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 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"); //priniting 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"); //priniting 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 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).

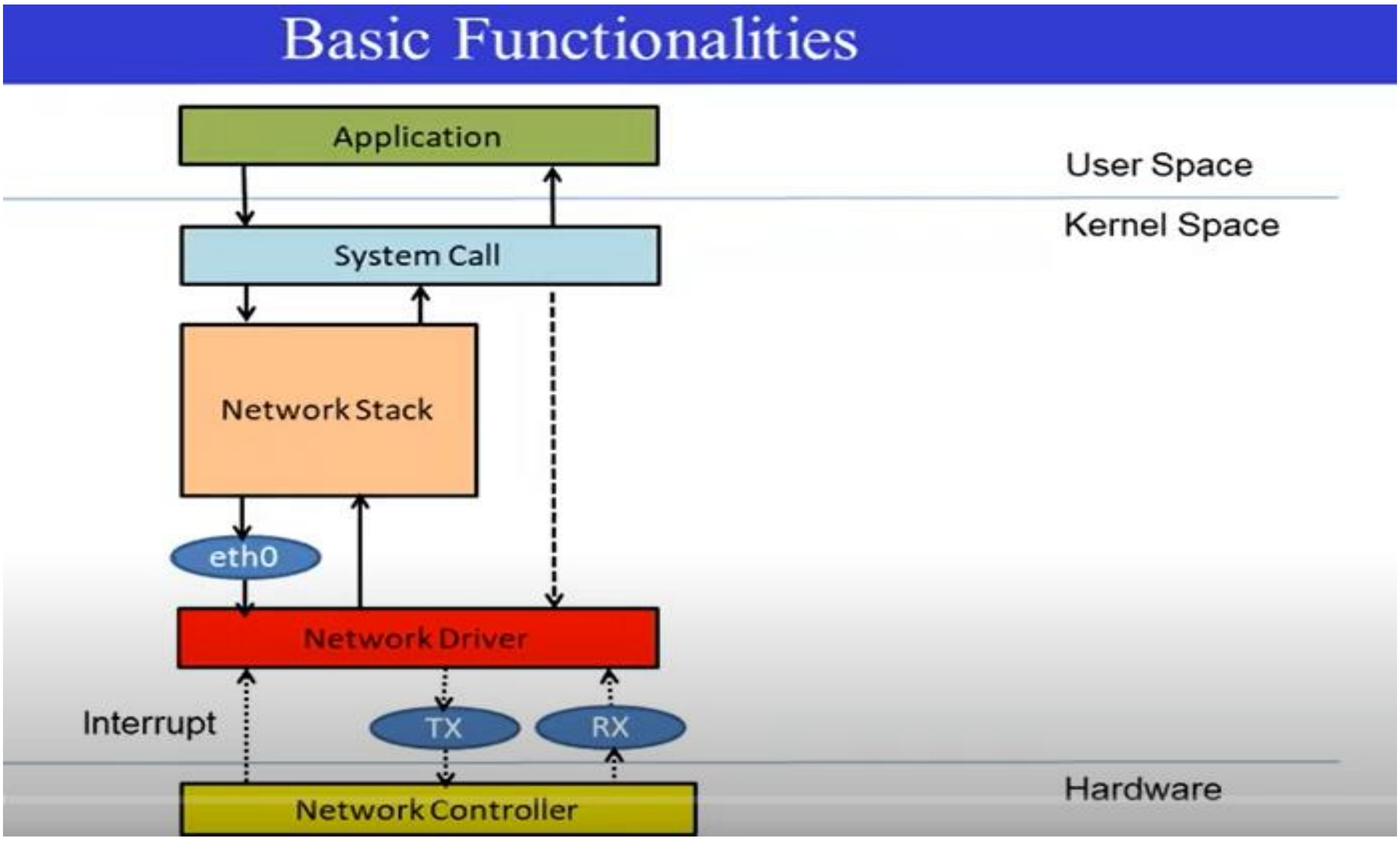
The modulename.o and the modulename.mod.o are linked together by modpost in the next stage to create the "modulename.ko" .

