RATHINAM TECHNICAL CAMPUS

RATHINAM TECHZONE

POLLACHI MAIN ROAD, EACHANARI, COIMBATORE-641021.

MASTER OF COMPUTER APPLICATION

RECORD NOTE BOOK

23MCP21 EMBEDDED LINUX DEVICE DRIVES ON RUGGED BOARD LABORATORY

NAME :

REGISTER NUMBER :

YEAR/SEMESTER :

ACADEMIC YEAR :

RATHINAM TECHNICAL CAMPUS

RATHINAM TECHZONE

POLLACHI MAIN ROAD, EACHANARI, COIMBATORE-641021.

BONAFIDE CERTIFICATE

NAME :

ACADEMIC YEAR :

YEAR/SEMESTER :

BRANCH :

UNIVERSITY REGISTER NUMBER: …….

Certified that this is the bonafide record of work done by the above student in the

_____Laboratory during the year 2024-2025.

Head of the Department **Staff-in-Charge**

Submitted for the Practical Examination held on _____

Internal Examiner **External Examiner**

INDEX

S.No	Date	Experiment Name	Marks	Staff Sign

INDEX

S.No	Date	Experiment Name	Marks	Staff Sign

EX NO: 01 DATE:	Compile and Load a Simple Hello World Module

AIM:

To compile and load a simple "Hello World" Linux kernel module to understand the basics of writing and loading kernel modules.

ALGORITHM:

STEP 1: Start the process.

STEP 2: Write the Kernel Module Code:

- Create a .c file, e.g., hello_world.c.
- Write the module initialization function hello_init to print "Hello, World!" when the module is loaded.
- Write the module exit function hello_exit to print "Goodbye, World!" when the module is unloaded.
- Include module_init and module_exit macros to register the functions.
- Add module information (author, license, description).

STEP 3: Create a Makefile:

- Create a Makefile in the same directory as your .c file.
- Define obj-m += hello_world.o to specify the object file to be compiled.
- Set the kernel directory path (KDIR).
- Write rules for compiling (make) and cleaning (make clean) the module.

STEP 4: Open Terminal:

Navigate to the directory where the hello_world.c and Makefile are located.

STEP 5: Compile the Kernel Module:

Run the make command to compile the module and generate the .ko file.

STEP 6: Insert (Load) the Module:

Use sudo insmod hello_world.ko to load the module into the Linux kernel.

STEP 7: Check the Kernel Logs:

Run dmesg | tail to verify that "Hello, World!" was printed in the kernel log after the module was loaded.

STEP 8: Remove (Unload) the Module:

Use sudo rmmod hello_world to unload the module from the kernel.

STEP 9: Check the Kernel Logs Again:

Run dmesg | tail to confirm that "Goodbye, World!" was printed when the module was unloaded.

STEP 10: Clean Up:

Run make clean to remove generated files (e.g., .o, .ko).

STEP 11: End the process.

PROGRAM:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
/* Init function - called when module is loaded */
static int __init hello_init(void)
{
   printk(KERN_INFO " Hello, World! Kernel Module Loaded.\n");
   return 0;
}
/* Exit function - called when module is removed */
static void __exit hello_exit(void)
{
   printk(KERN_INFO " Goodbye, World! Kernel Module Unloaded.\n");
}
/* Registering init and exit functions */
module_init(hello_init);
module_exit(hello_exit);
```

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("A Simple Hello World Kernel Module");
```

Makefile:

```
obj-m += hello_world.o
# Kernel directory for the host (your development machine)
KDIR = /lib/modules/$(shell uname -r)/build
# Kernel directory for the target (Rugged board)
TDIR = /home/deva/RuggedBoard/RB-A5D2x/SD_card_boot/Kernel_source/linux-rba5d2x
# Default target: Build the module for the host system
host:
	make -C $(KDIR) M=$(shell pwd) modules
# Build the module for the target system (Rugged board)
target:
	make -C $(TDIR) M=$(shell pwd) modules
# Clean the build files
clean:
	make -C $(KDIR) M=$(shell pwd) clean
```

OUTPUT:

```
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ make
make -C /lib/modules/5.15.0-67-generic/build M=/home/deva/Code/LDD_lab/Ex_1 modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-67-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
  You are using:           gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
  CC [M]  /home/deva/Code/LDD_lab/Ex_1/hello_world.o
  MODPOST /home/deva/Code/LDD_lab/Ex_1/Module.symvers
  CC [M]  /home/deva/Code/LDD_lab/Ex_1/hello_world.mod.o
  LD [M]  /home/deva/Code/LDD_lab/Ex_1/hello_world.ko
  BTF [M] /home/deva/Code/LDD_lab/Ex_1/hello_world.ko
Skipping BTF generation for /home/deva/Code/LDD_lab/Ex_1/hello_world.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-67-generic'
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ file hello_world.ko
hello_world.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), BuildID[sha1]=816c24d37abd5d2c53e2d4b721ed7efc3451f0e6, with debug_info, not stripped
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ sudo insmod hello_world.ko
[sudo] password for deva:
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ dmesg | tail
[  212.213727] usb 3-8: Manufacturer: Realtek USB2.0 Finger Print Bridge
[  233.798509] usb 3-8: USB disconnect, device number 14
[  234.373348] usb 3-8: new high-speed USB device number 15 using xhci_hcd
[  234.527473] usb 3-8: New USB device found, idVendor=2808, idProduct=a658, bcdDevice= 1.10
[  234.527485] usb 3-8: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[  234.527488] usb 3-8: Product: FocalTech Fingerprint Device
[  234.527491] usb 3-8: Manufacturer: Realtek USB2.0 Finger Print Bridge
[  386.095310] hello_world: loading out-of-tree module taints kernel.
[  386.095370] hello_world: module verification failed: signature and/or required key missing - tainting kernel
[  386.096098]  Hello, World! Kernel Module Loaded.
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ sudo rmmod hello_world
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$
```

```
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ . /opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ make target
make -C /home/deva/RuggedBoard/RB-A5D2x/SD_card_boot/Kernel_source/linux-rba5d2x M=/home/deva/Code/LDD_lab/Ex_1 modules
make[1]: Entering directory '/home/deva/RuggedBoard/RB-A5D2x/SD_card_boot/Kernel_source/linux-rba5d2x'
  CC [M]  /home/deva/Code/LDD_lab/Ex_1/hello_world.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/deva/Code/LDD_lab/Ex_1/hello_world.mod.o
  LD [M]  /home/deva/Code/LDD_lab/Ex_1/hello_world.ko
make[1]: Leaving directory '/home/deva/RuggedBoard/RB-A5D2x/SD_card_boot/Kernel_source/linux-rba5d2x'
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ file hello_world.ko
hello_world.ko: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), BuildID[sha1]=0c8a8eb958eeccbb2210166aa2ac87bcd2c5981d, not stripped
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ cp hello_world.ko /var/lib/tftpboot/
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$ ls /var/lib/tftpboot/
hello_world.ko
deva@mrdevanagalingam:~/Code/LDD_lab/Ex_1$
```

```
root@rugged-board-a5d2x-sd1:/data# tftp -r hello_world.ko -g 192.168.1.26
root@rugged-board-a5d2x-sd1:/data# ls
hello_world.ko
root@rugged-board-a5d2x-sd1:/data# chmod 777 hello_world.ko
root@rugged-board-a5d2x-sd1:/data# insmod hello_world.ko
hello_world: loading out-of-tree module taints kernel.
 Hello, World! Kernel Module Loaded.
root@rugged-board-a5d2x-sd1:/data# dmesg | tail
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chan5 for rx DMA transfers
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chan6 for tx DMA transfers
IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
macb f8008000.ethernet eth0: link up (10/Full)
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
random: crng init done
macb f8008000.ethernet eth0: link down
macb f8008000.ethernet eth0: link up (10/Full)
hello_world: loading out-of-tree module taints kernel.
 Hello, World! Kernel Module Loaded.
root@rugged-board-a5d2x-sd1:/data# rmmod hello_world
 Goodbye, World! Kernel Module Unloaded.
root@rugged-board-a5d2x-sd1:/data# dmesg | tail
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chan6 for tx DMA transfers
IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
macb f8008000.ethernet eth0: link up (10/Full)
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
random: crng init done
macb f8008000.ethernet eth0: link down
macb f8008000.ethernet eth0: link up (10/Full)
hello_world: loading out-of-tree module taints kernel.
 Hello, World! Kernel Module Loaded.
 Goodbye, World! Kernel Module Unloaded.
root@rugged-board-a5d2x-sd1:/data#
```

RESULT:

EX NO: 02 DATE:	**Experiment with Simple Timer Delays.**

AIM:

To Experiment with Simple Timer Delays.

ALGORITHM:

STEP 1: We choose 3 timer delays 1.)Busy-Wait Delay (mdelay).

2.)Sleep Delay (msleep).

3.) High-Resolution Timer(hrtimer) .

STEP 2: Write a C program file named as **ldd_exp2.c** for simple timer delays that have been mentioned above.

STEP 3: Then find or navigate to the manual compilation folder from github for RUGGED BOARD A5D2X and get the path for put into the Makefile or compile Purpose.

STEP 4: Then Make a file named as **Makefile** for compilation purpose.

STEP 5: In that make file give **obj m += ldd_exp2.o** we can also add for **make -C** target and host kernel's version path and M=present work directory.

STEP 6: Then enable the POKY-TINY TOOLCHAIN for 32 bit machine compilation because our RUGGED BOARD A5D2X is 32 bit machine.

STEP 7: Command to enable the 'poky-tiny' toolchain is **". /opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi".**

STEP 8: After enable the POKY-TINY toolchain give command **"make target"** for compilation for Rugged board A5D2X.

STEP 9: Then we will get **kernel object** file with extension **".ko".**

STEP 10: After the above step:09 is completed we will naavigate to the **MANUAL COMPILED linux-rba5d2x** folder and get the 2 binary files.

1.] /home/rameshchlm/BSP_BOOT/linux-rba5d2x/arch/arm/boot/dts/a5d2x-rugged_board.dtb **[.dtb] [device tree blob]**.

2.] /home/rameshchlm/BSP_BOOT/linux-rba5d2x/arch/arm/boot/zimage **zimage file.**

STEP 11: By step: 10 We will get a 2 files put that files into a rugged board a5d2x sd card's Boot Partition.

STEP 12: Then Boot the RB-A5D2X Board and **transfer** the 32 bit file name ldd_exp2.ko from **host machine** to the **target machine** RB-A5D2X board by TFTP.

STEP 13: And then receive that host transferred .ko file by **TFTP .**

STEP 14: After sucessfully received the kernel object file on the target machine RB-A5D2X we need to **insert the module** by using command **"insmod ldd_exp2.ko".**

STEP 15: After the above step is completed give command **"dmesg"** to see the output on the kernel.

