Operating System
PROJECT-TOPIC

DEVICE DRIVERS For USB IN LINUX

Mentors

Ass. Prof Rajni Aeron, Prof Subrat K. Dash, Ass. Prof Sunil Kumar

Group Members:

Deepank Agarwal : Y13UC078

Nikita Agarwal : Y13UC177

Reema Issarani : Y13UC223

Shreya Arora : Y13UC267

INDEX :

1. Abstract
2. Introduction
3. 1st Module : USB Device Detection in Linux
4. 2nd Module : USB Endpoints and their types
5. 3rd Module : Data Transfer to and from USB Devices
6. References

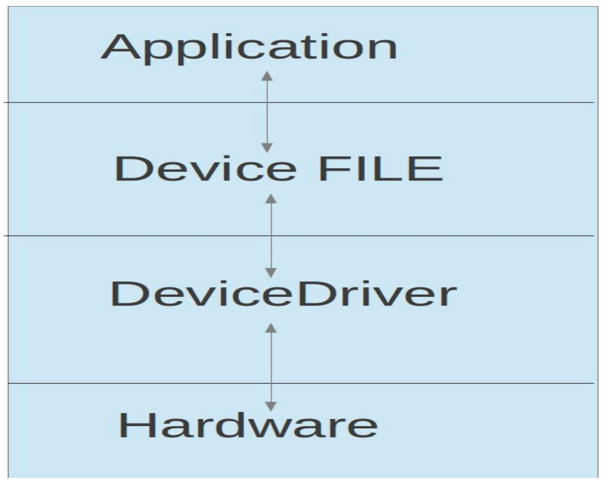
ABSTRACT :

A driver never runs by itself. It is similar to a library that is loaded for its functions to be invoked by a running application. It is written in C, but lacks a main() function. Moreover, it will be loaded/linked with the kernel, so it needs to be compiled in a similar way to the kernel, and the header files you can use are only those from the kernel sources, not from the standard /usr/include.

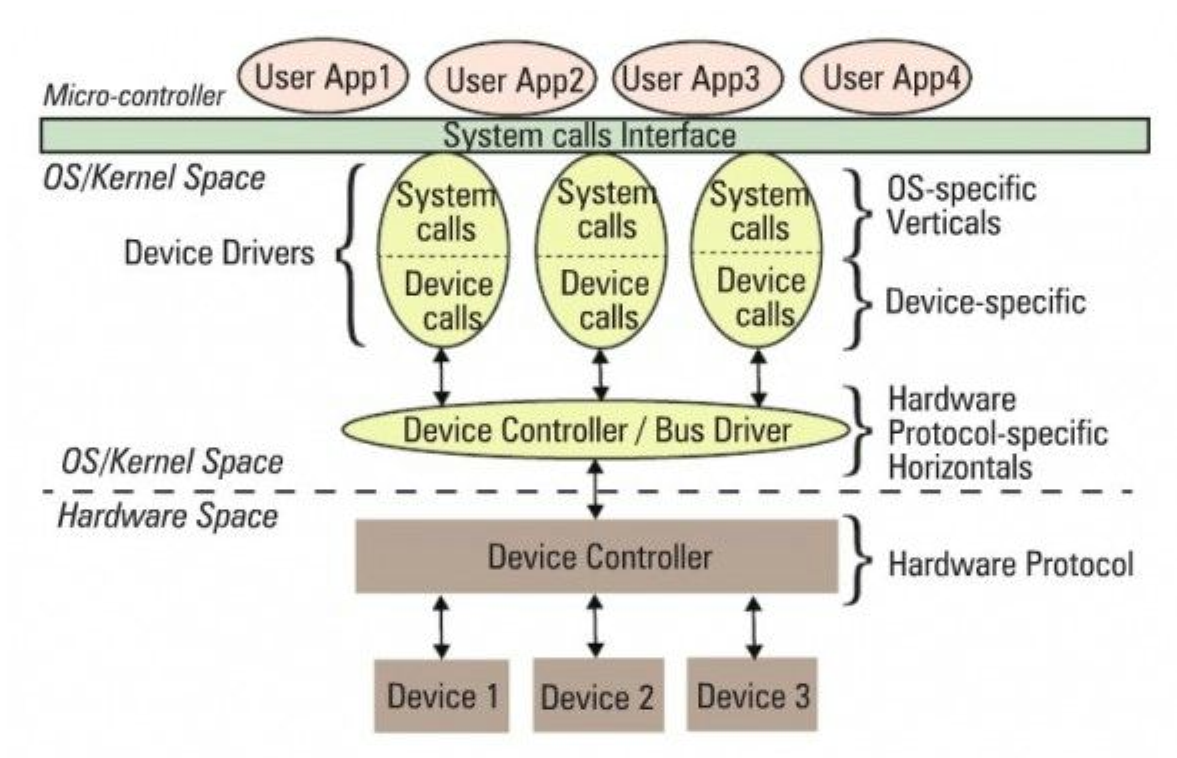
Kernel is that it is an object-oriented implementation in C, as we will observe even with our first driver. Any Linux driver has a constructor and a destructor. The module's constructor is called when the module is successfully loaded into the kernel, and the destructor when rmmod succeeds in unloading the module. These two are like normal functions in the driver, except that they are specified as the init and exit functions, respectively, by the macrosmodule_init() and module_exit(), which are defined in the kernel header module.h.

