Project 3 Report

CSC 460

Group #2

Shreyas Devalapurkar - V00827994

Rickus Senekal - V00826364

Table of Contents

List of Figures and Tables

Abstract

The foundation of this report is based on the details surrounding the implementation of a semi-autonomous roomba robot for use within a game of defend your castle. The functionality built for the robot included being able to move based on joystick input over bluetooth, detect a light source, shoot a laser by adjusting a servo pan-tilt kit, detect virtual walls as well as actual barriers. The objective of the game was to either defend your castle whilst on defense or to try and shoot the opponent castles/robots down whilst on offense.

In order to build this robot, functionality from our previous TTA scheduler was used. Periodic and sporadic tasks were utilized as necessary to perform certain tasks at specific times and for specific durations. The roomba library was used to control the movement of the roomba while also allowing other functionality such as being able to play a musical tune using the robot. A uart library was used to allow for communication between our joystick remote station and the base station situated on the roomba itself. Finally, a servo library was used to help control the movement of the servo motors situated on the roomba.

1. Introduction

This section will lay the foundation in describing the remainder of the report's contents. It will outline the purpose of the report while discussing the scope and overview of the project. The introduction of the document will serve as a precursor to the remainder of the report.

1.1. Purpose

The contents of this report will discuss our group's implementation of phase one and two of project three. The report's contents can be applied to the entire project and are not limited to a single feature of the project. The purpose of this documentation is to explain in detail each of the previously mentioned phases. It will describe the system overview, our implementation process, and any issues we ran into while working on this project.

1.2. Project Overview and Scope

For this project, we were tasked with building a Roomba robot that could play the **Defend Your Castle** game. What this entailed was defending yourself and your castle, while trying to shoot down the opponent castle/robots, among a group of semi-autonomous mobile robots.

Phase one included building a remotely-controlled Roomba robot that could perform the following tasks:

- Switch modes every 30 seconds from manual control to still/rotating
- Detecting being shot at and then killed
- Getting killed if crossing an infrared (IR) sensor (river)
- Only shooting the laser for a maximum of 10 seconds

Phase two included adding some semi-autonomous behaviours to what was already built in phase one such as:

- Detecting a wall and backing away from it
- Detecting an IR source (virtual wall) and avoiding it

The scope of this project was well outlined by Dr. Mantis Cheng (http://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p3.html) [1] and this outline was used throughout the development process to ensure completeness and correctness of our project.

2. Discussion

This section will explain our project implementation in detail, covering both phases of the project. It will present a system architecture overview and with the use of code snippets and images, will thoroughly explain how our system was designed and built. We will also explore and discuss some of the challenges we faced throughout our journey.

2.1. System Architecture and Overview

Our basic roomba system was composed of two subsystems; the base station on the roomba robot itself and the remote station on our controller used to control the roomba.

2.1.1. Base Station Design and Architecture

In order to enable the arduino board to send commands to the roomba robot and control it, we had to wire up a lot of different components from the arduino to the roomba. The list of components used is detailed below:

- One laser
- One arduino ATMega 2560 microcontroller

- One pan-tilt servo motor kit
- One digital light sensor
- One large red solo cup
- One bluetooth module
- One breadboard

A detailed block diagram of this hardware wiring is shown in Figure 1.0.

