Exercise Manual
For

CE3103
Embedded Programming

Practical Exercise #3

GPIO Interface Access
through
User Space and Kernel Space
(on Raspberry Pi Embedded Platform)

Venue: Hardware Laboratory 2
(Location: N4-01b-05)

COMPUTER ENGINEERING COURSE

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Learning Objectives

These practical exercises are designed to enable students to practice programming techniques to access GPIO interface from user space and kernel space. Access from user space is to be done by developing a C based application program with system calls to the descriptor file based GPIO interface, while Loadable Kernel Module is to be used to access the GPIO interface from kernel space. The GPIOs used in the exercises are those available on the RPi board that are connected to several LEDs and a Push-Button.

Equipment and accessories required

i) One Raspberry Pi (RPi 3) board with SDCard installed with Raspbian.
ii) One Ubuntu based PC installed (for SSH remote access to RPi)
iii) One Ethernet cable.
iv) One Micro-USB adapter/cable (for supplying power to RPi board)

---

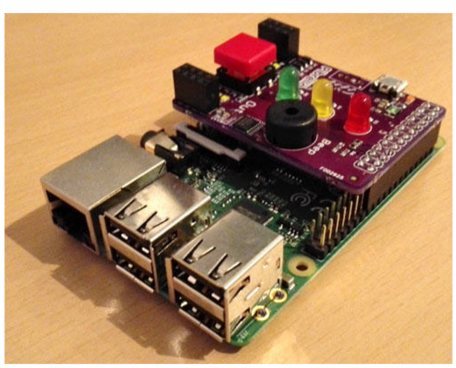
1. Introduction

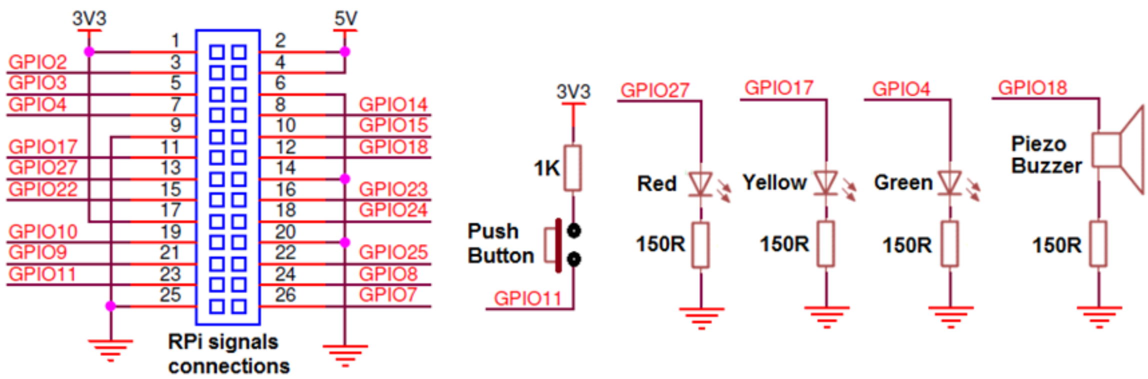
In embedded systems that use operating system such as Linux, programs are usually coded to run in the user space. To access the underlying hardware such as the General Purpose Input/Output (GPIO), the user space programs will then use a file based interface made available by the operating system. This separation of the user space from the hardware serves as a boundary restriction that protects the hardware from errant or malicious user programs.

On the other hand, Kernel modules are pieces of code that run in the kernel space, and hence can have unrestricted access to the underlying hardware. A loadable kernel nodule can also be loaded and unloaded into the kernel space upon demand. This feature allows the user to extend the functionality of the kernel without the need to reboot the system. It is particularly useful for system that can have different components/hardware been added or removed dynamically from the system during its execution.

2. Hardware Platform

For these exercises, you will learn how to develop programs for user space and kernel space that perform simple I/O functions on the RPi platform. The following figure shows the Pibrella add-on board that contains basic electronics components interfaced to the RPi board through its expansion headers.





