VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Device Driver Programming (Using USB)

PROJECT REVIEW REPORT

Submitted by M.J.Prajeeth kumar
(18BCE2354)
Shravan.A.J(18BCE2399)

Prepared For

OPERATING SYSTEMS (CSE2005) – PROJECT COMPONENT

Submiited to

Prof. Vijayasherly V

Introduction

A program that controls a selected form of device that could be attached to our computer is a driver. There are device drivers for printers, displays, read-only storage readers, floppy disk drives,and so on.A drivercould bea software system element that lets the software system and a device communicate with one another. A driver communicates with the device to that the hardware connects through the pc bus. The driver problems commands to the device, once the job program interrupts a routine within the driver. Once the device sends information back to the driver, the driver could invoke routines within the original calling program.

The purpose of a device driver is to handle requests created by the kernel with reference to a specific form of device. There’s a well-defined and consistent interface for the kernel to form these requests. By using analytic device-specific code in device drivers and by having a uniform interface to the kernel, adding a replacement device is simpler. a device driver could be a software system module that resides among the Digital UNIX system kernel and is that the code interface to a hardware device or devices. A hardware device could be a peripheral, like a controller, tape controller, or network controller device. In general, there's one driver for every form of hardware device.

The importance the device driver can be seen by the functions of a driver that is encapsulation which hides low-level device protocol details from the client unification makes similar devices look the same protection with the cooperation of OS only authorized applications can use device Multiplexing such as the Multiple applications can use the device concurrently. Device drivers isolate low-level, device-specific details from the system calls, which can remain general and uncomplicated. Because each device differs so, kernels cannot practically handle all the possibilities. Instead, each configured device plugs a device driver into the kernel. To add a new device or capability to the system, just plug in its driver.

Problem Statement-

Aims to implement USB pen drive for the working of modules and kernels and how the modules are loaded into the kernel for the execution of the code.

It also aims at the registration and deregistration of the device in the UNIX system.

Abstract

As a device driver program our project aims at implementing USB Pen Drive for the working of modules and kernels and howthe modules are loaded in kernelfor execution of the code. It also aims at the registration and deregistration of the device in the UNIX system. The drawback with the general device driver is that it often manages an entire set of identical device controller interfaces. But with the digital UNIX we can statistically configure more device drivers into the kernel. For the implementation, a directory will be made which will have 2 files that are kernel and module file and the coding will be done in C language. The reason for using such technique is that when we isolate a device specific code in device drivers and then by having a consistent interface to the kernel, adding a new device gets easier. Since UNIX is a monolithic kernel, so kernel and module form are to be implemented together to avoid recompiling of the kernel when the driver is added. As a result, when a pen drive is connected to device driver and as soon as we remove it, it will be disconnected. So, when USB pen drive and the device are registered properly then we would conclude that our kernel and module have been properly implemented. The main advantage of such model is that when a USB controller module is updated it does not require a complete recompilation, which as a result reduces the errors. Hence, adding a new device only need a new module driver rather than a new kernel, which offers much flexibility.

LITERATURE SURVEY:

In Linux, a device driver is code that implements a user-space or kernel-space abstraction of a physical device.

Device drivers come in several different types, depending on what abstraction they provide. Serial port drivers, for example, often implement the character device type. The frame-buffer driver type is normally used to enable user-space applications to write to an LCD or CRT display. Ethernet drivers allow the Linux kernel's TCP/IP protocol stack to send packets over an Ethernet network.

A device driver abstraction is purely that, an abstraction. The underlying hardware associated with an Ethernet device driver may not actually be an Ethernet controller, for example; there may not even be any physical hardware at all! Such is the case for the USB-net driver, which allows a USB Device to communicate with a USB Host as though the two were connected via Ethernet. (The USB-net driver packages Ethernet packets and forwards them to the USB host/device drivers for transmission).

For many device driver types, user applications communicate with the driver via a pseudofile called a device node. Device nodes look like files, and applications can even open() and close() them. But when data is written to a device node, the data is passed to the node's associated driver, and not stored in a filesystem.

A block device driver implements an abstraction commonly associated with hard drives and other data-block-oriented media. A block device offers persistent storage, which (generally) allows applications to "seek" data within the device. Properly-implemented block device drivers can be controlled by the kernel's Virtual File System functionality, allowing all manner of devices to be used to store nearly all kernel-supported file systems with little additional effort.