Network Device driver stack:

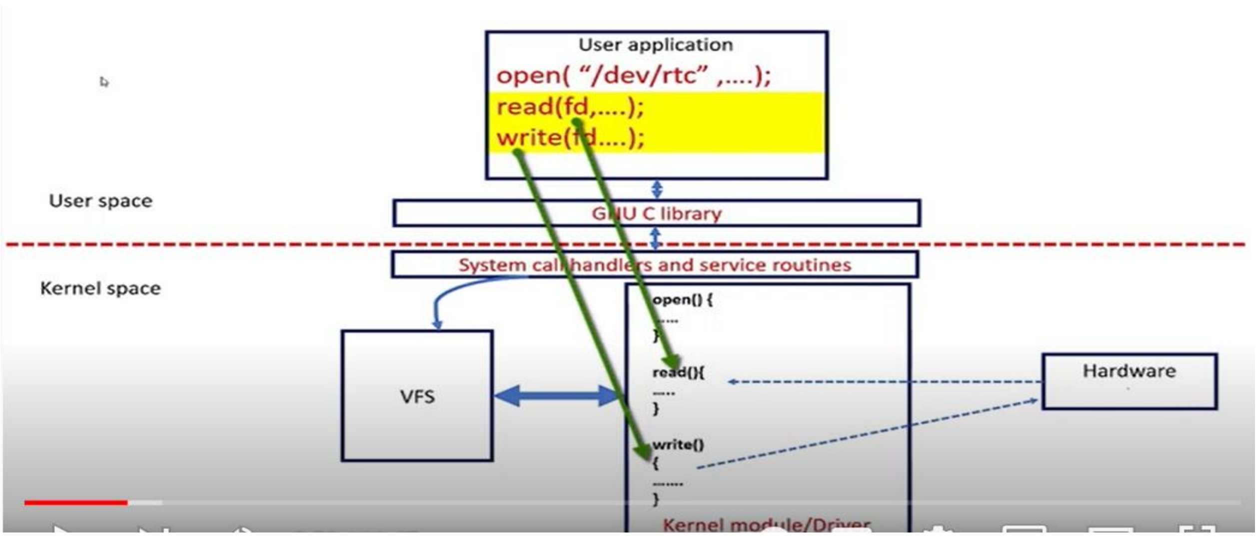
Struct net_dvice is used for network device.

Device Allocation: alloc_netdev()

Device Registration: register_netdevice()



Character device driver:



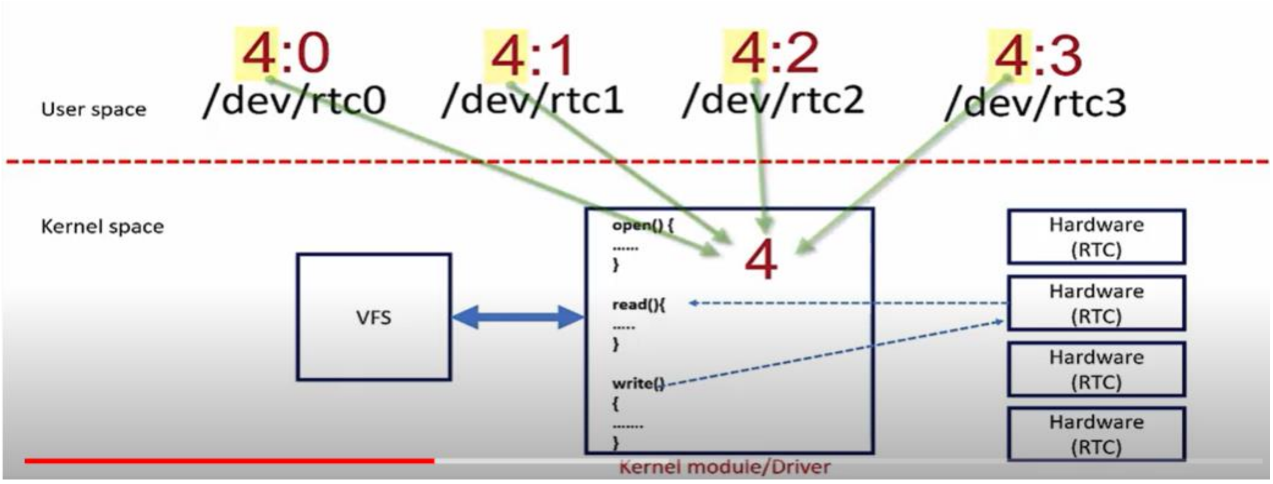
When we want to communicate(open/read/write) to device. Application calls system call (open/read/write)

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.