Introduction

A driver drives, manages, controls, directs and monitors the entity under its command.a specific piece of hardware could be controlled by a piece of software,known as device driver.Bus drivers provide hardware-specific interfaces for the corresponding hardware protocols, and are the bottom-most horizontal software layers of an operating system (OS). Over these sit the actual device drivers. These operate on the underlying devices using the horizontal layer interfaces, and hence are device-specific.



However, the whole idea of writing these drivers is to provide an abstraction to the user, and so, at the other “end”, these do provide an interface (which varies from OS to OS). In short, a device driver has two parts, which are: a) device-specific, and b) OS-specific.



Dynamically loadable drivers are more commonly called modules and built into individual files with a .ko (kernel object) extension. Every Linux system has a standard place under the root of the file system (/) for all the pre-built modules. They are organised similar to the kernel source tree structure, under /lib/modules/<kernel_version>/kernel.

__init and __exit

__init and __exit are not special keywords. Kernel C is just standard C with some additional extensions from the C compiler, GCC. Macros __init and __exit are just two of these extensions. However, these do not have any relevance in case we are using them for a dynamically loadable driver, but only when the same code gets built into the kernel. All functions marked with __init get placed inside the init section of the kernel image automatically, by GCC, during kernel compilation; and all functions marked with __exit are placed in the exit section of the kernel image.

the benefit of this is all functions with __init are supposed to be executed only once during bootup (and not executed again till the next bootup). So, once they are executed during bootup, the kernel frees up RAM by removing them (by freeing the init section). Similarly, all functions in the exit section are supposed to be called during system shutdown.

Now, if the system is shutting down anyway, why do you need to do any cleaning up? Hence, the exit section is not even loaded into the kernel

1ST MODULE: USB device detection in Linux

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. Figure 1 shows a top-to-bottom view of the USB subsystem in Linux.

So after plugging in the USB With **product id : 0951 and Vendor id : 1624**, uas and usb_storage gets loaded in the kernel space

```
nikita@nikita-pc:~$ lsusb
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 006: ID 0c45:64ad Microdia
Bus 001 Device 005: ID 0bda:0129 Realtek Semiconductor Corp. RTS5129 Card Reader
 Controller
Bus 001 Device 004: ID 04f3:0042 Elan Microelectronics Corp.
Bus 001 Device 007: ID 0cf3:e004 Atheros Communications, Inc.
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 002: ID 0951:1624 Kingston Technology DataTraveler G2
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
nikita@nikita-pc:~$ lsmod
Module                  Size  Used by
uas                    27255  0
usb_storage            66545  2 uas
ipt_MASQUERADE         12880  1
xt_conntrack           12760  1
ipt_REJECT             12541  2
xt_tcpudp              12884  4
iptable_filter         12810  1
```

so we removed these loaded modules using

1) sudo rmmod uas
2) sudo rmmod usb_storage

After this we need to make kernel module of our coded device driver.This is done by creating a MAKEFILE and using the command **make** as a super user on **pen.c**