PROGRAM:

ldd_exp2.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/delay.h>        // For mdelay, msleep
#include <linux/hrtimer.h>      // For hrtimer
#include <linux/ktime.h>
static struct hrtimer my_hrtimer;
ktime_t kt;
// Timer callback function
enum hrtimer_restart my_hrtimer_callback(struct hrtimer *timer)
{
   pr_info("High-resolution timer callback called\n");
   return HRTIMER_NORESTART;  // Prevent the timer from restarting
}
static int __init my_driver_init(void)
{
   pr_info("Initializing Timer Delay Driver\n");
   // Busy wait for 1 millisecond
   pr_info("Starting mdelay for 1 ms\n");
   mdelay(1);
   pr_info("mdelay complete\n");
```

```
    // Sleep for 10 milliseconds
    pr_info("Starting msleep for 10 ms\n");
    msleep(10);
    pr_info("msleep complete\n");
    // Set up high-resolution timer for 100 ms delay
    pr_info("Setting up high-resolution timer for 100 ms\n");
    kt = ktime_set(0, 100 * 1000000); // 100 ms in nanoseconds
    hrtimer_init(&my_hrtimer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    my_hrtimer.function = &my_hrtimer_callback;
    hrtimer_start(&my_hrtimer, kt, HRTIMER_MODE_REL);
    return 0;
}
static void __exit my_driver_exit(void)
{
    // Cancel the high-resolution timer
    pr_info("Exiting Timer Delay Driver\n");
    hrtimer_cancel(&my_hrtimer);
    pr_info("High-resolution timer canceled\n");
}
module_init(my_driver_init);
module_exit(my_driver_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("BY MCA");
MODULE_DESCRIPTION("Simple Timer Delay Example Driver for RBA5D2X");
MODULE_VERSION("1.0");
```

Makefile

```
obj-m += ldd_exp2.o
KDIR := /lib/modules/5.15.0-122-generic/build
TDIR := /home/rameshchlm/BSP_BOOT/linux-rba5d2x
```

host:

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

target:

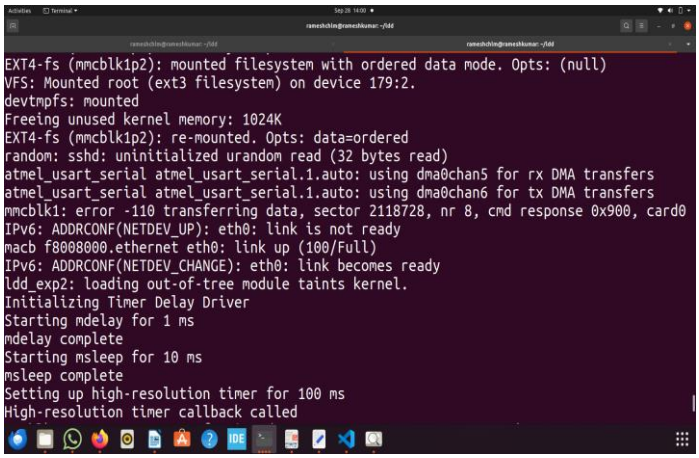
make -C $(TDIR) M=$(PWD) modules

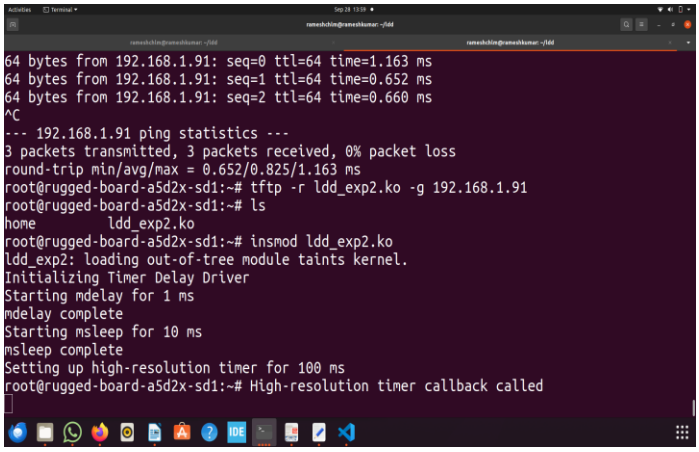
clean:

make -C $(TDIR) M=$(PWD) clean

make -C $(KDIR) M=$(PWD) clean

OUTPUT:





RESULT:

EX NO: 03 DATE:	**Experiment with Kernel Memory Allocation Using kmalloc and kfree.**

AIM:

To create a Experiment with Kernel Memory Allocation Using kmalloc and kfree

ALGORITHM:

STEP 1: Start the process.

STEP 2: Create a directory as Kmalloc to save your files.

STEP 3: Inside the Kmalloc directory, Create a file as kmalloc_example.c

STEP 4: After creating the kmalloc_example.c file, enable the poky tiny cross compilation

$. /opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi

STEP 5: Download the toolchain for the Rugged Board,

$ **git clone https://github.com/rugged-board/linux-rba5d2x.git**

$ make defconfig

$ make

STEP 6: Compile the kmalloc_example.c file, by using the Toolchain by using the toolchain directory.

STEP 7: Then copy the kmalloc_example.ko file to tftp location,

$ /var/lib/tftpboot/

STEP 8: Connect the Rugged Board by using minicom, then get the file from the host,

 # tftp -r 192.168.1.30 -g kmalloc_example.ko

STEP 9: And finally insert the Module in Kernal of the Rugged Board,

 # insmod kmalloc_example.ko

STEP 10: Stop the process.

PROGRAM:

kmalloc_exaample.c

```
#include <linux/init.h>        // Needed for module initialization
#include <linux/module.h>      // Needed for all modules
#include <linux/kernel.h>      // Needed for kernel info
#include <linux/slab.h>        // Needed for kmalloc and kfree

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Rathinavel Pandian");
MODULE_DESCRIPTION("A simple example of kmalloc and kfree");

static int __init my_module_init(void) {
   char *my_memory;

   // Allocate 100 bytes of memory
   my_memory = kmalloc(100, GFP_KERNEL);
   if (!my_memory) {
      printk(KERN_ALERT "Memory allocation failed!\n");
      return -ENOMEM;
   }

   printk(KERN_INFO "Memory allocated successfully at address: %p\n", my_memory);

   // Use the allocated memory
   snprintf(my_memory, 100, "Hello from kmalloc!");

   printk(KERN_INFO "Stored string: %s\n", my_memory);

   // Free the allocated memory
   kfree(my_memory);
   printk(KERN_INFO "Memory freed successfully.\n");

   return 0;
}

static void __exit my_module_exit(void) {
   printk(KERN_INFO "Module exiting.\n");
}

module_init(my_module_init);
module_exit(my_module_exit);
```

Makefile

```
obj-m += kmalloc_example.o

KDIR := /home/rathinavel/boot/kernel_source/linux-rba5d2x

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

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

OUTPUT:

```
root@rugged-board-a5d2x-sd1:/data# ls
hello.ko            kmalloc_example.ko
root@rugged-board-a5d2x-sd1:/data# insmod kmalloc_example.ko
kmalloc_example: loading out-of-tree module taints kernel.
Memory allocated successfully at address: c3b5a280
Stored string: Hello from kmalloc!
Memory freed successfully.
root@rugged-board-a5d2x-sd1:/data# dmesg | tail
Freeing unused kernel memory: 1024K
EXT4-fs (mmcblk1p2): re-mounted. Opts: data=ordered
random: sshd: uninitialized urandom read (32 bytes read)
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chan5 for rx DMA transfers
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chan6 for tx DMA transfers
random: crng init done
kmalloc_example: loading out-of-tree module taints kernel.
Memory allocated successfully at address: c3b5a280
Stored string: Hello from kmalloc!
Memory freed successfully.
root@rugged-board-a5d2x-sd1:/data#
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB0
```

RESULT:

EX NO: 04 DATE:	Write a SYSFS based Linux Device Driver

AIM:

To implement a device driver based on SYSFS.

ALGORITHM:

Step 1: Write the code hello.c.

Step 2: Create the Makefile in the same directory.

Step 3: Build the module using make host (for host) or make target (for target).

Step 4: Load the module with sudo insmod hello.ko.

Step 5: Check kernel logs using dmesg | tail.

Step 6: Verify device creation in /dev/etx_device and/sys/kernel/etx_sysfs.

Step 7: Test sysfs read/write using cat and echo commands.

Step 8: Unload the module with sudo rmmod hello and clean up using make clean.

PROGRAM:

SOURCE CODE:

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

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kdev_t.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/device.h>

#include<linux/slab.h>                 //kmalloc()

#include<linux/uaccess.h>              //copy_to/from_user()

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

```
#include<linux/kobject.h>
 #include <linux/err.h>
 volatile int etx_value = 0;
 dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;
struct kobject *kobj_ref;
/*
** Function Prototypes
*/
static int      __init etx_driver_init(void);
static void     __exit etx_driver_exit(void);
 /***** Driver functions ********/
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t  etx_read(struct file *filp,
                   char __user *buf, size_t len,loff_t * off);
static ssize_t  etx_write(struct file *filp,
                   const char *buf, size_t len, loff_t * off);
 /***** Sysfs functions ********/
static ssize_t  sysfs_show(struct kobject *kobj,
                   struct kobj_attribute *attr, char *buf);
static ssize_t  sysfs_store(struct kobject *kobj,
                   struct kobj_attribute *attr,const char *buf, size_t count);
struct kobj_attribute etx_attr = __ATTR(etx_value, 0660, sysfs_show, sysfs_store);
```

```
/*
** File operation sturcture
*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .release        = etx_release,
};
/*
** This function will be called when we read the sysfs file
*/
static ssize_t sysfs_show(struct kobject *kobj,
                struct kobj_attribute *attr, char *buf)
{
        pr_info("Sysfs - Read!!!\n");
        return sprintf(buf, "%d", etx_value);
}
/*
** This function will be called when we write the sysfsfs file
*/
static ssize_t sysfs_store(struct kobject *kobj,
                struct kobj_attribute *attr,const char *buf, size_t count)
{
```

```
        pr_info("Sysfs - Write!!!\n");
        sscanf(buf,"%d",&etx_value);
        return count;
}
/*
** This function will be called when we open the Device file
*/
static int etx_open(struct inode *inode, struct file *file)
{
        pr_info("Device File Opened...!!!\n");
        return 0;
}
/*
** This function will be called when we close the Device file
*/
static int etx_release(struct inode *inode, struct file *file)
{
        pr_info("Device File Closed...!!!\n");
        return 0;
}
 /*
** This function will be called when we read the Device file
*/
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
```

```
        pr_info("Read function\n");
        return 0;
}
/*
** This function will be called when we write the Device file
*/
static ssize_t etx_write(struct file *filp,
                const char __user *buf, size_t len, loff_t *off)
{
        pr_info("Write Function\n");
        return len;
}
/*
** Module Init function
*/
static int __init etx_driver_init(void)
{
        /Allocating Major number/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                pr_info("Cannot allocate major number\n");
                return -1;
        }
        pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
         /Creating cdev structure/
        cdev_init(&etx_cdev,&fops);
```

```
        /Adding character device to the system/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            pr_info("Cannot add the device to the system\n");
            goto r_class;
        }
        /Creating struct class/
        if(IS_ERR(dev_class = class_create(THIS_MODULE,"etx_class"))){
            pr_info("Cannot create the struct class\n");
            goto r_class;
        }
        /Creating device/
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){
            pr_info("Cannot create the Device 1\n");
            goto r_device;
        }
        /*Creating a directory in /sys/kernel/ */
        kobj_ref = kobject_create_and_add("etx_sysfs",kernel_kobj);
        /Creating sysfs file for etx_value/
        if(sysfs_create_file(kobj_ref,&etx_attr.attr)){
                pr_err("Cannot create sysfs file......\n");
                goto r_sysfs;
      }
        pr_info("Device Driver Insert...Done!!!\n");
        return 0;
    r_sysfs:
        kobject_put(kobj_ref);
```

```
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
 r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        cdev_del(&etx_cdev);
        return -1;
}
/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
{
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &etx_attr.attr);
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}
 module_init(etx_driver_init);
module_exit(etx_driver_exit);
 MODULE_LICENSE("GPL");
MODULE_AUTHOR("RuggedBoard");
MODULE_DESCRIPTION("Simple Linux device driver (sysfs)");
```

