Assessed Coursework 3 — Systems Programming

1 Overview

The aim of this coursework is to develop a simple, systems-level application in C and ARM assembler. The ARM code should run on the CPUlator. The C code can run on any Linux platform.

The learning objective of this coursework is for students to obtain 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 and contrast it to main stream programming.

This CW should be **done in pairs**, declared no later than 1 week before submission by signing up to this on-line allocation sheet.

The gitlab repo for the coursework, with code templates, is here .

2 Lab Environment

No physical or remote lab access is required for the CW. The programming can be done on a local machine for the C part, and on the online CPUlator for the Assembler part (see below). If needed, the departmental machine `jove` can be used via ssh or x2go, and has all necessary software installed to do the C programming, and it supports cross-compilation for ARM so that most of the Assembler programming can be done there as well (except for programming the HEX display).

The **C coding** for this coursework can be done on a local machine, or in the MACS VM, that has a C compiler, and the gcc compilation toolchain (including assembler and linker) installed. No external devices are programmed and no hardware specific aspects need to be considered for the C coding.

The **ARM Assembler coding** is best done on the web-based, online CPUlator. The hardware specific aspects of programming the HEX device, will only run on the CPUlator, or directly on hardware with the DE1-SOC by Altera. It is recommended, that you first test the behaviour of the matching function (and this part can be done with platform indepent ARM code), and then move on to displaying this information on the HEX display.

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 core logic of the application should be implemented in C and can run on an Linux platform The ARM Assembler code should control the devices available through the CPUlator and should run on this web-based interface.

Application: MasterMind is a two player game between a codemaker 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 codemaker selects N 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 codemaker 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 his 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 of the rules see this MasterMind Wikipedia page.

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

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

This is the **sample sequence** of input and output operations, running the **application**, corresponding to the example above. This uses an encoding of 1 for R (red), 2 for G (green) and 3 for B (blue). The numbers after the text `Input seq (len 3):` are the input provided by the user. The two numbers on the following line are the results for exact and approximate matches. The first line, showing the secret sequence, should only be displayed with the debug flag turned on (`-d`).

```
> ./cw3 -d
Contents of the sequence (of length 3):  2 1 1
Input seq (len 3): 1 2 3
0 2
Input seq (len 3): 3 2 1
1 1
Input seq (len 3): 2 1 1
3 0
SUCCESS after 3 iterations; secret sequence is  2 1 1
```

Coding: The application should be developed in two parts. The *core logic* of the game, that is entering a guess, the program answering with the number of *exact* (right color and right position) and *approximate matches* (right color but wrong position) , and testing for termination of the game, needs to be implemented in **C**, and can be developed stand-alone for any Linux platform. This C version needs to allow for full game-play and implement the requirements below. **ARM Assembler** code should be developed for just the *matching* operation described above. That is, given the secret and the guess, calculate the number of exact and approximate matches, and display these two numbers on two of the HEX displays provided by the CPUlator. The code doesn't need to be linked to the C implementation, and it should run stand-alone with sample inputs provided below, running on the CPUlator.

As optional task, integrate both C and Assembler code in one application, that can run stand-alone on the CPUlator, and allows for complete game-play (using buttons as input devices) as discussed above. 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).

Part 1: Game functionality in C: The **game logic** of the application (written in C) must provide the following functionality, with the application acting as code keeper (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. If the *debug flag* is on (`-d`), the first line should show the secret sequence.
2. The application proceeds in rounds of guesses and answers, as in the sample for the board game.
3. In each round the player enters a single-space separated sequence of numbers in a terminal window.
4. Numbers should be entered from the command line as a single-space separated list of integer values.
5. **(optional)** 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.
6. **(optional)** The digits of the input sequence should be printed, separated by 1 space each, e.g. `1 2 3`
7. 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).
8. To communicate the answer, print two numbers separated by a single space, repesenting the number of exact and of approximate matches, e.g. `2 1`
9. If the hidden sequence has been guessed successfully, the string `SUCCESS` should be shown on the Terminal and the program should terminate. Otherwise the application should enter the next turn.

Note: The application should act as **code-keeper**, so needs to generate a (secret) random sequence and answer with the number of exact and approximate matches in the sequence above.
Note: Steps marked as **(optional)** are only needed in an integrated version, which is optional to this spec.

Command-line usage: The application shall provide a command-line interface to test its functionality in an automated way, like this (options are shown in `[]` brackets, values to options, where needed, are shown in <> brackets):

```
./cw3 [-v] [-d] [-u] [-s] <secret sequence>
```

If run without any options, the program should show the behaviour specified above (without showing the secret sequence at the beginning). If run with the `-d` (debug) 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 `-u` (unit-testing) option, it should only run one unit test on 2 input sequences (without spaces!), and print the number of exact and approximate matches, e.g.

```
./cw3 -u 123 321
```

should print (on the terminal, just for debugging)

```
1 exact
2 approximate
```

Part 2: Matching function in ARM Assembler: Additionally and separately from the main logic of the above implementation, a **matching function** for sequences, as described above should be implemented in **ARM Assembler code** and run on the CPUlator:

- The data section of the code should define two input sequences (of bytes), taking the roles of secret and of guess sequences, as described above.
- An ARM Assembler subroutine `match` should be defined that takes pointers to the secret and guess sequence as inputs (in registers `R0` and `R1`), and returns the number of exact and approx matches in registers `R0` and `R1`.
- The subroutine should conform to the ARM procedure call conventions (Sec 6.1, p19).
- The main code of the Assembler implementation should call the `match` sub-routine with pre-defined secret and guess strings
- The result should be displayed on the 2 right-most HEX displays on the CPUlators, showing the number of exact matches first, and the number of approximate matches second (see picture below).

Testing: Test Part 1 (game logic in C) by running the game (in debug mode, which shows the secret sequence at the beginning) with a random secret sequence 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 (`u`) with the sequences `121` and `313`) and show the result. Test Part 2 (matching function and display in ARM Assembler) by using the input from the data shown here (replace USERID with your HWU user-id): test data.

4 Submission

You must submit the complete project files, containing the source code (separate files for the C and for the ARM Assembler code), a stand-alone executable of the C program for the core logic of the game, and the report (in `.pdf` format) as one `.zip` file no later than **3:30 PM on Thursday 8th April 2021**. You should also submit the C code and ARM Assembler code in separate files by pushing to a forked version of the CW2 github repository. The main function driving the application should be called `cw3`, as discussed in "Command-line Usage" above. Submission must be through *Vision*, submitting all of the above files in one `.zip` file. Additionally, you should push the final version of your code to the *gitlab repository*. **This coursework is worth 40% 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 will be mandatory demos, where each pair will have to present the implementation, explain its functionality and implementation, and you must be prepared for answering knowledge questions about the implementation and about programming external devices in general.

5 Report Format

The report should have at 1 – 3 pages and needs to cover the following:

- A short discussion of the code structure, specifying the functionality of the main functions (for each of the C and Assembler part)

- A discussion of the behaviour of the main functions in the C and Assembler programs, realising the specified behaviour.
- A sample execution of the program in debug mode (see Testing discussion above)
- 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)	24
Report Quality Contents matching the structure in Section 5; discussion of program logic (C) and of the core Assembler functions; summary of learning outcomes achieved.	4
Code Quality Code quality (both C and ARM Assembler), clear function interfaces, sufficient comments.	12
Total marks	40