Lab 1
Using the Raspberry Pi 3 – User Space, Command Line and Kernel Space

Week 1

Objective

In the first part of this lab, students will learn how to develop applications for the Raspberry Pi 3boards (RPi3). Students will also learn how to access the General Purpose Input/Output (GPIO) ports on the RPi3 from a user-space program.

Setting up the RPi3

Before we can start with the first lab assignment, we need to set up the Raspberry Pi boards. For that, we will use the document "Lab How Tos.pdf", which can be found on the Canvas website, under *Modules > Laboratory*. Have this document ready for the first lab session. You should also look at the document "Lab Guidelines", which can be found in the same Canvas module.

Make sure you can create a "Hello World" project on Eclipse. Run it on the RPi3.

General Purpose I/O (GPIO)

The Raspberry Pi 3 has many I/O ports and a total of 28 Digital, general purpose I/O lines available which are located on the header labeled "GPIO". All of the I/O lines can be programmed individually as inputs or outputs. There are many registers associated to the GPIO ports, which are aligned on 32-bit (4 byte) boundaries.

To use the ports as inputs or outputs, they need to be configured accordingly. The Function Select registers (GPFSEL) are used for that purpose. Modifying individual bits in the GPFSEL registers changes whether the corresponding I/O pins are inputs (default) or outputs.

When configured as inputs, the value of the GPIO pins can be read from the GPIO Pin Level registers (GPLEV) registers. When configured as outputs, the GPIO Pin Output Set registers (GPSET) are used to set the pins, and the GPIO Pin Output Clear registers (GPCLR) are used to clear the pins.

For more details, you can check chapter 6 on the "BCM2835_ARM-Peripherals" document. Be careful, that document has some typos and errors. Read the "General notes for the Raspberry Pi3 development".

In order to use the I/O registers in our applications, we need to map these register's addresses into our program memory. In *user space* programs, this is done by using the mmap() function. This function requires a *file descriptor*, so we need to open up the special file /dev/mem in our program first, using the open() command. Since we are mapping 32-bit registers into our program, we can use an unsigned long ptr for access to each port.

Doing the mapping above and accessing the registers directly can be a bit tricky at first. Fortunately, there is a nice library called **WiringPi** (https://projects.drogon.net/raspberry-pi/wiringpi/functions/) which provides an API with many functions that make interfacing with the GPIO ports a lot simpler. WiringPi also includes a command-line utility, *gpio*, which allows to program and setup the GPIO pins directly from a terminal or using shell scripts.

In this first week of Lab 1, you will use the WiringPi library to accomplish the tasks described in the Lab Procedure section below. In the second week you will learn how to do the mapping directly, among other interesting things.

Auxiliary Board

You will have access to an auxiliary board, which can be connected to the RPi3. This auxiliary board contains five push buttons, three LEDs, and a speaker. The push buttons should be connected to GPIO ports configured as inputs, and the LEDs and speaker should be connected to GPIO ports configured as outputs. Figure 1 shows an image of the auxiliary board.

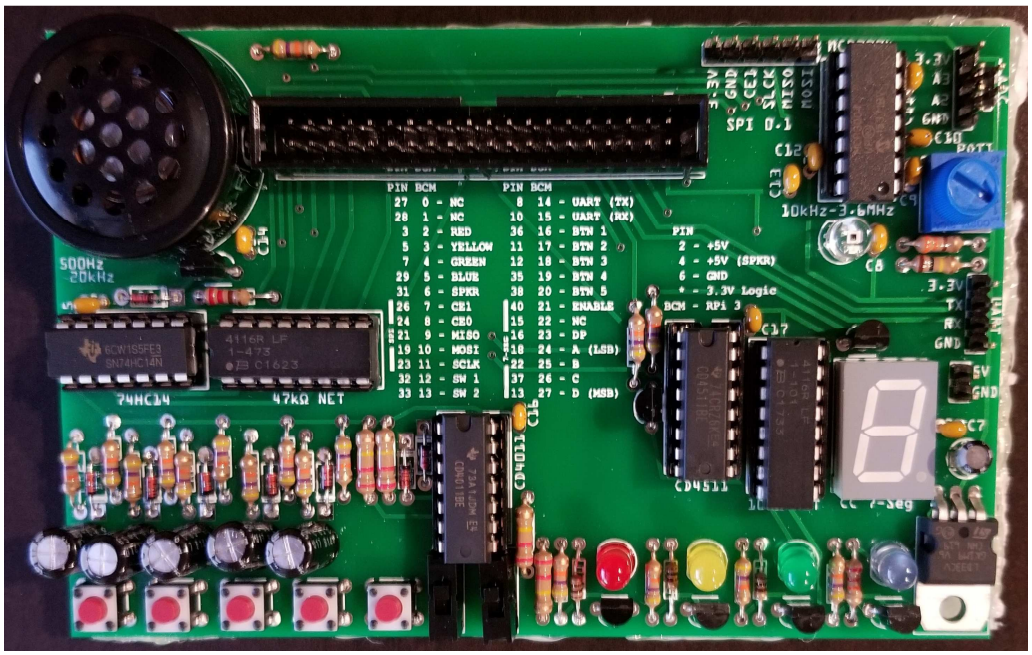

Figure 1. Auxiliary board.