MODULE_VERSION("1.0");

Makefile

obj-m += hello.o

KDIR = /lib/modules/$(shell uname -r)/build

TDIR = /home/rameshchlm/BSP_BOOT/linux-rba5d2x

host:

	make -C $(KDIR) M=$(shell pwd) modules

target:

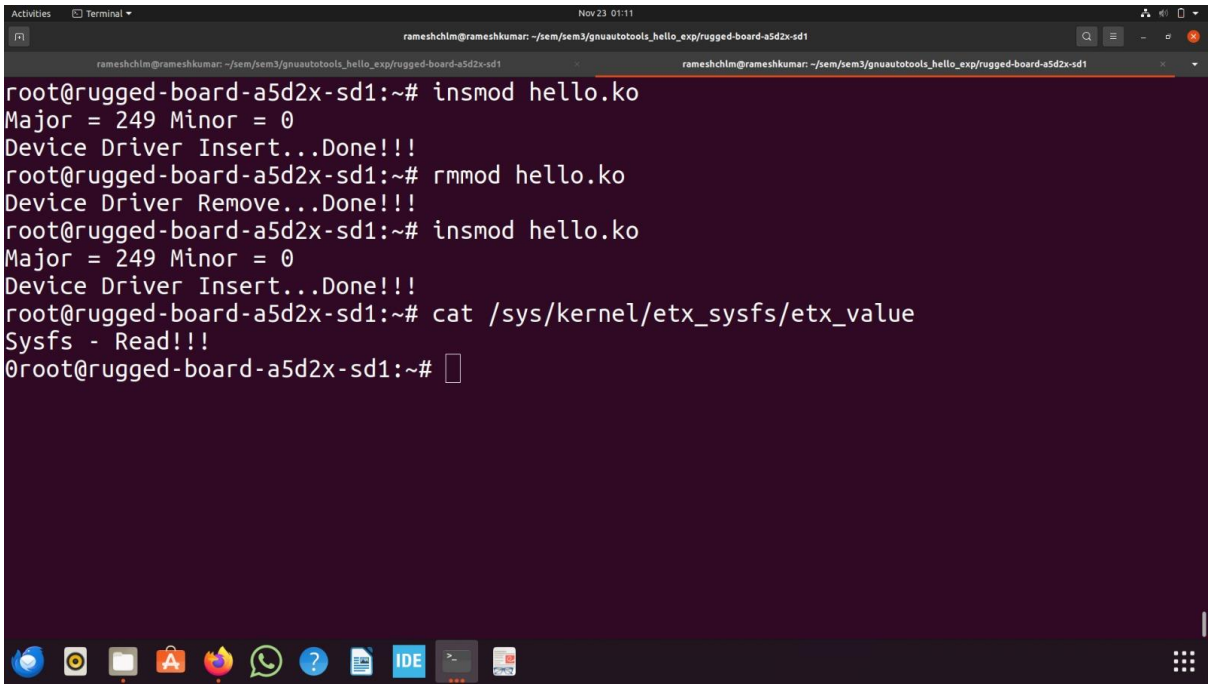
	make -C $(TDIR) M=$(shell pwd) modules

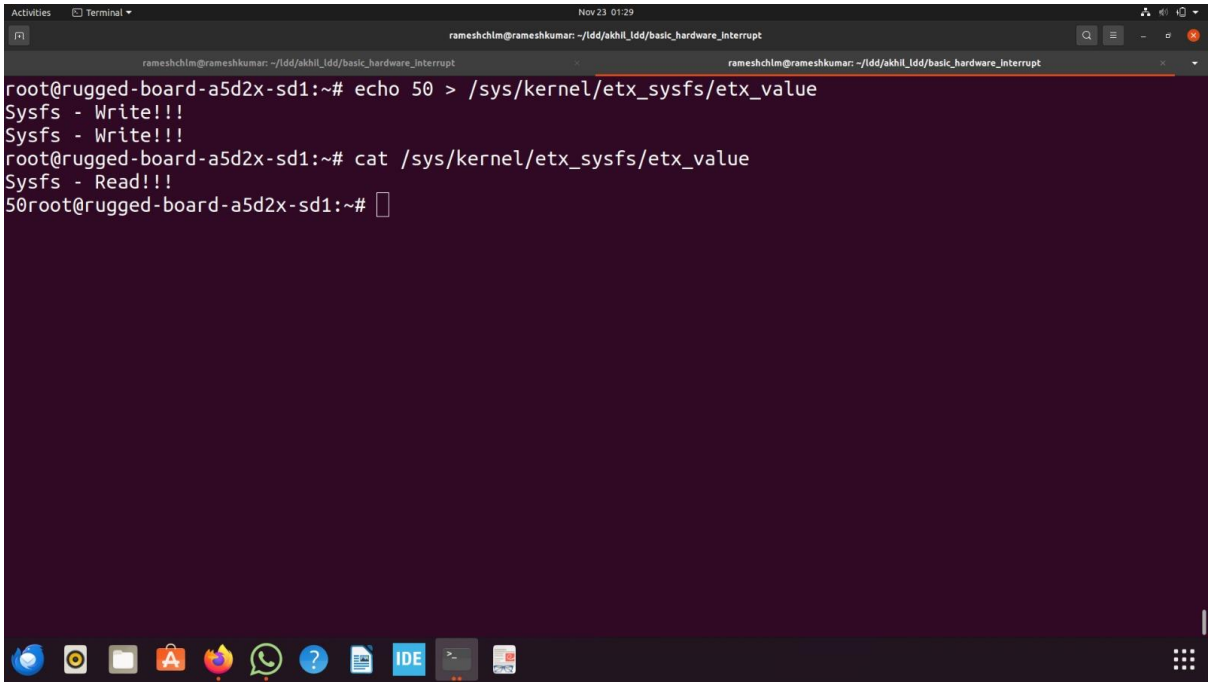
clean:

	make -C $(KDIR) M=$(shell pwd) clean

	make -C $(TDIR) M=$(shell pwd) clean

OUTPUT:




Activities
Terminal
Nov 23 01:29
rameshchlm@rameshkumar: ~/ldd/akhil_ldd/basic_hardware_interrupt
rameshchlm@rameshkumar: ~/ldd/akhil_ldd/basic_hardware_interrupt
rameshchlm@rameshkumar: ~/ldd/akhil_ldd/basic_hardware_interrupt
root@rugged-board-a5d2x-sd1:~# echo 50 > /sys/kernel/etx_sysfs/etx_value
Sysfs - Write!!!
Sysfs - Write!!!
root@rugged-board-a5d2x-sd1:~# cat /sys/kernel/etx_sysfs/etx_value
Sysfs - Read!!!
50root@rugged-board-a5d2x-sd1:~#
IDE

RESULT:

EX NO: 05 DATE:	**Experiment with Simple Character Device Operations**

AIM:

To experiment with Simple Character Device Operations

ALGORITHM:

STEP 1: Create the Character Device Driver Source File exp5_ldd.c

This driver includes standard character device operations (open, read, write, ioctl, and release) and supports mutex synchronization, dynamic allocation of device buffers, and permissions setup with uevent.

STEP 2: Navigate to the Manual Compilation Directory

Go to the folder where you plan to compile the driver, which should contain your kernel source code for the Rugged Board A5D2X.

STEP 3: Write the Makefile for Compilation

Create a Makefile for compiling exp5_ldd.c to generate a kernel module (exp5_ldd.ko).

STEP 4: Enable the POKY-TINY Toolchain for 32-bit Compilation

Since your Rugged Board A5D2X is 32-bit, enable the poky-tiny toolchain by setting up the environment:

COMMAND:

. /opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi

This will set up the toolchain for cross-compiling the driver on your host system.

STEP 5: Compile the Driver

Run the following command to compile the driver:

COMMAND:

make

If everything is set up correctly, you should get a kernel module file named **exp5_ldd.ko.**

STEP 6: Prepare the Device Tree Blob and Kernel Image for the SD Card

To boot the board properly, copy the necessary kernel files to the Boot partition of the Rugged Board A5D2X's SD card:

- **Device Tree Blob** (a5d2x-rugged_board.dtb): /home/rameshchlm/BSP_BOOT/linux-rba5d2x/arch/arm/boot/dts/a5d2x-rugged_board.dtb
- **Kernel Image** (zImage): /home/rameshchlm/BSP_BOOT/linux-rba5d2x/arch/arm/boot/zimage

STEP 7: Boot the Rugged Board and Transfer the .ko File

1. Boot the Rugged Board A5D2X.
2. Use TFTP to transfer the exp5.ko file from the host machine to the target machine:

On the host machine:

COMMAND:

tftp -g -r exp5_ldd.ko <rugged_board_ip_address>

On the target machine (Rugged Board A5D2X)**:**

tftp -r exp5_ldd.ko -g <host_ip_address>

STEP 8: Insert the Kernel Module on the Target Board

After transferring exp5.ko to the target, insert the module by running:

COMMAND:

insmod exp5_ldd.ko

STEP 9: i] List the device driver by command:

ls /dev/

ii]then check the device driver operation by

echo "Hello world" /dev/chardriver_name

iii]then check whether the characters are inserted by

cat /dev/chardriver_name

STEP 10: Verify Driver Functionality with dmesg

Use dmesg to check for any messages from your driver, such as initialization logs or any output generated by printk statements in the code:

