1. Introduction

Laboratories for the Embedded Systems part of ENCE260 will start in the second week of Term 4. Over the next three weeks you will be taken through a set of laboratory exercises to help you become familiar with embedded system programming and the UC FunKit. These labs are designed to assist you to complete your second assignment, which you can start at any time after team allocations. Demonstrations will be held during the last week of Term 4.

The purpose of this specific lab is to allow you to become familiar with input/output (I/O) ports and modular programming on an AVR microcontroller.

2. Getting started - Find and download the ucfk4 software libraries

2.1. Login to to the eng-git server: https://eng-git.canterbury.ac.nz

2.2. Open up a terminal window.

2.3. Using the mkdir command, create a new course directory somewhere from your working directory, e.g.,

```
../courses/ence260/
```

From this directory Step 2.4 will **automatically** create a new directory to hold your FunKit library.

2.4. At the command line, enter the following (note you will have to type rather than copy & paste from the .pdf):

```
git clone https://eng-git.canterbury.ac.nz/rmc84/ence260-ucfk4.git
```

Do not change the URL above (rmc84 is my (Richard Clare) user code – you do not need to change this to yours in the URL. You will be prompted to login after entering the above command – use your UC username and login.

A new directory called ence260-ucfk4 should appear.

2.5. Change your directory to the ence260-ucfk4 directory using the cd command:

```
cd ence260-ucfk4
```

2.6. In this directory are assorted sub-directories containing example applications, driver modules, testing utilities, etc. that are needed for the labs and assignment.

2.7. To list the directories and print a description of their contents type:

```
more README
```

You should see something like:

```
The directory structure is:
apps         --- contains a sub-directory for each application
drivers      --- device driver modules (hardware independent)
drivers/avr  --- device driver modules specifically for AVR architecture
drivers/test --- device driver modules for test scaffold
doc          --- documentation
etc          --- miscellaneous scripts and makefile templates
fonts        --- fonts and font creation program
utils        --- utility modules
labs         --- contains the laboratory exercises
```

3. Testing the UCFK4 board

For now, let's do a quick check to see that your UCFK4 board works. Here's what to do:

3.1. Change your working directory to ence260-ucfk4/apps/space12:

```
cd apps/space12
```

3.2. Connect the UCFK4 to the USB port of your computer with a USB cable having a mini-B USB connector. The green power LED should light.

3.3. Compile and load the space12 program into the flash memory of the UCFK4 microcontroller using:

```
make program
```

This will also start the program running. You should see some text displayed on the LED matrix. In the Erskine labs, you may see the error message:

```
dfu-programmer failed to release interface 0
```

Ignore this error throughout the course. Your program is still being compiled and loaded on the microcontroller. Changing the USB port can sometimes make this go away.

If you get the error

```
dfu-programmer: no device present.
```

This is probably because you have a program running on the microcontroller already. You need to press the reset switch (S2) before running make program. If you are using your own machine, this is because you have not set the USB permissions properly in your installation.

If you have installed avr-gcc on your own computer and it is using a newer version than 7.3.0 you may end up with many warnings of the form:

```
warning: array subscript 0 is outside array bounds of 'volatile uint8_t[0]'
{aka 'volatile unsigned char[]'} [-Warray-bounds]
```

The program will still compile and load, however it may be best to install an older version of avr-gcc (7.3.0 or similar) to avoid these warnings in the labs and the assignment.

3.4. Push the navswitch down. This will start the game. The navswitch can move your gun back and forth; pushing it fires a shell. Your task is to shoot the aliens before they get to the ground. If this is too easy, there are harder levels.

3.5. Change your working directory up two levels to the ence260-ucfk4 directory using:

```
cd ../..
```

4. Accessing the lab folders

Over the next three weeks you will be developing code based on examples contained within the /lab folder. Several lab folders from 1 to 3 are provided. For this specific lab you will be working in \lab1-ex1 to \lab1-ex5 folders.

