GENERAL PURPOSE INPUT OUTPUT CHARACTER DRIVER

Contents

- **What is GPIO?**
- GPIO or General-Purpose Input Output is one of the most frequent terms which you might have come across with. A GPIO is a signal pin on an integrated circuit or board that can be used to perform digital input or output functions. The GPIO behavior (input or output) is controlled at the run time by the application software/firmware by setting a few registers. Typical applications include reading/writing values from/to analog or digital sensors/devices, driving a led, driving a clock for I2C communication, Generating triggers for external components, Issuing Interrupts, etc.
- All the microcontrollers will be having a few registers to control the gpio functions. Those register names will vary based on the microcontroller.`
-
- **What is the GPIO port?**
- There will be many GPIO pins in the microcontroller. Those pins are grouped by some name. So, that one group is called GPIO port. Simply, the collection of GPIOs is called GPIO port. For example, If you take the STM32 microcontroller, it has 5 GPIO ports (PORT A, PORT B, PORT C, PORT D, PORT E). Each port has 16 GPIO pins. Like this, other microcontrollers will have few GPIO ports.

How to access GPIO in Linux kernel

If anyone wants to access the GPIO from the Kernel GPIO subsystem, they have to follow the below steps.

- Verify whether the GPIO is valid or not.
- If it is valid, then you can request the GPIO from the Kernel GPIO subsystem.
- Export the GPIO to sysfs (This is optional).
- Set the direction of the GPIO
- Make the GPIO to High/Low if it is set as an output pin.
- Set the debounce-interval and read the state if it is set as an input pin. You enable IRQ also for edge/level triggered.
- Then release the GPIO while exiting the driver or once you are done.

Validate the GPIO:

- Before using the GPIO, we must check whether the GPIO that we are planning to use is valid or not. To do that we have to use the below API.

bool gpio_is_valid(int gpio_number);

- **gpio_number :** GPIO that you are planning to use
- It returns **false** if it is not valid otherwise, it returns **true**.

- This API determines whether the GPIO number is valid or not. Sometimes this call will return **false** even if you send a valid number. Because that GPIO pin might be temporarily unused on a given board.
- **Request the GPIO**
- Once you have validated the GPIO, then you can request the GPIO using the below APIs.
- **int gpio_request(unsigned gpio, const char *label);**
- **gpio :** GPIO that you are planning to use.
- **label :** label used by the kernel for the GPIO in sysfs. You can provide any string that can be seen in **cat /sys/kernel/debug/gpio.** You can see the GPIO assigned to the particular GPIO.
- It returns **0** in success and a negative number in failure.
- **.**There are other variants also available. You can use any one of them based on your need.
- **int gpio_request_one(unsigned gpio, unsigned long flags, const char *label);**
  – Request one GPIO.
- **int gpio_request_array(struct gpio *array, size_t num);** – Request multiple GPIOs.

Export the GPIO

- For debugging purposes, you can export the gpio which is allocated using the **gpio_request()**to the sysfs using the below API
- **int gpio_export(unsigned int gpio, bool direction_may_change);**
- **gpio :** GPIO that you want to export.
- **direction_may_change :** This parameter controls whether user space is allowed to change the direction of the GPIO.
- **true** – Can change, **false** – Can't change.
- Returns zero on success, else an error.

Unexport the GPIO

- If you have exported the GPIO using the **gpio_export(),** then you can unexport this using the below API.

void gpio_unexport(unsigned int gpio);

- **gpio :** GPIO that you want to unexport
- **Set the direction of the GPIO**
- When you are working on the GPIOs, you have set the GPIO as output or input. The below APIs are used to achieve that.
- This API is used to set the GPIO direction as input.

int gpio_direction_input(unsigned gpio);

- **gpio :** GPIO that you want to set the direction as input.
- Returns zero on success, else an error.

- This API is used to set the GPIO direction as output.

int gpio_direction_output(unsigned gpio, int value);

- **gpio :** GPIO that you want to set the direction as output.
- **value :** The value of the GPIO once the output direction is effective.

Change the GPIO value:

- Once you set the GPIO direction as an output, then you can use the below API to change the GPIO value.
- **gpio_set_value(unsigned int gpio, int value);**
- **gpio :** GPIO that you want to change the value.
- **value :** value to set to the GPIO. **0** – Low, **1** – High.
- Read the GPIO value
- You can read the GPIO's value using the below API.
- **int gpio_get_value(unsigned gpio);**
- **gpio :** GPIO that you want to read the value.
- It returns the GPIO's value. 0 or 1.

Release the GPIO:

- Once you have done with the GPIO, then you can release the GPIO which you have allocated previously. The below API help in that case.

void gpio_free(unsigned int gpio);

- **gpio :** GPIO that you want to release.
- There are other variants also available. You can use any one of them based on your need.

void gpio_free_array(struct gpio *array, size_t num);

Release multiple GPIOs.

What Is Interrupt?

An interrupt is a signal emitted by a device attached to a computer or from a Program within the computer. It requires the operating system to stop and figure out what to do next. An interrupt temporarily stops or terminates a service or a current process.

Importance Of Interrupt In Character Driver

An interrupt is a hardware signal from a device to a CPU. It tells the CPU that the device needs attention, and that the CPU should stop performing what it is doing and respond to the device.