The above circuit schematic diagram shows the components that are available on the Pibrella board which consists of the following:

- 3 LED displays connected through the GPIO4, GPIO17 and GPIO27 pins of RPi board
- 1 Push button input connected through RPi's GPIO11
- 1 Piezo Buzzer driven through RPi's GPIO18

3. Exercise A: User Space Access of GPIO

3.1 Use the program code (incomplete) given in Appendix A as reference,
- ➢ code a program **gpio.c** to blink the Green and Yellow LEDs on the Pibrella board.

3.2 With reference to the lecture note on Topic 5 (or google on internet),
- ➢ add a function in your program to monitor the Push Button that is connected to GPIO11 pin
- ➢ code the program to blink the red LED whenever the Push Button is detected to be pressed.

4. Exercise B: Kernel Space Access of GPIO

4.1. Use the program code provided in the Appendix B,
- ➢ code a Loadable Kernel Module **hello_lkm.c** program for the RPi board.
- ➢ compile the module and test that it works as expected.

(Refer to Topic 6 lecture notes on on detailed procedure.)

4.2 You are now going to modify your LKM code to perform simple I/O functions on the RPi platform using the Integer-based GPIO interface, where every GPIO in the system is represented by a simple unsigned integer.

The GPIOs can be easily accessed and controlled from kernel space using the various functions that are provided through the <linux/gpio.h> (of kernel space), which are shown on the next page.

```
static inline bool gpio_is_valid(int number)
static inline int  gpio_request(unsigned gpio, const char *label)
static inline int  gpio_export(unsigned gpio, bool direction_may_change)
static inline int  gpio_direction_output(unsigned gpio, int value)
static inline int  gpio_set_value(unsigned gpio)
static inline int  gpio_direction_input(unsigned gpio)
static inline int  gpio_get_value(unsigned gpio)
static inline int  gpio_set_debounce(unsigned gpio, unsigned debounce)
static inline void gpio_unexport(unsigned gpio)
static inline void gpio_free(unsigned gpio)
static inline int  gpio_to_irq(unsigned gpio)
```

The followings describe the typical sequence of enabling and accessing the GPIO using the above functions:

i) Request to use the GPIO on the system using `gpio_request()`. This is a way to claim the ownership of the GPIO, preventing other drivers from accessing the same GPIO.
Note:
- If in doubt, you can first check whether the GPIO number is valid on the system by using `gpio_is_valid()`.
- GPIO should be returned to the system once done by calling `gpio_free()`.

ii) After taking the ownership of the GPIO, one can set the direction using `gpio_direction_output()` or `gpio_direction_input()`.

iii) GPIO's output state can be toggled using `gpio_set_value()` when configured as output.

iv) Debounce-interval and reading of the GPIO's state when configured as input can be performed using `gpio_set_debounce()` and `gpio_get_value()`.

v) GPIO line can also be mapped to IRQ using `gpio_to_irq()`. One can then define the edge/level that the interrupt should be triggered upon, and register a handler that will be run whenever the interrupt occurs. (Refer to lecture note for more detail).

- Appendix C shows the (incomplete) program code of the LKM based GPIO device driver **gpio_lkm.c**.
- Study and complete the program code.
- Execute the program to show that the device driver functions as expected.

Appendix A: Sample program code for accessing GPIO from user space - gpio.c

```
#include <linux/gpio.h> /* you may want to look at the contents of this file, to understand
                           the various elements of the data structure shown in the code below */
   :
int main(int argc, char *argv[ ])
{ int fd0 = open("/dev/gpiochip0",O_RDWR);  // open the file descriptor
  struct gpiochip_info cinfo;
  ioctl(fd0,GPIO_GET_CHIPINFO_IOCTL,&cinfo); // get the chip information
  fprintf(stdout, "GPIO chip 0: %s, \"%s\", %u lines\n", cinfo.name, cinfo.label, cinfo.lines);

  struct gpiohandle_request req_GY; // Green and Yellow
  struct gpiohandle_data data_GY;    // for data bit
  req_GY.lines = 2;  // 2 pins in this handler
  req_GY.lineoffsets[0] = 4;  //pin 4 - Green LED
  req_GY.lineoffsets[1] = 17;  // pin 17 - Yellow LED
  req_GY.flags = GPIOHANDLE_REQUEST_OUTPUT; // set them to be output
  data_GY.values[0] = 1; // set initial value of Green LED to High (ON)
  data_GY.values[1] = 0; // set initial value of Yellow LED to Low (OFF)
  ioctl(fd0, GPIO_GET_LINEHANDLE_IOCTL, &req_GY); // now get the line handler req_GY

  for (int i = 0; i < 5; ++i){
    ioctl(req_GY.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data_GY); // output data bits
       usleep(1000000);   //sleep for 1 second
       data_GY.values[0] = !data_GY.values[0]; // toggle
       data_GY.value[1] = !data_GY.values[1];
    } //for
    close(req_GY.fd); // release line
    close(fd0); // close the file
    exit(EXIT_SUCCESS);
}//main
```