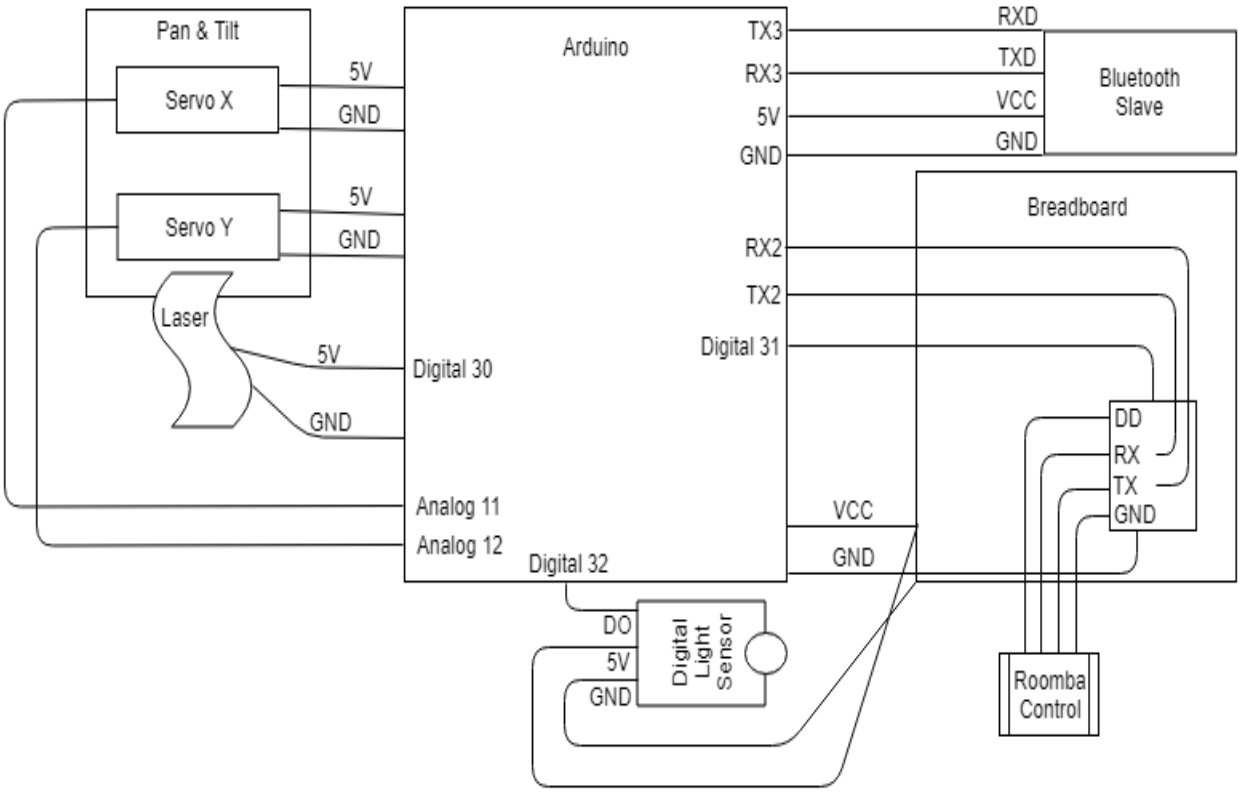

Figure 1.0 - Block diagram of base station components and wiring

All Roombas manufactured since 2005 have a serial port interface that accepts a male 7-pin mini-DIN connector, allowing the roomba to communicate with a device running UART at TTL levels (0-5 V) [2] which is what we took advantage of and plugged into. As shown below in Figure 2.0, this mini-DIN connector was used to transfer power from the roomba to the arduino and also for communication purposes.

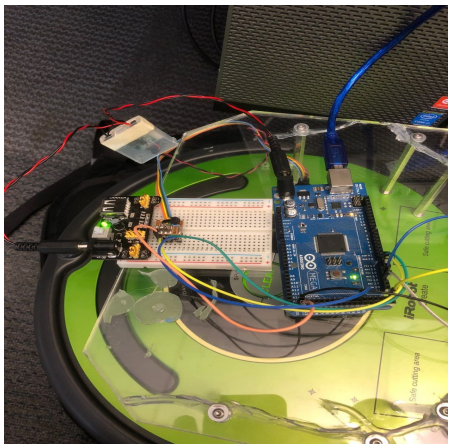

Figure 2.0 - Arduino board mounted on roomba with mini-DIN connector and power supply

Initially it was not necessary, but as we added more components to our base station such as the servo motors, we needed to add a power supply as also shown in Figure 2.0, as this would allow us to supply sufficient power to the various components of the system without draining the power of the roomba.

2.1.2. Remote Station Design and Architecture

Since our final robot had to be able to be controlled from a remote station via bluetooth, we needed to develop and build this station. We decided to create a fancy design for our remote station controller. Since we knew that we would be using joysticks to control the roomba, we wanted an ergonomic design for which we dismantled an XBOX 360 controller and plugged in our custom joysticks and bluetooth panel. The final controller design is shown in Figure 3.0.

Figure 3.0 - Custom XBOX 360 controller to control our roomba

The right joystick was calibrated and plugged in to act as the roomba movement control mechanism, whereas the left one was used to control the servo pan-tilt kit situated on the roomba to aim the laser. The click functionality of the right joystick was used to activate our laser which would be used to shoot enemy robots or their castles during gameplay. The components used to build our remote station were as follows:

- One XBOX 360 controller
- One Arduino ATMega 2560 microcontroller
- Two analog joysticks
- One bluetooth module
- One power supply

A detailed block diagram of this hardware wiring is shown in Figure 4.0.