Frame-buffer device drivers provide for direct user-space access to video frame buffers: the memory space used by an LCD controller to store the image actually visible on the LCD panel. XFree86, Qtopia, Micro-windows and other GUI libraries interact directly with frame buffer memory. Frame-buffer drivers are mostly informational; they provide standardized interfaces that allow applications to set and query video modes and refresh rates. User applications use the frame-buffer driver's mmap() system call to locate frame buffer memory.

Ethernet drivers provide an Ethernet-specific abstraction that focuses almost exclusively on data exchange with the network media access controller (MAC). Other network-related functions, like IP packet assembly and decoding, are handled within the kernel's network protocol stacks and do not directly influence the implementation of an Ethernet device driver.

GAPS IDENTIFIED:

Linux is a monolithic operating system, consequently making it rapid and easy to make driver packages being robust and additionally makes it appropriate for mistake-detection of modules which if not loaded properly, doesn’t modify or impact the gadget’s overall performance or working. Also, for every module, an object report is created mentioning the goal and listing for itself.

Overview of the proposed system

Preliminary design

The role of a driver is to provide mechanisms which allows normal user to access protected parts of its system, in particular ports, registers and memory addresses normally managed by the operating system. One of the good features of Linux is the ability to extend at run time the set of the features offered by the kernel. Users can add or remove functionalities to the kernel while the system is running. These “programs” that can be added to the kernel at run time are called “module” and built into individual files with .ko (Kernel object) extension. The Linux kernel takes advantages of the possibility to write kernel drivers as modules which can be uploaded on request. This method has different advantages:

• The kernel can be highly modularized, in order to be compatible with the most possible hardware.

• A kernel module can be modified without need of recompiling the full kernel.

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. Hardware-space detection is done by the USB host controller — typically a native bus device, like a PCI device on x86 systems. 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.
After this, it is up to various drivers, interfaces, and applications (which are dependent on the various Linux distributions), to have the user-space view of the detected devices.

Frameworks, Architecture and Modules of the proposed system-

The project involves the following modules-

- Module
- Drive Insertion
- Authentication
- Kernel Code
- Display of Pen drive Details
- Registration and De-registration

Module

A module is an object file prepared in a special way. The Linux kernel can load a module to its address space and link the module with itself. The Linux kernel is written in 2 languages: C and assembler (the architecture dependent parts). The development of drivers for Linux OS is possible only in C and assembler languages, but not in C++ language (as for the Microsoft Windows kernel). It is connected with the fact that the kernel source pieces of code, namely, header files, can contain C++ key words such as new, delete and the assembler pieces of code can contain the ‘::’ lexeme.
The module code is executed in the kernel context. It rests some additional responsibility in

the developer: if there is an error in the user level program, the results of this error will affect mainly the user program; if an error occurs in the kernel module, it may affect the whole system. But one of the specifics of the Linux kernel is a rather high resistance to errors in the modules’ code. If there is a noncritical error in a module (such as the dereferencing of the null pointer), the oops message will be displayed (oops is a deviation from the normal work of Linux and in this case, the kernel creates a log record with the error description). Then, the module, in which the error appeared, is unloaded, while the kernel itself and the rest of modules continue working. However, after the oops message, the system kernel can often be in an inconsistent state and the further work may lead to the kernel panic.

Drive Insertion

The insertion of the pen drive into the module is done with the help of the module file. The module is present in the directory of the laptop and as well as the pen drive. The insertion is done with the help of the command ‘lsusb’ . It will show the pen drive connected to which port and it will help in understanding the different port details of the laptop or to the system into which the program is installed into and displaying port number and to check whether the pen drive is properly connected to the port and the port number is assigned into the program for accessing the details of the pen drive.

Authentication

The authentication of accessing the details of the pen drive is done by the command “sudo su”. This is done in order to protect the internal details of the pen drive. The Authentication plays a very important role in prevention from misuse and it blocks such person. This will keep a check of number of users can keep the details of the USB pen drive and access it.

