Coursework 2: Systems Programming

Nicolas Sparagano & Mark Schmieg

A coursework report that includes a specification, code analysis, and summary of the task.

Heriot Watt University

Hardware Software Interface

F28HS

28/03/2018

Specification

Problem

"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 a Raspberry Pi2, with the following attached devices: two LEDs, a button, and an LCD (with attached potentiometer). The devices should be connected to the RPi2 via a breadboard, using the RPi2 kit that was handed out early in the course."

Hardware

"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."

Sample Execution

```
   sudo ./MasterMind d
»  Raspberry Pi button controlled LED
   (button in 19, led out 13, led out 5)
»  ###Debug mode###
»  Instructions:
»  -Red = 1 Presses
»  -Green = 2 Presses
»  -Blue = 3 Presses
»  =============
»  Secret: 1 3 3
»  =============
»  Player please guess the secret code
»  Press the button...
»  --You pressed 2 times--
»  Press the button...
»  --You pressed 1 times--
»  Press the button...
»  --You pressed 3 times--
»  ------------
»  Guess: 2 1 3
»  ------------
»  Exact: 1
»  Approx: 1
»  Try again
»  =============
»  Secret: 1 3 3
»  =============
»  Press the button...
»  --You pressed 1 times--
»  Press the button...
»  --You pressed 3 times--
»  Press the button...
»  --You pressed 3 times--
»  ------------
»  Guess: 1 3 3
»  ------------
»  Exact: 3
»  Approx: 0
»  SUCCESS! You have won in 2 rounds
»  end main.
```

Code structure

As there were two of us working on the code, and some required pieces of code were going to be clustered, the program is structured using a wide variety of methods and pointers to make the code 'cleaner' and easier to understand. For example, to turn on an LED the program calls `LEDOn(LEDR);` where `LEDOn` is a function and `LEDR` is a variable denoting the pin number of the red LED.

The program (as written in the `main` method) first checks that the number of arguments is correct (1 or 2). If there are 2 arguments then the program checks that the second is the string "d" meaning the user wants to run the program in debugging mode. This is flagged in the program by setting a variable, `debugMode`, to 1 (true) instead of 0 (false). Throughout the program, the console will display debugging information, such as the secret code, if this value is 1.

The game mastermind uses the colours red, green and blue, which in our system are represented by 1, 2 and 3 button presses respectively. For example, instead of a code in the form of [G G B], our system represents this as [2 2 3]. This allows the user to input a code using only one button.

At the start of the program, before the game starts, the program needs to set the pins for the button, red LED and green LED. This is done with calls to a specific `setMode` function for each of the three components. Once this is done, a secret code (three digits from 1 to 3 inclusive) is generated by a specific code-making function: `CodeCreator`. Then the user is prompted for their first guess (a series of button presses) which is displayed back to them in the console whilst being recorded to then be used by a code-checking function. This function, `checker`, takes the secret code and the player's guess and determines the number of exact digits and the number of approximate digits which are output as a struct of two integers representing exact and approx. respectively. The number of rounds/guesses is incremented, having been initialised at 0.

This begins a loop which only stops when the number of exact digits equals 3 (all 3 digits are exactly correct). Otherwise the loop prompts the user for another guess, checks the number of exact and approx. digits in the guess, and then increments the round/guesses counter. Once the loop is finished (the secret code has been correctly guessed), the user is told how many rounds/guesses it took them and the program ends by clearing the GPIOs of the three components.

Functions

Initially we used C when interacting with the GPIO pins to finalise the logic of our mastermind game, but then changed the C-based methods into in-line assembler methods. Because of the way we structured our program, these functions are almost entirely written in in-line assembler, and called upon when input/output is needed. This allows us to call clear and obviously-named functions whenever we want to access the hardware. Most of these functions use C variables within the in-line assembler to allow more flexible functions; only one function is needed to access multiple components as needed.

Constants: LEDR = 5, LEDG = 13, BUTTON = 19

```
void setMode (int pin)
```

- This sets the modes of the pins allowing us to use the GPIO-connected components.

```
int readButton (int pin)
```

- This reads from the button and returns 1 (true) if the pin that is called with this function detects an input.

```
void LEDOn (int pin)
```

- This turns the LED on at the pin called with this function.

```
void LEDOff (int pin)
```

- This function is identical to `LEDOn` except the offset used is different to write into the location that represents turning the LED off instead of the location for on.

```
void LEDBlink (int n, int pin)
```

- This function doesn't access the hardware, but instead calls `LEDOn` and `LEDOff` a variable amount of times, effectively making the referenced LED flash.

```
int * getInput()
```

- This function doesn't access the hardware, but instead calls `buttonPress`, which itself calls `readButton`, and records the returned value (the number of times the button is pressed). Then `LEDBlink` is called with the red LED and a value of 1 (flashing the red LED once), then `LEDBlink` with the green LED and the number of recorded button presses. This is done 3 times, once for each digit that needs to be input by the player. Finally `LEDBlink` is called for the red LED and a value of 2.

Summary

We successfully managed to complete the logic of the program, and had a fully working program that was written entirely in C without much difficulty. The logic behind checking the number of exact and approximate digits in the guessed code was eventually found to be flawed. But through testing and pen-and-paper working, this logic was corrected and tested to verify it worked properly.

Another issue came from the button hardware itself; pressing the button registered an inconsistent number of button presses. For example, pressing the button once could register as 8 button presses, regardless of how carefully the button was pressed. Our solution to this was to have a method that was called when an input from the user was needed, `buttonPress`. This would check every 200ms if the button was pressed using a method, `readButton`, increment a counter if so, and then pause for 200ms. This would loop 20 times, effectively giving the user 4 seconds to input using the button, while the delay upon a button press prevented the device detecting multiple button presses where there were none.

The main learning points from this coursework are a better familiarity with C, especially with GPIO pin. But mostly the main thing we learned was how to use in-line assembler with C, and how to use assembler on the Raspberry Pi to use the GPIO pins. These two assembler parts of the coursework were easily the hardest as they were new to us.

Nicolas Sparagano **(H00243900)**

Mark Schmieg **(H00238262)**