Author : Malinda Sulochana Silva

Dept.of Electrical and Electronic Engineering

Faculty of Engineering,University of Peradeniya

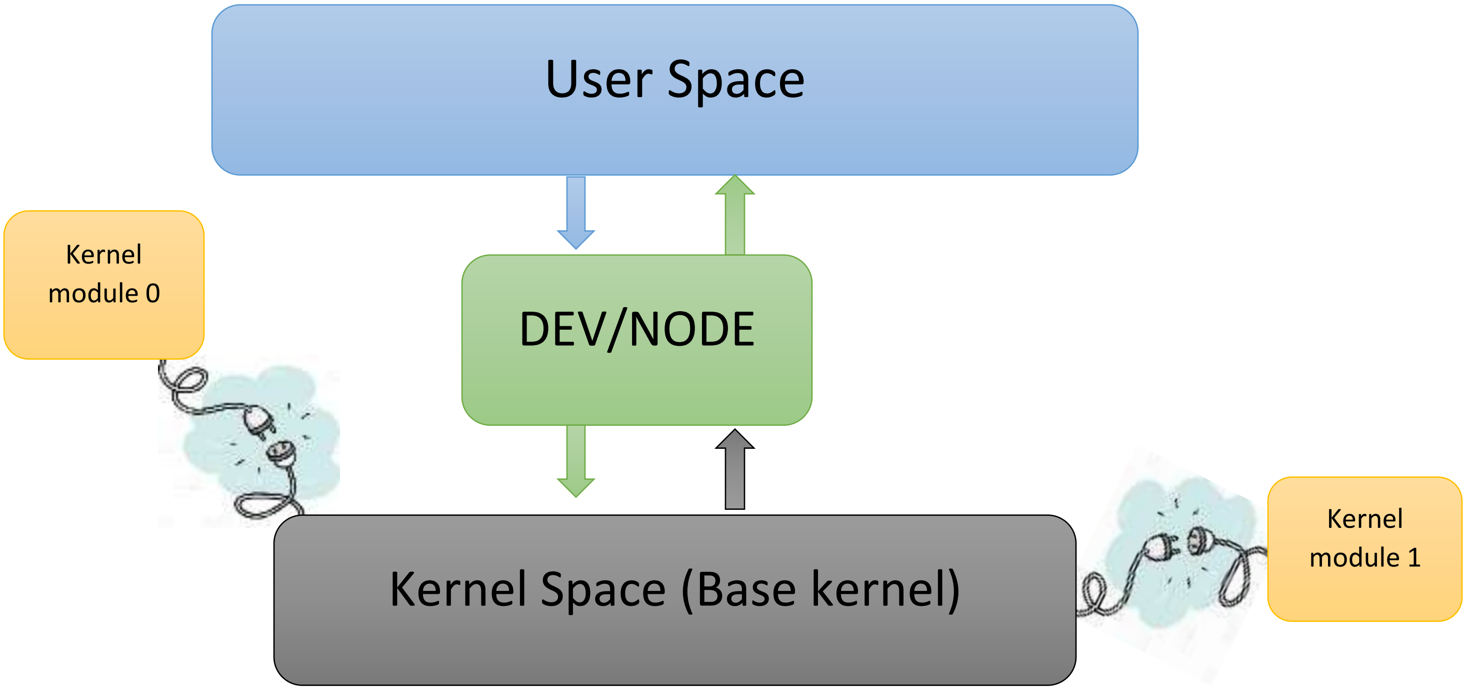
This chapter will discuss on how to

- Make a kernel module
- Implement Timer events
- And implement Interrupts in Kernel space.

And furthermore, Development of a user application (In User Space) to cater with the kernel through IOCTL (Input/output Control) commands.

For easiness of implementation let's target on Ubuntu which is a Linux kernel bases Operating system.

Overview of Kernel Space and the User space



Let's look at the Program Structure of a kernel Module

[Includes which would be needed]

```
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
```

[Definitions and Declarations]

```
#define KBD_IRQ          1      /* IRQ number for keyboard (i8042) */
#define KBD_DATA_REG      0x60
struct cdev my_cdev;
static int    majorNumber;

static int     my_open( struct inode *, struct file *);
static ssize_t my_read( struct file * , char *, size_t, loff_t *);.etc
```

[Initializing routine]

All the initialization should be here
This is going to be invoked during the plugging of the module to the kernel

[Other routines]

Including subroutines, Interrupt Handlers, Callback routines etc.

[Finalizing routine]

All used memories, interrupt requests should release and should ready to be unplugged from the kernel

Finally the Initializing routine and finalizing routines are called.

```
module_init(<Initializing routine>);  //these are kernel macros
module_init(<finalizing routine>);
```

Procedure to make a device driver

First of all you have to have a Major number and a Minor for your device driver in order to identify the device, this can be dynamically allocated or even can be allocated manually.

Then you have to have a device class and a Device name.

