Assessed Coursework 2 — Systems Programming

1 Overview

The aim of this coursework is to develop a simple, systems-level application in C and ARM Assembler, running on a Raspberry Pi with attached devices.

The learning objective of this coursework is for students to obtain detailed understanding of the interaction beween embedded hardware and external devices, in order to control this interaction in low-level code. The programming skills will cover detailed resource management and time sensitive operations. Design choices regarding languages, tools, and libraries chosen for the implementation need to be justified in the accompanying report. This coursework will develop personal abilities in articulating system-level operations and identifying performance implications of given systems through the written report that should accompany the complete implementation.

The report needs to critically reflect on the software development process for (embedded) systems programming contrast it to main stream programming, and comment on performance-relevant design choices as well as impact on resource consumption.

This CW should be **done in pairs**, declared no later than 1 week before submission: sign-up form.

2 Lab Environment

Hardware environment: As hardware platform, a Raspberry Pi 2 or 3 with the starter kit introduced in the course should be used. You can get these by appointment from "Hurt, Adrian" `<A.C.Hurt@hw.ac.uk>` in EM 1.32 (Edinburgh). The Raspberry Pi can be directly connected to the machines in the Linux Lab (EM 2.50) using a KVM switch. See the Lecture slides ("Course Overview") and the technical HOWTOs (Canvas) on this topic. You can then use the keyboard and mouse for input and the monitor for output and work as if on a standard Linux machine.

Software environment: The SD card that is part of the starter kit uses Raspbian (32-bit) as Linux-based operating system. It is recommended that you stick to this version, although there might be a more recent version of Raspbian available.

For developing the code you should use the standard GNU toolchain (`gcc`, `as`, `ld`) that comes with the version of Raspbian that is installed on the SD card. It is recommended that you use the GNU debugger (`gdb`) for debugging your code. See the Lecture slides in the Programming Languages track of the course.

3 Embedded Systems Programming: Master Mind Application

In this assignment, you are required to implement a simple instance of the MasterMind board-game, using C and ARM assembler as implementation language. The application needs to run on an Raspberry Pi 2 or 3, with the following attached devices: two LEDs, a button, and an LCD (with attached potentiometer for controlling contrast). The devices should be connected to the RPi2 via a breadboard, using the RPi2 kit that was handed out early in the course.

Application: MasterMind is a two player game between a codekeeper and a codebreaker. Before the game, a sequence length of N and a number of C colours for an arbitrary number of pegs are fixed. Then the codekeeper selects N coloured pegs and places them into a sequence of N slots. This (hidden) sequence comprises the code that should be broken. In turns, the codebreaker tries to guess the hidden sequence, by composing a sequence of N coloured pegs, choosing from the C colours. In each turn the codekeeper answers by stating how many pegs in the guess sequence are both of the right colour and at the right position, and how many pegs are of the right colour but not in the right position. The codebreaker uses this information in order to refine their guess in the next round. The game is over when the codebreaker successfully guesses the code, or if a fixed number of turns has been played. For details see this MasterMind Wikipedia page.

Below is a **sample sequence** of moves (R red, G green, B blue) for the board-game:

```
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.
```

This is the **sample sequence** of input (IN) and output (OUT) operations in the running **application** when picking the `Secret`, followed by the first `Guess1` and the first `Answ1`, corresponding to the example above. This uses an encoding of 1 for R (red), 2 for G (green) and 3 for B (blue). `Press 1` means, press the button once, `Green Blink 1` means blink the green LED once etc; `//` starts a comment for this example and is not part of the input/output):