Appendix B: Simple LKM program code - hello_lkm.c

```
#include <linux/init.h>   // needed by the macros module_init and exit
#include <linux/module.h> // needed by all modules
#include <linux/kernel.h> // needed by KERN_ definition

static int __init hello_init(void)
{ printk(KERN_ALERT "Hello from kernel world\n");
  return 0; // 0 for success, negative for failure
}

Static void __exit hello_exit(void)
{ printk(KERN_ALERT "Goodbye from kernel world\n");
}

module_init(xxxx); // macro to execute module's initialize routine
module_exit(xxxx); // macro to execute module's exit routine

MODULE_LICENSE("GPL");
MODULE_AUTHOR(“CE3103");
MODULE_DESCRIPTION("Simple Hello module");
MODULE_VERSION(“V1");
```

Appendix C: GPIO Device Driver LKM - gpio_lkm.c

```
#include <linux/module.h>
    :
#include <linux/gpio.h>
#include <linux/interrupt.h>

static unsigned int ledGreen = 4;        // GPIO4 (through header pin 7) connected to Green LED
static unsigned int pushButton = 11;     // GPIO11 (through pin 23)connects to push Button
static unsigned int irqNumber;           // share IRQ num within file
static bool ledOn = 0;                   // used to toggle state of LED


// The GPIO IRQ Handler function
static irq_handler_t rpi_gpio_isr(unsigned int irq, void *dev_id, struct pt_regs *regs)
{       :                                              // toggle the LED state
        :                                              // set LED accordingly
   printk(KERN_ALERT "GPIO Interrupt!\n"));
   return (irq_handler_t) IRQ_HANDLED;                 // announce IRQ handled
}


// The LKM exit function
static void __exit rpi_gpio_exit(void)
{  gpio_set_value(ledGreen, 0);                        // turn the LED off
   gpio_free(ledgreen);                                // free the LED GPIO
   gpio_free(pushbutton);                              // free the Button GPIO
   free_irq(irqNumber, NULL);                          // free the IRQ number, no *dev_id
   printk(KERN_ALERT "Goodbye from the GPIO LKM!\n");
}

// The LKM initialization function
static int __init rpi_gpio_init(void)
{  int result = 0;
   printk(KERN_ALERT " Initializing the GPIO LKM\n");
```

```
    ledOn = true;                               // default for LED is ON
    gpio_request(ledGreen, "sysfs");            // request for LED GPIO
    gpio_direction_output(ledGren, ledOn);      // set in output mode and turn on LED

    gpio_request(pushButton, "sysfs");          // request for push Button GPIO
    gpio_direction_input(pushButton);           // set up as input
    gpio_set_debounce(pushButton, 1000);        // debounce delay of 1000ms

    irqNumber = gpio_to_irq(pushButton);        // map pushbutton to IRQ number
    printk(KERN_ALERT "Button mapped to IRQ: %d\n", irqNumber);

    // Requests for an interrupt line
    result = request_irq(irqNumber,             // interrupt number requested
             (irq_handler_t) rpi_gpia_isr,      // isr handler function
             IRQF_TRIGGER_RISING,               // trigger on rising edge
             "rpi_gpio_handler",                // used in /proc/interrupts
             NULL);                             // *dev_id for shared interrupt lines - NULL
    return result;
 }
```