WiringPi uses its own GPIO numbering scheme. You need to be careful when connecting your boards, and when using the wiringPi API. Figure 2 shows the RPi3 GPIO pinout, and Figure 3 shows the wiringPi GPIO pinout.

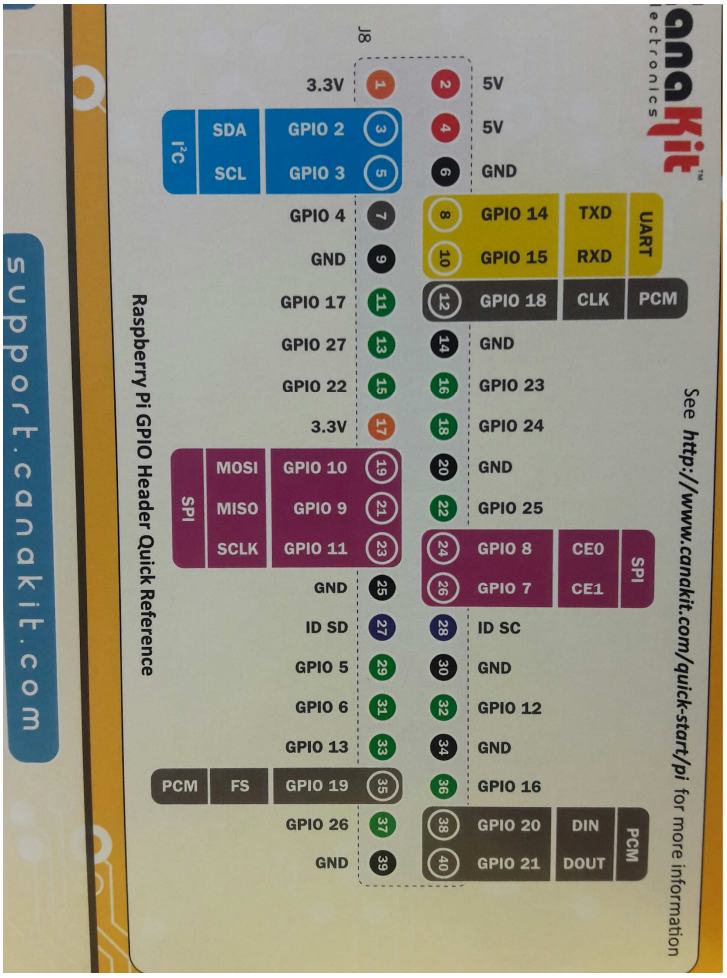

Figure 2. RPi3 pinout.