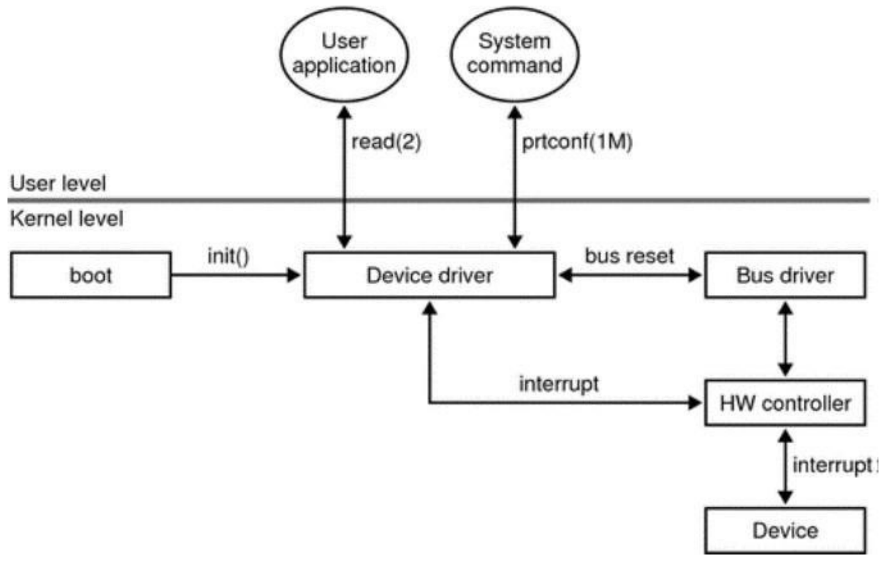
Kernel

The kernel and its modules are built into a practically single program module. That is why it is worth remembering that within one program module, one global name space is used. To clutter up the global name space minimally, one should monitor that the module exports only the necessary minimum of global characters and that all exported global characters have the unique names (the good practice is to add the name of the module, which exports the character, to the name of the character as a prefix). Here we are using make function to implement the details of kernel.

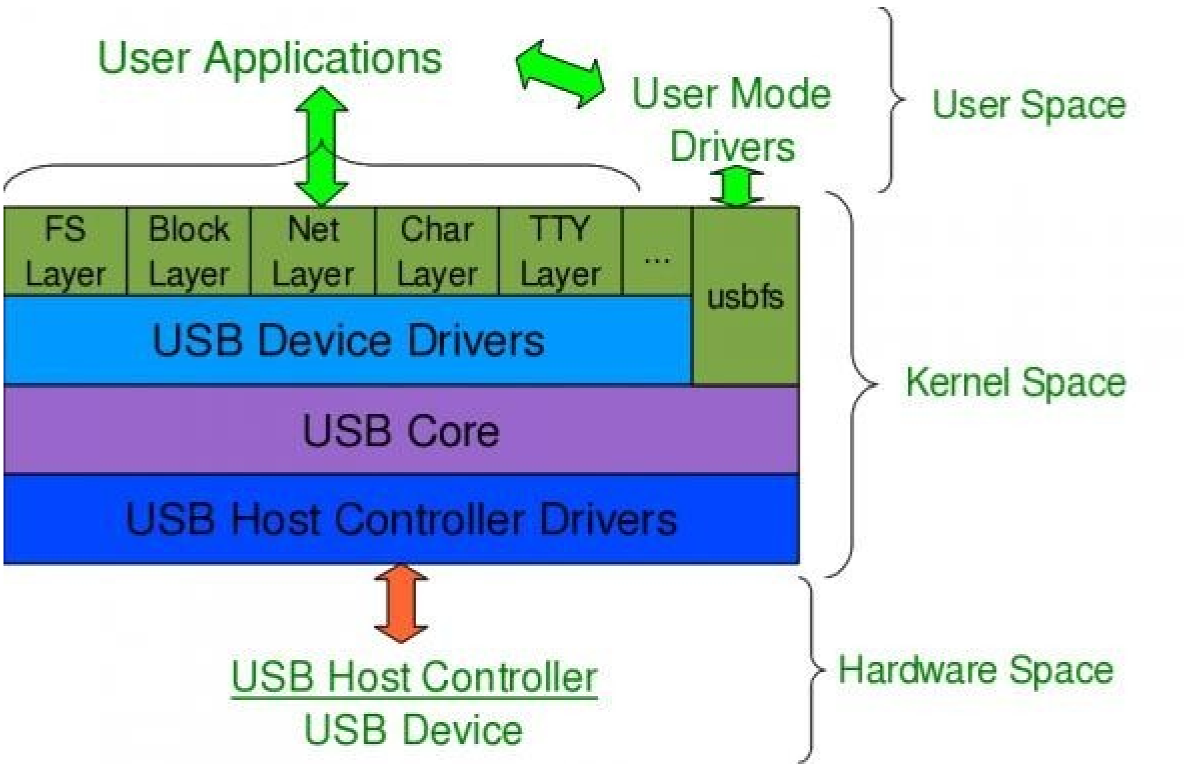
Display of pen drive Details

The details of the USB pen drive are displayed in a sequential fashion. Showing the things needed to be displayed like vendor ID, product ID, manufacture ID etc. Thus the internal details was able to be accessed by the USB drive which is the work of the drive is accomplished. The command used is ‘lsusb -v’. to get the details.