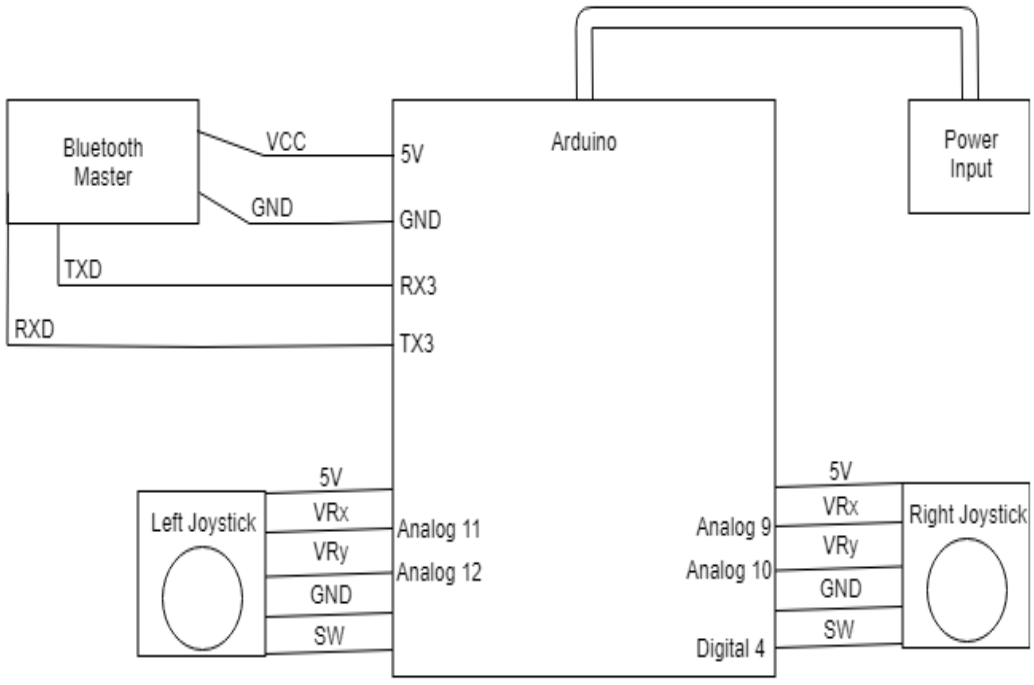

Figure 4.0 - Block diagram for remote station components

2.2. Implementation Overview

There were many pieces of functionality that needed to be implemented in this project. In phase one, we had to demonstrate a working remotely-controlled roomba that met all the rules of the game; switching modes every 30 seconds, detecting being shot at and then killed, killed if crossed the river, and only shooting the laser for at most 10 seconds before running out of ammunition. In phase two, we had to implement semi-autonomous behaviors; the Roomba must be able to roam and avoid obstacles on its own, e.g., detecting the virtual wall and then backing away, and also bumping into a physical "barrier" and then backing away.

2.2.1. Initial Base Station Implementation

The first thing we worked on for this project was being able to get the base station to control the roomba and send commands to it using wired uart. We figured that if we could get this working effectively, then it would not be too difficult to port it over to bluetooth when we added in our remote station. One thing we really had to keep in mind was which receiver and transmitter channel our various components were communicating over. When these channels were incorrectly wired or programmed, they caused major issues and took a long time to troubleshoot.

2.2.1.1. Wired Joysticks Controlling Roomba

In order to read in our analog joystick values without the help of an Arduino function such as *analogRead*, we had to implement our own function to read and interpret such values. This implementation is shown in Figure 5.0 below.

```
// ADC reader (analogRead)
uint16_t readADC(uint8_t channel) {
        ADMUX = (ADMUX & 0xF0 ) | (0x07 & channel);
        ADCSRB = (ADCSRB & 0xF7) | (channel & (1 << MUX5));
        ADCSRA |= (1 << ADSC);
        while ((ADCSRA & (1 << ADSC)));
        return ADCH;
}
```

Figure 5.0 - Reading in analog values

In order to initialize the analog to digital reader convertor, we set the ADC prescaler to a 128 - 125 KHz sample rate at 16MHz. We then set the ADC reference to AVCC and left adjusted the ADC value in order to allow for easy 8-bit reading of the value. We then were able to enable ADC and start a conversion to read in our value. Once we had our value, we compared it to threshold values to identify our joystick dead zones as well as figure out which direction we wished to move our roomba in. Once we had these values, we were able to simply call the pre-provided *Roomba_Drive* function with the respective speed and radius values to control the movement of our robot. It was good to see that this method of reading in values from the joystick worked because we could now transition to the remote station and work on getting the bluetooth communication working.

2.2.2. Remote Station Implementation

The remote station had one basic function; sending joystick values to the base station so the base station could interpret them and control the roomba accordingly. There were three values we cared about: right joystick movement, left joystick movement, and right joystick button press. The code snippets below illustrate how we received the values necessary from the joysticks and sent them over to our base station.

```
leftXAxis = readADC(leftJoyXPin); // Actually the right one
leftYAxis = readADC(leftJoyYPin);
rightXAxis = readADC(rightJoyXPin); // Actually the left one
rightYAxis = readADC(rightJoyYPin);
```

Figure 6.0 - Reading in joystick input values

```
if (PING & (1 << PG5)){
        PORTH &= ~0x20;
        leftJoyButton = 0;
        } else{
        PORTH |= 0x20;
        leftJoyButton = 1;
}
```

Figure 7.0 - Detecting a digital pin signal for button press

```
servoPan = rightXAxis;
servoTilt = rightYAxis;
```

Figure 8.0 - Setting rightX and rightY axis values (left joystick) as servo movement values

```
radiusRoomba = (leftXAxis - leftXAxisCenter);

speedRoomba = -(leftYAxis - leftYAxisCenter);
```

Figure 9.0 - Identifying radius and speed of roomba

Once these values were read in from the joystick and specific speed and radius values were read in, all that was remaining was to send these values over to the base station. In order to do so, we had to send the high and low bytes of these values over separately so that they could be received as individual bytes by the base station. Figure 10.0 shows how we sent our data packets over from the remote station to the base station. The uart library's *uart_putchar* function was used to enable this functionality.

```
uart_putchar(HIGH_BYTE(speedRoomba), CH_2);
uart_putchar(LOW_BYTE(speedRoomba), CH_2);

uart_putchar(HIGH_BYTE(radiusRoomba), CH_2);
uart_putchar(LOW_BYTE(radiusRoomba), CH_2);

uart_putchar(HIGH_BYTE(servoPan), CH_2);
uart_putchar(LOW_BYTE(servoPan), CH_2);

uart_putchar(HIGH_BYTE(servoTilt), CH_2);
uart_putchar(LOW_BYTE(servoTilt), CH_2);

uart_putchar(HIGH_BYTE(leftJoyButton), CH_2);
uart_putchar(LOW_BYTE(leftJoyButton), CH_2);
```

Figure 10.0 - Sending over all joystick values from remote to base station

2.2.3. Final Base Station Implementation

In our base station, we had to read in the values coming from the remote station and use them to drive our roomba as well as move the pan-tilt kit while firing the laser. Moving the roomba was quite easy as it involved using the pre-built *uart_getbyte* function from the library and reading in the values which were then used in the pre-built *Roomba_Drive* function. This section will focus on the more difficult sections of the implementation process such as controlling the servo motor, switching states every 30 seconds, firing the laser for only 10 seconds max, and semi-autonomous behaviours.

2.2.3.1. Controlling Roomba State

During each run of the main periodic task, checks were done to determine which state the roomba is currently in. It is the result of these checks that determined which code would be executed during the current cycle. A global variable kept track of the roomba's current state. The available states included **User-Controlled**, **Stationary, Reverse,** and **Dead**.

The initial state was user-controlled, in which the user has complete control of all movements and actions. All other states take priority over the user-controlled state, and thus each other state will interrupt all control if switched to.

According to the rules designed for project 3, each roomba must switch into a stationary state in which no forward or backwards movement can occur. When in this state, the roomba is only allowed to spin and control the pan-tilt system. This allows for more strategic gameplay within the competition period. Implementation of the automatic state switching was done by adding a sporadic task with a delay of 30 seconds into the main periodic task. Upon activation, the sporadic task sets the roomba's current state to stationary. During initial execution of this code, notice was taken of the possibility of the sporadic task buffer overflowing due to a sporadic task being added every 150ms, and only executing every 30s. To avoid this glitch, a global boolean variable was added to keep track of whether or not the sporadic task had already been added. When added the boolean value was set to 1 and upon execution of the sporadic task, it was reset to 0.

The **reverse** state disabled all manual input and moved the roomba backwards for a period of 1 second. This state was included to make sure that upon activation of the semi-autonomous behaviour, as described in Section 2.2.3.2, the roomba would reverse until it was safe to reactivate the user-controlled state.