COMMAND:

dmesg

PROGRAM:

exp5_ldd

```
#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>
#include <linux/slab.h>  // for kmalloc and kfree
#include <linux/mutex.h> // for mutex
#define MAX_DEV 2
#define BUFFER_SIZE 128
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;
    char *buffer; // To hold user data
    struct mutex mutex; // For synchronization
```

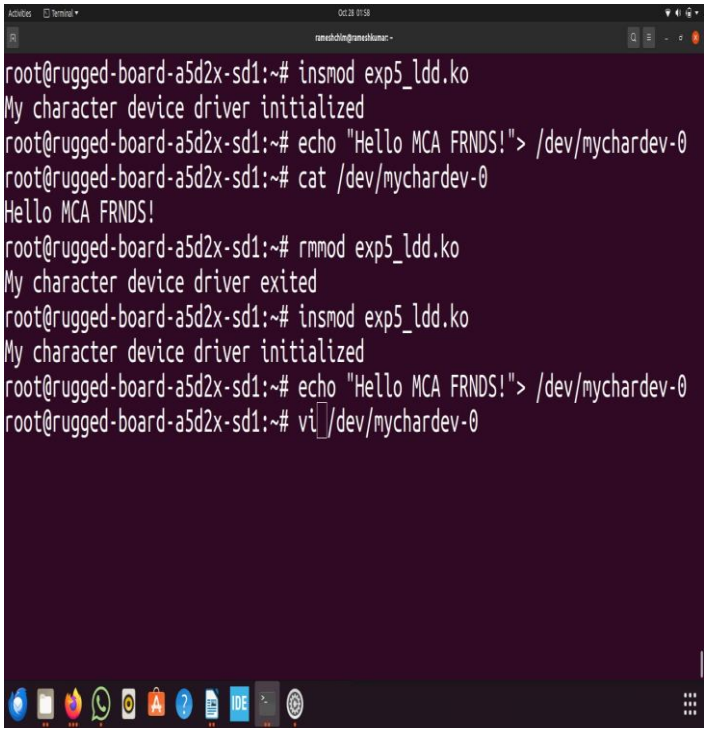
```
};
static int dev_major = 0;
static struct class *mychardev_class = NULL;
static struct mychar_device_data mychardev_data[MAX_DEV];
static int mychardev_uevent(struct device *dev, struct kobj_uevent_env *env) {
    add_uevent_var(env, "DEVMODE=%#o", 0666);
    return 0;
}
static int __init mychardev_init(void) {
    int err, i;
    dev_t dev;
    err = alloc_chrdev_region(&dev, 0, MAX_DEV, "mychardev");
    if (err < 0) {
        printk(KERN_ERR "Failed to allocate char device region\n");
        return err;
    }
    dev_major = MAJOR(dev);
    mychardev_class = class_create(THIS_MODULE, "mychardev");
    if (IS_ERR(mychardev_class)) {
        unregister_chrdev_region(dev, MAX_DEV);
        return PTR_ERR(mychardev_class);
    }
    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;
        mychardev_data[i].buffer = kmalloc(BUFFER_SIZE, GFP_KERNEL);
        if (!mychardev_data[i].buffer) {
            printk(KERN_ERR "Failed to allocate memory for device %d\n", i);
            while (--i >= 0) {
```

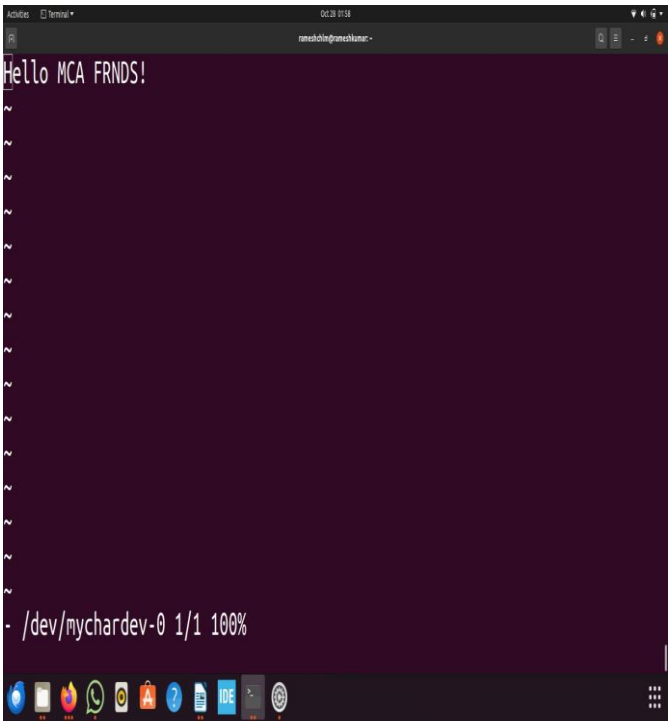
```
            kfree(mychardev_data[i].buffer);
            cdev_del(&mychardev_data[i].cdev);
        }
        class_destroy(mychardev_class);
        unregister_chrdev_region(dev, MAX_DEV);
        return -ENOMEM;
    }
    mutex_init(&mychardev_data[i].mutex);
    cdev_add(&mychardev_data[i].cdev, MKDEV(dev_major, i), 1);
    device_create(mychardev_class, NULL, MKDEV(dev_major, i), NULL,
"mychardev-%d", i);
  }

  printk(KERN_INFO "My character device driver initialized\n");
  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));
    kfree(mychardev_data[i].buffer);
    cdev_del(&mychardev_data[i].cdev);
  }
  class_unregister(mychardev_class);
  class_destroy(mychardev_class);
  unregister_chrdev_region(MKDEV(dev_major, 0), MAX_DEV);

  printk(KERN_INFO "My character device driver exited\n");
}
static int mychardev_open(struct inode *inode, struct file *file) {
```

```
    struct mychar_device_data *dev_data = container_of(inode->i_cdev, struct
mychar_device_data, cdev);
    file->private_data = dev_data;
    printk(KERN_INFO "MYCHARDEV: Device open\n");
    return 0;
}
static int mychardev_release(struct inode *inode, struct file *file) {
    printk(KERN_INFO "MYCHARDEV: Device close\n");
    return 0;
}
static long mychardev_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    printk(KERN_INFO "MYCHARDEV: Device ioctl\n");
    return 0;
}
static ssize_t mychardev_read(struct file *file, char __user *buf, size_t count, loff_t *offset) {
    struct mychar_device_data *dev_data = file->private_data;
    size_t datalen = strlen(dev_data->buffer);
    if (*offset >= datalen) {
        return 0; // End of file
    }
    if (count > datalen - *offset) {
        count = datalen - *offset; // Adjust count to remaining data
    }
    if (copy_to_user(buf, dev_data->buffer + *offset, count)) {
        return -EFAULT;
    }
    *offset += count; // Update offset for the next read
    printk(KERN_INFO "MYCHARDEV: Read %zu bytes\n", count);
    return count;
}
```

```
static ssize_t mychardev_write(struct file *file, const char __user *buf, size_t count, loff_t
*offset) {
    struct mychar_device_data *dev_data = file->private_data;
    size_t maxdatalen = BUFFER_SIZE - 1; // Reserve space for null terminator
    if (count > maxdatalen) {
        count = maxdatalen; // Limit the number of bytes to write
    }
    mutex_lock(&dev_data->mutex); // Lock for synchronization
    // Clear the buffer before copying to prevent garbage values
    memset(dev_data->buffer, 0, BUFFER_SIZE);
    if (copy_from_user(dev_data->buffer, buf, count)) {
        mutex_unlock(&dev_data->mutex);
        return -EFAULT;
    }
    dev_data->buffer[count] = '\0'; // Null-terminate the string
    *offset += count; // Update offset
    printk(KERN_INFO "MYCHARDEV: Wrote %zu bytes: %s\n", count, dev_data->buffer);
    mutex_unlock(&dev_data->mutex); // Unlock after the operation
    return count;
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Rameshkumar_Hari_BelthaArthi");
module_init(mychardev_init);
module_exit(mychardev_exit);
```

OUTPUT:





RESULT:

EX NO: 06 DATE:	**Experiment with IOCTL using Device Driver**

AIM:

To write the Program in IOCTL in Device Driver

ALGORITHM:

STEP 1: Start the Program
STEP 2: Write the program to open and read and write using Device driver
STEP 3: To create the Make file "Makefile"

```
obj-m += ioctl_driver.o
KDIR = /lib/modules/$(shell uname -r)/build
TDIR = /home/surya/Downloads/RB-a5d2x/linux-rba5d2x
host:
    make -C $(KDIR)  M=$(shell pwd) modules
target:
    make -C $(TDIR)  M=$(shell pwd) modules
clean:
    make -C $(KDIR)  M=$(shell pwd) clean
```

STEP 4: To enable to poky toolchain

. / opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi

STEP 5: To execute the app.c file and Makefile

app.c-> $CC app.c -o app
Makefile -> make target

STEP 6: Then copy the executable file "app" and "ioctl_driver.ko" in tftpboot

cp /*var*/lib/tftpboot/

STEP 7: Next to enter the Rugged Board Terminal.
STEP 8: Then to config ethernet

ifconfig 192.168.1.36

STEP 9: To move on the executable file tftpboot to RuggedBoard

tftp -r app -g 192.168.1.30
tftp -r ioctl_driver.ko -g 192.168.1.30

STEP 10: To change the file permision

chmod 777 ioctl_driver.ko
chmod 777 app

STEP 11: To insert the .ko file

insmod ioctl_driver.ko

STEP 12: To run on the file

./app

PROGRAM:

app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include<sys/ioctl.h>

#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int main()
{
        int fd;
        int32_t value, number;
        printf("*********************************\n");
        printf("*******WWW.phyuniversity.com*******\n");

        printf("\nOpening Driver\n");
        fd = open("/dev/etx_device", O_RDWR);
        if(fd < 0) {
                printf("Cannot open device file...\n");
                return 0;
        }

        printf("Enter the Value to send\n");
        scanf("%d",&number);
        printf("Writing Value to Driver\n");
        ioctl(fd, WR_VALUE, (int32_t*) &number);

        printf("Reading Value from Driver\n");
        ioctl(fd, RD_VALUE, (int32_t*) &value);
        printf("Value is %d\n", value);

        printf("Closing Driver\n");
        close(fd);
}
```

ioctl_driver.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
```