Thereafter these class and the device should be registered.

```
// Try to dynamically allocate a major number for the device -- more difficult but worth it
        majorNumber = register_chrdev(0, DEVICE_NAME, &my_fops); // 0 wil auto allocate major num
        if (majorNumber<0) {
          printk(KERN_ALERT "char driver failed to register a major number\n");
          return majorNumber;
        }
        printk(KERN_INFO "char driver: Major number %d\n", majorNumber);

// Register the device class
        CharClass = class_create(THIS_MODULE, CLASS_NAME);
        if (IS_ERR(CharClass)){             // Check for error and clean up if there is
                unregister_chrdev(majorNumber, DEVICE_NAME);
                printk(KERN_ALERT "Failed to register device class\n");
                return PTR_ERR(CharClass);      // Correct way to return an error on a pointer
        }

// Register the device driver
        CharDevice = device_create(CharClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
        if (IS_ERR(CharDevice)){            // Clean up if there is an error
                class_destroy(CharClass);       // Repeated code but the alternative is goto statements
                unregister_chrdev(majorNumber, DEVICE_NAME);
                printk(KERN_ALERT "Failed to create the device\n");
                return PTR_ERR(CharDevice);
        }
```

Since we are focusing on a char driver, there should be a way to communicate with the user space, for that we use the file operation structure in kernel.

```
        struct file_operations my_fops = {
        read    :       my_read,
        write   :       my_write,
        open    :       my_open,
        release :       my_close,
        unlocked_ioctl :        my_ioctl,
        owner   :       THIS_MODULE
        };
```

In here user space have the access to the read, write … functions. The arguments are defined in the kernel space functions which are related as above.

```
        static ssize_t my_read(struct file *filp, char *buff, size_t len, loff_t *off)
        {
                short count;
                printk("\n*****Some body is Reading me*******\n");
```

```
            count = copy_to_user(buff, msg, len);
            return 0;
    }
```

*Here the buff means a char pointer to the user space buffer and the msg means the kernel space buffer

Implementing an Interrupt

In the process of implementing an interrupt for the kernel module 4 things should be aware of,

- Selecting the interrupt request number (IRQ number)
- Requesting for an interrupt
- Implementing the interrupt handler also named as Interrupt service routine (ISR)
- Finally make sure to release/free the IRQ

```
•   #define KBD_IRQ          1      //IRQ number for the keyboard
•   request_irq(KBD_IRQ, kbd2_isr, IRQF_SHARED, "kbd2", (void *)kbd2_isr);
•   static irqreturn_t kbd2_isr(int irq, void *dev_id){
            //your ISR code goes here
                 return IRQ_HANDLED;
    }
        •   free_irq(KBD_IRQ, (void *)kbd2_isr);
```

Implementing an Timer callback event

In the process of implementing a Timer callback event for the kernel module few things should be aware of,

Declaring

```
static struct timer_list my_timer;
void my_timer_callback( unsigned long data );
```

Initializing

```
setup_timer(&my_timer, my_timer_callback, 0);/* setup your timer to call my_timer_callback */
mod_timer(&my_timer, jiffies + msecs_to_jiffies(200));/* setup timer interval to 200 msecs */
```

