors.
            2. Device Descriptor:
                a. USB devices can only have one device descriptor.
                b. The device descriptor includes information such as the Product and Vendor IDs.
                c. It is used to load the appropriate drivers.
            3. Configuration Descriptor:
                a. The configuration descriptor specifies values such as the amount of power this particular configuration uses if the device is self or bus-powered and the number of interfaces it has.

b. When a device is enumerated, the host reads the device descriptors and can make a decision of which configuration to enable.
c. A device can have more than one configuration, though it can enable only one configuration at a time.

4. **Interface Descriptor:**
   a. A device can have one or more interfaces.
   b. Each interface descriptor can have a number of endpoints and represents a functional unit belonging to a particular class.
5. **Endpoint Descriptor:**
   a. Each endpoint descriptor is used to specify the type of transfer, direction, polling interval, and maximum packet size for each endpoint.
   b. In other words, each endpoint is a source or sink of data.

d. USB DRIVER CODE:

i. USB driver_structure:

1. The USB driver needs to register itself with the Linux USB subsystem (USB core). So while registering we need to give 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. All of this information is passed to the USB subsystem via **usb_driver structure**.
2. Following snap shows how the code for usb_driver structure:
3.
```
//The structure needs to do is register with the linux subsystem
static struct usb_driver etx_usb_driver = {
    .name       = "Namal USB Driver",
    .probe      = etx_usb_probe,
    .disconnect = etx_usb_disconnect,
    .id_table   = etx_usb_table,
};
```
   a. **<name>:** The driver name should be unique among USB drivers, and should normally be the same as the module name.
   b. **<probe>:** The function needs to be called when a USB device is connected.
   c. **<disconnect>:** The function needs to be called when a USB device is disconnected.
   d. **<id_table>:** USB drivers use an ID table to support hotplugging.
4. **Id_table:**
   a. It holds a set of descriptors, and specialised data may be associated with each entry. That table is used by both user and kernel mode hotplugging support.
   b. The following code tells the hotplug scripts that this module supports a single device with a specific vendor and product ID:

5.
```
//usb_device_id provides a list of different types of USB devices that the driver supports
const struct usb_device_id etx_usb_table[] = {
    { USB_DEVICE( USB_VENDOR_ID, USB_PRODUCT_ID ) },    //Put USB device's Vendor and Product ID
    { } /* Terminating entry */
};
```

   a. In order to find the device vendor and product ID I used the command **usb-devices** in terminal which gives following result for my usb device:

6.
```
Bus=03 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  8 Spd=480 MxCh= 0
Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
Vendor=23a9 ProdID=ef18 Rev=01.00
Manufacturer=AI210
Product=Mass Storage
#Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA
If#=0x0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=usb-storage
```

7. **Probe:**
   a. When a device is plugged into the USB bus that matches the device ID pattern then the driver registered with the USB core, the probe function is called.
   b. Following is the snap of it:

8.
```
/*
** This function will be called when USB device is inserted.
*/
static int etx_usb_probe(struct usb_interface *interface,
                         const struct usb_device_id *id)
{
    unsigned int i;
    unsigned int endpoints_count;
    struct usb_host_interface *iface_desc = interface->cur_altsetting;

    dev_info(&interface->dev, "USB Driver Probed: Vendor ID : 0x%02x,\t"
             "Product ID : 0x%02x\n", id->idVendor, id->idProduct);

    endpoints_count = iface_desc->desc.bNumEndpoints;

    PRINT_USB_INTERFACE_DESCRIPTOR(iface_desc->desc);

     for ( i = 0; i < endpoints_count; i++ ) {
          PRINT_USB_ENDPOINT_DESCRIPTOR(iface_desc->endpoint[i].desc);
     }
    return 0;  //return 0 indicates we are managing this device
}
```

9. **Disconnect:**
   a. When a device is plugged out or removed, this function will be getting called.