Warning: even though you might be tempted to start moving or rearranging folders within this software library, don't do it. There are dependencies that cannot (easily) be changed. So, leave the directory structure alone and let's start writing some code.

5. Turning on an LED by writing to the I/O port registers

5.1. The next task is to write a program to turn on the blue LED connected to the micro-controller on pin #2 of port C. This pin needs to be initialised as an output so that we can drive the LED. To do this we must write to DDRC[1], the Data Direction Register for port C. Bit #2 of this register relates to the state of pin #2. To set a pin as an output, we need to write a logic 1 to the corresponding bit in DDRC. Note that if a pin is to be configured as an input the corresponding (to the pin) DDR bit value would be logic 0.

Open the file lab1-ex1/lab1-ex1.c in your favourite text editor. Here you should see a program skeleton that does nothing particularly exciting for the moment.

To set pin #2 as an output we will set the second bit of DDRC. Add the following line to the program:

```
DDRC |= (1 << 2);
```

This line should be placed in the initialisation section at the top of the main() function[2].

5.2. Next we need to set the state of the pin in order to drive the blue LED. To do this we need to write to bit #2 of the PORTC register[3]. The following code will do this:

```
PORTC |= (1 << 2);
```

You can place this code inside the main loop of the program.

5.3. Now compile the program by running the make program[4]

```
make
```

This should build the program lab1-ex1.out using the rules in the file called Makefile. If there are errors or warnings, fix them and try again!

---

[1] This is defined in the header file, avr/io.h

[2] Note that << is the left-shift operator and is useful here to create bit patterns for writing to registers. We start with a bit pattern for 1, 00000001, and left shift it twice (i.e., bit #2) to get 00000100, or 0x04 in hexadecimal.

[3] Also defined in the header file avr/io.h.

[4] make is an automated build utility and is useful for large projects with many files or when there are many compiler options.

5.4. Reset your UCFK4 board by pushing switch S2 and program it using the following command:

```
make program
```

This will load your program and provided there are no errors it will load your program to the UCFK4. With any luck, the blue LED will turn on. If not, check your program and try again.
Not that `make program` also re-compiles your program, so you do not need to enter both the `make` and `make program` commands every time.

6. Reading a button using the I/O port registers

Now we will deal with simple input. The goal is to modify your program to turn the LED on when a button (the white switch near R7) is pressed, and off when the button is released.

The button we will use is on pin #7 of port D. The data direction register for port D (DDRD) will need to have bit #7 set to zero. While this is the default, see if you can figure out the bit pattern for this and add code to ensure that this bit is cleared in DDRD.

To get the status of an input pin, we read its associated PIN register. The button state will be reflected in bit #7 of the PIND register.

6.1. Modify your previous program to use an if statement that checks the button pin and turns the LED on if the pin is high or off if the pin is low. Place this inside the while loop so that we are continuously checking the button state. Hint: you may want to refer to your Embedded Systems notes on Learn.

6.2. Compile the program using make and program the UCFK4 board as you did previously. You should be able to press and release the button to turn the blue LED on and off.

7. Abstracting the gory details

Abstraction is the key to programming. It allows us to hide unnecessary details. So instead of directly accessing the PORT registers from our main function, we will write functions that control the LED and read the button.

7.1. Open the `lab1-ex2/lab1-ex2.c` file and you will see some empty functions. Complete these functions so they can be used in main() to achieve the same functionality of the previous exercise.

7.2. Compile and program your UCFK4 board.

8. Writing simple C modules

The functions you have written to control the LED and read the button are useful for other programs. The best way to re-use these functions is to put related functions into their own files. In this case, the LED functions are put into their own file and the button functions are put into their own file. This approach is called modular programming. It helps to keep your

code tidy and manageable and facilitates building of large projects. It further helps to hide the gory details.