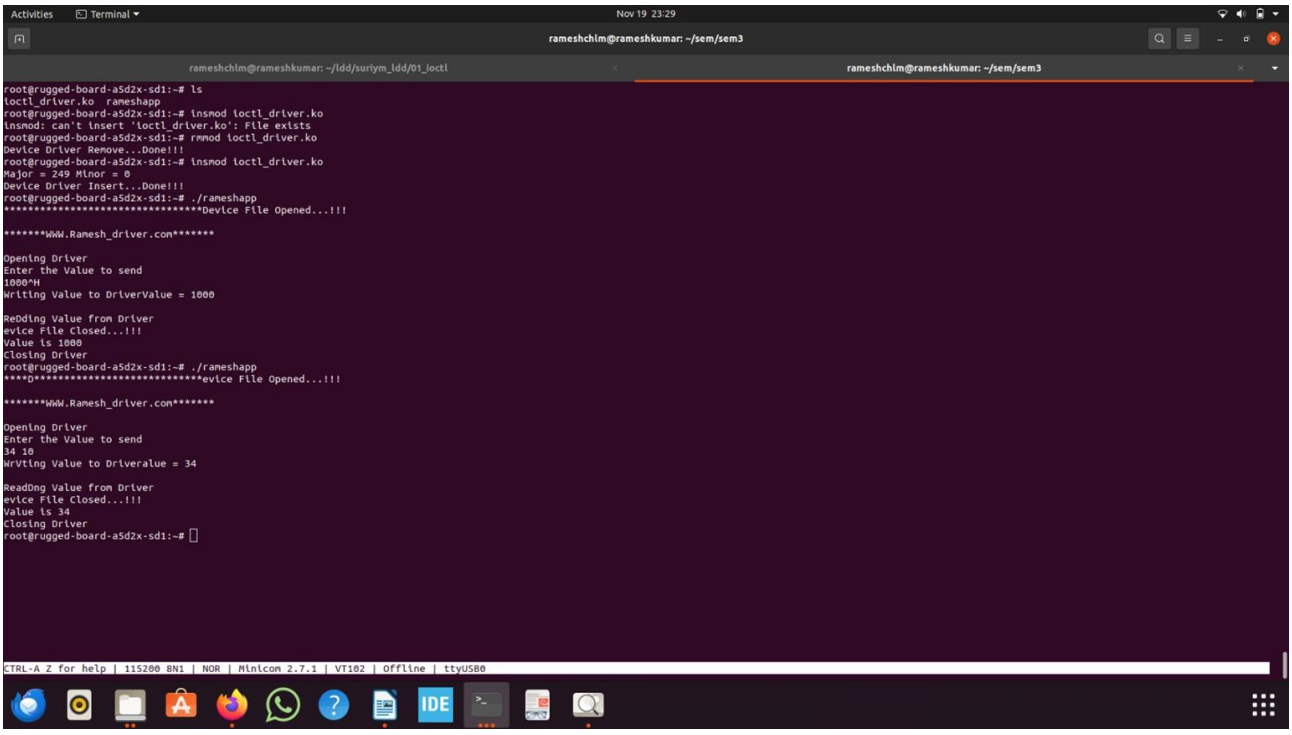
```
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include <linux/ioctl.h>
#include <linux/err.h>


#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

int32_t value = 0;

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

/*
** Function Prototypes
*/
static int      __init etx_driver_init(void);
static void     __exit etx_driver_exit(void);
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t  etx_read(struct file *filp, char __user *buf, size_t len,loff_t * off);
static ssize_t  etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);
static long     etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

/*
** File operation sturcture
*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .unlocked_ioctl = etx_ioctl,
        .release        = etx_release,
};

/*
** This function will be called when we open the Device file
*/
static int etx_open(struct inode *inode, struct file *file)
{
        pr_info("Device File Opened...!!!\n");
        return 0;
}
```

```
/*
** This function will be called when we close the Device file
*/
static int etx_release(struct inode *inode, struct file *file)
{
        pr_info("Device File Closed...!!!\n");
        return 0;
}

/*
** This function will be called when we read the Device file
*/
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        pr_info("Read Function\n");
        return 0;
}

/*
** This function will be called when we write the Device file
*/
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        pr_info("Write function\n");
        return len;
}

/*
** This function will be called when we write IOCTL on the Device file
*/
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Value = %d\n", value);
                        break;
                case RD_VALUE:
                        if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                        {
                                pr_err("Data Read : Err!\n");
                        }
                        break;
                default:
                        pr_info("Default\n");
                        break;
```

```
        }
        return 0;
}

/*
** Module Init function
*/
static int __init etx_driver_init(void)
{
        /*Allocating Major number*/
        if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
                pr_err("Cannot allocate major number\n");
                return -1;
        }
        pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

        /*Creating cdev structure*/
        cdev_init(&etx_cdev,&fops);

        /*Adding character device to the system*/
        if((cdev_add(&etx_cdev,dev,1)) < 0){
            pr_err("Cannot add the device to the system\n");
            goto r_class;
        }

        /*Creating struct class*/
        if(IS_ERR(dev_class = class_create(THIS_MODULE,"etx_class"))){
            pr_err("Cannot create the struct class\n");
            goto r_class;
        }

        /*Creating device*/
        if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){
            pr_err("Cannot create the Device 1\n");
            goto r_device;
        }
        pr_info("Device Driver Insert...Done!!!\n");
        return 0;

r_device:
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
```

```
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("RB");
MODULE_DESCRIPTION("Simple Linux device driver (IOCTL)");
MODULE_VERSION("1.5");
```

OUTPUT:



RESULT:

EX NO: 07 DATE:	**Experiment with GPIO Control in a Kernel Module**

AIM:

To write the Program for GPIO Control in a Kernel Module

ALGORITHM:

STEP 1: Start the Program
STEP 2: write the program to open and read and write using Device driver
STEP 3: To create the Make file "Makefile"

```
obj-m += gpio_driver.o
KDIR = /lib/modules/$(shell uname -r)/build
TDIR = /home/surya/Downloads/RB-a5d2x/linux-rba5d2x

host:
    make -C $(KDIR) M=$(shell pwd) modules
target:
    make -C $(TDIR) M=$(shell pwd) modules
clean:
    make -C $(KDIR) M=$(shell pwd) clean
```

STEP 4: To enable to poky toolchain

. / opt/poky-tiny/2.5.2/environment-setup-cortexa5hf-neon-poky-linux-musleabi

STEP 5: To execute the app.c file and Makefile

app.c-> $CC app.c -o app
Makefile -> make target

STEP 6: Then copy the executable file "app" and "gpio_driver.ko" in tftpboot

cp /*var*/lib/tftpboot/

STEP 7: next to enter the Rugged Board Terminal.
STEP 8: Then to config ethernet

ifconfig 192.168.1.36

STEP 9: To move on the executable file tftpboot to RuggedBoard

tftp -r app -g 192.168.1.30
tftp -r gpio_driver.ko -g 192.168.1.30

STEP 10: To change the file permision

chmod 777 gpio_driver.ko
chmod 777 app

STEP 11: To insert the .ko file

insmod gpio_driver.ko

STEP 12: To run on the file

./app

PROGRAM:

app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include<sys/ioctl.h>
#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)
void delay (int aa)
{
int i,j;
for(i=0; i<aa; i++)
{
for (j=0; j<100000; j++)
{}
}
}
int main()
{
	int fd;
	int32_t off=0, on=1;
	printf("*********************************\n");
	printf("*******WWW.phyuniversity.com*******\n");
	printf("\nOpening Driver\n");
	fd = open("/dev/etx_device", O_RDWR);
	if(fd < 0) {
		printf("Cannot open device file...\n");
		return 0;
	}
	while(1)
	{
		delay(100);
		ioctl(fd, WR_VALUE, (int32_t*)&off);
		delay(1000);
		ioctl(fd, WR_VALUE, (int32_t*)&on);
		delay(1000);
	}
	printf("Closing Driver\n");
	close(fd);
}
```

ioctl_driver.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include<linux/slab.h>                 //kmalloc()
#include<linux/uaccess.h>              //copy_to/from_user()
#include <linux/ioctl.h>
#include <linux/err.h>
#include <linux/gpio.h>

#define WR_VALUE _IOW('a','a',int32_t*)
#define RD_VALUE _IOR('a','b',int32_t*)

#define GPIO_PC_13 77

int32_t value = 0;
dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

/* Function Prototypes*/
static int      __init etx_driver_init(void);
static void     __exit etx_driver_exit(void);
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t  etx_read(struct file *filp, char __user *buf, size_t len,loff_t * off);
static ssize_t  etx_write(struct file *filp, const char *buf, size_t len, loff_t * off);
static long     etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

/*File operation sturcture*/
static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
        .write          = etx_write,
        .open           = etx_open,
        .unlocked_ioctl = etx_ioctl,
        .release        = etx_release,
};

int led_init(int gpio_num)
{
        int retval=0;
        retval= gpio_request(gpio_num,"sled");
```

```
        if(retval<0)
                return retval;
        retval= gpio_direction_output(gpio_num,1);
        return retval;
}
void led_on(int gpio_num)
{
        gpio_set_value(gpio_num,1);

}
void led_off(int gpio_num)
{
        gpio_set_value(gpio_num,0);
}
/*This function will be called when we open the Device file*/
static int etx_open(struct inode *inode, struct file *file)
{
        pr_info("Device File Opened...!!!\n");
        led_init(GPIO_PC_13);
        return 0;
}
/*This function will be called when we close the Device file*/
static int etx_release(struct inode *inode, struct file *file)
{
        pr_info("Device File Closed...!!!\n");
        return 0;
}
/*This function will be called when we read the Device file*/
static ssize_t etx_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
        pr_info("Read Function\n");
        return 0;
}
/*This function will be called when we write the Device file*/
static ssize_t etx_write(struct file *filp, const char __user *buf, size_t len, loff_t *off)
{
        pr_info("Write function\n");
        return len;
}
/*This function will be called when we write GPIO Kernel on the Device file*/
static long etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
        switch(cmd) {
                case WR_VALUE:
                        if( copy_from_user(&value ,(int32_t*) arg, sizeof(value)) )
                        {
                                pr_err("Data Write : Err!\n");
                        }
                        pr_info("Value = %d\n", value);
                        if(value)
```

```
                {
                        led_on(GPIO_PC_13);
                }
                else
                {
                        led_off(GPIO_PC_13);
                }
                break;
        case RD_VALUE:
                if( copy_to_user((int32_t*) arg, &value, sizeof(value)) )
                {
                        pr_err("Data Read : Err!\n");
                }
                break;
        default:
                pr_info("Default\n");
                break;
    }
    return 0;
}
/*Module Init function*/
static int __init etx_driver_init(void)
{
    /*Allocating Major number*/
    if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
            pr_err("Cannot allocate major number\n");
            return -1;
    }
    pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));
    /*Creating cdev structure*/
    cdev_init(&etx_cdev,&fops);
    /*Adding character device to the system*/
    if((cdev_add(&etx_cdev,dev,1)) < 0){
       pr_err("Cannot add the device to the system\n");
       goto r_class;
    }
    /*Creating struct class*/
    if(IS_ERR(dev_class = class_create(THIS_MODULE,"etx_class"))){
       pr_err("Cannot create the struct class\n");
       goto r_class;
    }
    /*Creating device*/
    if(IS_ERR(device_create(dev_class,NULL,dev,NULL,"etx_device"))){
       pr_err("Cannot create the Device 1\n");
       goto r_device;
    }
    pr_info("Device Driver Insert...Done!!!\n");
    return 0;

r_device:
```

```
        class_destroy(dev_class);
r_class:
        unregister_chrdev_region(dev,1);
        return -1;
}

/* Module exit function*/
static void __exit etx_driver_exit(void)
{
        device_destroy(dev_class,dev);
        class_destroy(dev_class);
        cdev_del(&etx_cdev);
        unregister_chrdev_region(dev, 1);
        pr_info("Device Driver Remove...Done!!!\n");
}
module_init(etx_driver_init);
module_exit(etx_driver_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("RB");
MODULE_DESCRIPTION("Simple Linux device driver (IOCTL)");
MODULE_VERSION("1.5");
```