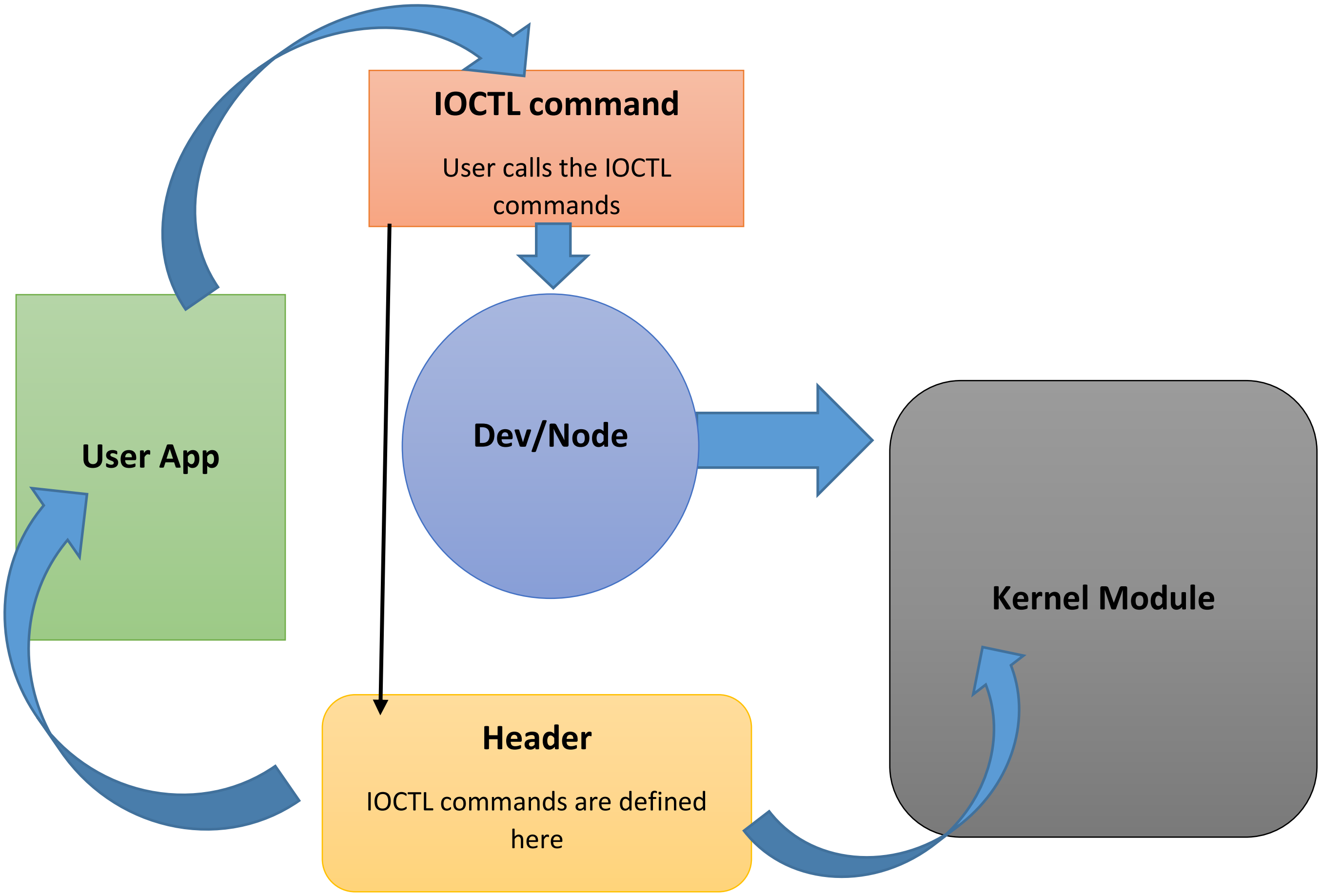
Timer callback routine

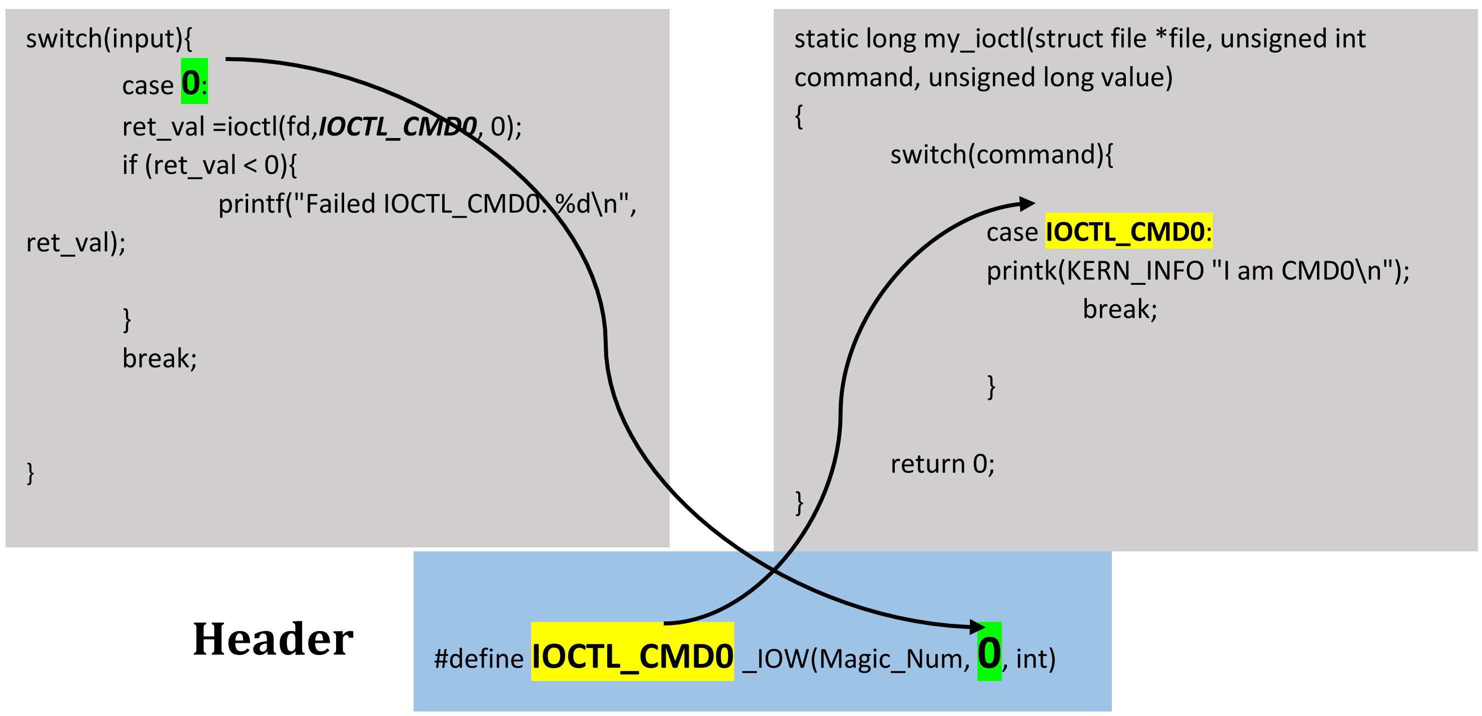
```
void my_timer_callback( unsigned long data )
{
         mod_timer(&my_timer, jiffies + msecs_to_jiffies(2000));/* setup timer interval to 2000 msecs */
   printk(KERN_ALERT "2 seconds Gone \n");

}
```

Deleting the Timer

```
del_timer(&my_timer);/* remove kernel timer when unloading module*/
```

How IOCTL works


IOCTL command
User calls the IOCTL commands
User App
Dev/Node
Kernel Module
Header
IOCTL commands are defined here



User program

```
switch(input){
        case 0:
        ret_val =ioctl(fd,IOCTL_CMD0, 0);
        if (ret_val < 0){
                printf("Failed IOCTL_CMD0.%d\n",
ret_val);

        }
        break;


}
```

Kernel Program

```
static long my_ioctl(struct file *file, unsigned int
command, unsigned long value)
{
        switch(command){

                case IOCTL_CMD0:
                printk(KERN_INFO "I am CMD0\n");
                        break;

                }

        return 0;
}
```

Header

```
#define IOCTL_CMD0 _IOW(Magic_Num, 0, int)
```

IOCTL makes it user friendly for the user app to communicate with the kernel module

Example code to capture keyboard press, evoke a timer event and User application to communicate with the kernel module.