8.1. Open the files lab1-ex3/lab1-ex3.c, lab1-ex3/led.c, lab1-ex3/led.h, lab1-ex3/button.c, and lab1-ex3/button.h. You will see some familiar functions to implement a module, however, these are now contained in separate files. Look at the structure of these files to see how a C module is created.

8.2. Implement the functions in led.c and button.c and compile your program. Have a look at the list of files generated by the make command. Do you notice anything different from the previous exercise? **Hint**: check the header file dependencies with the previous example and notice the differences in the makefile.

9. Using a port I/O module

At this point you should have some idea how to program an I/O port for an AVR microcontroller. Unfortunately, other microcontrollers have different ways of performing port I/O[5]. This makes it harder to write portable programs, that is, programs that you can use on different computers. A solution to the portability problem is to use an abstraction layer for an I/O port. So rather than manipulating the I/O port registers directly, we use functions to do this.

Open up the header file /drivers/avr/pio.h, which defines the Programmable Input Output for the UCFK API. Now consider the following code:

```
#include "pio.h"
#define LED_PIO PIO_DEFINE (PORT_C, 2)
int main (void)
{
        pio_config_set (LED_PIO, PIO_OUTPUT_LOW);
        while (1)
        {
                pio_output_low (LED_PIO);
                pio_output_high (LED_PIO);
        }
        return 0;
}
```

Things to note:

9.1. This program includes the header file /drivers/avr/pio.h that has the function prototypes for the port I/O (PIO) routines.

9.2. PIO_DEFINE is a macro that assigns a unique integer to identify the port I/O pin that drives the LED[6]. The PIO_DEFINE macro is defined in pio.h. The first argument to the macro is the port (PORT_C here), and the second argument is the bit/pin for the I/O being defined (bit #2 here).

[5] For example, some require a data direction register bit to be programmed with a 1 for an input.
[6] In this case pin #2 of port C.

9.3. The pio_config_set function sets the data direction register; in this case to be an output. It also sets the port register so the initial state of the desired pin specified by LED_PIO is low.

9.4. From drivers/avr/pio.h, we can see that PIO_INPUT defines an input, and the pio_input_get function returns the state of the input.

9.5. The while loop alternately sets the desired pin low then high. This will happen at a rate for your eye not to notice any flicker. Here's what to do:

9.5.1. Edit lab1-ex4/button.c and lab1-ex4/led.c. Instead of accessing the I/O port registers directly, use the PIO module[7].

9.5.2. Compile this code and program your board. You should see the same behaviour as if you were using the port registers directly.

9.5.3. Change directory to the lab1-ex4/doc directory and run make. This should generate some PDF files. View the files module_dependencies.pdf and callgraph.pdf[8].

9.5.4. Modify your program so that each time you press the button the state of the LED toggles. For example, when you push the button the LED should turn on. If you push the button again the LED should turn off. You will need to use a variable to keep track of the previous button state and compare it to the current state to detect an edge event. You will also need to keep track of the LED state in order to toggle it. Note this may not work perfectly due to switch bouncing, where one push may appear like multiple pushes.

10. Driving the LED matrix

The LED display consists of an array of LEDs arranged in a matrix of 7 rows and 5 columns. To turn on an LED in the matrix you need to drive its row and column low at the same time.

10.1. Open the file drivers/avr/system.h and have a look at the definitions. This file defines all the PIOs that interface to the peripherals on the UCFK4[9].

10.2. Modify the file lab1-ex5/lab1-ex5.c so that the top left LED on the LED display turns on using the PIO module (the USB connector is the top end of the board).

10.3. Modify the file lab1-ex5/lab1-ex5.c so that all the LEDs in the left column are on.

[7] Documentation for the PIO module can be found in the header file header file drivers/avr/pio.h.
[8] These show the module dependencies and program call graph (more on this in your lectures).
[9] See the UCFK documentation on Learn https://learn.canterbury.ac.nz/mod/page/view.php?id=4217084 for further information.