OUTPUT:

```
root@rugged-board-a5d2x-sd1:~# tftp -r rameshapp -g 192.168.1.30
root@rugged-board-a5d2x-sd1:~# tftp -r ioctl_driver.ko -g 192.168.1.30
root@rugged-board-a5d2x-sd1:~# ls
ex.ko            exp5_ldd.ko      ioctl_driver.ko  ldd_exp2.ko      rameshapp        suriya_app
root@rugged-board-a5d2x-sd1:~# chmod 777 ioctl_driver.ko
root@rugged-board-a5d2x-sd1:~# chmod 777 rameshapp
root@rugged-board-a5d2x-sd1:~# insmod ioctl_driver.ko
insmod: can't insert 'ioctl_driver.ko': File exists
root@rugged-board-a5d2x-sd1:~# rmmod ioctl_driver.ko
Device Driver Remove...Done!!!
root@rugged-board-a5d2x-sd1:~# insmod ioctl_driver.ko
Major = 249 Minor = 0
Device Driver Insert...Done!!!
root@rugged-board-a5d2x-sd1:~# ./rameshapp
****D**************************evice File Opened...!!!

*******WWW.RameshKumarNetwork.com*******

Opening Driver
Value = 0
Value = 1
Value = 0
Value = 1
Value = 0
Value = 1
Value = 0
Value = 1
Value = 0
```

RESULT:

EX NO: 08 DATE:	**Read and Write to a Simple Flash Memory Simulator**

AIM:

To develop a Linux platform driver that manages LED devices by interfacing with a platform device. The driver allows user-space applications to control an LED's state (ON or OFF) using a GPIO pin configured for the LED device.

ALGORITHM:

STEP 1: Initialize Platform Data

- Define a sled_platform_data structure containing the GPIO pin number required for LED control.
- Pass this platform data to the platform device during its registration.

STEP 2: Register the Platform Device

- Create and initialize a platform device (platform_sled) with the platform data.
- Implement a sled_release() function to handle device release when the device is unregistered.

STEP 3: Implement Platform Driver

- Define the sled_platform_driver with:
    - **Probe Function:** Called when the device is detected. It:
        - Initializes the GPIO for the LED.
        - Allocates device-specific data and links it to the platform device.
        - Registers a character device for communication with user space.
    - **Remove Function:** Cleans up resources when the device is removed.
- Register the platform driver with the Linux kernel.

STEP 4: Character Device Driver

- Define file operations (sled_fops) for the character device, including:
    - **Open:** Initialize the LED GPIO using the platform data.
    - **Read:** (Optional) Provide a way to read the LED status.
    - **Write:** Parse user input to turn the LED ON or OFF using the GPIO pin.
    - **Release:** Free any resources, if required.

- Use the copy_from_user() function to receive data from user space for controlling the LED.

STEP 5: Module Initialization

- Allocate a range of device numbers for the LED devices.
- Create a device class in /sys/class for sysfs entry.
- Register the platform driver to link it with the platform device.

STEP 6: Module Cleanup

- Unregister the platform driver.
- Destroy the device class and release the allocated device numbers.

PROGRAM:

```
//platform.h
struct sled_platform_data
{
    int gpio_num;
};
//sled_platform_driver.c
#include<linux/module.h>
#include<linux/fs.h>
#include<linux/cdev.h>
#include<linux/device.h>
#include<linux/kdev_t.h>
#include<linux/uaccess.h>
#include <linux/platform_device.h>
#include<linux/slab.h>
#include<linux/mod_devicetable.h>
#include <linux/gpio.h>
#include"platform.h"
char kbuff[20];
int gpio_number;
```

```
struct sledev_private_data
{
	struct sled_platform_data pdata;
	dev_t dev_num;
	struct cdev cdev;
};
/*Driver private data structure */
struct sledrv_private_data
{
	int total_devices;
	dev_t device_num_base;
	struct class *class_sled;
	struct device *device_sled;
};
struct sledrv_private_data sledrv_data;
int led_init(int gpio_num)
{
	int retval=0;
	retval= gpio_request(gpio_num,"sled");
	if(retval<0)
		return retval;
	retval= gpio_direction_output(gpio_num,1);
	return retval;
}
void led_on(int gpio_num)
{
	gpio_set_value(gpio_num,1);
}
void led_off(int gpio_num)
{
	gpio_set_value(gpio_num,0);
```

```
}
int sled_open(struct inode *in, struct file *fp)
{
	int minor_n;
	    struct sledev_private_data *sledev_data;
	    /*find out on which device file open was attempted by the user space */
	    minor_n = MINOR(in->i_rdev);
	    pr_info("minor access = %d\n",minor_n);
	    /*get device's private data structure */
	    sledev_data = container_of(in->i_cdev,struct sledev_private_data,cdev);
	    /*to supply device private data to other methods of the driver */
	    fp->private_data = sledev_data;
	    printk(KERN_INFO "This is sled open function\n");
	    led_init(sledev_data->pdata.gpio_num);
	    pr_info("In open function gpioi_number is %d\n",sledev_data->pdata.gpio_num);
	    return 0;
}
ssize_t sled_read (struct file *fp, char __user *buff, size_t sz, loff_t *offset)
{
	    printk(KERN_INFO "This is sled read function\n");
	    return 0;
}
ssize_t sled_write (struct file *fp, const char __user *buff, size_t sz, loff_t *offset)
{
	    printk(KERN_INFO "This is sled write function\n");
	struct sledev_private_data *sledev_data = (struct sledev_private_data*)fp->private_data;
	    copy_from_user(kbuff,buff,sz);
	    if(kbuff[0]=='O' && kbuff[1]=='N')
	        led_on(sledev_data->pdata.gpio_num);
	    else if(kbuff[0]=='O' && kbuff[1]=='F' && kbuff[2]=='F')
	            led_off(sledev_data->pdata.gpio_num);
```

```
        else
                printk(KERN_INFO "Invalid function");
        return 0;
}
int sled_release (struct inode *in, struct file *fp)
{
        printk(KERN_INFO "This is sled release function\n");
        return 0;
}
struct file_operations sled_fops= {
        .open=sled_open,
        .read=sled_read,
        .write=sled_write,
        .release=sled_release
};
/*Called when the device is removed from the system */
int sled_platform_driver_remove(struct platform_device *pdev)
{
      struct sledev_private_data *dev_data = dev_get_drvdata(&pdev->dev);
      device_destroy(sledrv_data.class_sled,dev_data->dev_num);
      cdev_del(&dev_data->cdev);
      kfree(dev_data);
      pr_info("%s:A device is removed\n",__func__);
      return 0;
}
int sled_platform_driver_probe(struct platform_device *pdev)
{
        int ret;
        struct sledev_private_data *dev_data;
        struct sled_platform_data *pdata;
```

```
    pr_info("%s:A device is detected\n",__func__);
    /*Get the platform data */
    pdata = (struct sled_platform_data*)dev_get_platdata(&pdev->dev);
    if(!pdata){
            pr_info("No platform data available\n");
            return -EINVAL;
    }
  /*Dynamically allocate memory for the device private data  */
    dev_data = devm_kzalloc(&pdev->dev, sizeof(*dev_data),GFP_KERNEL);
  //dev_data = kzalloc(sizeof(*dev_data),GFP_KERNEL);
    if(!dev_data){
            pr_info("Cannot allocate memory \n");
            return -ENOMEM;
    }
  /*save the device private data pointer in platform device structure */
    dev_set_drvdata(&pdev->dev,dev_data);
    dev_data->pdata.gpio_num = pdata->gpio_num;
    gpio_number = pdata->gpio_num;
  /* Get the device number */
    dev_data->dev_num = sledrv_data.device_num_base + pdev->id;
    /*Do cdev init and cdev add */
    cdev_init(&dev_data->cdev,&sled_fops);
  /*  Register a device (cdev structure) with VFS */
    dev_data->cdev.owner = THIS_MODULE;
    ret = cdev_add(&dev_data->cdev,dev_data->dev_num,1);
    if(ret < 0){
            pr_err("Cdev add failed\n");
    }
  /*populate the sysfs with device information */
sledrv_data.device_sled=device_create(sledrv_data.class_sled,NULL,dev_data-
>dev_num,NULL,"sledev-%d",pdev->id);
```