User Application

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <sys/ioctl.h>
#include <unistd.h>

#include "User.h"

int main(){

        int input = 0;
        char buff;
        int fd;
        int ret_val=0;

        fd = open("/dev/TEST_DRIVER",O_RDWR);

        if(fd == 0){
                printf("Error opening file");
```

```
    }else{

        printf("Enter 1/0 : ");
        scanf("%d", &input);
        //write(fd,&input,1);
        switch(input){
            case 0:
                ret_val =ioctl(fd,IOCTL_CMD0, 0);
                if (ret_val < 0){
                    printf("Failed IOCTL_CMD0: %d\n", ret_val);

                }
                break;
            case 1:
                ret_val = ioctl(fd, IOCTL_CMD1, 0);
                if (ret_val < 0){
                    printf("Failed IOCTL_CMD1: %d\n", ret_val);

                }
                break;
            case 2:
                ret_val = ioctl(fd, IOCTL_CMD2,&buff);
                if (ret_val < 0){
                    printf("Failed IOCTL_CMD2: %d\n", ret_val);

                }
                printf("Buffer : %d\r\n", buff);
                break;
            default:
                printf("Error input");

        }

        close(fd);

    }
return 0;
}
```

Header

```
#ifndef TEST_H

#define TEST_H

#define MY_MAJOR  200
#define MY_MINOR  0
#define MY_DEV_COUNT 1
#define DEVICE_NAME "TEST_DRIVER"        ///< The device will appear at /dev
#define CLASS_NAME  "TDRER"        ///< The device class -- this is a character device driver

#include <linux/ioctl.h>
```

```
#define Magic_Num 131

#define IOCTL_CMD0 _IOW(Magic_Num, 0, int)
#define IOCTL_CMD1 _IOW(Magic_Num, 1, int)
#define IOCTL_CMD2 _IOR(Magic_Num, 2, char*)

#define DEVICE_FILE_NAME "TEST_DRIVER"

#endif
```

Kernel Module

```
/*
 * This is a keyboard tracking kernal module;
 * once this module is pulgged into the kernal
 * all the track recods are reserved in the kernal.
 *
 * Devoloped by: Malinda Sulochana Silva
 * Organization: Zone24x7
 *
 * Copyright. All Rights Reserved.
 * */

#include <linux/module.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/init.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/list.h>
#include <linux/irq.h>
#include <linux/slab.h>
#include <linux/gpio.h>
#include <linux/time.h>
#include <linux/timer.h>
#include <linux/delay.h>
#include <linux/ioctl.h>
#include "User.h"
//___________________Definitions

#define KBD_IRQ             1       /* IRQ number for keyboard (i8042) */
#define KBD_DATA_REG        0x60    /* I/O port for keyboard data */
#define KBD_SCANCODE_MASK   0x7f
#define KBD_STATUS_MASK     0x80

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Malinda Sulochana Silva");
```

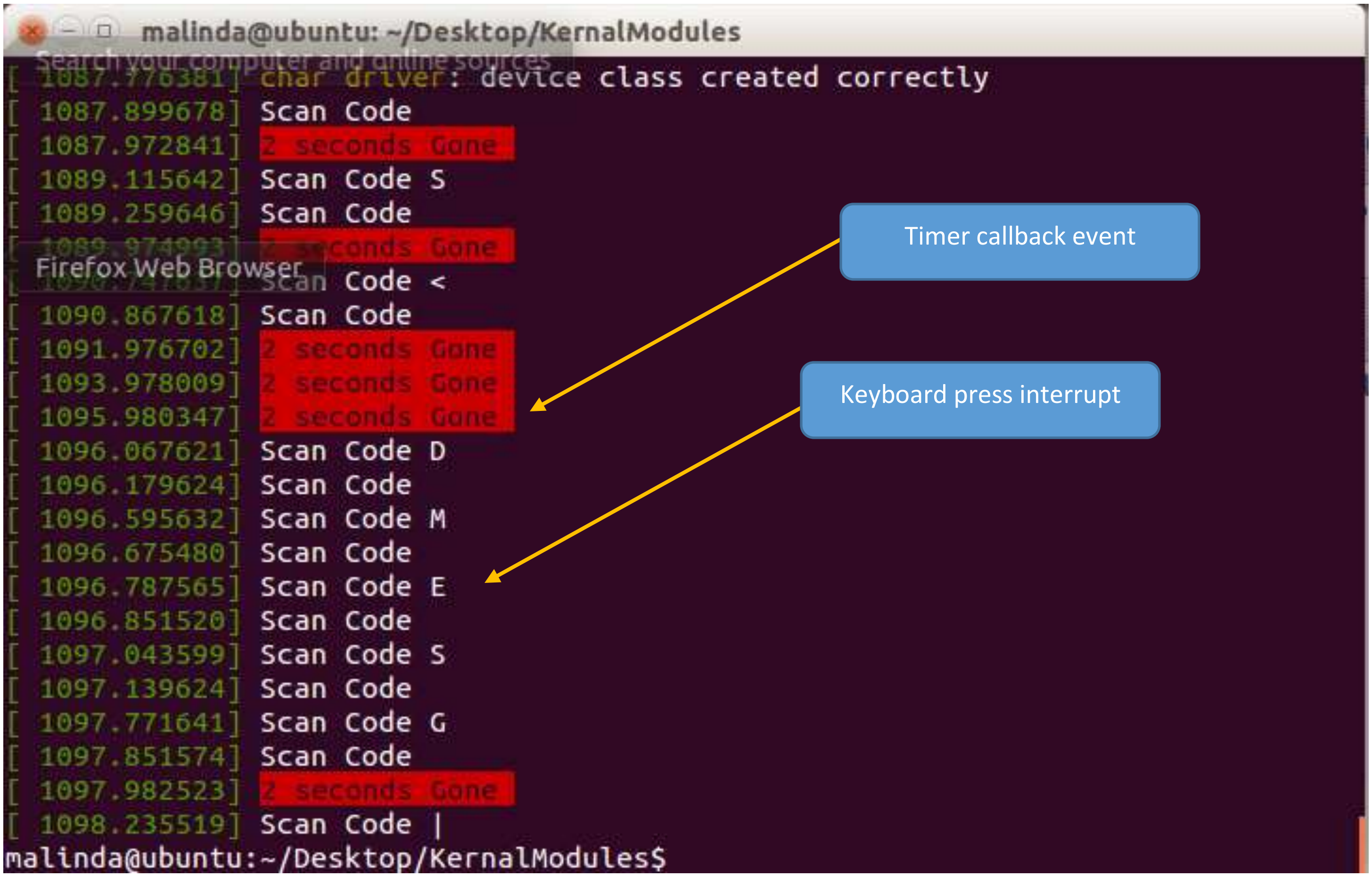
```
MODULE_DESCRIPTION("A Simple GPIO Device Driver module for Ubuntu");