Proposed System Model-



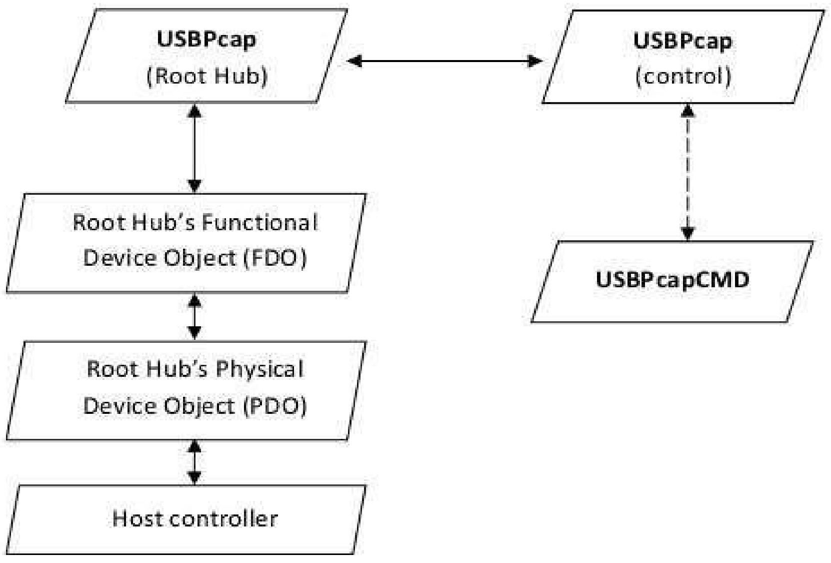
How a USB interacts with the OS-



PROPOSED SYSTEM ANALYSIS AND DESIGN-

To perform some useful functions in the device, processes need to be accessed to the peripherals connected to the computer, that are thus controlled by the kernels through device drivers.

A device driver stands for the computer program that enables the operating system to interact and implement with a hardware device. It provides the operating system with the details of how to control, communicate and contact with a certain piece of hardware. As device drivers are thus handled differently by each kind of the kernel design, but in each and every case, the kernel has to provide the I/O to allow the drivers to physically access their devices through some port in the device or memory location. The block diagram is represented as follows:


USBPcap (Root Hub)
USBPcap (control)
Root Hub's Functional Device Object (FDO)
USBPcapCMD
Root Hub's Physical Device Object (PDO)
Host controller

The implementation of USB PEN DRIVE device driver theme is to gain the practical knowledge of working of modules and kernels, where we learn how the module is loaded in kernel for the execution of the code. In this project, we aim to implement support for loadable kernel modules. An ideal set of tools in Linux for supporting the kernel module include some additional OS facilities for inserting a module, removing a module and thus stating the currently loaded modules. Some other slightly advanced features include about determining the dependencies among the modules or loading or unloading the kernel modules based on its dependency with the other modules.

System calls from user processes. The kernel calls a device driver to perform the I/O operations on the device such as the open (2), read(2), and ioctl (2) functions

User-level requests. The kernel calls the device drivers to send requests from the commands such as the prtconf(1M).

Device interrupts. The kernel calls a device driver to handle the interrupts generated by a certain device.

Bus reset. The kernel calls the device driver to perform the following operations like re-initializing the driver or the device or the both when the bus is reset.