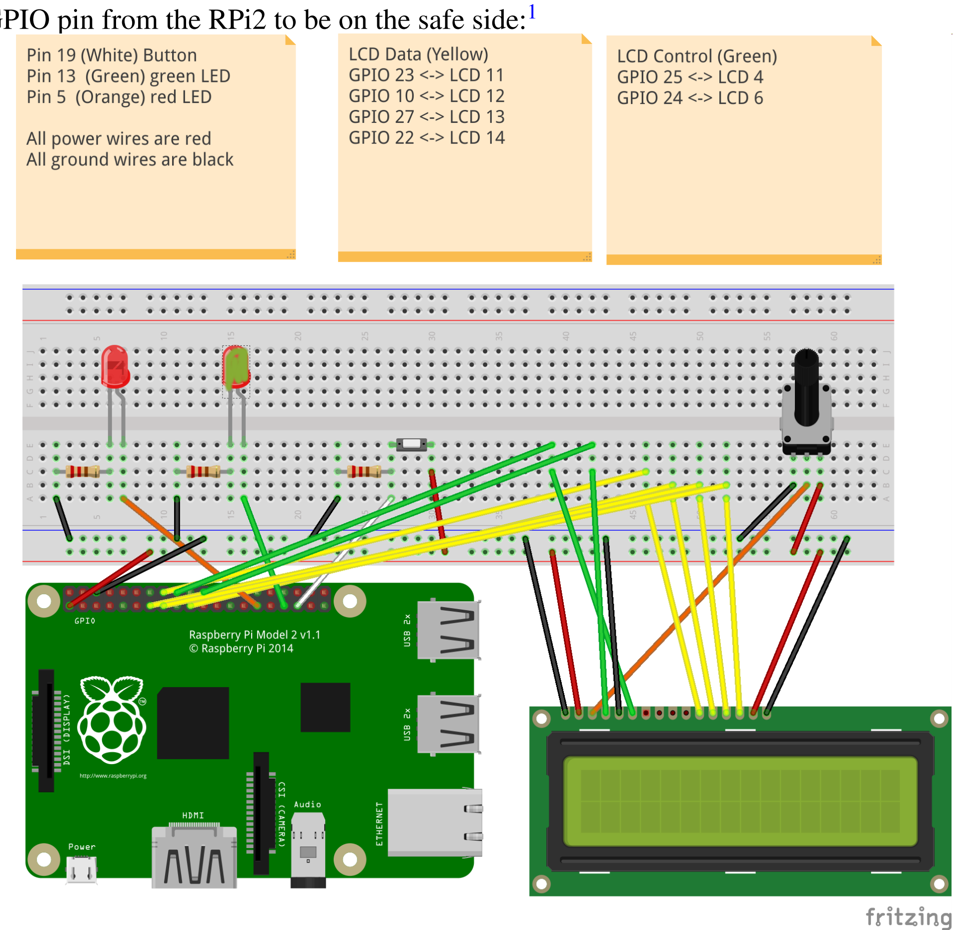
```
OUT: Secret:  1  2  2   // show secret

IN:  <Press 3> <Pause>  // first input
OUT: <Red Blink>        // first input accepted
OUT: <Green Blink 3>    // echo the input value
IN:  <Press 1> <Pause>  // second input
OUT: <Red Blink>        // second input accepted
OUT: <Green Blink 1>    // echo the input value
IN:  <Press 2> <pause>  // third input
OUT: <Red Blink>        // third input accepted
OUT: <Green Blink 2>    // echo the input value
OUT: <Red Blink 2>      // input completed

OUT: <Green Blink 1>    // first no. in answer (exact matches)
OUT: <Red Blink 1>      // separator
OUT: <Green Blink 1>    // second no. in answer (contained matches)
OUT: <Red Blink 3>      // next round starts
...                     // next round
```

Coding: The application should be developed on the Raspberry Pi 2 or 3, using the device configuration below, with inlined ARM assembler code to directly control the attached devices through GPIO connections. No external libraries for programming the devices should be used in the final application. It is recommended to encode the C colours as numbers from 1 to C, and to display the sequence of pegs as a sequence of numbers. To test the application, a setting of *3 colours (C=3) and a sequence length of 3 should be used (N=3)*. In a "debug" mode the program should print the secret sequence at the beginning, so that the answers given can be checked, and each entered sequence (the guess) with the corresponding answer (as 2 numbers).

Wiring/Devices: Two LEDs should be used as output devices: one (green) LED for data, and another (red) LED for control information (e.g. to separate parts of the input and to start a new round). The **green data LED (right)** should be connected to the RPi2 using **GPIO pin 13**. The **red control LED (left)** should be connected to the RPi2 using **GPIO pin 5**. A **button** should be used as input device. It should be connected to **GPIO pin 19**. An LCD should be used as an additional output device. It should be connected as follows:

LCD	GPIO	LCD	GPIO
1	(GND)	9	(unused)
2	(3v Power)	10	(unused)
3	(Potentiometer)	11 (DATA4)	23
4 (RS)	25	12 (DATA5)	10
5 (RW)	(GND)	13 (DATA6)	27
6 (EN)	24	14 (DATA7)	22
7	(unused)	15 (LED+)	(3v Power)
8	(unused)	16 (LED-)	(GND)