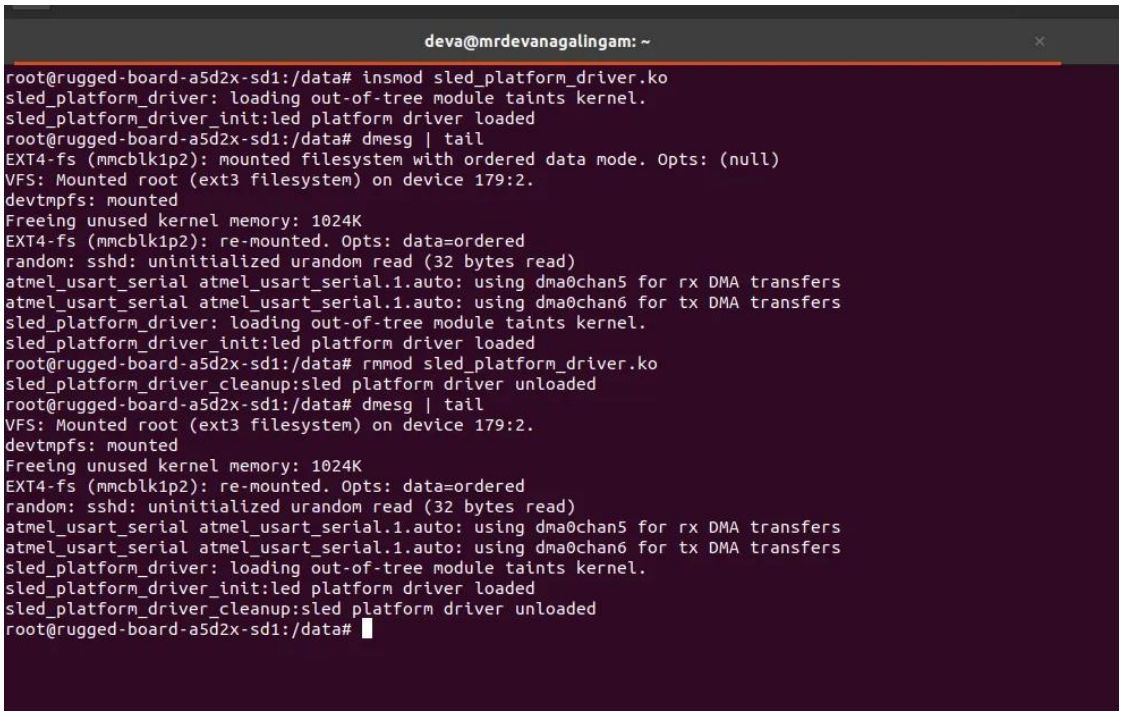
```
                if(IS_ERR(sledrv_data.device_sled)){
                        pr_err("Device create failed\n");
                        ret = PTR_ERR(sledrv_data.device_sled);
                        cdev_del(&dev_data->cdev);
                }
        pr_info("%s:Device gpio number = %d\n",__func__,dev_data->pdata.gpio_num);
         pr_info("%s:A probe was detected\n",__func__);
         return 0;
}
struct platform_driver sled_platform_driver =
{
         .probe = sled_platform_driver_probe,
         .remove = sled_platform_driver_remove,
         .driver = {
                 .name = "sled"
          }
};
static int __init sled_platform_driver_init(void)
{
      int ret;
         ret = alloc_chrdev_region(&sledrv_data.device_num_base,0,1,"sldevs");
      /*create device class under /sys/class/ */
         sledrv_data.class_sled = class_create(THIS_MODULE,"sled_class");
         if(IS_ERR(sledrv_data.class_sled)){
                 pr_err("%s:Class creation failed\n",__func__);
                 ret = PTR_ERR(sledrv_data.class_sled);
            unregister_chrdev_region(sledrv_data.device_num_base,1);
            return ret;
         }
         platform_driver_register(&sled_platform_driver);
         pr_info("%s:led platform driver loaded\n",__func__);
```

```
        return 0;
}
static void __exit sled_platform_driver_cleanup(void)
{
    platform_driver_unregister(&sled_platform_driver);
    class_destroy(sledrv_data.class_sled);
    unregister_chrdev_region(sledrv_data.device_num_base,1);
    pr_info("%s:sled platform driver unloaded \n",__func__);
}
module_init(sled_platform_driver_init);
module_exit(sled_platform_driver_cleanup);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("RB");
MODULE_DESCRIPTION("Sled platform driver which handles platform devices\n");
```

MAKEFILE:

```
obj-m += sled_device.o sled_platform_driver.o
KDIR = /lib/modules/$(shell uname -r)/build
TDIR = /home/deva/RuggedBoard/RB-A5D2x/SD_card_boot/Kernel_source/linux-rba5d2x
host:
    make -C $(KDIR)  M=$(shell pwd) modules
target:
    make -C $(TDIR)  M=$(shell pwd) modules

clean:
    make -C $(KDIR)  M=$(shell pwd) clean
```

OUTPUT:



RESULT:

EX NO: 09 DATE:	**Basic USB Device Driver Experiment**

AIM:

To create and test a basic USB driver on a rugged board, detecting USB device connections and disconnections.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Write the Program for Basic USB driver with probe and disconnect functions

STEP 3: Identify the USB Device, Plug in the USB device, retrieve Vendor ID and Product ID using lsusb

Bus 002 Device 003: ID 1234:5678 USB Device

STEP 4: To create the Make file "Makefile"

```
obj-m += basic_usb_driver.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
```

STEP 5: Cross-Compile for Rugged Board

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules
```

STEP 6: To execute the app.c file and Makefile

app.c-> $CC app.c -o app

Makefile -> make target

STEP 7: Transfer the basic_usb_driver.ko file to the rugged board

```
# scp basic_usb_driver.ko user@rugged_board_ip:/path/on/board
```

STEP 8: To load the driver

```
# sudo insmod basic_usb_driver.ko
```

STEP 9: Check the kernel log to verify that the driver has been loaded

```
# dmesg | tail
```

STEP 10: Test the Driver , To Plug and unplug the USB device

Connection Message:

USB Device connected: Vendor ID=1234, Product ID=5678

Disconnection Message:

USB Device disconnected

STEP 11: Unload and Clean up the driver after testing.

sudo rmmod basic_usb_driver

CODING:

app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#define USB_ON _IOW('u', 'o', int32_t*)  // Command to turn USB on
#define USB_OFF _IOW('u', 'f', int32_t*) // Command to turn USB off
void delay(int milliseconds) {
    int i, j;
    for (i = 0; i < milliseconds; i++) {
        for (j = 0; j < 100000; j++) {}
    }
}
int main() {
    int fd;
    int32_t usb_on = 1, usb_off = 0;
    printf("*********************************\n");
    printf("***** USB Device Control ******\n");
    printf("*********************************\n");
```

```
    // Open the USB device file
    fd = open("/dev/usb_device", O_RDWR);
    if (fd < 0) {
        printf("Cannot open USB device file...\n");
        return 0;
    }
    while (1) {
        printf("Turning USB device ON...\n");
        ioctl(fd, USB_ON, (int32_t*)&usb_on); // Send USB ON command
        delay(1000);
        printf("Turning USB device OFF...\n");
        ioctl(fd, USB_OFF, (int32_t*)&usb_off); // Send USB OFF command
        delay(1000);
    }
    printf("Closing USB device file...\n");
    close(fd);
    return 0;
}
```

Usb_driver.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/usb.h>
/* Define the USB device ID */
static struct usb_device_id usb_table[] = {
    { USB_DEVICE(0x1234, 0x5678) },  // Replace with actual Vendor and Product IDs
    {}
};
MODULE_DEVICE_TABLE(usb, usb_table);

/* Probe function */
static int usb_probe(struct usb_interface *interface, const struct usb_device_id *id) {
```

```
    printk(KERN_INFO "USB Device connected: Vendor ID=%04X, Product ID=%04X\n",
id->idVendor, id->idProduct);
    return 0;
}
/* Disconnect function */
static void usb_disconnect(struct usb_interface *interface) {
    printk(KERN_INFO "USB Device disconnected\n");
}
/* Define USB driver structure */
static struct usb_driver usb_drv = {
    .name = "basic_usb_driver",
    .id_table = usb_table,
    .probe = usb_probe,
    .disconnect = usb_disconnect,
};
/* Register the driver */
static int __init usb_init(void) {
    return usb_register(&usb_drv);
}
/* Deregister the driver */
static void __exit usb_exit(void) {
    usb_deregister(&usb_drv);
}
module_init(usb_init);
module_exit(usb_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Basic USB Driver for Rugged Board");
```

OUTPUT:

```
root@rugged-board-a5d2x-sd1:~# insmod usb_driver.ko
usb_driver: loading out-of-tree module taints kernel.
usbcore: registered new interface driver my_usb_driver
root@rugged-board-a5d2x-sd1:~#
```

```
EXT4-fs (mmcblk1p2): recovery complete
EXT4-fs (mmcblk1p2): mounted filesystem with ordered data mo)
VFS: Mounted root (ext3 filesystem) on device 179:2.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
EXT4-fs (mmcblk1p2): re-mounted. Opts: data=ordered
random: sshd: uninitialized urandom read (32 bytes read)
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chans
atmel_usart_serial atmel_usart_serial.1.auto: using dma0chans
mmcblk1: error -110 transferring data, sector 2113536, nr 8,0
mmcblk1: error -110 transferring data, sector 2118856, nr 480
mmcblk1: error -110 transferring data, sector 2118912, nr 160
usb_driver: loading out-of-tree module taints kernel.
usbcore: registered new interface driver my_usb_driver
root@rugged-board-a5d2x-sd1:~#
```

RESULT:

EX NO: 10 DATE:	Basic Hardware Interrupt

AIM:

To implement a basic hardware interrupt-based Linux kernel module that toggles an LED state based on a button press using GPIOs, demonstrating interrupt handling and kernel-user space interaction.

ALGORITHM:

Step 1: Write the driver code hardware_interrupt.c and user application hardware_interrupt_app.c.

Step 2: Create the Makefile in the same directory.

Step 3: Build the module using make host (for host) or make target (for target).

Step 4: Load the module with sudo insmod hardware_interrupt.ko.

Step 5: Export and configure GPIO pins if necessary using echo <gpio_number> > /sys/class/gpio/export.

Step 6: Run the user application ./hardware_interrupt_app to monitor instructions.

Step 7: Observe kernel log messages using dmesg -w to check button press and LED state.

Step 8: Unload the module with sudo rmmod hardware_interrupt and clean up using make clean.

PROGRAM:

hardware_interrupt.c:

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

#include <linux/gpio.h>

#include <linux/interrupt.h>

#define BUTTON_GPIO_PIN 76  // Replace with your button GPIO

#define LED_GPIO_PIN 77     // Replace with your LED GPIO
```

```
static int irq_number;
static bool led_state = false;
// Interrupt handler
static irqreturn_t button_irq_handler(int irq, void *dev_id) {
    // Read the button state
    int button_state = gpio_get_value(BUTTON_GPIO_PIN);
    if (button_state) {
        pr_info("Button pressed! LED is %s\n", led_state ? "ON" : "OFF");
        // Toggle LED state only on button press
        led_state = !led_state;
        gpio_set_value(LED_GPIO_PIN, led_state);
    } else {
        pr_info("Button not pressed! LED remains %s\n", led_state ? "ON" : "OFF");
    }
    return IRQ_HANDLED;
}
static int __init button_led_init(void) {
    int ret;
    // Request GPIOs for button and LED
    if ((ret = gpio_request_one(BUTTON_GPIO_PIN, GPIOF_IN, "button_gpio")) ||
        (ret = gpio_request_one(LED_GPIO_PIN, GPIOF_OUT_INIT_LOW, "led_gpio"))) {
        pr_err("GPIO request failed.\n");
        return ret;
    }
    // Map GPIO to IRQ
    if ((irq_number = gpio_to_irq(BUTTON_GPIO_PIN)) < 0) {
        pr_err("Failed to get IRQ number for GPIO %d\n", BUTTON_GPIO_PIN);
        ret = irq_number;
```

```
        goto err_gpio_free;
    }
    // Request IRQ
    if ((ret = request_irq(irq_number, button_irq_handler, IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING, "button_irq", NULL))) {
        pr_err("Failed to request IRQ %d\n", irq_number);
        goto err_gpio_free;
    }
    pr_info("Driver initialized.\n");
    return 0;
err_gpio_free:
    gpio_free(BUTTON_GPIO_PIN);
    gpio_free(LED_GPIO_PIN);
    return ret;
}
static void __exit button_led_exit(void) {
    free_irq(irq_number, NULL);
    gpio_free(BUTTON_GPIO_PIN);
    gpio_free(LED_GPIO_PIN);
    pr_info("Driver exited.\n");
}
module_init(button_led_init);
module_exit(button_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Button-LED Driver with Button State Detection");
```

hardware_interrupt_app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
   printf("************\n");
   printf("** BUTTON-LED TEST PROGRAM **\n");
   printf("************\n");
   printf("This program doesn't directly control the GPIOs because the driver handles it.\n");
   printf("Press the button connected to GPIO to toggle the LED state.\n");
   printf("Monitoring driver log messages for button press and LED state.\n");
   printf("Run the following command in another terminal to observe the kernel log:\n");
   printf("    dmesg -w\n");
   printf("\nPress Ctrl+C to exit.\n");
   // Simulate continuous monitoring
   while (1) {
      sleep(1); // Keep the application running
   }
   return EXIT_SUCCESS;
```

MAKEFILE:

```
obj-m := hardware_interrupt.o
KDIR := /lib/modules/$(shell uname -r)/build
TDIR := /home/rameshchlm/BSP_BOOT/linux-rba5d2x
PWD := $(shell pwd)
host:
	$(MAKE) -C $(KDIR) M=$(PWD) modules
```

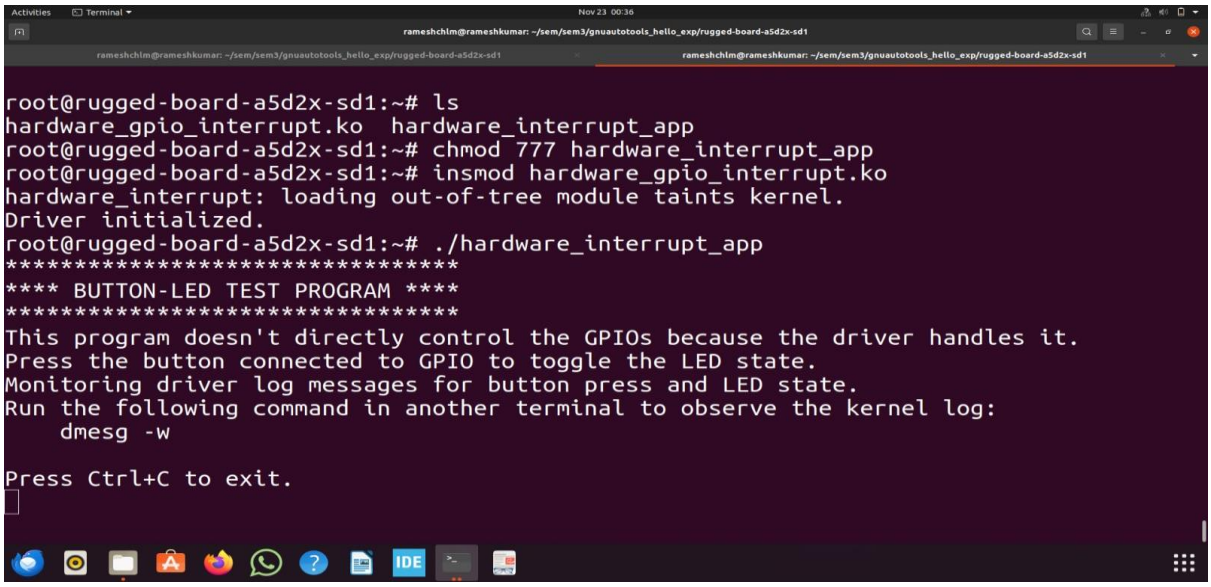
target:

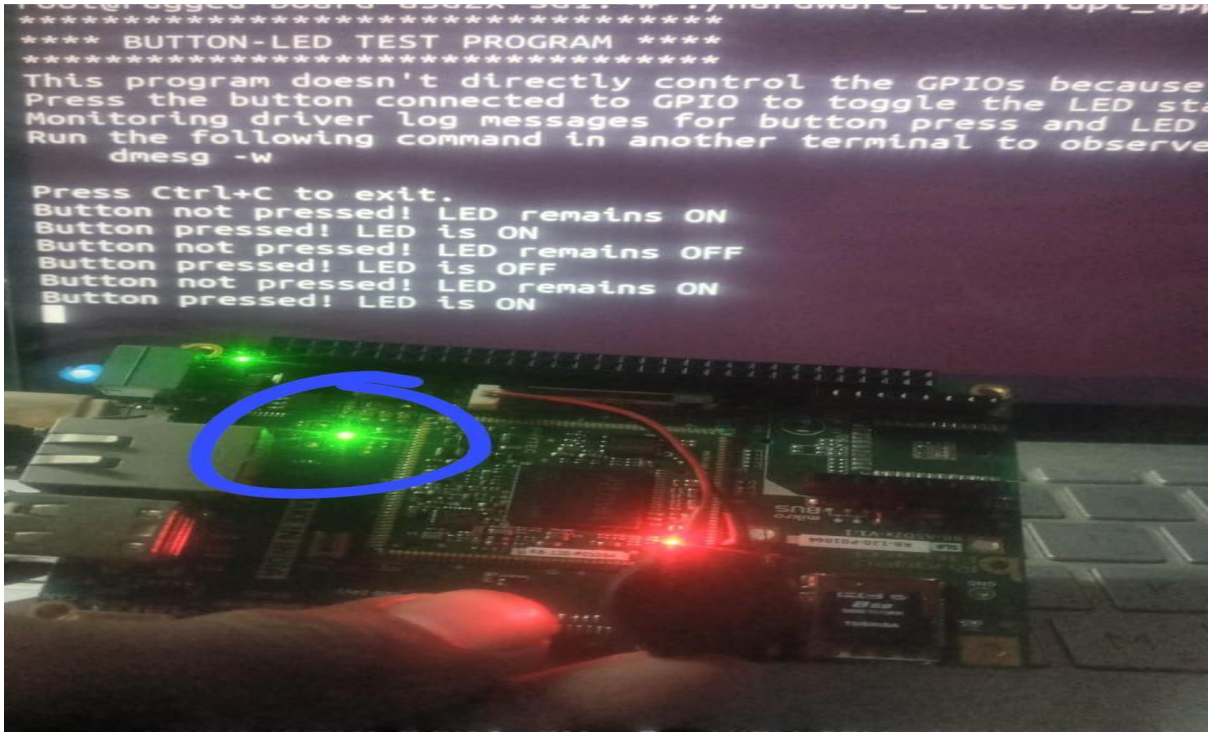
$(MAKE) -C $(TDIR) M=$(PWD) modules

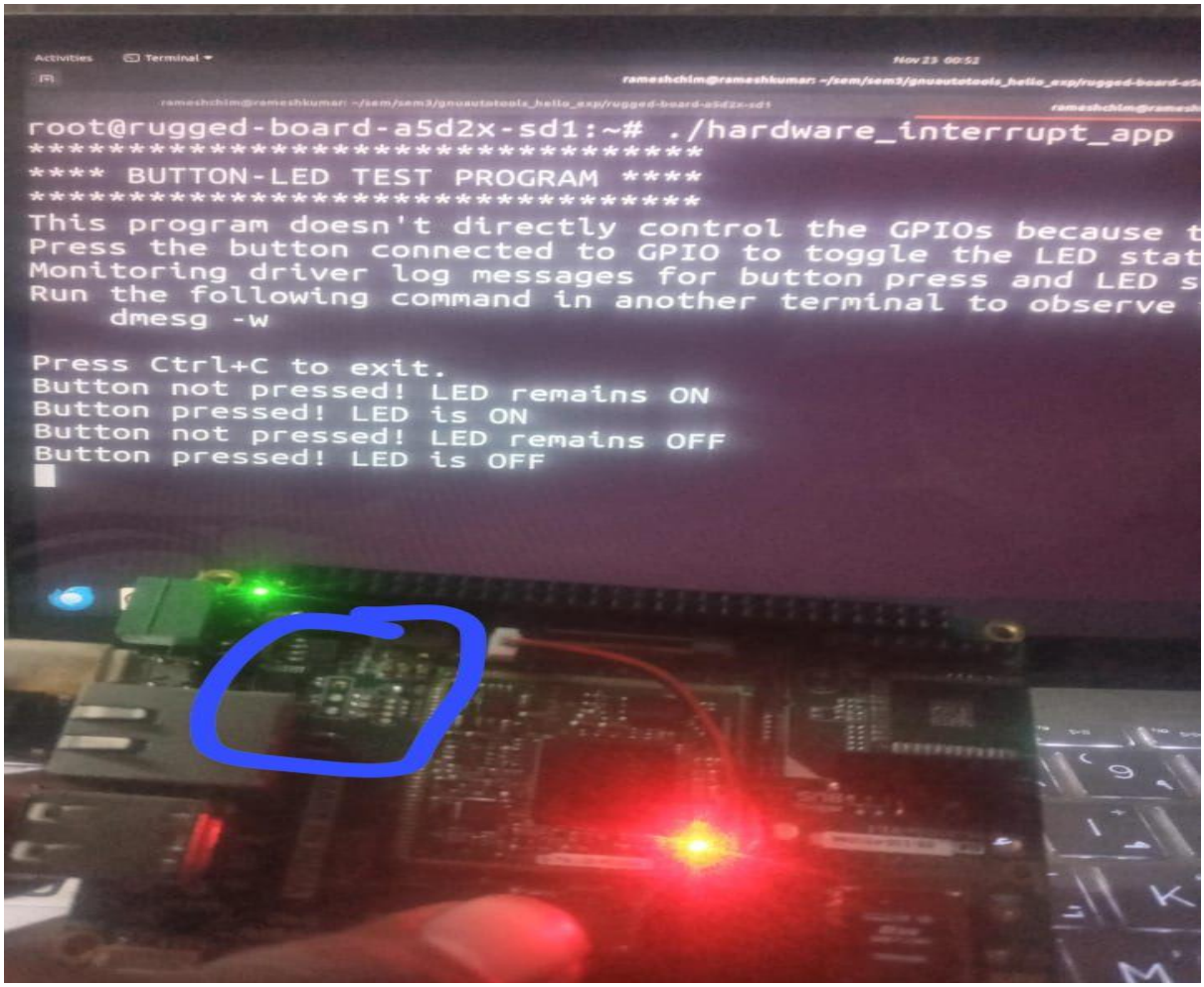
clean:

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

OUTPUT:






Activities
Terminal
root@rugged-board-a5d2x-sd1:~# ./hardware_interrupt_app
****
**** BUTTON-LED TEST PROGRAM ****
****
This program doesn't directly control the GPIOs because t
Press the button connected to GPIO to toggle the LED stat
Monitoring driver log messages for button press and LED s
Run the following command in another terminal to observe
dmesg -w
Press Ctrl+C to exit.
Button not pressed! LED remains ON
Button pressed! LED is ON
Button not pressed! LED remains OFF
Button pressed! LED is OFF

RESULT: