F28HS COURSEWORK 2 REPORT

Dubai Campus

Group 97 · Zaid Alshami · João Labriola

CW2: Systems Programming

Demo video: cw2-video.mov

cw2-video.mov (sharepoint.com)

Problem Specification

This project draws inspiration from the Mastermind board game and is crafted using C and ARM Assembly programming languages. It is tailored to function on Raspberry Pi models 2 and 3 and incorporates external hardware like buttons and LED indicators. The game consists of a codebreaker and codekeeper. The codekeeper (RPi a.k.a computer), creates a secret sequence of N colors, each color ranging from C to available ones. At each round the user tries guessing the sequence by pressing buttons on the RPi, and the codekeeper, indicates at the end of each round (through the use of LEDs and LCD screen) how many colors the user guesses exactly correct (same color same peg) and how many were approximately correct (the color is present in the sequence, but the location is different and hasn't been guessed). An example case:

```
Secret:  R  G  G
================
Guess1:  B  R  G
Answ1:           1 1
Guess2:  R  B  G
Answ2:           2 0
Guess3:  R  G  G
Answ3:           3 0
Game finished in 3 moves.
```

Hardware Specification & Functions

Green LEDs

Three Resistors used to ground LEDs and button

LCD Display, used to display messages to the user. LCD used four data cables, two for control, two for a power supply, three for a ground connection, and one to connect to its potentiometer.

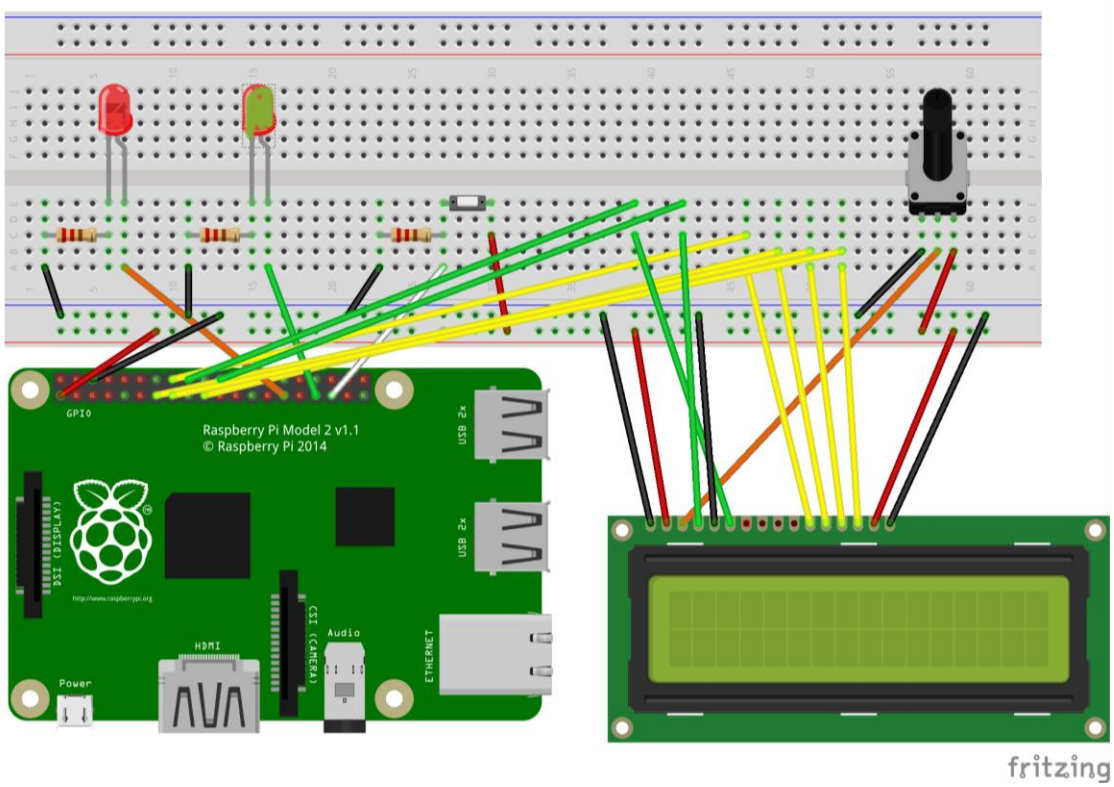
Button, used to read input from the user.