This creates different file step by step till pen.ko is created.Files created are

1)Module.symvers

2)modules.order

3)pen.mod.c

4)pen.mod.o

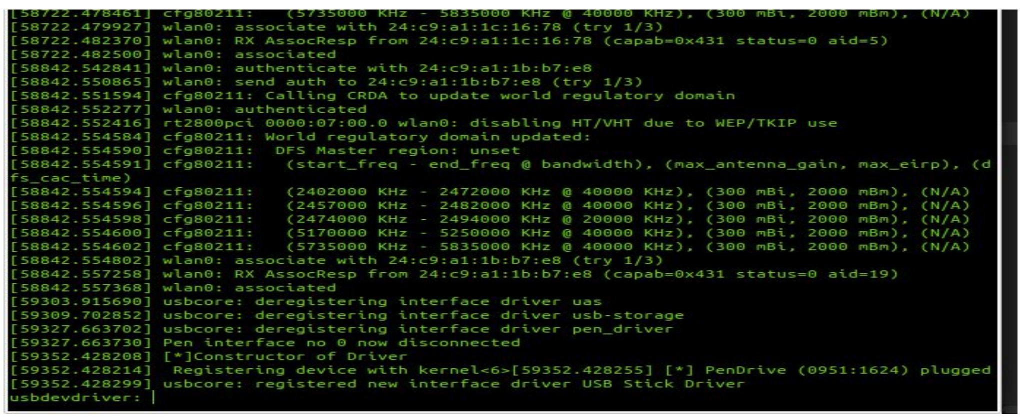
5)pen.o

6)pen.ko

After this we need to register pen.ko with the kernel so as when the USB is connected to the Linux system by using the command **insmod pen.ko as a super user** ,

Using dmesg ,will show the kernel logs

[*]Constructor of driver module is made incharge of it.



2^{ND} MODULE: USB endpoints and their types

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 the default control endpoint point as zero.
- **Interrupt** — for small and fast data transfers, typically of up to 8 bytes. Examples include data transfer for serial ports, human interface devices (HIDs) like keyboards, mouse, etc.
- **Bulk** — for big but comparatively slower data transfers. A typical example is data transfers for mass-storage devices.
- **Isochronous** — for big data transfers with a bandwidth guarantee, though data integrity may not be guaranteed. Typical practical usage examples include transfers of time-sensitive data like audio, video, etc.

Additionally, all but control endpoints could be “in” or “out”, indicating the direction of data transfer; “in” indicates data flow from the USB device to the host machine, and “out”, the other way.

Technically, an endpoint is identified using an 8-bit number, the most significant bit (MSB) of which indicates the direction — 0 means “out”, and 1 means “in”. Control endpoints are bidirectional, and the MSB is ignored.

with access to the struct usb_device handle for a specific device, all the USB-specific information about the device can be decoded,

- Build the driver (pen_info.ko file) by running make.
- Load the driver using insmod pen_info.ko.
- Plug in the pen drive (after making sure that the usb-storage driver is not already loaded).
- Unplug the pen drive.
- Check the output of dmesg for the logs.

```
ames
[57511.052032] usb 3-1: ep 0x2 - rounding interval to 128 microframes, ep desc says 255 microfra
mes
[57511.052394] Pen Drive interface no. 0 now probed: (0951:1624)
[57511.052399] ID->NumEndpoints: 02
[57511.052402] ID->InterfaceClass: 08
[57511.052404] ED[0]->EndpointAddress: 0x81
[57511.052406] ED[0]->Attributes: 0x02
[57511.052408] ED[0]->MaxPacketSize: 0x0200 (512)
[57511.052411] ED[1]->EndpointAddress: 0x02
[57511.052413] ED[1]->Attributes: 0x02
[57511.052416] ED[1]->MaxPacketSize: 0x0200 (512)
[57511.078484] usbcore: registered new interface driver usb-storage
[57511.081435] usbcore: registered new interface driver uas
pen-info: ls
Makefile   modules.order   pen_info.c   pen_info.ko     pen_info.mod.o
Makefile~  Module.symvers  pen_info.c~  pen_info.mod.c  pen_info.o
pen-info:
```

3RD MODULE: Data Transfer to and from USB Devices

USB, being a hardware protocol, forms the usual horizontal layer in the kernel space. And hence, for it to provide an interface to user-space, it has to connect through one of the vertical layers.

Also, we do not need to get a free unreserved character major number, but can use the character major number 180, reserved for USB-based character device files. Moreover, to achieve this complete character driver logic with the USB horizontal in one go, the following are the APIs declared in **<linux/usb.h>**:

int usb_register_dev(struct usb_interface *intf, struct usb_class_driver *class_driver);

void usb_deregister_dev(struct usb_interface *intf, struct usb_class_driver *class_driver);

, to achieve the hot-plug-n-play behaviour for the (character) device files corresponding to USB devices, these are instead invoked in the probe and disconnect callbacks, respectively.

The first parameter in the above functions is the interface pointer received as the first parameter in both probe and disconnect. The second parameter, struct usb_class_driver, needs to be populated with the suggested device file name and the set of device file operations, before invoking **usb_register_dev**. For the actual usage, refer to the functions pen_probe and **pen_disconnect** in the code listing of pen_driver.c below.

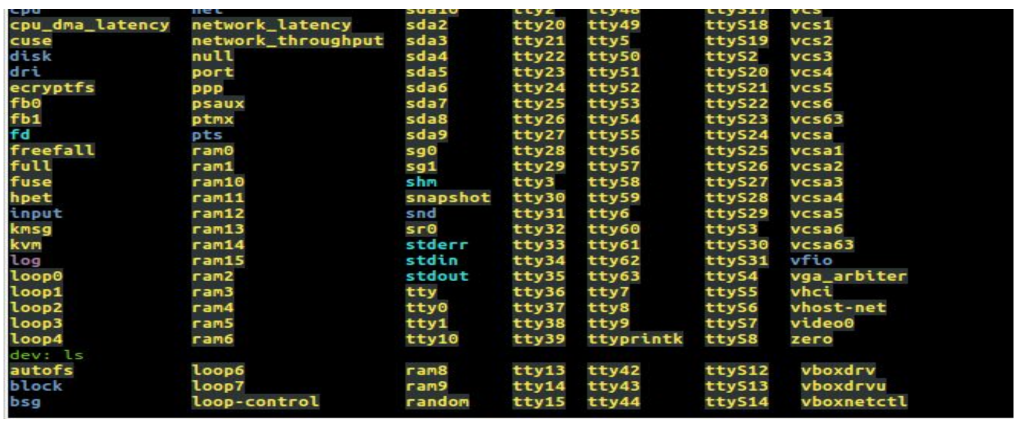
Moreover, as the file operations (write, read, etc.,) are now provided, that is exactly where we need to do the data transfers to and from the USB device. So, pen_write and pen_ read below show the possible calls to usb_bulk_msg() (prototyped in <linux/usb.h>) to do the transfers over the pen drive's bulk end-points 0x01 and 0x82, respectively.

Note : That a pen drive belongs to a USB mass storage class, which expects a set of SCSI-like commands to be transacted over the bulk endpoints. So, a raw read/write as shown in the code listing below may not really do a data transfer as expected, unless the data is appropriately formatted.

Commands that need to be followed are :

- Build the driver (pen_driver.ko) by running make.
- Load the driver using insmod pen_driver.ko.
- Plug in the pen drive (after making sure that the usb-storage driver is not already loaded).
- Check for the dynamic creation of /dev/pen0 (0 being the minor number obtained — check dmesg logs for the value on your system).
- Possibly try some write/read on /dev/pen0 (you most likely will get a connection timeout and/or broken pipe errors, because of non-conforming SCSI commands).
- Unplug the pen drive and look for /dev/pen0 to be gone.