Get the IRQ number for the GPIO

int gpio_to_irq(unsigned gpio);
gpio: which gpio pin do you want select as an interrupt line.

How to Request IRQ

request_irq(unsigned int irq,irq_handler_t handler,unsigned long irqflags,
const char*devname,void*devid);
irq:Interrupt line to allocate
handler:Function to be called when the IRQ occurs
irqflags: interrupt type flags
devname:an ascii name for the claiming device
devid: a cookie passed back to the handler function

Interrupt Flags

IRQF_TRIGGER_RISING
IRQF_TRIGGER_FALLING
IRQF_TRIGGER_HIGH
IRQF_TRIGGER_Low

Set the debounce-interval:

The below API is used for sets debounce time for a GPIO.raspberry pi is not supporting that. That's why we have commented on that line in our source code. Instead, we used some software hack to eliminate multiple times IRQ being called for a single rising edge transition. You can remove the software hack and uncomment that gpio_set_debounce() if your microcontroller supports this.

Int gpio_set_debounce(unsigned gpio,unsigned debounce);

gpio : GPIO that you want to set the debounce value.
Debounce:Delay of the debounce.
It returns 0 on success. Otherwise, <0.

IRQ_HANDLED

It is a MACRO which means that we did have a valid interrupt and handled it.

local_irq_restore();

This macro defined in the same header file and does the opposite thing it restores the interrupt flag and enables interrupts.

local_irq_save();

Defined in the include/linux/irqflags.h header file.
It saves the state of the IF flag of the eflags register and disables interrupts on the local processor.

Hardware Required

Raspberry Pi
Breadboard
Resistor
LED
Push-button

Bring up Raspberry PI

Install Raspberry Pi OS (64-bit) with desktop in the SD card.
Then install the kernel header using sudo apt install raspberrypi-kernel-header

Accessing the input GPIO in Linux Kernel
Verify the GPIO is valid or not.
If it is valid, then you can request the GPIO from the Kernel GPIO subsystem.
Set the direction of the GPIO as an input
Set the debounce-interval
Read the GPIO.
You enable IRQ also for edge/level triggered if you need it.
Then release the GPIO while exiting the driver or once you are done.

Testing the Device Driver:
Build the driver by using Makefile (sudo make)
Load the driver using sudo insmod driver.ko
Just press the Push button.
That output LED should be toggled