Potentiometer, used to control the brightness of the LCD display.

The RPi used had rows (represented on the diagram as red and blue lines), which were separated by two halves. Therefore, each half required one ground and power cable connected to the RPi. The left half where LEDs and button were connected used a 3v3 power connection, whilst the other half, containing the LCD connections, required a 5v connection to function correctly.

Below is the wiring used for each connection. Following, a diagram representing the connections, however it is worth noting, our data and control connections for the LCD were connected directly to the pins on the RPi (not through a breadboard).

Pin 19 (White) Button	LCD Data (Yellow)	LCD Control (Green)
Pin 13 (Green) green LED	GPIO 23 <-> LCD 11	GPIO 25 <-> LCD 4
Pin 5 (Orange) red LED	GPIO 10 <-> LCD 12	GPIO 24 <-> LCD 6
All power wires are red	GPIO 27 <-> LCD 13	
All ground wires are black	GPIO 22 <-> LCD 14	



Code structure

The code compiled and executed in the video and use cases (which will be presented later in the report), where the compilation of two files, master-mind.c and lcdBinary.c.

lcdBinary.c

This file contained functions used in the game logic and set-up.

digitalWrite and writeLED were functions that passed a value of one or zero to a register location (physical memory location in the RPi), equivalent to that of the pin number passed as an argument. These were used to send signals to devices. pinMode similarly to the previous function wrote a value of one or zero to a memory location to that of the pin number. However the location was one which stored values of zero or one corresponding to the type of hardware that was connected to that pin was (i.e input or output). All three of the above were written in a c file, but inline assembler language, which is efficient in communicating with hardware, due to its “shorter distance” to hardware (more on this later).

Another function in this file is waitForButton, which “heard”/polled for button presses by the user, and returned the amount of presses.

master-mind.c

This contained the main logic and auxiliary functions of the program. The main function used in the game logic:

initSequece() which randomly created the secret sequence to be guessed by the codebreaker;

showSeq() printed the secret sequence;

countMatches() accepted two sequences and returned a pointer to an array which represented the amount of exact and approximate matches of the second to first sequence;

showMatches() printed the matches of two sequences. Uses a call the previous function;

readSeq() read a value and saved it in the sequence array the pointer given in its argument.

readNum() which asks for a guess from the terminal;

initTimer() started an interval timer, once finished the function sets up a handler timer_handler(), which changes the value of a global volatile variable from 0 to. So that the waitForButton() function uses to know when to stop polling for button presses;

blinkN() accepts a pin connected to an LED and blinks it c times, by writing 0/1 to its register blinking the LED;

printMessageLcd() accepts an lcd object and using the puts() function it writes the message to that lcd. The puts() function writes a string of 16 characters to the LCD. Thus, if longer than 16 characters this function iterates through the longer string and puts the string on the LCD 16 characters at a time, simulating a scrolling message;

The main function is composed of all the hardware set up, and variables initializations required. The main game logic starts in a while loop, that keeps iterating until the found variable is set to 1. Each iteration represents one round of the game (also updating the attempts variable at each round).

Inside the round logic, a for loop iterates an arbitrary number of times, representing the number of “pegs”/elements in the secret sequence. At each iteration the user is prompted to press the button 1-C (number of colors) times, to “pick a color”, once done, the user is

prompted his/her guesses. The green LED blinks the amount of time the codebreaker pressed the button. Red LED blinks twice once the round of input is finished.

The LCD and LEDs then display the amount of exact and approximate guesses (green LED blinks n times, n being the number of exact guesses, then a red blink, then green n times to represent the approximate guesses). The red LED then blinks 3 times to display the end of the round.

At the end of the iteration (while loop) the program sets the found variable to one if there are 3 exact matches or breaks out of the loop if user reached maximum number of attempts (10).

Once the game finishes, if successful the green LED blinks three times, with red LED on, and prints a message to the LCD displaying success and the amount of attempts it took to find the secret sequence.

Performance and Resource Consumption

When dealing with software that controls, “speaks”, directly to software performance is highly important. The coursework focuses on writing performance effective and resource conscious coding design.

