**Device driver programming:**

**How to configure and install kernel:**

1) Download the kernel source code.

2) Extract the Source Code: tar xvf linux-5.9.6.tar.xz

3) Configure Kernel:

* Navigate to the code directory using the cd command (cd linux-5.9.6)
* Copy the existing configuration file using the cp command (cp -v /boot/config-$(uname -r) .config)
* To make changes to the configuration file, run the make command (make menuconfig)

4) Build the Kernel (make -j 4)

5) Install the required modules with this command (sudo make modules\_install)

6) Install the kernel (sudo make install)

7)Update initramfs

8)Update grub.

**Check installed kernel version: uname -r**

**check installed kernel date: uname -v**

**Kernel is installed in path: /lib/modules/$(uname-r)**

### lsmod – List Modules that Loaded Already

lsmod command will list modules that are already loaded in the kernel as shown beblow.

# lsmod

Module Size Used by

ppp\_deflate 12806 0

zlib\_deflate 26445 1 ppp\_deflate

bsd\_comp 12785 0

..

### 2. insmod – Insert Module into Kernel

insmod command will insert a new module into the kernel as shown below.

**# insmod /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko**

# lsmod | grep "squash"

squashfs 35834 0

### 3. modinfo – Display Module Info

modinfo command will display information about a kernel module as shown below.

# modinfo /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko

filename: /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko

license: GPL

author: Phillip Lougher

description: squashfs 4.0, a compressed read-only filesystem

srcversion: 89B46A0667BD5F2494C4C72

depends:

intree: Y

vermagic: 3.5.0-19-generic SMP mod\_unload modversions 686

### 4. rmmod – Remove Module from Kernel

rmmod command will remove a module from the kernel. You cannot remove a module which is already used by any program.

# rmmod squashfs.ko

### 5. modprobe – Add or Remove modules from the kernel

modprobe is an intelligent command which will load/unload modules based on the dependency between modules. Refer to [modprobe commands](https://www.thegeekstuff.com/2010/11/modprobe-command-examples/) for more detailed examples.

## II. Write a Simple Hello World Kernel Module

### 1. Installing the linux headers

You need to install the linux-headers-.. first as shown below. Depending on your distro, use apt-get or yum.

# apt-get install build-essential linux-headers-$(uname -r)

### 2. Hello World Module Source Code

Next, create the following hello.c module in C programming language.

#include <linux/module.h> // included for all kernel modules

#include <linux/kernel.h> // included for KERN\_INFO

#include <linux/init.h> // included for \_\_init and \_\_exit macros

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Ankit");

MODULE\_DESCRIPTION("A Simple Hello World module");

static int \_\_init hello\_init(void)

{

printk(KERN\_INFO "Hello world!\n"); **//printing to kernel buffer.**

return 0; // Non-zero return means that the module couldn't be loaded.

}

static void \_\_exit hello\_cleanup(void)

{

printk(KERN\_INFO "Cleaning up module.\n"); **//printing to kernel buffer.**

}

module\_init(hello\_init);

module\_exit(hello\_cleanup);

Warning: All kernel modules will operate on kernel space, a highly privileged mode. So be careful with what you write in a kernel module.

### 3. Create Makefile to Compile Kernel Module

The following makefile can be used to compile the above basic hello world kernel module.

obj-m += hello.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

Use the [make command](https://www.thegeekstuff.com/2010/08/make-utility/) to compile hello world kernel module as shown below.

# make

make -C /lib/modules/3.5.0-19-generic/build M=/home/lakshmanan/a modules

make[1]: Entering directory `/usr/src/linux-headers-3.5.0-19-generic'

CC [M] /home/lakshmanan/a/hello.o

Building modules, stage 2.

MODPOST 1 modules

CC /home/lakshmanan/a/hello.mod.o

LD [M] /home/lakshmanan/a/hello.ko

make[1]: Leaving directory `/usr/src/linux-headers-3.5.0-19-generic'

The above will create hello.ko file, which is our sample Kernel module.

### 4. Insert or Remove the Sample Kernel Module

Now that we have our hello.ko file, we can insert this module to the kernel by using insmod command as shown below.

# insmod hello.ko

# dmesg | tail -1

[ 8394.731865] Hello world!

# rmmod hello.ko

# dmesg | tail -1

[ 8707.989819] Cleaning up module.