10.
```
/*
** This function will be called when USB device is removed.
*/
static void etx_usb_disconnect(struct usb_interface *interface)
{
    dev_info(&interface->dev, "USB Driver Disconnected\n");
}
```

ii. Register the USB device driver to the USB Subsystem (USB core):

1. The following API is used for this purpose:
   a. **usb_register (struct usb_driver * your_usb_driver);**

2.
```
//register the USB device
static int __init etx_usb_init(void)
{

    return usb_register(&etx_usb_driver);
}
```

iii. Deregister the USB device driver to the USB Subsystem (USB core):

1. The following API is used for this purpose:
    a. **usb_deregister (struct usb_driver * your_usb_driver);**

2.
```
//deregister the USB device
static void __exit etx_usb_exit(void)
{

    usb_deregister(&etx_usb_driver);
}
```

iv. Initialise and exit function:

1. Unitil now I completed all the requirements of the usb driver or module. Now it's time to initialise it, following commands are used for it:
    a. **static int __init etx_usb_init(void)**
2. Following command used for exit function:
    a. **static void __exit etx_usb_exit(void)**

3.
```
module_init(etx_usb_init);
module_exit(etx_usb_exit);
```

v. Execution:

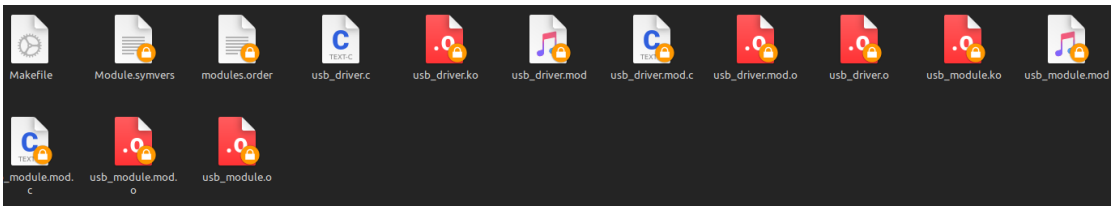
1. **Makefile:**
    a. It's time to build the driver by using Makefile:

b.
```
obj-m += usb_driver.o

KDIR = /lib/modules/$(shell uname -r)/build

all:
        make -C $(KDIR)  M=$(shell pwd) modules

clean:
        make -C $(KDIR)  M=$(shell pwd) clean
```

2. Now by writing **sudo make** in the terminal I got the following files generated:
3. 

4. Where the loadable kernel module (usb_driver.ko) is used to insert in the module which is an object file that is used to extend the kernel of the Linux Distribution.

5. **Load the driver:**
    a. The command is:
        i. **sudo insmod usb_driver.ko**
        ii. I got following message by writing the **dmesg**:
    b. 
```
55081.756994] wlp3s0: associated
55151.988424] usbcore: registered new interface driver Namal USB Driver
awad@jawad-ThinkPad-T430s:~/Desktop/usb_module/attemp1$
```
    c. Which means the usb driver is successfully loaded in the kernel.

vi. **Testing:**

1. The driver is linked with the USB subsystem.
2. Now connect the USB device with the correct vendor id and product id. Now it's time to see if the etx_usb_probe function is getting called or not.
3. Check the **dmesg**
4. Hence the module is working fine and the following output is shown:

```
[61099.192036] usb 3-2: new high-speed USB device number 14 using xhci_hcd
[61099.340799] usb 3-2: New USB device found, idVendor=23a9, idProduct=ef18, bcdDevice= 1.00
[61099.340804] usb 3-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[61099.340806] usb 3-2: Product: Mass Storage
[61099.340809] usb 3-2: Manufacturer: AI210
[61099.341650] Namal USB Driver 3-2:1.0: USB Driver Probed: Vendor ID : 0x23a9, Product ID : 0xef18
[61099.341654] USB_INTERFACE_DESCRIPTOR:
[61099.341656] ----------------------------
[61099.341658] bLength: 0x9
[61099.341660] bDescriptorType: 0x4
[61099.341661] bInterfaceNumber: 0x0
[61099.341664] bAlternateSetting: 0x0
[61099.341665] bNumEndpoints: 0x2
[61099.341666] bInterfaceClass: 0x8
[61099.341667] bInterfaceSubClass: 0x6
[61099.341670] bInterfaceProtocol: 0x50
[61099.341671] iInterface: 0x0

[61099.341673] USB_ENDPOINT_DESCRIPTOR:
[61099.341675] ------------------------
[61099.341676] bLength: 0x7
[61099.341678] bDescriptorType: 0x5
[61099.341679] bEndPointAddress: 0x2
[61099.341681] bmAttributes: 0x2
[61099.341683] wMaxPacketSize: 0x200
[61099.341685] bInterval: 0x0
```