<u>Implementation-</u>

<u>Source code-</u>

Stick-driver program-

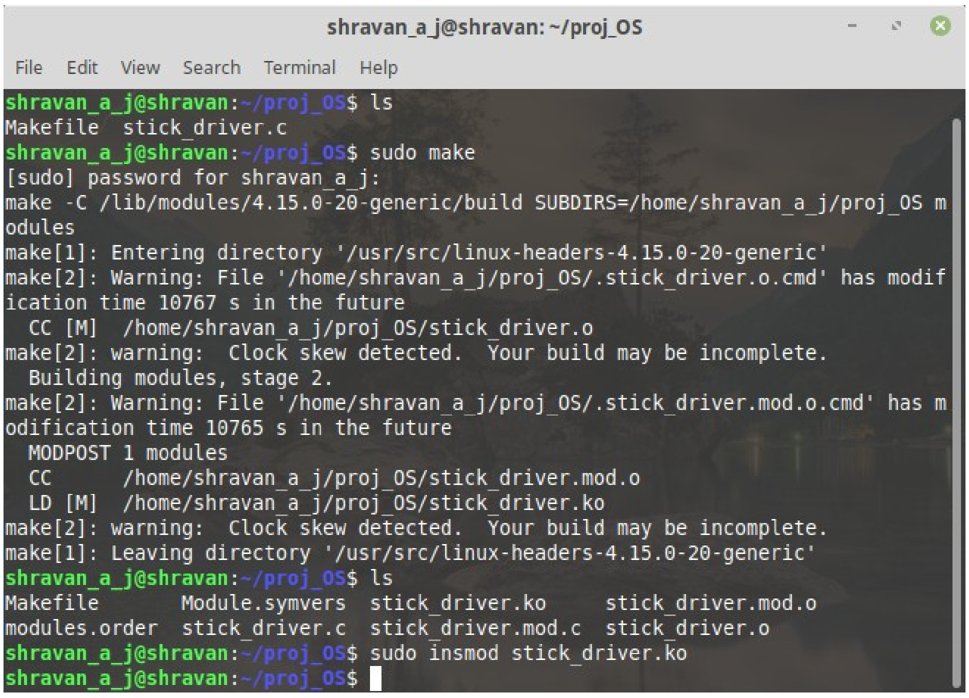
```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/usb.h>

//probe function
static int pen_probe(struct usb_interface *interface, const struct usb_device_id *id)
{
   printk(KERN_INFO "[*] Shravan's Pen drive (%04x:%04x)plugged\n",id->idVendor,id->idProduct);
   return 0;
}

static void pen_disconnect(struct usb_interface *interface){
   printk(KERN_INFO "[*] Shravan's Pen drive removed\n");
}

 //usb_device_id
static struct usb_device_id pen_table[]={
   //0781:558a
   {USB_DEVICE(0x0781,0x558a)},
   {} /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb,pen_table);

 //usb driver
static struct usb_driver pen_driver=
{
   .name="Shravan's USB Stick-Driver",
   .id_table=pen_table,   //usb_device_id
   .probe=pen_probe,
   .disconnect=pen_disconnect,
};

static int __init pen_init(void){
   int ret=-1;
   printk(KERN_INFO "[*]Constructor of USB driver");
   printk(KERN_INFO "\tRegistering Driver with Kernel");
   ret=usb_register(&pen_driver);
   printk(KERN_INFO "\tRegistration is complete");
   return ret;
}

static void __exit pen_exit(void){
   //deregister
```

```
    printk(KERN_INFO "[*]Destructor of USB driver");
    usb_deregister(&pen_driver);
    printk(KERN_INFO "\tDeregistration complete!");
}

module_init(pen_init);
module_exit(pen_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Shravan and Prajeeth");
MODULE_DESCRIPTION("USB Pen Registration Driver");
```

Makefile-

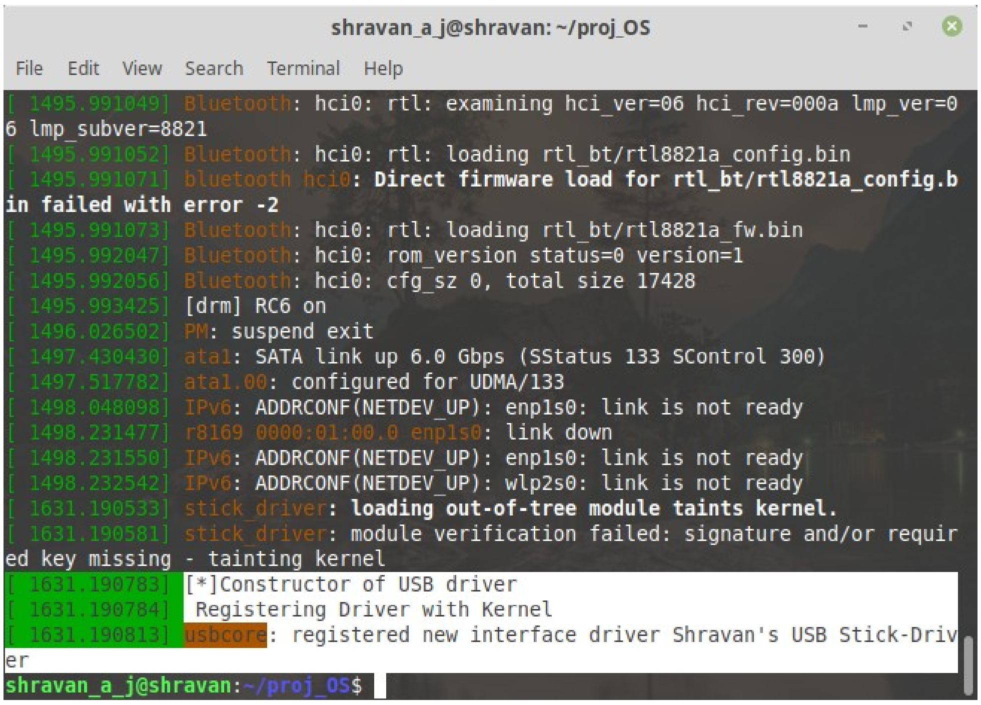
```
obj-m := stick_driver.o

KERNEL_DIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell pwd)
all:
	$(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD) modules

clean:
	rm -rf *.o *.ko *.mod *.symvers *.order *~
```