When a module is inserted into the kernel, the **module\_init** macro will be invoked, which will call the function hello\_init. Similarly, when the module is removed with rmmod, **module\_exit** macro will be invoked, which will call the hello\_exit. Using dmesg command, we can see the output from the sample Kernel module.

**Output file generated when we compile our module:**

**modules.order**

--------------------------------------------------

This file records the order in which modules appear in Makefiles. This

is used by modprobe to deterministically resolve aliases that match

multiple modules.

---------------------------------------------------------------------

If the files are present the source file is compiled to a "modulname.o", and "modulename.mod.c" is created which is compiled to "modulename.mod.o".   
The modulename.mod.c is a file that basically contains the information about the module (Version information etc).   
The modulename.o and the modulename.mod.o are linked together by modpost in the next stage to create the "modulename.ko" .

### Network Device driver stack:

**Struct net\_dvice is used for network device.**

**Device Allocation: alloc\_netdev()**

**Device Registration: register\_netdevice()**

Diagram

Description automatically generated

**Character device driver:**

Diagram

Description automatically generated

When we want to communicate(open/read/write) to device. Application calls system call (open/read/write)

User space system call is connected to driver system call implementation is taken care by VFS. Our device driver has to get registered with vfs.

**When you use open system call on device file, how does the kernel connect to open system call to intended driver’s open call?**

**Kernel uses device no. Assign no. to driver**

**Below there are 4 instances of rtc type devices but driver will be same for all devices. There are 4 files created by driver. Communication with device will be done using device files.**

A picture containing diagram

Description automatically generated

**Device number is combination of major number and minor number.**

Diagram

Description automatically generated

**Major no. and minor no. can be checked in system by cd to dev and check ls-l command:**



**Connection establishment b/w device file access and driver:**

**Driver creates the device no., device file, makes char device registration with VFS using CDEV\_ADD and**

**Implements the driver’s file operation methods for open. Read, write etc.**

When user program uses open system call on device file(dev/rtc), system call is handled by VFS first. VFS gets the device no. and compares it with driver registration list, that means this driver to get registered with VFS using device no. that we call character device add, CDEV\_ADD (). When VFS gets open call from application, it opens a file by creating new file object and linking it to corresponding inode object.

### 

### Diagram Description automatically generated

### 

Graphical user interface, text, application

Description automatically generated

**Below creation calls are written in Init function of driver and deletion calls in Exit function of driver.**

Text, application, chat or text message

Description automatically generated

Table

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**Device Files: The device file allows transparent communication b/w user-space application and hardware.**

**Device file creation:**

1. **Manually:** We can create device file manually by mknod:

**mknod -m <permissions> <name> <device type> <major> <minor>**

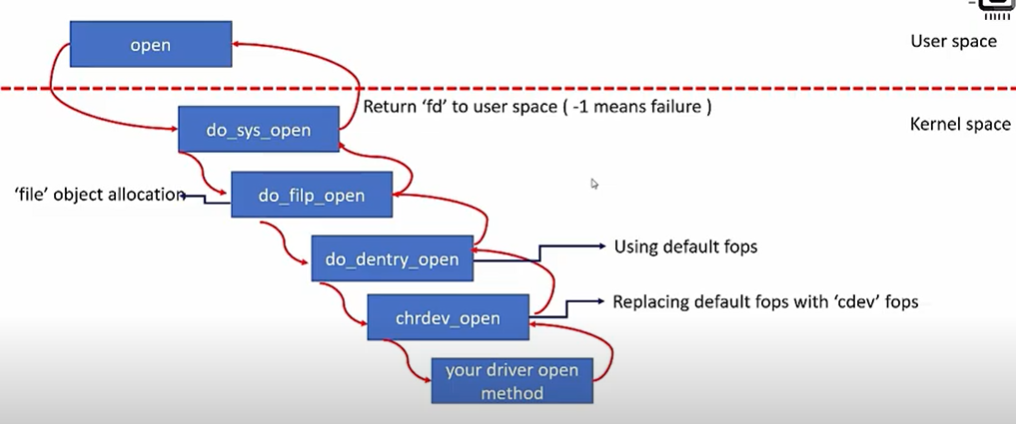
sudo mknod -m 666 /dev/etx\_device c 246 0

1. **Automatically:** The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically.

**When device file is created** -> special\_init\_inode() function is called -> inode object created in memory -> in memory inode object's device no is initialized, and inode object's file ops is initialized with dummy default file operation methods.

**When we call open method ()->**

Do\_file\_open creates file object and now it calls do\_dentry\_open(). do\_dentry\_open() uses default ops. It calls default chrdev\_open() which replaces default fops with cdev fops.



Graphical user interface, text, application, email

Description automatically generated

Diagram

Description automatically generated

**Char driver code:**

#include <linux/init.h>

#include <linux/module.h>

#include <linux/cdev.h>

#include <linux/device.h>

#include <linux/kernel.h>

#include <linux/uaccess.h>

#include <linux/fs.h>

#define MAX\_DEV 2

static int mychardev\_open (struct inode \*inode, struct file \*file);

static int mychardev\_release (struct inode \*inode, struct file \*file);

static long mychardev\_ioctl (struct file \*file, unsigned int cmd, unsigned long arg);

static ssize\_t mychardev\_read (struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offset);

static ssize\_t mychardev\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*offset);

static const struct file\_operations mychardev\_fops = {

.owner = THIS\_MODULE,

.open = mychardev\_open,

.release = mychardev\_release,

.unlocked\_ioctl = mychardev\_ioctl,

.read = mychardev\_read,

.write = mychardev\_write

};

struct mychar\_device\_data

{

struct cdev cdev;

};

static int dev\_major = 0;

static struct class \*mychardev\_class = NULL;

static struct mychar\_device\_data mychardev\_data[MAX\_DEV];

static int mychardev\_uevent(struct device \*dev, struct kobj\_uevent\_env \*env)

{

add\_uevent\_var(env, "DEVMODE=%#o", 0666);

return 0;

}

static int \_\_init mychardev\_init(void)

{

int err, i;

dev\_t dev;

**//Allocates a range of char device numbers. The major number will be chosen dynamically, and returned (along with the first minor number) in**

**dev. Returns zero or a negative error.**

err = alloc\_chrdev\_region(&dev, 0, MAX\_DEV, "mychardev");

dev\_major = MAJOR(dev);

mychardev\_class = class\_create(THIS\_MODULE, "mychardev");

mychardev\_class->dev\_uevent = mychardev\_uevent;

for (i = 0; i < MAX\_DEV; i++)

{

**//initialize a cdev structure**

cdev\_init(&mychardev\_data[i].cdev, &mychardev\_fops);

mychardev\_data[i].cdev.owner = THIS\_MODULE;

**//add a char device to the system**

cdev\_add(&mychardev\_data[i].cdev, MKDEV(dev\_major, i), 1);

**//creates a device and registers it with sysfs**

device\_create(mychardev\_class, NULL, MKDEV(dev\_major, i), NULL, "mychardev-%d", i);

}

return 0;

}

static void \_\_exit mychardev\_exit(void)

{

int i;

for (i = 0; i < MAX\_DEV; i++) {

device\_destroy(mychardev\_class, MKDEV(dev\_major, i));

}

class\_unregister(mychardev\_class);

class\_destroy(mychardev\_class);

unregister\_chrdev\_region(MKDEV(dev\_major, 0), MINORMASK);

}

static int mychardev\_open(struct inode \*inode, struct file \*file)

{

printk("MYCHARDEV: Device open\n");

return 0;

}

static int mychardev\_release(struct inode \*inode, struct file \*file)

{

printk("MYCHARDEV: Device close\n");

return 0;

}

static long mychardev\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

printk("MYCHARDEV: Device ioctl\n");

return 0;

}

static ssize\_t mychardev\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

uint8\_t \*data = "Hello from the kernel world!\n";

size\_t datalen = strlen(data);

printk("Reading device: %d\n", MINOR(file->f\_path.dentry->d\_inode->i\_rdev));

if (count > datalen) {

count = datalen;

}

if (copy\_to\_user(buf, data, count)) {

return -EFAULT;

}

return count;

}

static ssize\_t mychardev\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

size\_t maxdatalen = 30, ncopied;

uint8\_t databuf[maxdatalen];

printk("Writing device: %d\n", MINOR(file->f\_path.dentry->d\_inode->i\_rdev));

if (count < maxdatalen) {

maxdatalen = count;

}

ncopied = copy\_from\_user(databuf, buf, maxdatalen);

if (ncopied == 0) {

printk("Copied %zd bytes from the user\n", maxdatalen);

} else {

printk("Could't copy %zd bytes from the user\n", ncopied);

}

databuf[maxdatalen] = 0;

printk("Data from the user: %s\n", databuf);

return count;

}

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Oleg Kutkov <elenbert@gmail.com>");

module\_init(mychardev\_init);

module\_exit(mychardev\_exit);