//__________________Declerations

unsigned long j, stamp_1, stamp_half, stamp_n;
int malinda=123;
int n=2;

static char   *msg=NULL;
struct cdev my_cdev;
static int    majorNumber;

static int     my_open( struct inode *, struct file *);
static ssize_t my_read( struct file * , char *, size_t, loff_t *);
static ssize_t my_write(struct file * , const  char *, size_t, loff_t *);
static int     my_close(struct inode *, struct file *);
static long    my_ioctl(struct file *, unsigned int, unsigned long);
static irqreturn_t kbd2_isr(int irq, void *dev_id);

static struct timer_list my_timer;
void my_timer_callback( unsigned long data );
static struct class*  CharClass  = NULL; ///< The device-driver class struct pointer
static struct device* CharDevice = NULL; ///< The device-driver device struct pointer


/*++++++++++++++++++++++++++++++++_MAIN_++++++++++++++++++++++++++++++++++++*/
/*||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||*/

/*____________________________File Operations___________________________*/
struct file_operations my_fops = {
       read    :      my_read,
       write   :      my_write,
       open    :      my_open,
       release :      my_close,
 unlocked_ioctl :      my_ioctl,
       owner   :      THIS_MODULE
};
/*====================================================================*/

/*-----------------------------------------------------------------*/
/*_________________________File Operation Routines_____________________*/
static int my_open(struct inode *inod, struct file *fil)
{
   printk("\n*****Some body is opening me*******\n");
   return 0;
}
static ssize_t my_read(struct file *filp, char *buff, size_t len, loff_t *off)
{
            short count;
            printk("\n*****Some body is Reading me*******\n");
            count = copy_to_user(buff, msg, len);
            return 0;
```

```
}
static ssize_t my_write(struct file *filp, const char *buff, size_t len, loff_t *off)
{
        short count;
        memset(msg, 0, 32);
        printk("\n*****Some body is writting to me*******\n");
        // -- copy the string from the user space program which open and write this device
        count = copy_from_user( msg, buff, len );
        printk("%s\n",msg);
        return count;
}
static int my_close(struct inode *inod, struct file *fil)
{
        printk("\n*****Some body is Closing me*******\n");
        return 0;
}
/*====================================================================*/

/*------------------------------------------------------------------*/
/*______________________________IOCTL Routine____________________________*/
static long my_ioctl(struct file *file, unsigned int command, unsigned long value)
{
        switch(command){

                case IOCTL_CMD0:
                printk(KERN_INFO "I am CMD0\n");
                        break;

                case IOCTL_CMD1:
                printk(KERN_INFO "I am CMD1\n");
                        break;

                case IOCTL_CMD2:
                        printk(KERN_INFO "I am CMD2\n");
                        break;

                default:
                        printk(KERN_INFO "I am default");
                        break;
                }

        return 0;
}
/*====================================================================*/




/*------------------------------------------------------------------*/
/*__________________________________Timer________________________________*/
void my_timer_callback( unsigned long data )
{
```

```
            mod_timer(&my_timer, jiffies + msecs_to_jiffies(2000));/* setup timer interval to 2000 msecs */
     printk(KERN_ALERT "2 seconds Gone \n");

}
/*========================================================================*/

/*---------------------------------------------------------------------------*/
/*______________________________Interrupt Handler_________________________________*/

static irqreturn_t kbd2_isr(int irq, void *dev_id)
{
   char scancode;
   char key1=0;
   scancode = inb(KBD_DATA_REG);
   if((int)scancode==30){key1='A';}
   else if((int)scancode==48){key1='B';}
   else if((int)scancode==46){key1='C';}
   else if((int)scancode==32){key1='D';}
   else if((int)scancode==18){key1='E';}
   else if((int)scancode==33){key1='F';}
   else if((int)scancode==34){key1='G';}
   else if((int)scancode==35){key1='H';}
   else if((int)scancode==23){key1='I';}
   else if((int)scancode==36){key1='J';}
   else if((int)scancode==37){key1='K';}
   else if((int)scancode==38){key1='L';}
   else if((int)scancode==50){key1='M';}
   else if((int)scancode==49){key1='N';}
   else if((int)scancode==24){key1='O';}
   else if((int)scancode==25){key1='P';}
   else if((int)scancode==16){key1='Q';}
   else if((int)scancode==19){key1='R';}
   else if((int)scancode==31){key1='S';}
   else if((int)scancode==20){key1='T';}
   else if((int)scancode==22){key1='U';}
   else if((int)scancode==47){key1='V';}
   else if((int)scancode==17){key1='W';}
   else if((int)scancode==45){key1='X';}
   else if((int)scancode==21){key1='Y';}
   else if((int)scancode==44){key1='z';}
   else if((int)scancode==14){key1='<';}
   else if((int)scancode==28){key1='|';}
   else if((int)scancode==57){key1='_';}
   else if((int)scancode==57){key1=':';}
   /* NOTE: i/o ops take a lot of time thus must be avoided in HW ISRs */
   //just like printk included with KERN_ALERT
  //pr_info("Scan Code %c %s\n", key1 & KBD_SCANCODE_MASK,scancode & KBD_STATUS_MASK ?
                                                                                "Released" : "Pressed");
            pr_info("Scan Code %c \n", key1 & KBD_SCANCODE_MASK);

   return IRQ_HANDLED;
```

```
}
/*==============================================================================*/