This means that the 4 data connections to the LCD display are connected to these 4 GPIO pins on the RPi2: 23, 10, 27, 22. All devices should be connected to the RPi2 using a breadboard. Below is a **Fritzing diagram** that visualises the entire wiring. Note that the *middle* pin of the potentiometer needs to be wired to LCD 3, and the other 2 legs to ground and power as shown in this version. Also, use the 3.3V and not the 5V power GPIO pin from the RPi2 to be on the safe side:[1]



[1]you can connect the RPi2 GPIO pins directly to the LCD pins, but it's usally easier to use a breadboard as shown

LCD: Note that the wiring for the LCD display matches the one discussed in Fig 9-2 (p 202), of the Adventures in Raspberry Pi book, and Chapter 9 has been handed out at the beginning of the course. The project file of the above Fritzing diagram application can be downloaded here. You can view it using the Fritzing diagram application. Follow this FAQ on how to download and install the Fritzing diagram application. The main advantage of viewing the diagram inside the application is that you can zoom in to examine the wiring, and you get info on the pin numbers when you hover the mouse over them.

Sample code for the control of LED, button and LCD can be found on the sample sources section of the Course Information page for F28HS. **Note that the low-level operations of setting the mode of a pin, writing to an LED or LCD, and reading from a button need to be encoded in ARM Assembler**. The **matching function also needs to be implemented in ARM Assembler** (see below). It's recommended, though, that you first develop a pure C implementation, and then replace the C functions for the above operations with ARM Assembler functions. The rest can be done in C, and you can draw on the sample code to build the functionality of the application.

It is strongly recommended that you do the lab sheets for LED, LCD and button control, before starting the CW. However, you have to adapt the wiring used in the lab-sheets to the one prescribed in the CW spec.

Game functionality in C: The **game logic** of the application (written in C) must provide the following functionality, with the application acting as codekeeper (i.e. generating a random, hidden sequence and answering) and the user as code breaker (i.e. entering guess sequences) (see the sample sequence above):

1. The application proceeds in rounds of guesses and answers, as in the sample for the board game.
2. In each round the player enters a sequence of numbers.
3. A number is entered using the button as an input device. Each number is entered as the number of button presses, i.e. press twice for an input of two etc.
4. A fixed time-out should be used to separate the input of successive numbers. Use timers (either on C or Assembler level), as introduced in the lectures.
5. The red control LED should blink once to acknowledge input of a number.
6. Then the green data LED should repeat the input number by blinking as many times as the button has been pressed.
7. Repeat this sequence of button-input and LED-echo for each element of the input sequence.
8. Once all values have been entered and echoed, the red control LED should blink two times to indicate the end of the input.
9. As an answer to the guess sequence, the application has to calculate the numbers of exact matches (same colour and location) and approximate matches (same colour but different location).
10. To communicate the answer, the green data LED should first blink the number of exact matches. Then, as a separator, the red control LED should blink once. Finally, the green data LED should blink the number of approximate matches.
11. Finally, the red control LED should blink three times to indicate the start of a new round.
12. If the hidden sequence has been guessed successfully, the green LED should blink three times while the red LED is turned on, otherwise the application should enter the next turn.

13. When an LCD is connected, the output of exact and approximate matches should additionally be displayed as two separate numbers on an 16x2 LCD display (see below).
14. On successful finish, a message "SUCCESS" should be printed on the LCD, followed by the number of attempts required.

Note: The Raspberry Pi should act as **codekeeper**, so needs to generate a (secret) random sequence and answer with the number of exact and approximate matches in the sequence above.

Command-line usage: The application shall provide a command-line interface to test its functionality in an automated way, like this:

```
./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, e.g.

```
./cw2 -u 123 321
```

should print (on the terminal, just for debugging)