Drivercode

```
/* raising interrupt using GPIO pin on raspberry pi 3b+*/

/***************************************************************************//**
*
*
*
*
***************************************************************************/
#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/delay.h>
#include <linux/uaccess.h>  //copy_to/from_user()
#include <linux/gpio.h>     //GPIO
#include <linux/interrupt.h>
#include <linux/err.h>
#include<linux/workqueue.h>  /* for workqueueus */


/* Since debounce is not supported in Raspberry pi, I have addded this to disable
** the false detection (multiple IRQ trigger for one interrupt).
** Many other hardware supports GPIO debounce, I don't want care about this even
** if this has any overhead. Our intention is to explain the GPIO interrupt.
** If you want to disable this extra coding, you can comment the below macro.
** This has been taken from : https://raspberrypi.stackexchange.com/questions/8544/gpio-interrupt-debounce
**
** If you want to use Hardaware Debounce, then comment this EN_DEBOUNCE.
**
*/
#define EN_DEBOUNCE

#ifdef EN_DEBOUNCE
#include <linux/jiffies.h>

extern unsigned long volatile jiffies;
unsigned long old_jiffie = 0;
#endif

//LED is connected to this GPIO
#define GPIO_21_OUT (21)

//LED is connected to this GPIO
```

```
#define GPIO_25_IN  (25)

//GPIO_25_IN value toggle
unsigned int led_toggle = 0;

//This used for storing the IRQ number for the GPIO
unsigned int GPIO_irqNumber;


/*--------------------------------------------------------*/
/*tasklet function declaration */
void tasklet_fn(struct tasklet_struct *);

void workqueue_fn(struct work_struct *work);


/* declaring  work queue static method */
//DECLARE_WORK(workqueue, workqueue_fn);
DECLARE_DELAYED_WORK(workqueue, workqueue_fn);
/* declaring tasklet static method */
DECLARE_TASKLET(tasklet, tasklet_fn);


//Interrupt handler for GPIO 25. This will be called whenever there is a raising edge detected.
static irqreturn_t gpio_irq_handler(int irq,void *dev_id)
{
  static unsigned long flags = 0;
  pr_info("interrupt raised\n");

#ifdef EN_DEBOUNCE
   unsigned long diff = jiffies - old_jiffie;
   if (diff < 30)
   {
     return IRQ_HANDLED;
   }old_jiffie = jiffies;
#endif

  local_irq_save(flags);
  led_toggle = (0x01 ^ led_toggle);                        // toggle the old value
  gpio_set_value(GPIO_21_OUT, led_toggle);                 // toggle the GPIO_21_OUT
  pr_info("Interrupt Occurred : GPIO_21_OUT : %d ",gpio_get_value(GPIO_21_OUT));
  /* scheduling work queue */
  //schedule_work(&workqueue);
            //schedule_work_on(2, &workqueue);  /* scheduling work on cpu 2 */
           schedule_delayed_work(&workqueue,msecs_to_jiffies(3000));
           printk(KERN_INFO "Delayed workqueue scheduled\n");
           // tasklet_schedule(&tasklet);    /* scheduling tasklet */
  local_irq_restore(flags);
  return IRQ_HANDLED;
}

void tasklet_fn(struct tasklet_struct *tasklet)
{

           pr_info("-----------------------------------\n");
           pr_info("executing tasklet function\n");
           pr_info("the data: %d\n",45);
```

```
        pr_info("---------------------------------------\n");


}

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

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);


/*------------------- work queue function ----------------*/
void workqueue_fn(struct work_struct *work)
{
        pr_info("---------------------------------------\n");
        pr_info("executing work queue function\n");

pr_info("the data: %d\n",45);
        pr_info("---------------------------------------\n");



}

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

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);


/*------------------- work queue function ----------------*/
void workqueue_fn(struct work_struct *work)
{
        pr_info("---------------------------------------\n");
        pr_info("executing work queue function\n");
        pr_info("---------------------------------------\n");




}
/**************** 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);
/*********************************************************/
```

```
//File operation structure
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 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)
{
  uint8_t gpio_state = 0;

  //reading GPIO value
  gpio_state = gpio_get_value(GPIO_21_OUT);

  //write to user
  len = 1;
  if( copy_to_user(buf, &gpio_state, len) > 0) {
    pr_err("ERROR: Not all the bytes have been copied to user\n");
  }

  pr_info("Read function : GPIO_21 = %d \n", gpio_state);


  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)
{
  uint8_t rec_buf[10] = {0};

  if( copy_from_user( rec_buf, buf, len ) > 0) {
    pr_err("ERROR: Not all the bytes have been copied from user\n");
  }

  pr_info("Write Function : GPIO_21 Set = %c\n", rec_buf[0]);

  if (rec_buf[0]=='1') {
    //set the GPIO value to HIGH
    gpio_set_value(GPIO_21_OUT, 1);
  } else if (rec_buf[0]=='0') {
    //set the GPIO value to LOW
    gpio_set_value(GPIO_21_OUT, 0);
  } else {
    pr_err("Unknown command : Please provide either 1 or 0 \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_err("Cannot allocate major number\n");
    goto r_unreg;
  }
  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_del;
  }

  /*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 \n");
    goto r_device;
  }

  //Output GPIO configuration
  //Checking the GPIO is valid or not
  if(gpio_is_valid(GPIO_21_OUT) == false){
    pr_err("GPIO %d is not valid\n", GPIO_21_OUT);
    goto r_device;
  }

  //Requesting the GPIO
  if(gpio_request(GPIO_21_OUT,"GPIO_21_OUT") < 0){
    pr_err("ERROR: GPIO %d request\n", GPIO_21_OUT);
    goto r_gpio_out;
  }

  //configure the GPIO as output
  gpio_direction_output(GPIO_21_OUT, 0);

  //Input GPIO configuratioin
  //Checking the GPIO is valid or not
  if(gpio_is_valid(GPIO_25_IN) == false){
    pr_err("GPIO %d is not valid\n", GPIO_25_IN);
  goto r_gpio_in;
  }

  //Requesting the GPIO
  if(gpio_request(GPIO_25_IN,"GPIO_25_IN") < 0){
    pr_err("ERROR: GPIO %d request\n", GPIO_25_IN);
    goto r_gpio_in;
  }

  //configure the GPIO as input
  gpio_direction_input(GPIO_25_IN);

  /*
  ** I have commented the below few lines, as gpio_set_debounce is not supported
  ** in the Raspberry pi. So we are using EN_DEBOUNCE to handle this in this driver.
  */
 #ifndef EN_DEBOUNCE
  //Debounce the button with a delay of 200ms
  if(gpio_set_debounce(GPIO_25_IN, 200) < 0){
    pr_err("ERROR: gpio_set_debounce - %d\n", GPIO_25_IN);
    //goto r_gpio_in;
```

```
  }
#endif

  //Get the IRQ number for our GPIO
  GPIO_irqNumber = gpio_to_irq(GPIO_25_IN);
  pr_info("GPIO_irqNumber = %d\n", GPIO_irqNumber);

  if (request_irq(GPIO_irqNumber,             //IRQ number
                  (void *)gpio_irq_handler,   //IRQ handler
                   IRQF_TRIGGER_RISING,       //Handler will be called in raising edge
                  "etx_device",               //used to identify the device name using this IRQ
                  NULL)) {                    //device id for shared IRQ
    pr_err("my_device: cannot register IRQ ");
    goto r_gpio_in;
  }



  pr_info("Device Driver Insert...Done!!!\n");
  return 0;
r_gpio_in:
  gpio_free(GPIO_25_IN);
r_gpio_out:
  gpio_free(GPIO_21_OUT);
r_device:
  device_destroy(dev_class,dev);
r_class:
  class_destroy(dev_class);
r_del:
  cdev_del(&etx_cdev);
r_unreg:
  unregister_chrdev_region(dev,1);

  return -1;
}

/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
{
  free_irq(GPIO_irqNumber,NULL);
  gpio_free(GPIO_25_IN);
  gpio_free(GPIO_21_OUT);
  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("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("A simple device driver - GPIO Driver (GPIO Interrupt) ");
MODULE_VERSION("1.33");
```