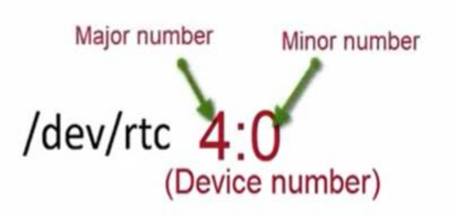
When you use open system call on device file, how does the kernel connect to open system call to intended driver's open call?

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.



Device number is combination of major number and minor number.



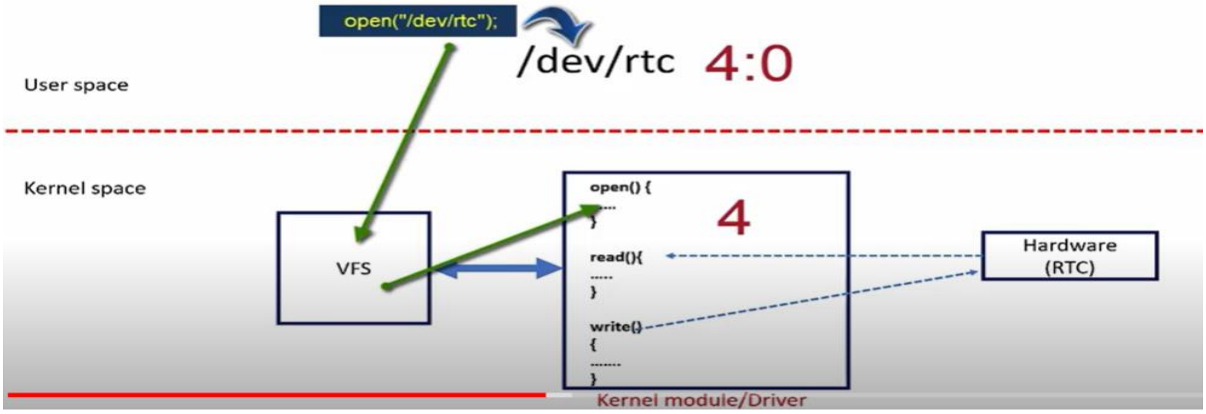
Major no. and minor no. can be checked in system by cd to dev and check ls-l command:

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.



Kernel APIs and utilities to be used in driver code

alloc_chrdev_region();	1.Create device number
cdev_init(); cdev_add();	2. Make a char device registration with the VFS
class_create(); device_create();	3. Create device files

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

Creation	Deletion
alloc_chrdev_region();	unregister_chrdev_region();
cdev_init(); cdev_add();	cdev_del();
class_create(); device_create();	class_destroy(); device_destroy();

Kernel Header file details

Kernel functions and data structures	Kernel header file
alloc_chrdev_region() unregister_chrdev_region()	include/linux/fs.h
cdev_init() cdev_add() cdev_del()	include/linux/cdev.h
device_create() class_create() device_destroy() class_destroy()	include/linux/device.h
copy_to_user() copy_from_user()	include/linux/uaccess.h
VFS structure definitions	inclue/linux/fs.h

Device number representation

- The device number is a combination of major and minor numbers
- In Linux kernel, dev_t (typedef of u32) type is used to represent the device number.
- Out of 32 bits, 12 bits to store major number and remaining 20 bits to store minor number
- You can use the below macros to extract major and minor parts of dev_t type variable

```
dev_t device_number;
int minor_no = MINOR(device_number);
int major_no = MAJOR(device_number);
```

- You can find these macros in linux/kdev_t.h
- If you have, major and minor numbers , use the below macro to turn them into dev_t type device number

```
MKDEV(int major, int minor);
```

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

2) **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.

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;

    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++) {
        cdev_init(&mychardev_data[i].cdev, &mychardev_fops);
        mychardev_data[i].cdev.owner = THIS_MODULE;

        cdev_add(&mychardev_data[i].cdev, MKDEV(dev_major, i), 1);

        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);
```