**Nicholas Masci**

**CSC 155 - Systems Programming**

**Final Project - Robi Drivers**

**Professor Thomas G. Re**

**Table of Contents**

Manual 1 - 7

I2C.h 1

MotorHat.h 1 - 2

PWM.h 3

Robot.h 3 - 5

Sensor.h 5 - 6

Servo.h 6 - 7

Problems Encountered During Project 8 - 10

Relevant Topics Covered in Class 11 - 12

Lab 2 11

Lab 3 11

Lab 4 11

I2C and Slave Select 11

Raspbian Operating System 12

Extra Credit 13 - 14

Works Cited 15

**Manual**

**I2C.h**

**int i2cInit(int address);**

* This function is needed to initialize the communication with the i2c bus. This function should be the first function that is called in main
* It is also called by several other functions in the project
* Only the address is needed for this function.
* Returns 0 on success, 1 on failure.

**void write8(unsigned char reg, unsigned char value);**

* This function is needed to write data to the hat, and is used by many other functions in the program
* This will likely never be called directly.

**unsigned int readU8(unsigned char reg);**

* This function is needed to read data from the hat, and is likewise used by many other functions in this project.
* It will likely never be called directly.
* Returns the value that it reads.

**MotorHat.h**

**void initMotors();**

* This function initializes each of the motors with their proper in1, in2, and pwm pins.
* There are 2 motors used in this project, motors[0] is the left motor, and motors[1] is the right motor.
* This function only needs to be called once, and should usually be the second function called, after i2cInit.

**void initHat();**

* This function initializes the motor hat itself, setting it as the current slave to the Pi and setting the proper PWM frequency for the motors.
* Needs to be called every time you want to control the motors instead of the servo, and will probably be the third function you call in main.
  + Must set the motor hat as the current slave when you want to control the motors, and the servo hat as the slave when you want to control the servo.

**void run(unsigned char command, unsigned char motorID);**

* This function is used by Robot.c in order to give Robi commands. It handles the actual motors' movements using setPin
* Will likely not need to be called directly.

**void setSpeed(int speed, unsigned char motorID);**

* This function sets the speed of the specified motor. It is also used in the functions of Robot.c in order to control Robi
* Will likely not need to be called directly.

**void setPin(unsigned char pin, unsigned char value);**

* This function uses PWM.c's setPWM in order to make one of Robi's motors move. The aforementioned run function uses this function in order to smoothly handle commands to be sent to Robi's motors.
* It will likely not need to be called directly.

**PWM.h**

**void setAllPWM(int on, int off);**

* This function will send a command to all channels at once on ROBI. It is useful for controlling all motors at once, but I preferred to set them all individually.

**void PWMInit(unsigned int address);**

* This function initializes an I2C connection on the given address and prepares it to receive PWM communications.
* Used by initHat in MotorHat.c, and initServo in Servo.c.
* It will likely not need to be called directly.

**void setPWMFreq(int freq);**

* This function sets the PWM frequency to the frequency given to it.
* This is important, as the motors and the servo both use different frequencies to communicate with the Pi.
* It is not called directly, but instead used by the various init functions.

**void setPWM(int channel, int on, int off);**

* This function sends a command to a specific PWM channel, which is helpful for controlling Robi's motors individually.
* Will not be called directly, but is instead used in other functions to streamline control of Robi.

**Robot.h**

**void \_left\_speed(int speed);**

* This function acts as a wrapper for the setSpeed function, to be used on the left motor. It ensures that the speed is between 0 and 255, inclusive, and sets the left motor's speed using setSpeed().
* This function will not be called directly, and is instead used in the forward, backward, left, and right functions.

**void \_right\_speed(int speed);**

* This function is similar to the \_left\_speed function, but as the name would imply, it is to be used on the right motor.
* Will not be called directly.

*The following five functions should be used with delay(x), where x is the amount of time, in milliseconds, you want the command to take place.*

*For example:*

forward(128);

delay(1000);

*Will make Robi move forward at speed 128 for 1 second.*

*Speed should be a value between 0 and 255, inclusive.*

**void stop();**

* This function simply sends a RELEASE command to both motors.
* This function should be called whenever you want Robi to stop.

**void forward(int speed);**

* This function will make Robi move forward at the given speed.
* My Robi's left wheel is backwards, so motors[0] receives the opposite instruction, BACKWARD. If this code does not work properly on your Robi, you may need to swap the instruction being given to motors[0] for this function and all the others within this file.
* For example, in this function, you would change...

run(BACKWARD, 0) -> run(FORWARD, 0)

* This function should be called when you want Robi to go forward.

**void backward(int speed);**

* This function will make Robi move backward at the given speed.
* This function has the same problem with the wheel (as do all the others in this file), so you may again need to change the instruction sent to the left wheel (motors[0]).
* This function should be called when you want Robi to go backward.

**void right(int speed);**

* This function will make Robi rotate clockwise (turn right)
* It should be called when you want Robi to face to the right.
* The speed and delay used with this function will affect how far to the right Robi will turn.

**void left(int speed);**

* This function will make Robi rotate counterclockwise (turn left)
* It should be called when you want Robi to face to the left.
* The speed and delay used with this function will affect how far to the left Robi will turn.

**Sensor.h**

**int initSensor();**

* This function simply calls bcm2835\_init(). It was originally called in get\_distance, which eventually caused segmentation faults when running the program for too long.
* Should just be called once after all other initializations have taken place.

**double get\_time();**

* This function returns the current time. It is used by get\_distance in order to measure the distance from the sensor.
* It will not be called directly.

**void gpio\_reset();**

* This function sets pulldowns for the two middle pins of the sensor, and selects them to be used as inputs to the Pi.
* Used at the end of get\_distance, but will not be called directly.

**double get\_distance();**

* This function will use the sensor to get Robi's distance from the obstacle in front of him.
* Measures the amount of time it takes for sound from the sensor to return, and uses that value to determine the distance.
* Should be called on a loop periodically in main in order to make sure Robi does not crash
* Sometimes, very high or very low (negative) values will be returned by this function. The high values are ignored altogether, and low values are changed to instead be 500cm (And then are ignored as well)

**Servo.h**

**void initServo();**

* This function prepares the servo to be controlled by the Pi.
* Must be called every time you want to control the servo, after you have been controlling the motors.
  + Control has to be passed between the motor hat and the servo hat, they cannot both be initialized at the same time.

**void setServoPulse(int channel, int pulse);**

* This function is not directly called, but it is used in the following three functions.
* The given channel, that the servo is plugged into, will be sent a pulse, which will cause the servo to move to that position.
* 500 is the minimum, turning the servo all the way to the right
* 2500 is the maximum, turning the servo all the way to the left

**void look\_left();**

* This function will be called when you want Robi to look left.
* Uses setServoPulse in order to make the servo move to its maximum actuation.

**void look\_right();**

* This function will be called when you want Robi to look right.
* Uses setServoPulse in order to make the servo move to its minimum actuation.

**void look\_mid();**

* This function will be called when you want Robi to look straight ahead.
* Uses setServoPulse in order to make the servo move to the middle, which is the average of the maximum and minimum actuation values.

**Problems Encountered During Project**

| **Problem Description** | **Action(s) Taken** |
| --- | --- |
| Sensor became very hot; smelled like smoke whenever it was used | Upon consulting Professor Re, he pointed out that the Power and Ground wires were backwards - the sensor was fried. New sensor was obtained and wired properly. |
| Servo would not move at all when given commands | There were two different reasons for this:   1. Servo was wired backwards. Thankfully, it did *not* get fried, and worked properly upon turning the wires around 2. Upon inspecting my PWM.c program, I noticed I never called i2cInit(address), and therefore no communication was ever established with the servo hat to begin with   After fixing both these issues, the servo began taking commands |
| Servo does not move properly | The limits defined in the python file were too small; Upon playing with these values, an upper limit of 2500 and a lower limit of 500 allowed the servo to properly move left and right at its maximum range. |
| Robi’s “head” does not move properly; seems to not attach properly to the servo | Thankfully, the “head” of my lab partner, Nick Geoghegan’s, Robi was able to properly attach to the servo. He allowed me to use his head in order to demonstrate my project. |
| Could not get Robi to turn on using the battery pack | Upon consulting Professor Re, he pointed out that the battery pack should be plugged into the Pi. |
| Sensor periodically gives very high values (2000’s-ish) and very low values (negative 10-thousands-ish) | High values were simply left alone, as the odds of them appearing when Robi is about to crash is very low. Additionally, distance readings are taken quickly enough that these high values shouldn’t affect anything  However, the low values would cause Robi to “see” an obstacle that isn’t really there, so I added code to Sensor.c to handle any negative distance values and ignore them (Sets the distance to be returned as 500 cm) |
| Commanding both wheels to go FORWARD would cause Robi to spin to the left, commanding both motors to go BACKWARD causes Robi to spin right, etc. | The left motor appears to be installed backwards, so the left motor (motors[0]) was given opposite instructions to those given to the right motor (motors[1])  For example, FORWARD will command the left motor to go *backward*, but the right motor will go forward. This will make both wheels spin the same direction, allowing Robi to proceed forward. |
| Running the code for an extended period of time causes a segmentation fault from bcm2835\_init() | Upon reviewing my code, I noticed that sensor.c calls bcm2835\_init() EVERY time it runs, causing it to eventually not be able to properly allocate memory. Moving the init into its own separate function and calling it ONCE at the beginning of main allowed the program to proceed as normal without any segmentation faults. |
| It is difficult to get Robi to stop moving when the program is killed. | Implemented a countdown in main.c. After turning *n* times, Robi will stop. *n* = 8 in the code I handed in, but can be easily changed via the demoCounter variable. |

**Relevant Topics Covered in Class**

**Lab 2 - Reading /etc/passwd; Writing to STDOUT**

In this lab, we wrote data directly to standard out in order to print it to the terminal. Additionally, we had to read data from files stored on the Raspberry Pi. This information was useful in reading from and writing to the different registers on Robi’s various hats in order to have him move around.

**Lab 3 - LED Blink Lab**

For this lab, pins on the Pi were selected to be used as outputs, lighting up LEDs on a breadboard. Knowledge of the bcm2835\_gpio\_fsel function, used to select a pin on the Pi for I/O, was also highly important for the final project. Each motor, the sensor, and the servo were all connected to different pins, and it was important to ensure proper procedures were taken to send output to each of these devices (Motors moving, sensor sending pulses, servo turning).

**Lab 4 - Button Binary Counter La**b

This lab, in addition to outputs, also dealt with taking inputs. The bcm2835\_gpio\_set\_pud function was necessary for the button in this lab to be recognized as an input, selecting it as the current slave for the Pi. For the final project, bcm2835\_gpio\_set\_pud was used for the sensor in order to allow it to measure the distance and return it to the Pi. Without knowledge of this function, there would be no way for the sensor to fully communicate with the Pi.

**I2C and Slave Select**

As mentioned previously, knowledge of I2C and Slave Select was very important for this project. All components of Robi communicate with the Pi via I2C, and therefore have to be selected each time you want to communicate with a different component.

**Raspbian Operating System**

The Raspberry Pi runs the Raspbian operating system, which is closely related to Debian. The operating system is the main reason we are using a Raspberry Pi for this project, as opposed to an Arduino, which does not have an operating system. On an Arduino, all code is run directly on the hardware, meaning you are limited to one specific language. However, a Raspberry Pi running Debian is, quite literally, a tiny computer. It can do anything a computer can, including store multiple files, install new programs, and run and compile code in a variety of languages. This allows us to have a lot more freedom with programming Robi.

**Extra Credit**

The extra credit for this project was to have Robi look to the left and right, get the distances on each side of himself, and then proceed in the direction that has an obstacle farther away. To accomplish this, Robi’s servo hat must be used. The servo hat allows Robi’s “head”, which holds the distance sensor, to turn and check in different directions. Though it is only one additional file, there were several bumps in the road along the way.

The servo example code did not seem to work on my Robi, so I ported it to C and simply had to hope that it worked. It did, eventually, but not at first. The first of several issues I encountered was that the servo was wired to the hat upside-down. Flipping the wires around the other way fixed that problem, but the servo still would not move. Upon reviewing PWM.c, which Servo.c depends on, I noticed I never called i2cInit in my PWMInit function. That means there was never any connection established with the servo hat to begin with. Once these problems were solved, the servo seemed to be trying to move, but would not do anything of note. I tried changing the upper and lower limits, and eventually found that an upper limit of 2500, and a lower limit of 500 were the proper limits for the servo. I could finally make Robi look around!

Once the servo code was up-and-running, it was implementation time. In main.c, I had initialized the motor hat, then the servo. I noticed the wheels wouldn’t move! Then, I tried initializing the servo first, instead. The wheels moved, but now the head wouldn’t turn. Finally, I figured it out. Upon initializing a given hat, that hat is selected to be the slave. You cannot have them both selected at once, so the initialization functions had to be called several more times in main.c in order to control all parts of Robi. Once this had been established, the actual changes made to main.c were very small. Instead of simply backing up and turning to the right, I now had Robi look to the left, check that distance, look to the right, check the right distance, look straight ahead, compare the distances, and then turn in the direction that had the higher distance.

After a little more testing, I noticed Robi was beginning to stop and turn out of nowhere. I looked at my program’s output, and saw that Robi was occasionally returning a distance in the negative 15,000’s-or-so centimeters. To remedy this, to some degree, at least, I added an if statement in Sensor.c that will change any negative distances to be a value of 500cm. This is a fairly large value, and has no meaning, but it allows the program to continue relatively normally, preventing Robi from suddenly turning out of nowhere. Once this was put into place, the program was fully functional. Nick Geoghegan and I set up an obstacle course at the top floor of Starr Hall, using notebooks, backpacks, laptops, and any other materials we had on hand, and watched Robi navigate through it with ease. The feeling of watching something I created successfully navigate around is a feeling I will not soon forget.

**Works Cited**

[1] HC-SR04 Ultrasonic Range Sensor on the Raspberry Pi <https://thepihut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi>

[2] Adafruit 16-Channel PWM/Servo HAT & Bonnet for Raspberry Pi Setup Tutorial <https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/attach-and-test-the-hat>

[3] bcm2835 I2C Documentation <https://www.airspayce.com/mikem/bcm2835/group__i2c.html>

[4] bcm2835 GPIO Documentation <https://www.airspayce.com/mikem/bcm2835/group__gpio.html>

[5] Adafruit Motor HAT Python Library <https://github.com/adafruit/Adafruit-Motor-HAT-Python-Library>

[6] Professor Thomas G. Re - Introduction to Serial Communications

[7] Professor Thomas G. Re - Introduction to BEE and the Raspberry Pi