/*-----------------------------------------------------------------------------*/
/*_____________________________Driver  Initialization_______________________*/
int Start(void)
{
    // Try to dynamically allocate a major number for the device -- more difficult but worth it
    majorNumber = register_chrdev(0, DEVICE_NAME, &my_fops); //auto allocate major num
    if (majorNumber<0) {
      printk(KERN_ALERT "char driver failed to register a major number\n");
      return majorNumber;
    }
    printk(KERN_INFO "char driver: Major number %d\n", majorNumber);

    // Register the device class
    CharClass = class_create(THIS_MODULE, CLASS_NAME);
    if (IS_ERR(CharClass)){                // Check for error and clean up if there is
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to register device class\n");
        return PTR_ERR(CharClass);          // Correct way to return an error on a pointer
    }

    // Register the device driver
    CharDevice = device_create(CharClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);
    if (IS_ERR(CharDevice)){               // Clean up if there is an error
        class_destroy(CharClass);           // Repeated code but the alternative is goto statements
        unregister_chrdev(majorNumber, DEVICE_NAME);
        printk(KERN_ALERT "Failed to create the device\n");
        return PTR_ERR(CharDevice);
    }
    printk(KERN_INFO "char driver: device class created correctly\n"); // Made it! device was initialized

    /*_________________Registering an Interrupt Handler_________________*/
    request_irq(KBD_IRQ, kbd2_isr, IRQF_SHARED, "kbd2", (void *)kbd2_isr);

    /*_________________Setting an Timer Callback event_________________*/
    setup_timer(&my_timer, my_timer_callback, 0);/* setup your timer to call my_timer_callback */
    mod_timer(&my_timer, jiffies + msecs_to_jiffies(200));/* setup timer interval to 200 msecs */

    return 0;
}
/*=============================================================================*/

/*------------------------------------------------------------------------------------------------------------------------*/
/*_______________________________Driver  Disposing__________________________*/

void Dispose(void)
{
        device_destroy(CharClass, MKDEV(majorNumber, 0));     // remove the device
        class_unregister(CharClass);                          // unregister the device class
        class_destroy(CharClass);                             // remove the device class
        unregister_chrdev(majorNumber, DEVICE_NAME);          // unregister the major number
        printk(KERN_INFO "TEST_DRIVER char driver: Goodbye \n");
        free_irq(KBD_IRQ, (void *)kbd2_isr);
        del_timer(&my_timer);/* remove kernel timer when unloading module*/
}
/*==================================================================*/



module_init( Start );
module_exit( Dispose );
```

OUTPUTS:

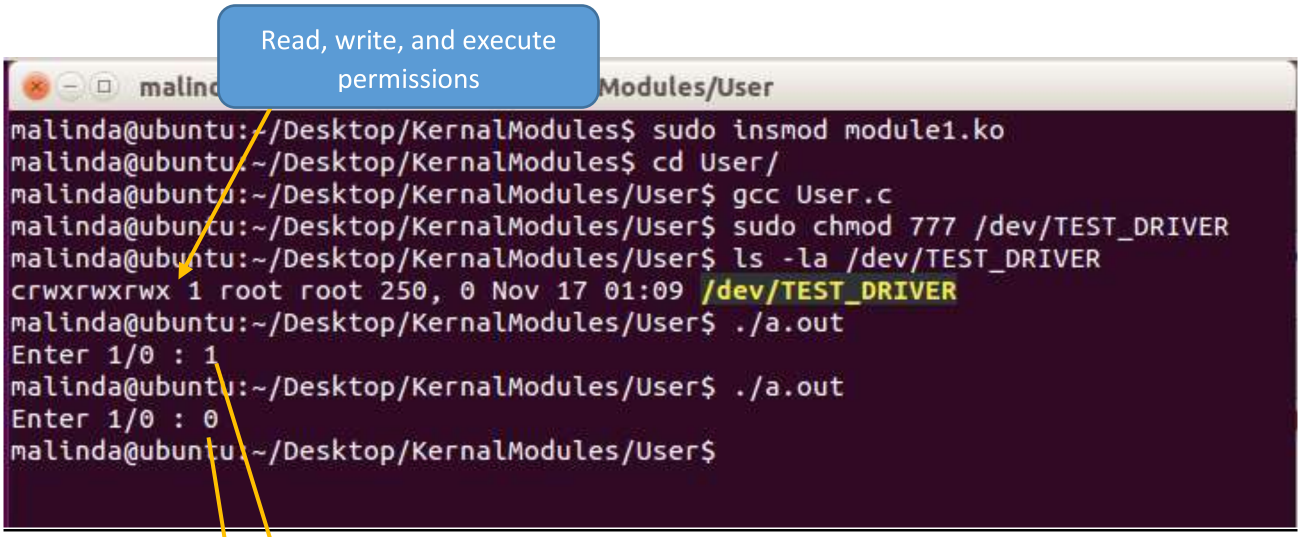


malinda@ubuntu: ~/Desktop/KernalModules
Search your computer and online sources
Firefox Web Browser