Procfs in Linux

Introduction

- Most of Linux users have at least heard of proc. Some of you may wonder why this folder is so important.
- On the root, there is a folder titled “proc”.
- This folder is not really on /dev/sda or wherever you think the folder resides. This folder is a mount point for the procfs (Process Filesystem) which is a filesystem in memory.
- Many processes store information about themselves on this virtual filesystem.
- Procfs also stores other system information.
- proc files can also be used to control and modify kernel behavior on the fly. The proc files need to be writable in this case. but they are meant to retrieve info from kernel related process. The proc file system is also very useful when we want to debug a kernel module. While debugging we might want to know the values of various variables in the module or maybe the data that the module is handling. In such situations, we can create a proc entry for ourselves and dump whatever data we want to look into in the entry.

Creating procfs directory:

You can create the directory under /proc/* using the below API.

struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);

name: The name of the directory that will be created under /proc.
parent: In case the folder needs to be created in a subfolder under/proc a pointer to the same is passed else it can be left as NULL.

CreatingProcessEntry:

The creation of proc entries has undergone a considerable change in kernel version 3.10 and above. In

this post, we will see one of the methods we can use in Linux kernel version 3.10. Let us see how we can create proc entries in version 3.10.The function is defined in proc_fs.h

struct proc_dir_entry *proc_create (const char *name, umode_t mode, struct proc_dir_entry *parent, const struct file_operations *proc_fops)

name: The name of the proc entry
mode: The access mode for proc entry
parent: The name of the parent directory under /proc. If NULL is passed as a parent, the /proc directory will be set as a parent.

proc_fops: The structure in which the file operations for the proc entry will be created.

Note: The above proc_create is valid in the Linux Kernel v3.10 to v5.5. From v5.6, there is a change in this API. The fourth argument const struct file_operations *proc_fops is changed to const struct proc_ops *proc_ops.

If you are using the kernel version below 3.10, please use the below functions to create proc entry.

create_proc_read_entry();
create_proc_entry() ;

The create_proc_entry is a generic function that allows creating both the read as well as the write entries.
create_proc_read_entry is a function specific to create only read entries.

It is possible that most of the proc entries are created to read data from the kernel space that is why the kernel developers have provided a direct function to create a read proc entry.

Now, we need to create file_operations structure proc_fops in which we can map the read and write functions for the proc entry.In version less than 3.10 the same file operations structure is used. but later they created separate structure proc_fops

```
static struct proc_ops proc_fops =
{
        .proc_open = open_proc,
        .proc_read = read_proc,
        .proc_write = write_proc,
        .proc_release = release_proc
};
```

Remove Proc Entry:

Proc entry should be removed in the Driver exit function using the below function.

void remove_proc_entry(const char *name, struct proc_dir_entry *parent);

Basic GPIO Character driver code using procfs functions:

```
#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/proc_fs.h>
#include <linux/err.h>

/*
** I am using the kernel 5.10.27-v7l. So I have set this as 510.
** If you are using the kernel 3.10, then set this as 310,
** and for kernel 5.1, set this as 501. Because the API proc_create()
** changed in kernel above v5.5.
**
*/
#define LINUX_KERNEL_VERSION  510

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

int32_t value = 0;
char etx_array[20]="try_proc_array\n";
static int len = 1;


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

/*
** 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);
static long     etx_ioctl(struct file *file, unsigned int cmd, unsigned long arg);

/****************** Procfs Functions *******************/
static int      open_proc(struct inode *inode, struct file *file);
static int      release_proc(struct inode *inode, struct file *file);
static ssize_t  read_proc(struct file *filp, char __user *buffer, size_t length,loff_t *
offset);
static ssize_t  write_proc(struct file *filp, const char *buff, size_t len, loff_t * off);

/*
** 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,
};
#if ( LINUX_KERNEL_VERSION > 505 )

/*
** procfs operation sturcture
*/
static struct proc_ops proc_fops = {
        .proc_open = open_proc,
        .proc_read = read_proc,
        .proc_write = write_proc,
        .proc_release = release_proc
};
#else //LINUX_KERNEL_VERSION > 505
/*
** procfs operation structure
*/
static struct file_operations proc_fops = {
        .open = open_proc,
        .read = read_proc,
        .write = write_proc,
```

```
        .release = release_proc
};

#endif //LINUX_KERNEL_VERSION > 505

/*
** This function will be called when we open the procfs file
*/
static int open_proc(struct inode *inode, struct file *file)
{
    pr_info("proc file opend.....\t");
    return 0;
}

/*
** This function will be called when we close the procfs file
*/
static int release_proc(struct inode *inode, struct file *file)
{
    pr_info("proc file released.....\n");
    return 0;
}

/*
** This function will be called when we read the procfs file
*/
static ssize_t read_proc(struct file *filp, char __user *buffer, size_t length,loff_t *
offset)
{
    pr_info("proc file read.....\n");
    if(len)
    {
        len=0;
    }
    else
    {
        len=1;
        return 0;
    }

    if( copy_to_user(buffer,etx_array,20) )
    {
        pr_err("Data Send : Err!\n");
    }
```

```
    return length;;
}

/*
** This function will be called when we write the procfs file
*/
static ssize_t write_proc(struct file *filp, const char *buff, size_t len, loff_t * off)
{
    pr_info("proc file wrote.....\n");

    if( copy_from_user(etx_array,buff,len) )
    {
        pr_err("Data Write : Err!\n");
    }

    return len;
}

/*
** 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_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;
        }

        /*Create proc directory. It will create a directory under "/proc" */
        parent = proc_mkdir("etx",NULL);

        if( parent == NULL )
        {
            pr_info("Error creating proc entry");
            goto r_device;
        }

        /*Creating Proc entry under "/proc/etx/" */
        proc_create("etx_proc", 0666, parent, &proc_fops);

        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)
{
        /* Removes single proc entry */
        //remove_proc_entry("etx/etx_proc", parent);

        /* remove complete /proc/etx */
        proc_remove(parent);

        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("surya");
MODULE_DESCRIPTION("Simple Linux device driver (procfs)");
MODULE_VERSION("1.6");
```

Sysfs in Linux Kernel

Introduction

- Sysfs is a virtual filesystem exported by the kernel, similar to /proc. The files in Sysfs contain information about devices and drivers. Some files in Sysfs are even writable, for configuration and control of devices attached to the system.
- Sysfs is always mounted on /sys.
- The directories in Sysfs contain the hierarchy of devices, as they are attached to the computer.
- Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices. The sysfs is tied to the device driver model of the kernel. The procfs is used to export the process-specific information and the debugfs is used to use for exporting the debug information by the developer.
- Before getting into the sysfs we should know about the Kernel Objects.

Kernel Objects

- The heart of the sysfs model is the kobject. Kobject is the glue that binds the sysfs and the kernel, which is represented by struct kobject and defined in <**linux/kobject.h**>. A struct

kobject represents a kernel object, maybe a device or so, such as the things that show up as directory in the sysfs filesystem.

- Kobjects are usually embedded in other structures.

```
#define KOBJ_NAME_LEN 20
struct kobject
{
 char *k_name;
 char name[KOBJ_NAME_LEN];
 struct kref kref;
 struct list_head entry;
 struct kobject *parent;
 struct kset *kset;
 struct kobj_type *ktype;
 struct dentry *dentry;
 };
```

struct kobject

- **name** (Name of the kobject. Current kobject is created with this name in sysfs.)
- **parent** (This is kobject's parent. When we create a directory in sysfs for the current kobject, it will create under this parent directory).
- **ktype** (the type associated with a kobject).
- **kset** (a group of kobjects all of which are embedded in structures of the same type).
- **sd** (points to a sysfs_dirent structure that represents this kobject in sysfs.).
- **kref** (provides reference counting)

It is the glue that holds much of the device model and its sysfs interface together.So, kobject is used to create kobject directory in /sys.

Create a directory in /sys

struct kobject * kobject_create_and_add (const char * name, struct kobject * parent);

This function is used to create directory.

name : the name for the kobject.

parent: the parent kobject of this kobject, if any.

If you pass kernel_kobj to the second argument, it will create the directory under /sys/kernel/. If you pass firmware_kobj to the second argument, it will create the directory under /sys/firmware/. If you pass fs_kobj to the second argument, it will create the directory under /sys/fs/. If you pass NULL to the second argument, it will create the directory under /sys/.

This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned. When you are finished with this structure, call kobject_put(); and the structure will be dynamically freed when it is no longer being used.

Create Sysfs file

Using the above function, we will create a directory in /sys. Now we need to create a sysfs file, which is used to interact user space with kernel space through sysfs. So, we can create the sysfs file using sysfs attributes.

Attributes are represented as regular files in sysfs with one value per file. There are loads of helper functions that can be used to create the kobject attributes. They can be found in the header file sysfs.h

Kobj_attribute is defined as,

```
struct kobj_attribute
{
struct attribute attr;
ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count);
};
```

- **attr:** the attribute representing the file to be created,
- **Show:** the pointer to the function that will be called when the file is read in sysfs,
- **store :** the pointer to the function which will be called when the file is written in sysfs.

- We can create an attribute using __ATTR macro.
  __ATTR(name, permission, show_ptr, store_ptr);

Store and Show functions

Then we need to write show and store functions.

ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);

ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count);

Store function will be called whenever we are writing something to the sysfs attribute,

Show function will be called whenever we are reading the sysfs attribute.

Create sysfs file

To create a single file attribute we are going to use 'sysfs_create_file'.

int sysfs_create_file (struct kobject * kobj, const struct attribute * attr);

- **kobj:** object we're creating for.
- **attr:** attribute descriptor.

One can use another function ' sysfs_create_group ' to create a group of attributes.

Remove function

Once you have done with the sysfs file, you should delete this file using **sysfs_remove_file();**

These functions should be used in exit/cleanup functions.

void sysfs_remove_file (struct kobject * kobj, const struct attribute * attr);

- **Kobj:** object we're creating for.
- **attr:** attribute descriptor.

Basic GPIO Character driver code using procfs functions:

```
#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;
```

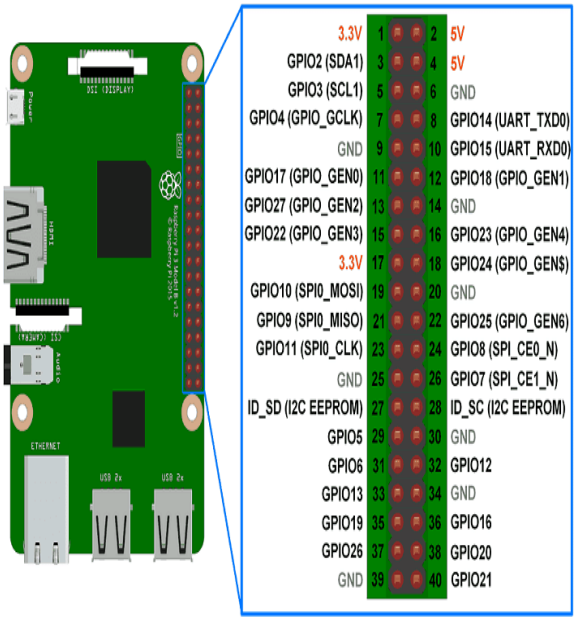
```
}

/*
** 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("surya");
MODULE_DESCRIPTION("Simple Linux device driver (sysfs)");
MODULE_VERSION("1.8");
```

BareMetal programming for GPIO pins for raspberry pi

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL29	FSEL29 - Function Select 29 000 = GPIO Pin 29 is an input 001 = GPIO Pin 29 is an output 100 = GPIO Pin 29 takes alternate function 0 101 = GPIO Pin 29 takes alternate function 1 110 = GPIO Pin 29 takes alternate function 2 111 = GPIO Pin 29 takes alternate function 3 011 = GPIO Pin 29 takes alternate function 4 010 = GPIO Pin 29 takes alternate function 5	R/W	0
26-24	FSEL28	FSEL28 - Function Select 28	R/W	0
23-21	FSEL27	FSEL27 - Function Select 27	R/W	0
20-18	FSEL26	FSEL26 - Function Select 26	R/W	0
17-15	FSEL25	FSEL25 - Function Select 25	R/W	0
14-12	FSEL24	FSEL24 - Function Select 24	R/W	0
11-9	FSEL23	FSEL23 - Function Select 23	R/W	0
8-6	FSEL22	FSEL22 - Function Select 22	R/W	0
5-3	FSEL21	FSEL21 - Function Select 21	R/W	0
2-0	FSEL20	FSEL20 - Function Select 20	R/W	0

Table 6-4 – GPIO Alternate function select register 2



0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W

Bare metal code for gpio

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/poll.h>
#include <linux/ioport.h>
#include <linux/errno.h>
#include <linux/proc_fs.h>
#include <asm/io.h>
#include <linux/interrupt.h>
#include <linux/sched.h>
#define PHERIADDR 0X3F200000
#define FSEL 0X08
#define SET 0x1C
#define RSET 0X28
unsigned  int* virtualaddr;
unsigned  int* config;
unsigned  int* set;
unsigned  int* reset;
int init(void)
{

        printk("driver is initilalized\n");
        virtualaddr=ioremap(PHERIADDR,PAGE_SIZE);
        config=(unsigned int *)((char*)virtualaddr+FSEL);
        set=(unsigned int *)((char*)virtualaddr+SET);
        reset=(unsigned int *)((char*)virtualaddr+RSET);
        *config=*config&(~(0x7 << 18));
        *config=*config|(0x1 << 18);
        *set=*set|(0x1 << 26);
        return 0;
}
void exit(void)
{
        printk("driver is unloaded\n");
        *reset=*reset|(0x1 << 26);
        iounmap(virtualaddr);

}
module_init(init);
module_exit(exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Low Level Learning");
MODULE_DESCRIPTION("Test of writing drivers for RASPI");
MODULE_VERSION("1.0");
```

KERNEL TIMERS

What is a timer in general?

A timer is a specialized type of clock used for measuring specific time intervals.

There are two types of time:

1.Absoluet

ime

2.Relative

time

Absolute time :-It is used to know the date &time of the day,we depend on hardware chip called real-time clock (RTC).

Relative time:

It is used to schedule the task or to create delays,from kernelpoint of view we call it as kernel timers.

Kernel timers are classified in to two different parts:

- Standard timer
- High resolution timer

Standard Timers:

These are measured in terms of jiffies.

Jiffies can be incremented from system bootup time.

Jiffies is defined in terms of constants called Hertz's,which is the number of times jiffies is incremented in one second.

If 1000 interrupts raises then it will be one

jiffie. These are declared in **<linux/jiffies.h>**

header file.

To generate the delay in terms of milli seconds we will take the help of jiffies

Jiffies is a variable which is defined globally and can be accessed directly in any driver

(we will have read only permissions).

The timer is represented by a kernel data structure struct timer_list which is defined in **<linux/timer.h>** header file

```
struct timer_list

{

unsigned long expires;

void (*function)(unsigned long);

unsigned long data;

};
```

For initialization need to use this **timer_setup** function.

void timer_setup(timer, function, data

Argument:

timer – the timer to be initialized

function – Callback function to be called when the timer expires. In this callback function, the argument will be **struct timer_list ***.

data – data has to be given to the callback function

Modifying Kernel Timer's timeout

int mod_timer(struct timer_list * timer, unsigned long expires);

This function is used to modify a timer's timeout.

This is a more efficient way to update the expires field of an active timer

(if the timer is inactive it will be activated)

The function returns whether it has modified a pending timer or not.

Return values:

0 -**mod_timer**of an inactive timer1 -

mod_timer of an active timer **Stop a**

Kernel Timer

The below functions will be used to deactivate the kernel timers.

int del_timer(struct timer_list * timer);

This will deactivate a timer.

This works on both active and inactive timers.

The function returns whether it has deactivated a pending timer or not.

Return values:

- - **`del_timer`** of an inactive timer

1 - **`del_timer`** of an active timer

Check Kernel Timer status

int timer_pending(const struct timer_list* timer);

This will tell whether a given timer is currently running, or not.

Return values:

The function returns whether the timer is pending or not.

0 – **timer** is not pending

• – **`timer`** is pending

CODE

```
/***************************************************************************//**
 *  \file       driver.c
*
 *  \details    Simple Linux device driver (Kernel Timer)
*
 *  \author     EmbeTronicX
*
 *  \Tested with Linux raspberrypi 5.10.27-v7l-embetronicx-custom+
*
*******************************************************************************/#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/timer.h> #include
 <linux/jiffies.h>#include
 <linux/err.h>
//Timer Variable
#define TIMEOUT 5000            //milliseconds

 static struct timer_list etx_timer;static
 unsigned int count = 0;


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


 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);
/******************************************************/


//File operation structure

static struct file_operations fops =
{
        .owner          = THIS_MODULE,
        .read           = etx_read,
```

```
                .write          = etx_write,
                .open           = etx_open,
                .release        = etx_release,
};

 //Timer Callback function. This will be called when timer expiresvoid
 timer_callback(struct timer_list * data)
{
        /* do your timer stuff here */
        pr_info("Timer Callback function Called [%d]\n",count++);


        /*
            Re-enable timer. Because this function will be called only first time.If we re-enable this will
            work like periodic timer.
        */
```

```
        mod_timer(&etx_timer, jiffies + msecs_to_jiffies(TIMEOUT));
}


/*
** 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_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;
    }

    /* setup your timer to call my_timer_callback */
    timer_setup(&etx_timer, timer_callback, 0);                    //If you face some issues and
 using older kernel version, then you can try setup_timer API(Change Callbackfunction's argument to
 unsingned long instead of struct timer_list *.

    /* setup timer interval to based on TIMEOUT Macro */ mod_timer(&etx_timer,
    jiffies + msecs_to_jiffies(TIMEOUT));

    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)
{
    /* remove kernel timer when unloading module */
    del_timer(&etx_timer); 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("EmbeTronicX <embetronicx@gmail.com>");
 MODULE_DESCRIPTION("A simple device driver - Kernel Timer");
 MODULE_VERSION("1.21");
```

MAKEFILE

```
 obj-m += driver.o
```

```
  KDIR = /lib/modules/$(shell uname -r)/buildall:
      make -C $(KDIR)    M=$(shell pwd) modules


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

OUTPUT

Build the driver by using Makefile (sudo make)

- Load the driver using **`sudo insmod driver.ko`**
- Now see the Dmesg (**`dmesg`**)

```
linux@embetronicx-VirtualBox: dmesg

[ 2253.635127] Device Driver Insert...Done!!!
[ 2258.642048] Timer Callback function Called [0]
[ 2263.647050] Timer Callback function Called [1]
[ 2268.652684] Timer Callback function Called [2]
[ 2273.658274] Timer Callback function Called [3]
[ 2278.663885] Timer Callback function Called [4]
[ 2283.668997] Timer Callback function Called [5]
[ 2288.675109] Timer Callback function Called [6]
[ 2293.680160] Timer Callback function Called [7]
[ 2298.685771] Timer Callback function Called [8]
[ 2303.691392] Timer Callback function Called [9]
[ 2308.697013] Timer Callback function Called [10]
[ 2313.702033] Timer Callback function Called [11]
[ 2318.707772] Timer Callback function Called [12]
```

- See timestamp. That callback function is executing every 5 seconds.
- Unload the module using **`sudo rmmod driver`**

High Resolution Timer

Kernel Timers are bound to **jiffies**. But this High Resolution Timer (HRT) is bound with 64-bit **nanoseconds** resolution.

There are many ways to check whether high resolution timers are available,

In the **/boot**directory, check the kernel config file. It should have a line like **CONFIG_HIGH_RES_TIMERS=y**.

- Check the contents of **`/proc/timer_list`**. For example, the **.resolution** entry showing 1 nanosecond and event_handler as hrtimer_interrupt in /proc/timer_list indicate that high resolution timers are available.
- Get the clock resolution using the **`clock_getres`** system call.