Outputs-


shravan_a_j@shravan: ~/proj_OS
File Edit View Search Terminal Help

```
shravan_a_j@shravan:~/proj_OS$ ls
Makefile  stick_driver.c
shravan_a_j@shravan:~/proj_OS$ sudo make
[sudo] password for shravan_a_j:
make -C /lib/modules/4.15.0-20-generic/build SUBDIRS=/home/shravan_a_j/proj_OS m
odules
make[1]: Entering directory '/usr/src/linux-headers-4.15.0-20-generic'
make[2]: Warning: File '/home/shravan_a_j/proj_OS/.stick_driver.o.cmd' has modif
ication time 10767 s in the future
  CC [M]  /home/shravan_a_j/proj_OS/stick_driver.o
make[2]: warning:  Clock skew detected.  Your build may be incomplete.
  Building modules, stage 2.
make[2]: Warning: File '/home/shravan_a_j/proj_OS/.stick_driver.mod.o.cmd' has m
odification time 10765 s in the future
  MODPOST 1 modules
  CC      /home/shravan_a_j/proj_OS/stick_driver.mod.o
  LD [M]  /home/shravan_a_j/proj_OS/stick_driver.ko
make[2]: warning:  Clock skew detected.  Your build may be incomplete.
make[1]: Leaving directory '/usr/src/linux-headers-4.15.0-20-generic'
shravan_a_j@shravan:~/proj_OS$ ls
Makefile       Module.symvers  stick_driver.ko     stick_driver.mod.o
modules.order  stick_driver.c  stick_driver.mod.c  stick_driver.o
shravan_a_j@shravan:~/proj_OS$ sudo insmod stick_driver.ko
shravan_a_j@shravan:~/proj_OS$
```