```
[ 1087.776381] char driver: device class created correctly
[ 1087.899678] Scan Code
[ 1087.972841] 2 seconds Gone
[ 1089.115642] Scan Code S
[ 1089.259646] Scan Code
[ 1089.974993] 2 seconds Gone
[ [illegible]] Scan Code <
[ 1090.867618] Scan Code
[ 1091.976702] 2 seconds Gone
[ 1093.978009] 2 seconds Gone
[ 1095.980347] 2 seconds Gone
[ 1096.067621] Scan Code D
[ 1096.179624] Scan Code
[ 1096.595632] Scan Code M
[ 1096.675480] Scan Code
[ 1096.787565] Scan Code E
[ 1096.851520] Scan Code
[ 1097.043599] Scan Code S
[ 1097.139624] Scan Code
[ 1097.771641] Scan Code G
[ 1097.851574] Scan Code
[ 1097.982523] 2 seconds Gone
[ 1098.235519] Scan Code |
malinda@ubuntu:~/Desktop/KernalModules$
```

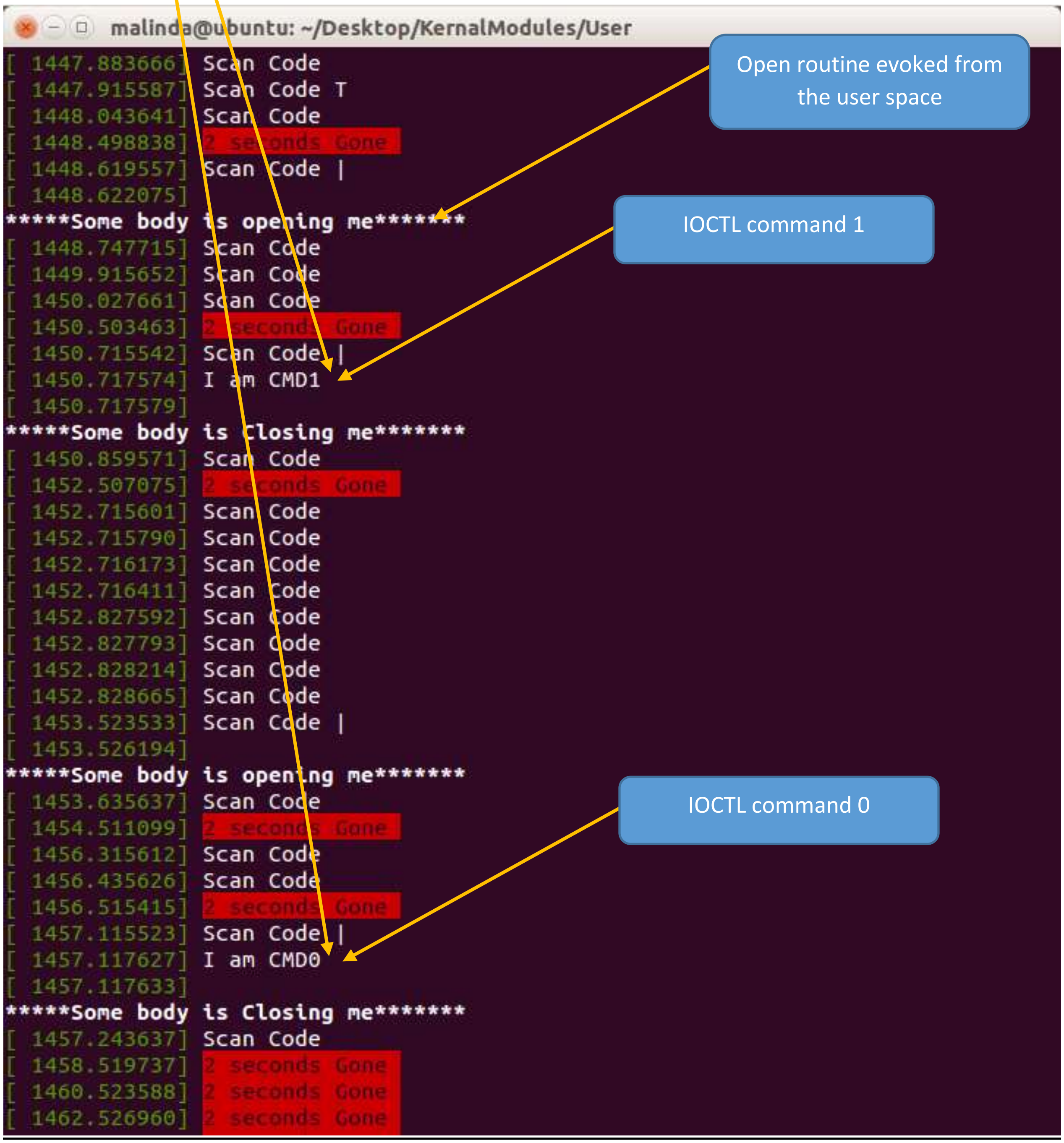
Timer callback event

Keyboard press interrupt



Read, write, and execute permissions

```
malinda@ubuntu:~/Desktop/KernalModules$ sudo insmod module1.ko
malinda@ubuntu:~/Desktop/KernalModules$ cd User/
malinda@ubuntu:~/Desktop/KernalModules/User$ gcc User.c
malinda@ubuntu:~/Desktop/KernalModules/User$ sudo chmod 777 /dev/TEST_DRIVER
malinda@ubuntu:~/Desktop/KernalModules/User$ ls -la /dev/TEST_DRIVER
crwxrwxrwx 1 root root 250, 0 Nov 17 01:09 /dev/TEST_DRIVER
malinda@ubuntu:~/Desktop/KernalModules/User$ ./a.out
Enter 1/0 : 1
malinda@ubuntu:~/Desktop/KernalModules/User$ ./a.out
Enter 1/0 : 0
malinda@ubuntu:~/Desktop/KernalModules/User$
```



malinda@ubuntu: ~/Desktop/KernalModules/User

Open routine evoked from the user space

IOCTL command 1

IOCTL command 0

```
[ 1447.883666] Scan Code
[ 1447.915587] Scan Code T
[ 1448.043641] Scan Code
[ 1448.498838] 2 seconds Gone
[ 1448.619557] Scan Code |
[ 1448.622075]
*****Some body is opening me*******
[ 1448.747715] Scan Code
[ 1449.915652] Scan Code
[ 1450.027661] Scan Code
[ 1450.503463] 2 seconds Gone
[ 1450.715542] Scan Code |
[ 1450.717574] I am CMD1
[ 1450.717579]
*****Some body is Closing me*******
[ 1450.859571] Scan Code
[ 1452.507075] 2 seconds Gone
[ 1452.715601] Scan Code
[ 1452.715790] Scan Code
[ 1452.716173] Scan Code
[ 1452.716411] Scan Code
[ 1452.827592] Scan Code
[ 1452.827793] Scan Code
[ 1452.828214] Scan Code
[ 1452.828665] Scan Code
[ 1453.523533] Scan Code |
[ 1453.526194]
*****Some body is opening me*******
[ 1453.635637] Scan Code
[ 1454.511099] 2 seconds Gone
[ 1456.315612] Scan Code
[ 1456.435626] Scan Code
[ 1456.515415] 2 seconds Gone
[ 1457.115523] Scan Code |
[ 1457.117627] I am CMD0
[ 1457.117633]
*****Some body is Closing me*******
[ 1457.243637] Scan Code
[ 1458.519737] 2 seconds Gone
[ 1460.523588] 2 seconds Gone
[ 1462.526960] 2 seconds Gone
```

Compiling the kernel Module and Makefile

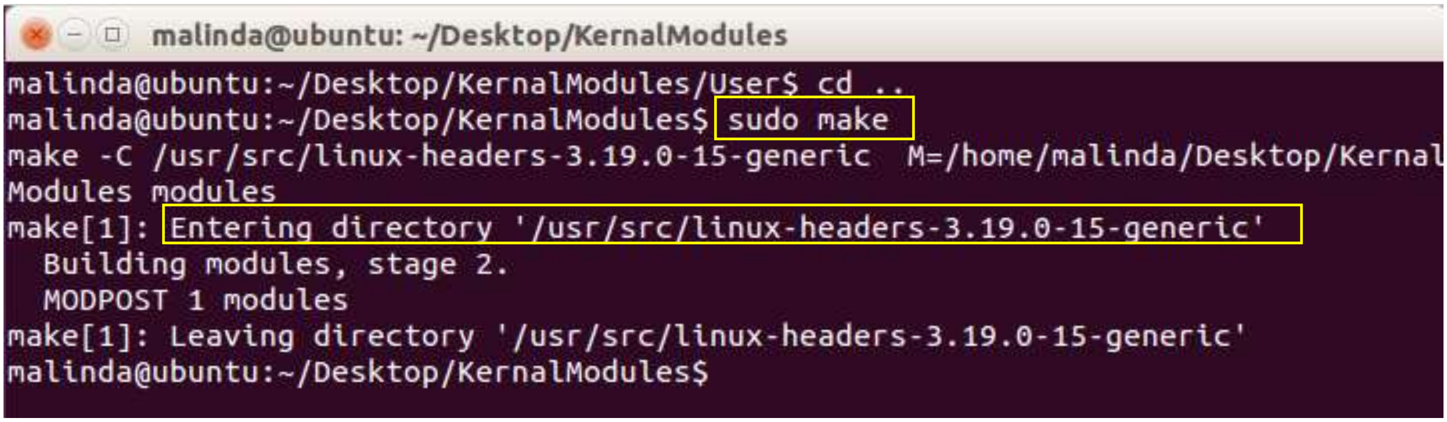
In order to compile the kernel module doing " gcc module.c" simply doesn't work. Because to build the kernel module it needs special header files, which we called as kernel headers. And especially these headers should be compatible with the relevant base kernel which you are going to plug your modules into.

What you have to do is, simply make a "Makefile" as follows

```
obj-m += module1.o


KDIR :=/usr/src/linux-headers-3.19.0-15-generic

PWD:= $(shell pwd)

default:

		$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:

		$(MAKE) -C $(KDIR) M=$(PWD) clean
```

Here KDIR means the directory where your kernel headers are located
PWD is where your kernel module exist (i.e module.c)
Thereafter what you have to do is type sudo make in the terminal
What it does is it will find a file named as (Makefile) inside the current directory and compile it

So, if the compilation is success you will get new bunch of files one having a name "<module.ko>". And that is your kernel object file. Now you just have to plug it into the kernel and test for it



```
malinda@ubuntu:~/Desktop/KernalModules/User$ cd ..
malinda@ubuntu:~/Desktop/KernalModules$ sudo make
make -C /usr/src/linux-headers-3.19.0-15-generic  M=/home/malinda/Desktop/Kernal
Modules modules
make[1]: Entering directory '/usr/src/linux-headers-3.19.0-15-generic'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-3.19.0-15-generic'
malinda@ubuntu:~/Desktop/KernalModules$
```

When compiling the user program you just have type "gcc User.c" and it will create a file "a.out"
Thereafter to run the file as usual
Just type "./a.out "

Things you should get into know about (Terminal tips)

- The node is create/created in the , /dev/< node_name >
- Make node manually sudo mknod /dev/<node_name> c majorNo MinorNo
- Taking read, write and execute permission sudo chmod 777 /dev/<node_name>
- Look into the kernel space messages dmesg
- Go to previous Directory cd ..
- List files ls
- Inserting a module sudo insmod <module.ko>
- Remove a module sudo rmmod <module>
- Check available modules lsmod