Usb Driver Version 2.0:

1. **Purpose:**
    a. **Modify the previous version in order to read and write the characters.**
    b. There are two groups of device files:
        i. Character files
            1. It is non buffered files that allow us to read and write data character by character.
        ii. Block files
            1. It is buffered files that allow us to read and write whole blocks of data.
    c. Now in order to register character device, I use **register_chrdev** function as bellow:
2.
```
int result = 0;

printk( KERN_NOTICE "Namal-USB-Driver: Is Successfully Registered.\n" );

result = register_chrdev( 0, device_name, &simple_driver_fops );
if( result < 0 )
{
    printk( KERN_WARNING "Namal-USB-Driver:  can\'t register character device with errorcode = %i\n", result );
    return result;
}

device_file_major_number = result;
printk( KERN_NOTICE "Namal-USB-Driver: registered character device with major number = %i and minor numbers 0...255\n"
    , device_file_major_number );
```
3. **File_operations:**
    a. The file_operations structure contains pointers to functions defined by the driver. Each field of the structure corresponds to the address of a function defined by the driver to handle a requested operation.
    b.
```
static struct file_operations simple_driver_fops =
{
    .owner = THIS_MODULE,
    .read = device_file_read,
};
```
4. **Read function:**
    a. The read function I write will read characters from a device. It is used in the file_operation function.

5.
```
static const char g_s_Hello_World_string[] = "Hello world from kernel mode!\n\0";
static const ssize_t g_s_Hello_World_size = sizeof(g_s_Hello_World_string);

/*=============================================================================================*/
static ssize_t device_file_read(
    struct file *file_ptr
    , char __user *user_buffer
    , size_t count
    , loff_t *possition)
{
    printk( KERN_NOTICE "Namal-USB-Driver: Device file is read at offset = %i, read bytes count = %u\n"
        , (int)*possition
        , (unsigned int)count );

    if( *possition >= g_s_Hello_World_size )
        return 0;

    if( *possition + count > g_s_Hello_World_size )
        count = g_s_Hello_World_size - *possition;

    if( copy_to_user(user_buffer, g_s_Hello_World_string + *possition, count) != 0 )
        return -EFAULT;

    *possition += count;
    return count;
}
```

6. **Load the module:**
    a. Load the module as the previous usb driver loaded.
7. **Character devices:**
    a. In order to show the character devices following command is used:
        i. **sed -n '/^Character/, /^$/ { /^$/ !p }' /proc/devices**

8.
```
jawad@jawad-ThinkPad-T430s:~$ sed -n '/^Character/, /^$/ { /^$/ !p }' /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 89 i2c
 99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
```

9. **Device file:**
    a. The /dev directory contains **the special device files for all the devices**. The device files are created during installation, and later with the /**dev/MAKEDEV** command.
    b. If I want to put device files in a specific folder, I use the command **mknod.**
    c. 

Conclusion:

- USB devices are complex, they consist of several logical units: one or more configurations, configurations have one or more interfaces, interfaces have one or more sets of settings, interfaces have zero or more endpoints.
- I tried my best and almost successfully built the usb driver. It worked fine. But little bit issues occur while reading the characters in the usb.
- I learned the basic level of how to make and load the linux module.
