Lab 1

Using the Raspberry Pi 3 – Kernel Module and Interrupt Service Routine

WEEK-1

Objective

In this part, students will learn how to access the General Purpose Input/Output (GPIO) ports on the RPi3 through a Kernel Module.

Prelab

- Before coming to this lab, look up the use of the functions **mmap** and **ioremap** and their arguments.
- Review the bit-wise operations in C/C++, and the concept of "Bit Masking".

Background

A **kernel module** is a piece of code that can be added to the kernel at run-time. It can then be removed, as well. Modules extend the functionality of the kernel while the system is up and running. That is, we do not need to rebuild an entire kernel to add new functions, and we do not need to build a large kernel that contains all functionality. There are different types of modules, including device drivers. You could find a sample code on how a write a kernel module at this link.

Lab Procedure

1. Turn ON/OFF LEDs using Kernel Module

You will create a kernel module that simply turns on the LEDs of the auxiliary board when it is installed, and turns the LEDs off when it is removed.

- In Kernel space, you do not have access to the *wiringPi* library. Therefore, you will need to access the GPIO ports directly through the GPSET, GPCLR and GPLEV registers. Before you can use the ports, you need to configure them as inputs or outputs through the GPFSEL register (see the week 1 part).

  Hint 1: You will need to use *ioremap* when accessing the I/O registers in a kernel module. You will need to include a specific header file.

  Hint 2: Consider the following function:

```
void iowrite32(u32 value, void *addr)
```

- Something to note about kernel modules is that they DO NOT contain a main(). Instead, they contain two functions:

  ```
  int init_module(void)
  void cleanup_module(void)
  ```

  The init_module function contains code that is run when the module is installed. This function should return 0 unless an error has occurred. cleanup_module is the code that is run when the module is removed.

  Alternatively, you can name your functions differently, i.e.,

  ```
  int when_installed(void)
  void when_removed(void)
  ```

  Your code should contain the lines:

  ```
  // function runs when module is installed
  module_init(when_installed);

  // function runs when module is removed
  module_exit(when_removed);
  ```

- To avoid warnings when installing the module, add the following line to your source code MODULE_LICENSE("GPL");

- To compile the module, you may need to define the symbols: __KERNEL__ and MODULE. Those macros indicate being part of the kernel, but not permanent, respectively. This can be done one of two ways. First, you can define the symbols at the beginning of your source code, which would look like #define MODULE. Another way is to define the symbols when you compile in Eclipse by going to the properties and clicking on symbols under the compiler settings.

  Important: a Kernel module is compiled into object code, but it is not linked into a complete executable. That is, the Linker should not be invoked. The TA will show you how to compile a Kernel module.

- There are three helpful shell commands to use when dealing with modules:

  1. lsmod: lists all of the modules currently installed in the kernel
  2. insmod NAME.ko: installs the module whose filename is NAME.ko
  3. rmmod NAME: removes the module with name: NAME (notice the .ko is not included)
  4. modinfo NAME.ko: Can get the information about the loaded module

  You need **sudo** to use 2 and 3.

- In order to debug your module code, you cannot use the printf() function. Instead, you can use the printk() function and then check the printed lines using dmesg command in the terminal. Modify your kernel module so that the message "MODULE INSTALLED" is printed when the module is installed, and the message "MODULE REMOVED" is printed when the module is removed. Install and remove your module several times. What do you see when you run the *dmesg* command?

WEEK-2

Objective

In the second part of the lab, students will learn how to trigger an interrupt through a push button. Also, write their own interrupt service routine (ISR) to handle the interrupt.

Prelab

- Investigate and include a brief description for the following functions: request_irq, free_irq.

You should look at *section 6 General Purpose I/O(GPIO)* in the manual: **BCM2837-ARM-Peripherals** (on Canvas under *Modules > Other Material*). Pay attention to the registers associated to configuring and detecting level and edge events. You will need those for this lab.

Lab Procedure

Detect Push-Button using an Interrupt

You will add an *Interrupt Service Routine (ISR)* to your modules from Week-1. The purpose of the *ISR* is to handle one of five events, which correspond to the five push buttons on the auxiliary board. The corresponding GPIO ports and associated registers should be configured so that pushing any of the buttons causes an interrupt. The handler will change state of the LEDs based on the button pressed. You will associate first four buttons to four LEDs on the board, respectively to turn them on individually. And link the last button to turn OFF all the LEDs in one go. You should configure these interrupts to be **rising-edge-sensitive**, so program the corresponding GPIO registers accordingly.

Check the file **ece4220lab1_isr.c**. It is a template that illustrates the basic steps needed to configure an interrupt and attach a handler.

Note *WiringPi* provides a way to configure GPIO interrupts in user space on the RPi. You may explore that approach for your future work.

References

- Wiring Pi
- Raspberry Pi GPIO Pinout