The functions which dealt directly to hardware in lcdBinary.c were written in assembly code as it has a better performance due to it being a lower-level programming language. Assembler deals directly with registers and memory access, thus at every step of writing code, the use of memory was kept to a minimum. The functions only used the top 5 registers of the CPU, thus making access to values quicker than if written in C. Thus, increasing the speed of writing and reading values from and to the register which controlled the pins in the Raspberry pi.

In the C parts of the program, consumption was tended to, through the usage of pointers instead of pre-set arrays. Pointers allowed the program to request the necessary memory at run-time, rather than allocated fixed amounts of memory, which can be wasteful. For example, the memory used to store sequences are pointers as the programs works for an arbitrary number, if the program were to use 3, 10 or 20 “pegs”, with an array it would have to allocate enough memory for all. With pointers memory used is only what is necessary for each case. Dynamically allocating memory also requires the freeing of such memory, which is tended to at the end of the program, to avoid memory leaks and effectively manage resources.

Other considerations include the use of volatile variables and functions, which prevents the compiler from applying optimizations which may affect the values and behavior of variables and functions. As well as atomic variables, which cannot be interrupted, ensuring that signal

handlers won't catch the program in an inconsistent state. These considerations become especially critical in such a program that is heavily focused on performance.

Functions which directly access the hardware include:

digitalWrite() · pinMode() · writeLED · readButton() · waitForButton()

The first four use mostly assembler, and last all in C, except for the call for readButton(), which is written in inline assembler.

Example Cases

The program accepts the execution of the program with the below usage.

./cw2 [-v] [-d] [-u <seq1> <seq2>] [-s <secret sequence>]

" If run without any options, the program should show the behaviour specified above. If run with the -d option it should run in debug mode, and show the secret sequence, the guessed sequence and the answer, as shown in the example above. If run with the -s option, the <secret sequence> should be used as the sequence to guess (this is useful in combination with the -d option to debug the program). If run with the -u option, it should run a unit test on 2 input sequences, <seq1> and <seq2>, and print the number of exact and approximate matches "

Verbose and unit test

```
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -v -u 123 321
1st argument = 123
2nd argument = 321
Settings for running the program
Verbose is ON
Debug is OFF
Unittest is ON
Testing matches function with sequences 123 and 321
1 exact
2 approximate
```

-v –u · runs the match function, testing the second sequence against the first, and displaying the number of exact and approximate matches between the two.

Verbose and secret sequence

```
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -v -s 111
Settings for running the program
Verbose is ON
Debug is OFF
Unittest is OFF
Secret sequence set to 111
Running program with secret sequence:
Secret:  R R R
Raspberry Pi LCD driver, for a 16x2 display (4-bit wiring)
Printing welcome message on the LCD display ...
Press ENTER to continue:
Sequence found
```

-s · runs the program with the secret sequence equals to the argument passed in the program call. This case was ran with 111, which equals Red Red Red.

Unit test cases

```
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -u 123 321
1 exact
2 approximate
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -u 123 122
2 exact
0 approximate
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -u 123 111
1 exact
0 approximate
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -u 123 132
1 exact
2 approximate
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -u 123 123
3 exact
0 approximate
```

-u · a few test cases

Debug test case

```
csyear2@raspberrypi:~/Desktop/cw2 $ sudo ./cw2 -d
Raspberry Pi LCD driver, for a 16x2 display (4-bit wiring)
Printing welcome message on the LCD display ...
Secret:  B G G
Press ENTER to continue:
Guess1:   G G R
Answer1:          11
Guess2:   B G R
Answer2:          20
Guess3:   B G G
Answer3:          30
Sequence found
```

-d · debug runs the program and prints the guesses and answers at each round. As well as prints the secret sequence ant the start

Summary

The mastermind game was effectively executed in the Raspberry Pi and built using C and Assembler. The project involved many learning areas, from performance and resource conscious programming, understanding the basics of writing software that was focused on hardware. This posed challenges as neither of us had previously wrote code in such manner. But it showed it to be great in deepening not only our efficiency in both languages but understanding the theoretical and practical concepts of dealing with hardware a lot better.

The assembly parts of the project were not as developed as the C parts. But it was an outstanding experience deepening our roots in the C language and understanding the compilation linking and execution of programs of this style.

Peer Assessment

Zaid Alshami 5/5

Joao Labriola 5/5