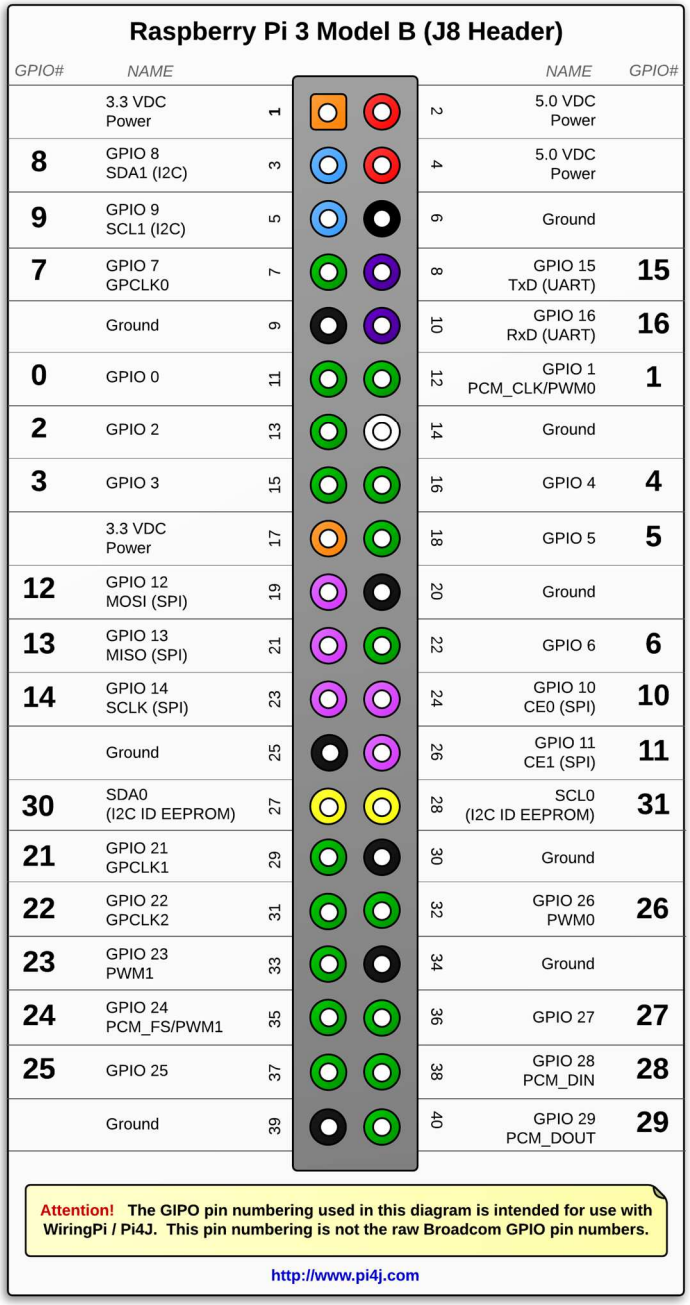

Figure 3. WiringPi pinout. Taken from http://pi4j.com/pins/model-3b-rev1.html

Lab Procedure

1. Turning On/Off the LEDs: Your first task is to write a C program that alternately turns On and Off two of the LEDs on the auxiliary board, every one second (when one LED is On, the other is Off, and vice-versa).

   Consider the following wiringPi functions:
   - int wiringPiSetup(void)
   - int wiringPiSetupGpio(void)
   - void pinMode(int Pin, int Value)
   - void digitalWrite(int Pin, int Value)

   Hint: You need to include the wiringPi.h header file in your source code, and you need to include the wiringPi library when linking your program (-l).

2. Producing a sound on the speaker: Your second task is to create a square wave and output it to the speaker on the auxiliary board.

   You are to create a program that first scans for user input through the keyboard. Valid entries are integers from 1 to 5. Depending on the number entered, the corresponding push button will then be used to activate the generation of a square wave on the speaker. The wave is generated by constantly toggling the value of the GPIO port connected to the speaker. For example, if the user enters a "1", then the square wave should start after the first button is pushed. If any other button is pushed, it should be ignored. Try different frequencies until you hear a "nice" sound.

   In addition to the wiringPi functions mentioned earlier, consider the following ones:
   - void pullUpDnControl(int Pin, int pud_mode)
   - int digitalRead(int Pin)

   Hint: Between toggles of the port, you need to "waste" time to create the square wave. How can you achieve that?

Post Lab Questions (Part 1):

1. Imagine that in the user-space program you want to terminate the sound by pressing a key on the keyboard. How could you do that? Are there any implementation difficulties? You do NOT have to implement it; only discuss it in the report.

2. How do you think you could solve the frequency issues to get a "nice", consistent sound in the speaker?

Week 2

Objective

In the second part of lab 1, students will learn how to access the General Purpose Input/Output (GPIO) ports on the RPi3 from the Linux command line, and through a Kernel Module.

Prelab

- Before coming to the lab on the second week, look up the use of the functions `mmap` and `ioremap` and their arguments.
- Review the bitwise operations in C/C++, and the concept of "Bit Masking".

Background

WiringPi comes with a GPIO utility that allows command line access to the GPIO ports in the Raspberry Pi. Check http://wiringpi.com/the-gpio-utility/ as you work on Part 1 of this week's lab procedure.

A **kernel module** is a piece of code that can be added to the kernel at runtime. 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.

Lab Procedure

3. Command Line: Your first task this second week will be to access the GPIO port directly from the command line. You will need to configure the corresponding ports as inputs or outputs.

    - Turn On and Off all the LEDs on the small auxiliary board. Try using both the GPIO numbering scheme and the wiringPi numbering scheme.
    - Read the status of a couple of the push buttons using the command line. Read the values without pressing the buttons, and while pressing the buttons. What values do you get? Are the results what you expected?
      Hint: In week 1, you needed to use the function pullUpDnControl(). What is the purpose of that function? There is a corresponding command line instruction that you may need…

4. Kernel Module: For your second task in week two, 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:
  ```
  module_init(when_installed);   // function runs when module is installed
  module_exit(when_removed);     // function runs when module is 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:
  i. `lsmod`: lists all of the modules currently installed in the kernel
  ii. `insmod` NAME.ko: installs the module whose filename is NAME.ko

iii. `rmmod` NAME: removes the module with name: NAME (notice the .ko is not included)

You need **sudo** to use ii. and iii.

- 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?