CW2: Systems Programming

Vishakha Heralal : H00421235
Hamid Faghipoor : H00393041
Heriot Watt University, Dubai
School Of Mathematical & Computer Sciences

Introduction

MasterMind is a two-player game that involves a codekeeper and a codebreaker. At the outset, the codekeeper sets the parameters by determining the sequence length (N) and the number of available colors (C) for the pegs. Subsequently, the codekeeper conceals a sequence of N colored pegs within N slots. The codebreaker's objective is to decipher this hidden sequence by formulating guesses consisting of N colored pegs, chosen from the C available colors. After each guess, the codekeeper provides feedback to the codebreaker, indicating the number of pegs that are both the correct color and in the correct position (exact matches), as well as the number of pegs that are the correct color but in the wrong position (approximate matches). Armed with this feedback, the codebreaker refines their strategy for subsequent rounds. The game concludes either when the codebreaker successfully deduces the hidden sequence or when a predetermined number of turns have been exhausted. MasterMind thus fosters strategic thinking and deduction skills as players navigate the challenge of unraveling the code within a finite number of attempts.

A simple instance of the Mastermind board game is implemented using C and ARM Assembler programming languages. This application then runs on Raspberry Pi 3 with the following attached devices: two LEDs, a button, and an LCD (with attached potentiometer for controlling contrast). The devices are connected to the RPi via a breadboard.

Hardware Setup:

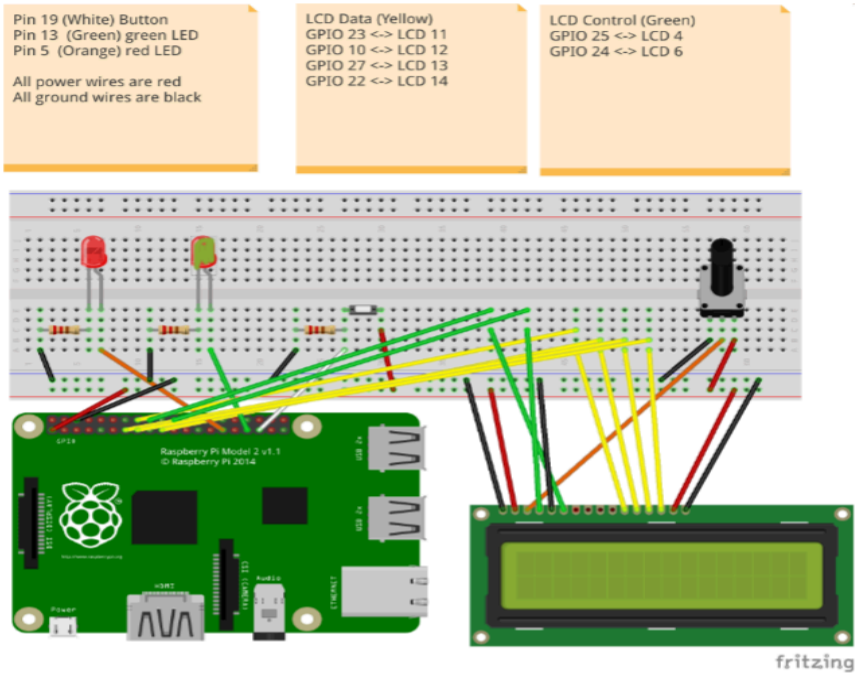
The Mastermind application is developed on Raspberry Pi 3 with the following attached devices: two LEDs, a button, and an LCD (with attached potentiometer for controlling contrast). The devices are connected to the RPi via a breadboard. The inline ARM assembler code is used to directly control the attached devices through GPIO connections.

The two LEDS are used as output devices. The green LED is used for data and the red LED is used for control information which is to separate the parts of the input and to start a new round.

The green data LED is connected to the RPi using GPIO pin 13 whereas the red control LED is connected using the GPIO pin 5. The button is used as an input device and it is connected to the RPi using the GPIO pin 19. The LCD is used as an additional output device and is connected in the following way:

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 RPi: 23, 10, 27, 22. In the table above we can see that the middle pin of the potentiometer is wired to LCD 3 and the other two legs are wired to ground and power. The above table also shows that the 3.3v Power GPIO pin from the RPi is used instead of the 5V Power. The diagram below shows a detailed visualization of the entire wiring setup for this application.



Implementation Details:

The code is organized into two main sections: `Master-Mind.c`, `Lcdbinary.c` each serving a distinct purpose in implementing the MasterMind game on a Raspberry Pi and providing low-level hardware control functions. In the development of the MasterMind game on a Raspberry Pi, the `Master-Mind.c` file orchestrates the entire game flow, from initialization to output display. This includes low-level hardware control functions for direct interaction with Raspberry Pi components.

Key to its performance optimization is the integration of inline assembly within functions like `countMatches`. This function, crucial for evaluating user guesses against the secret sequence, utilizes inline assembly to access hardware registers directly. This enables efficient GPIO pin manipulation, enhancing responsiveness and minimizing overhead.

The inline assembly integrates low-level operations into the higher-level context, ensuring optimal performance while maintaining code coherence. This approach leverages the Raspberry Pi's hardware capabilities for an immersive gaming experience without sacrificing efficiency.

`Lcdbinary.c` offers low-level hardware control functions for LEDs, buttons, and potentially LCD devices.The functions directly accessing hardware components and the parts utilizing C and/or assembly language:

digitalWrite:

- Language: Assembly
- Purpose: Controls the state of GPIO pins connected to LEDs, allowing toggling of their illumination.

writeLED:

- Language: Assembly
- Purpose: Similar to digitalWrite, it controls the state of GPIO pins specifically for LED control.

readButton:

- Language: Assembly
- Purpose: Reads the state of GPIO pins connected to buttons, determining whether a button is pressed or released.

waitForButton:

- Language: Assembly
- Purpose: Waits for a button press event by continuously monitoring the state of GPIO pins.

pinMode:

- Language: C
- Purpose: Configures the GPIO pin mode (input or output) for a specified pin, enabling control over its behavior.

lcdInit:

- Language: C (with possible inline assembly)
- Purpose: Initializes the LCD display, setting up communication parameters and configuring its operation mode.

lcdPutCommand:

- Language: C (with possible inline assembly)
- Purpose: Sends commands to the LCD display controller, configuring various display parameters and functionalities.

lcdClear:

- Language: C (with possible inline assembly)
- Purpose: Clears the contents of the LCD display, preparing it for the display of new data or messages.

These functions play critical roles in interfacing with hardware components such as LEDs, buttons, and LCD displays in embedded systems. While some functions primarily utilize assembly language for low-level hardware manipulation, others are implemented in C, providing a higher-level abstraction for handling complex tasks and logical operations. The combination of both languages allows for efficient and comprehensive control over hardware peripherals, enabling the development of robust embedded systems.

Performance-relevant design decisions across the codebase include the use of a timer mechanism to prevent indefinite waits during user input, enhancing responsiveness and user experience. Additionally, the adoption of inline assembly for low-level hardware control functions ensures efficient hardware interaction with minimal overhead. The structure facilitates modularity and reusability, with functions directly accessing hardware manipulating GPIO pins and implemented using inline assembly.

Command Line Interface

```
* Compile:
 gcc -c -o lcdBinary.o lcdBinary.c
 gcc -c -o master-mind.o master-mind.c
 gcc -o master-mind master-mind.o lcdBinary.o
 * Run:
 sudo ./master-mind
 ./master-mind [-v] [-d] [-u <seq1> <seq2>] [-s <secret
sequence>]
```

The commands above compile two source files, lcdBinary.c and master-mind.c, into object files using the GNU Compiler Collection (gcc). The -c flag instructs gcc to compile the source files without linking them, while the -o flag specifies the output filename for the resulting object files. Hence, gcc -c -o lcdBinary.o lcdBinary.c compiles lcdBinary.c into an object file named lcdBinary.o, and gcc -c -o master-mind.o master-mind.c similarly compiles master-mind.c into an object file named master-mind.o.

Following the compilation of the source files, the next step involves linking these object files to create an executable program. The command gcc -o master-mind master-mind.o lcdBinary.o performs this operation. Here, the -o flag specifies the output filename, which in this case is master-mind. This command combines the functionalities from the master-mind.o and lcdBinary.o object files to create the master-mind executable program.

Finally, the last command executes the master-mind program. The sudo ./master-mind command runs the master-mind program with elevated privileges, typically requiring administrative access (sudo) to execute. This step executes the compiled program, allowing it to perform its intended operations. Optionally, the command can be followed by various command-line options, such as -v, -d, -u <seq1> <seq2>, and -s <secret sequence>,

which offer different functionalities and settings for the program, as explained in the provided usage information.

In summary, these commands compile the source files into object files, link them to create an executable program, and then execute the resulting program with appropriate privileges, possibly accompanied by additional command-line options to specify various functionalities or settings.

Code Output:

```
Raspberry Pi LCD driver, for a 16x2 display (4-bit wiring)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,40)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,12)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,12)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,12)
lcdClear: lcdPutCommand(27768344,1) and
lcdPutCommand(27768344,2)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,1)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,2)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,6)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,20)

Preparing to display welcome message on the LCD display ...

lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,128)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,192)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,192)
Press ENTER to continue:
lcdClear: lcdPutCommand(27768344,1) and
lcdPutCommand(27768344,2)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,1)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,2)
```

```
Lets start the game...
round 1
Enter number 1: Press the button

Button pressed 2 times
Timer expired 1 times. Time took: 6.000059
Enter number 2: Press the button

Button pressed 1 times
Timer expired 2 times. Time took: 6.000056
Enter number 3: Press the button

Button pressed 3 times
Timer expired 3 times. Time took: 6.000055

your input sequence was 2 1 3!
Exact: 1, Approximate: 2
lcdClear: lcdPutCommand(27768344,1) and
lcdPutCommand(27768344,2)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,1)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,2)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,128)
lcdPutCommand: digitalWrite(25,0) and sendDataCmd(27768344,192)
^Z
[2]+  Stopped                 sudo ./master-mind -s 123
```

Conclusion

Throughout this coursework, significant progress has been made in developing a Master Mind game implementation for the Raspberry Pi platform. Key achievements include the successful implementation of the game logic, integration of low-level hardware control functions for LEDs and buttons, and efficient ARM Assembler integration to enhance performance. Robust testing infrastructure has been established, encompassing both unit tests and manual testing procedures, ensuring the correctness and reliability of the game logic. However, outstanding features such as LCD device integration and further optimization opportunities remain. From this coursework, valuable lessons have been learned, including the importance of modularity, optimization considerations, and effective testing methodologies. Overall, this project has provided valuable hands-on experience in embedded systems development, ARM assembly programming, and software testing, laying a solid foundation for future projects in similar domains.