We need to include the **<linux/hrtimer.h>**,in order to use kernel timers.Kernel timers are described by the **hrtimer**structure, defined in **`<linux/hrtimer.h>.`**

```
struct hrtimer {
  struct rb_node node;
  ktime_t expires;
  int (* function) (struct hrtimer *);
  struct hrtimer_base * base;
};
```

Where,

node– red black tree node for time ordered insertion

expires – the absolute expiry time internal in the hr timers representation. The time is related to the clock on which the timer is based.

function – timer expiry callback function. This function has an integer return value, which should be either **HRTIMER_NORESTART** or HRTIMER_RESTART for a recurring timer. In the restart case, the callback must set a new expiration time before returning.

base– pointer to the timer base (per CPU and per clock)

The **hrtimer**structure must be initialized by init_hrtimer_#CLOCKTYPE.

Initialize High Resolution Timer

Void hrtimer_init(struct hrtimer *timer,clockid_t clock_id,

enum hrtimer_mode mode);

Arguments:

timer– the timer to be initialized

clock_id– the clock to be used

The clock to use is defined in **./include/linux/time.h**and represents the various clocksthat the system supports .

mode– timer mode absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL)

Start High Resolution Timer

Once a timer has been initialized, it can be started with the below-mentioned function.

int hrtimer_start(struct hrtimer *timer, ktime_t time, const enumhrtimer_mode mode);

This call is used to (Re)start an hrtimer on the current CPU.

Arguments:

timer– the timer to be added

time– expiry time

mode– expiry mode: absolute (HRTIMER_MODE_ABS) or relative (HRTIMER_MODE_REL)

Returns: 0 on success 1 when the timer was active

Stop High Resolution Timer

Using the below function, we can able to stop the High Resolution Timer.

int hrtimer_cancel(struct hrtimer * timer);

This will cancel a timer and wait for the handler to finish.

Arguments:

timer– the timer to be canceled

Returns:

- 0 when the timer was not active
- 1 when the timer was active

Check High Resolution Timer's status

The below-explained functions are used to get the status and timings.

```
ktime_t hrtimer_get_remaining(const struct hrtimer * timer);
```

This is used to get the remaining time for the timer.

Arguments:

timer– hrtimer to get the remaining time

Returns: Returns the remaining time.

Using High Resolution Timer In Linux Device Driver

we have added the high resolution timer. The steps are mentioned below.

- Initialize and start the timer in the init function
- After the timeout, a registered timer callback will be called.
- In the timer callback function again we are forwarding the time period and return . We have to do this step if we want a periodic timer. Otherwise, we can ignore that time forwarding and return call back function.
- Once we are done, we can disable the timer.

CODE

```
 •   \file       driver.c
*
 •   \details    Simple Linux device driver (High Resolution Timer)
*
 •   \author     EmbeTronicX
*
 •   \Tested with Linux raspberrypi 5.10.27-v7l-embetronicx-custom+
*
*******************************************************************************/#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/hrtimer.h>#include
  <linux/ktime.h> #include
  <linux/err.h>
//Timer Variable
  #define TIMEOUT_NSEC        ( 1000000000L )          //1 second in nano seconds
  #define TIMEOUT_SEC         ( 4 )                    //4 seconds

  static struct hrtimer etx_hr_timer;static
  unsigned int count = 0;


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


  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);
/******************************************************/


//File operation structure
static struct file_operations fops =
```

```
{
        .owner          = THIS_MODULE,

        .read           = etx_read,
        .write          = etx_write,

        .open           = etx_open,
        .release        = etx_release,

};

 //Timer Callback function. This will be called when timer expiresenum hrtimer_restart
 timer_callback(struct hrtimer *timer)
{
        /* do your timer stuff here */

      pr_info("Timer Callback function Called [%d]\n",count++);
      hrtimer_forward_now(timer,ktime_set(TIMEOUT_SEC, TIMEOUT_NSEC));return
      HRTIMER_RESTART;
}
```

```
/*
** 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)
{
         ktime_t ktime;

        /*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;
    }

    ktime = ktime_set(TIMEOUT_SEC, TIMEOUT_NSEC); hrtimer_init(&etx_hr_timer,
    CLOCK_MONOTONIC, HRTIMER_MODE_REL);etx_hr_timer.function =
    &timer_callback;

    hrtimer_start( &etx_hr_timer, ktime, HRTIMER_MODE_REL);


    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)
{
    //stop the timer hrtimer_cancel(&etx_hr_timer);
    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("EmbeTronicX <embetronicx@gmail.com>"); MODULE_DESCRIPTION("A
 simple device driver - High Resolution Timer");MODULE_VERSION("1.22");
```

Makefile:

```
 obj-m += driver.o

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

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

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

Output:

- Build the driver by using Makefile (`sudo make`)
- Load the driver using **`sudo insmod driver.ko`**
- Now see the Dmesg (**`dmesg`**)

```
[  +0.000026] Device Driver Insert...Done!!!
[  +5.000073] Timer Callback function Called [0]
[  +5.000052] Timer Callback function Called [1]
[Apr24 15:05] Timer Callback function Called [2]
[  +5.000057] Timer Callback function Called [3]
[  +5.000058] Timer Callback function Called [4]
[  +5.000060] Timer Callback function Called [5]
[  +5.000053] Timer Callback function Called [6]
[  +5.000058] Timer Callback function Called [7]
[  +5.000064] Timer Callback function Called [8]
```

- `See timestamp. That callback function is executing every 5seconds.`
- Unload the module using **`sudo rmmod driver`**