Snippet before connecting the USB :



```
cpu_dma_latency  network_latency      sda2      tty20  tty49      ttyS18  vcs1
cuse             network_throughput   sda3      tty21  tty5       ttyS19  vcs2
disk             null                 sda4      tty22  tty50      ttyS2   vcs3
dri              port                 sda5      tty23  tty51      ttyS20  vcs4
ecryptfs         ppp                  sda6      tty24  tty52      ttyS21  vcs5
fb0              psaux                sda7      tty25  tty53      ttyS22  vcs6
fb1              ptmx                 sda8      tty26  tty54      ttyS23  vcs63
fd               pts                  sda9      tty27  tty55      ttyS24  vcsa
freefall         ram0                 sg0       tty28  tty56      ttyS25  vcsa1
full             ram1                 sg1       tty29  tty57      ttyS26  vcsa2
fuse             ram10                shm       tty3   tty58      ttyS27  vcsa3
hpet             ram11                snapshot  tty30  tty59      ttyS28  vcsa4
input            ram12                snd       tty31  tty6       ttyS29  vcsa5
kmsg             ram13                sr0       tty32  tty60      ttyS3   vcsa6
kvm              ram14                stderr    tty33  tty61      ttyS30  vcsa63
log              ram15                stdin     tty34  tty62      ttyS31  vfio
loop0            ram2                 stdout    tty35  tty63      ttyS4   vga_arbiter
loop1            ram3                 tty       tty36  tty7       ttyS5   vhci
loop2            ram4                 tty0      tty37  tty8       ttyS6   vhost-net
loop3            ram5                 tty1      tty38  tty9       ttyS7   video0
loop4            ram6                 tty10     tty39  ttyprintk  ttyS8   zero
dev: ls
autofs           loop6                ram8      tty13  tty42      ttyS12  vboxdrv
block            loop7                ram9      tty14  tty43      ttyS13  vboxdrvu
bsg              loop-control         random    tty15  tty44      ttyS14  vboxnetctl
```

Snippet after connecting the USB ,we can see pen0(character file created):

```
char              mei                  sda        tty19  tty48      ttyS18   vcs2
console           mem                  sda1       tty2   tty49      ttyS19   vcs3
core              net                  sda10      tty20  tty5       ttyS2    vcs4
cpu               network_latency      sda2       tty21  tty50      ttyS20   vcs5
cpu_dma_latency   network_throughput   sda3       tty22  tty51      ttyS21   vcs6
cuse              null                 sda4       tty23  tty52      ttyS22   vcs63
disk              pen0                 sda5       tty24  tty53      ttyS23   vcsa
dri               port                 sda6       tty25  tty54      ttyS24   vcsa1
ecryptfs          ppp                  sda7       tty26  tty55      ttyS25   vcsa2
fb0               psaux                sda8       tty27  tty56      ttyS26   vcsa3
fb1               ptmx                 sda9       tty28  tty57      ttyS27   vcsa4
fd                pts                  sg0        tty29  tty58      ttyS28   vcsa5
freefall          ram0                 sg1        tty3   tty59      ttyS29   vcsa6
full              ram1                 shm        tty30  tty6       ttyS3    vcsa63
fuse              ram10                snapshot   tty31  tty60      ttyS30   vfio
hpet              ram11                snd        tty32  tty61      ttyS31   vga_arbiter
input             ram12                sr0        tty33  tty62      ttyS4    vhci
kmsg              ram13                stderr     tty34  tty63      ttyS5    vhost-net
kvm               ram14                stdin      tty35  tty7       ttyS6    video0
log               ram15                stdout     tty36  tty8       ttyS7    zero
loop0             ram2                 tty        tty37  tty9       ttyS8
loop1             ram3                 tty0       tty38  ttyprintk  ttyS9
loop2             ram4                 tty1       tty39  ttyS0      uhid
loop3             ram5                 tty10      tty4   ttyS1      uinput
loop4             ram6                 tty11      tty40  ttyS10     urandom
loop5             ram7                 tty12      tty41  ttyS11     v4l
dev:
```

REFERENCES :

1) https://www.google.co.in/search?q=device+driver&biw=1301&bih=613&source=lnms&tbm=isch&sa=X&ved=0CAYQ_AUoAWoVChMItNev2630yAIVTBqUCh27zAKf#tbm=isch&q=device+driver+linux&imgrc=_

2) https://www.fsl.cs.sunysb.edu/kernel-api/re256.html

3) http://www.makelinux.net/ldd3/chp-13-sect-5

4) http://www-numi.fnal.gov/offline_software/srt_public_context/WebDocs/Errors/unix_system_errors.html

5) http://www.unix.com/programming/231791-what-use-char-__user-buf.html