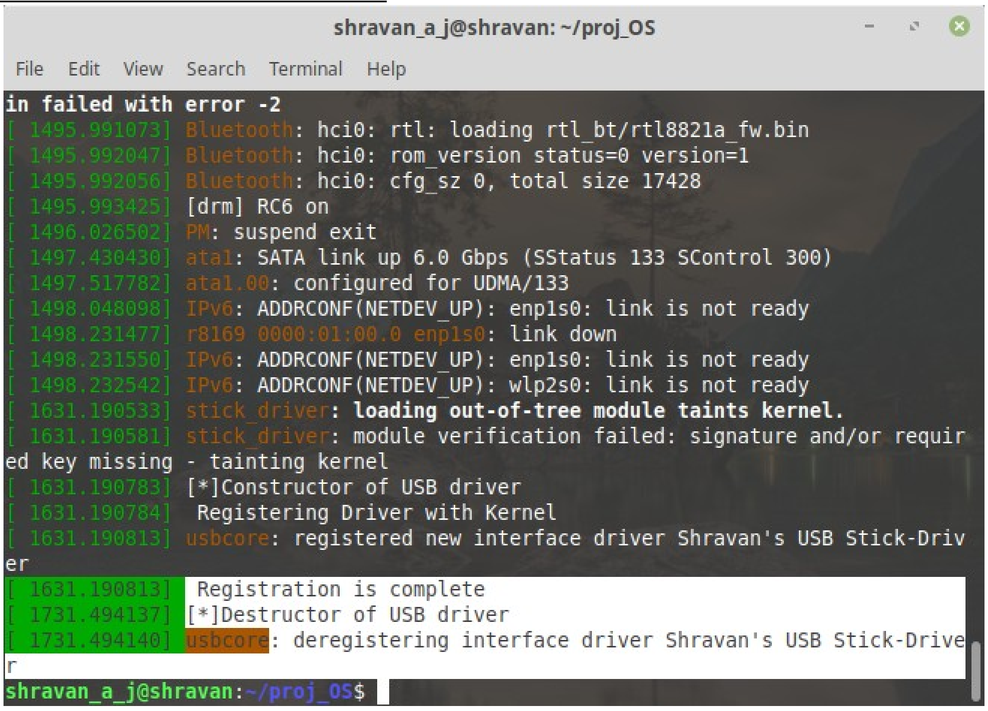
<u>When the module is inserted-</u>



shravan_a_j@shravan: ~/proj_OS
File Edit View Search Terminal Help

```
[ 1495.991049] Bluetooth: hci0: rtl: examining hci_ver=06 hci_rev=000a lmp_ver=0
6 lmp_subver=8821
[ 1495.991052] Bluetooth: hci0: rtl: loading rtl_bt/rtl8821a_config.bin
[ 1495.991071] bluetooth hci0: Direct firmware load for rtl_bt/rtl8821a_config.b
in failed with error -2
[ 1495.991073] Bluetooth: hci0: rtl: loading rtl_bt/rtl8821a_fw.bin
[ 1495.992047] Bluetooth: hci0: rom_version status=0 version=1
[ 1495.992056] Bluetooth: hci0: cfg_sz 0, total size 17428
[ 1495.993425] [drm] RC6 on
[ 1496.026502] PM: suspend exit
[ 1497.430430] ata1: SATA link up 6.0 Gbps (SStatus 133 SControl 300)
[ 1497.517782] ata1.00: configured for UDMA/133
[ 1498.048098] IPv6: ADDRCONF(NETDEV_UP): enp1s0: link is not ready
[ 1498.231477] r8169 0000:01:00.0 enp1s0: link down
[ 1498.231550] IPv6: ADDRCONF(NETDEV_UP): enp1s0: link is not ready
[ 1498.232542] IPv6: ADDRCONF(NETDEV_UP): wlp2s0: link is not ready
[ 1631.190533] stick_driver: loading out-of-tree module taints kernel.
[ 1631.190581] stick_driver: module verification failed: signature and/or requir
ed key missing - tainting kernel
[ 1631.190783] [*]Constructor of USB driver
[ 1631.190784]  Registering Driver with Kernel
[ 1631.190813] usbcore: registered new interface driver Shravan's USB Stick-Driv
er
shravan_a_j@shravan:~/proj_OS$
```

<u>When the module is removed-</u>

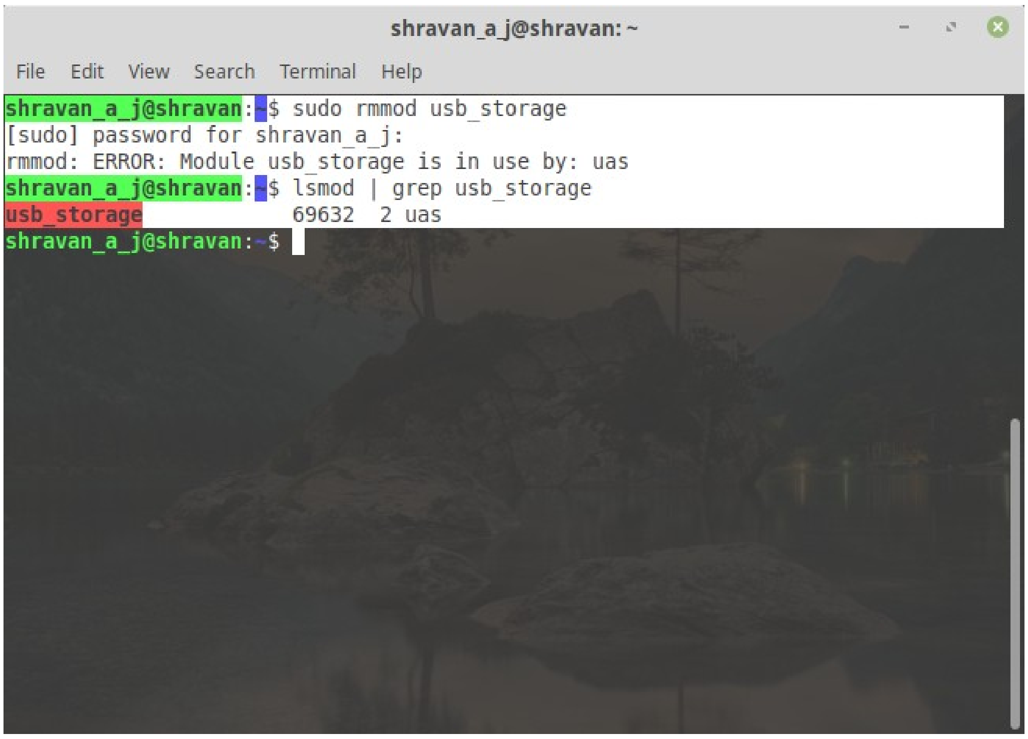


shravan_a_j@shravan: ~/proj_OS
File Edit View Search Terminal Help

```
in failed with error -2
[ 1495.991073] Bluetooth: hci0: rtl: loading rtl_bt/rtl8821a_fw.bin
[ 1495.992047] Bluetooth: hci0: rom_version status=0 version=1
[ 1495.992056] Bluetooth: hci0: cfg_sz 0, total size 17428
[ 1495.993425] [drm] RC6 on
[ 1496.026502] PM: suspend exit
[ 1497.430430] ata1: SATA link up 6.0 Gbps (SStatus 133 SControl 300)
[ 1497.517782] ata1.00: configured for UDMA/133
[ 1498.048098] IPv6: ADDRCONF(NETDEV_UP): enp1s0: link is not ready
[ 1498.231477] r8169 0000:01:00.0 enp1s0: link down
[ 1498.231550] IPv6: ADDRCONF(NETDEV_UP): enp1s0: link is not ready
[ 1498.232542] IPv6: ADDRCONF(NETDEV_UP): wlp2s0: link is not ready
[ 1631.190533] stick_driver: loading out-of-tree module taints kernel.
[ 1631.190581] stick_driver: module verification failed: signature and/or requir
ed key missing - tainting kernel
[ 1631.190783] [*]Constructor of USB driver
[ 1631.190784]  Registering Driver with Kernel
[ 1631.190813] usbcore: registered new interface driver Shravan's USB Stick-Driv
er
[ 1631.190813]  Registration is complete
[ 1731.494137] [*]Destructor of USB driver
[ 1731.494140] usbcore: deregistering interface driver Shravan's USB Stick-Drive
r
shravan_a_j@shravan:~/proj_OS$
```

Module ‘pen_probe’ could not be implemented since there is an already existing module called ‘usb_storage’ which already over-looks the same functions and cannot be removed.

<u>Following are its screen shots-</u>



```
shravan_a_j@shravan:~$ sudo rmmod usb_storage
[sudo] password for shravan_a_j:
rmmod: ERROR: Module usb_storage is in use by: uas
shravan_a_j@shravan:~$ lsmod | grep usb_storage
usb_storage            69632  2 uas
shravan_a_j@shravan:~$
```

Conclusion

Writing Linux USB device drivers is not a difficult task as the usb-skeleton driver shows. This driver, combined with the other current USB drivers, should provide enough examples to help a beginning author create a working driver in a minimal amount of time. The linux-usb-devel mailing list archives also contain a lot of helpful information.

We have implemented a code for usb device driver where we have shown registering of a USB pen drive in linux system. A directory was made which had 2 files that are Kernel and Module file. The module coding has been done in C language and Kernel helps in loading of module for device driver. When the pen drive is connected, it gets registered by the device driver and as soon as we remove it, it gets disconnected and the registration remains incomplete.

We conclude that our kernel and module have been properly implemented for the formation of device driver and usb pen drive is getting registered properly.

REFERENCES-

1. Linux Device Drivers, 3rd Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman: http://lwn.net/Kernel/LDD3/

2. The Linux Kernel Module Programming Guide by Peter Jay Salzman and Ori Pomeranz: http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

Linux Cross Reference http://lxr.free-electrons.com/ident

3.IOSR Journal of Electronics and Communication Engineering (IOSR-JECE) e-ISSN: 2278-2834,p- ISSN: 2278-8735.Volume 11, Issue 6, Ver. II (Nov.-Dec .2016), PP 46-52 www.iosrjournals.org DOI: 10.9790/2834-1106024652 www.iosrjournals.org

4. Coding for Pseudo Device by Linux Character Device Driver Navneet Kr. Pandey1 , Prof. Prabhakar Dubey2, and Saurabh Bhutani3 , 1 (M Tech Scholar Electronics Dept, RRIMT / Aktu, India) 2 (Prof. ECE,RRIMT/ Aktu,India) 3 (M Tech Scholar Electronics Dept, RRIMT / Aktu, India)

Tschudin, Peter Senna; Reveillere, Laurent; JIANG, Lingxiao; LO, David; and Lawall, Julia. Understanding the Genetic Makeup of Linux Device Drivers. (2013). 7th Workshop on Programming Languages and Operating Systems (PLOS). Research Collection School Of Information Systems.

5. “Linux Driver Tutorial: How to Write a Simple Linux Device Driver.” *Apriorit*, www.apriorit.com/dev- blog/195-simple-driver-for-linux-os.