```
1 exact
2 approximate
```

Matching function in ARM Assembler: The **matching function** for sequences, as described above should be implemented in **ARM Assembler code** and used in the C program for the game logic. An ARM Assembler sub-routine, matching ARM sub-routine calling conventions (AAPCS), should be implemented, either in a separate Assembler file or as inline Assembler, and be used from the C code to determine the number of exact and approximate matches as explained above. During development it's recommended that you first develop a pure C implementation of the matching function, and to use this version in order to test the game logic of the program. Then, move on and replace the C function for the matching function with an ARM Assembler function.

GitLab repository: To start your project fork the following GitLab repository, and then clone your forked copy onto a local machine. Ideally, do this directly on the Raspberry Pi. If you don't have a network connection, clone onto a local machine and then transfer the files to the Raspberry Pi, using for example a USB stick. Remember, you can develop the C parts, including unit testing for the matching function, on any Linux machine, and the repo contains a CI script for automated unit testing, using the `-u` option. The GitLab repo contains template code with a *suggested* structure for the master mind C program in `master-mind.c` and for the ARM Assembler matching function in `mm-matches.s`. Check the comments in these two files as well as the top-level `README.md` file in the GitLab repository.

Testing: Use the unit testing provided in the templates for C and Assembler code, to test the matching function, and report the results (number of successful tests) in your report. Unit tests are triggered automatically in the CI pipeline when you upload a new version to GitLab. Beyond this, test the game logic of the application by running the game (in debug mode, which shows the secret sequence at the beginning) with a given secret sequence (using the `-s` option) until you find the secret sequence. Add the sequence of interactions as screenshot or cut-and-paste text to the report. Also, run a unit-testing setup (using the `-u` option) with the sequences `121` and `313`) and show the result: `./cw2 -u 121 313`

Video: As a demo of running your application, you should produce a video, that executes the program from the command line, and goes through the stages of the game. Show how you press the buttons, use audio to discuss the input that you are providing and the expected behaviour, and show the responses from the application via LEDs and LCD display. Show at least 3 rounds, and finish with providing the correct input. The video should be uploaded together with the other data files. If there are limitations of file size, upload the video to OneDrive, and provide a link to the video in the report (and make that link prominent).

4 Submission

You must submit the complete project files, containing the source code, a stand-alone executable, the report (in `.pdf` format), and a video of a demo of the application, as one `.zip` file no later than **3:30 PM on Tuesday 29$^{\text{th}}$ March 2022**. The main executable driving the application should be called `cw2`, as discussed in "Command-line Usage" above. Submission must be through Canvas, submitting all of the above files in one `.zip` file. **This coursework is worth 50% of the module's mark.**

You are marked for the functionality of the application, but also for code quality and the discussion in the short report. The marking scheme for this project is attached. **This project should be done in pairs.** Following the submission, there might be selected demos for submissions that don't sufficiently explain behaviour in the report.

5 Report Format

The report should have about 2 – 5 pages and needs to cover the following:

- A short problem specification
- The hardware specification and wiring that is used as hardware platform
- A short discussion of the code structure, specifying the functionality of the main functions.
- A discussion of performance-relevant design decisions, and implications on resource consumption.
- A list of functions directly accessing the hardware (for LEDs, Button, and LCD display) and which parts of the function use assembler and which use C
- The name and an interface discussion (what are the inputs, what is the output of the sub-routine) of the matching function implemented in ARM Assembler.
- A sample execution of the program in debug mode
- A summary, covering what was achieved (and what not), outstanding features, and what you have learnt from this coursework

Marking Scheme

Criteria	Marks
Meeting system requirements and functionality (as specified in Section 3)	30
Report Quality Contents matching the structure in Section 5; discussion of program logic and of the core functions, controlling the GPIO interface; summary of learning outcomes achieved.	7
The Application Code quality (both C and ARM Assembler), clear function interfaces, sufficient comments.	8
Assessed Lab The implementation of the "Traffic Lights" lab sheet needs to be demo'ed in a lab slot.	5
Total marks	50

Professional Conduct and Plagiarism

This is a **pair project** and you will have to identify the contributions of each group member in the report. Each group member needs to contribute a substantial implementation task to the project. Where external resources have been used, these need to be clearly identified and referenced.

A check on source code plagiarism will be performed on all submissions. Confirmed plagiarism will result in disciplinary procedures depending on the scale of misconduct.

Plagiarism:
This project is assessed as an **pair project**. You will work with your partner on the implementation and should split up tasks in a reasonable way to arrive at a complete implementation of the specification. You will have to identify the contributions of each group member in the report. You can discuss general technical issues related to this work with other students, however, you must not share concrete pieces of code or text. Readings, web sources and any other material that you use from sources other than lecture material must be appropriately acknowledged and referenced. Plagiarism in any part of your report will result in referral to the disciplinary committee, which may lead to you losing all marks for this coursework and may have further implications on your degree. For details see this link

Late Submission Policy

The standard **penalty of -30%** of the maximum available mark applies to late submissions. No submissions will be accepted after 5 working days beyond the submission deadline.