2.2.3.2. Enabling Semi-Autonomous Behaviour

To encourage semi-autonomous behaviour, the ability for the roomba to sense its environment was very important. This was done by handling events, such as triggering of the front bumper, and the virtual-wall/IR sensor attached to the roomba. We implemented a function called *roomba_sense* which was responsible for detecting a bumper hit or a virtual wall, and this is shown in Figure 11.0 below.

```
void roomba_sense() {
        uart_putchar(149, CH_1);
        uart_putchar(2, CH_1);
        uart_putchar(7, CH_1);
        uart_putchar(13, CH_1);

        if (uart_bytes_received(CH_1) == 2){
                bumper_hit = uart_get_byte(0, CH_1);
                river_hit = uart_get_byte(1, CH_1);
                if (bumper_hit & 0x03 != 0 || river_hit != 0){
                        roomba_state = 2;
                        Scheduler_AddSporadicTask(50, change_state_to_user, true, 5);
                }
        }
}
```

Figure 11.0 - Sensing bumper hits and the virtual wall

Looking at the roomba documentation, we identified that we could query the 149th sequence and get the first two bits of the 7th packet (bumper hits) and the value of the 13th packet (virtual wall) and then look at those values to see if they returned a 1 or a 0. If the value was a 1, we changed roomba states to **reverse** and added a sporadic task to run after 1 second. When the sporadic task actually executed, it just reset the state back to **user-controlled**. Since our reading in values periodic task ran much more often than any of these sporadic tasks, the change in state was identified sooner and the robot moved backwards for a second until the sporadic task executed.

2.2.3.3. Firing Laser - Ammunition

One of the specifications for our project was that our roomba must be able to fire a laser to shoot down opponent roombas and/or their castles. Now, what made this even more interesting and

challenging to implement was that we were only able to fire a laser for a total of 10 seconds before running out of laser ammunition. Another specification was that you needed to shoot a target for a total of 2 seconds in order for that target to be considered dead. Keeping these two specifications in mind, we decided to implement our laser firing system as follows:

- Pressing the joystick button would turn on the laser
- Once on, the laser would remain on for a total of 2 seconds regardless of subsequent presses
- Once the 2 seconds passed, the laser would automatically turn off
- We would have a total of 5 shots or bullets

In order to implement this, we used a sporadic task with a delay of exactly 2 seconds, meaning that it would execute in exactly 2 seconds. This allowed us to not implement any new timer code; we were able to take advantage of our TTA scheduler and its ability to schedule tasks exactly after a certain amount of time in the future. This sporadic task would reset a variable called *just_shot* which indicated whether or not a shot was fired. Along with this, every time a shot was fired, a global variable counter would be incremented to ensure we do not get more than 5 bullets total. This is shown in Figures 12.0 and 13.0 below.

```
if (LeftJoyButton == 1){
        if (just_shot == 0){
                shot_count = shot_count + 1;
                if (shot_count < 6 ){
                        PORTB |= 0x20;
                        Scheduler_AddSporadicTask(100, shot_fired, true, 5);
                        just_shot = 1;
                } else{
                        PORTB &= ~0x20;
                }
        }

}
```

Figure 12.0 - Logic to determine if we can shoot the laser, timing 2 seconds from shot

```
void shot_fired(bool state){
        just_shot = 0;
        PORTB &= ~0x20;
}
```

Figure 13.0 - Resetting the *just_shot* variable

We believe this implementation approach actually gave us a competitive advantage rather than having a laser firing mechanism where a click enabled the laser and a subsequent click disabled it. This way, we never wasted any ammunition and knew exactly how many bullets we had. Also, we never faced a situation in battle where we had some firing power left but it was < 2 seconds in duration, and as a result, useless (because all our fires were exactly 2 seconds in duration).

2.2.3.4. Sensing Our Demise

In order to properly detect the death of our own roomba, a function was implemented and executed every 15ms to check the current state of the digital light sensor. An analog sensor was initially used, but removed due to its noisy behaviour. A digital pin was set as input via the DDRC and PORTC AVR GCC registers. The value of the digital pin was checked using the PINC register, along with a bit shift and bitwise 'and' operation.

As can be seen in Figure 14.0, a Solo cup was internally lined with reflective foil tape. The light sensor was installed facing the back of the cup and a protective "sun-visor" was added onto the cup in order to isolate any incoming light. Once light was shone into the cup, it would reflect back onto the sensor activating the digital pin it was plugged into.

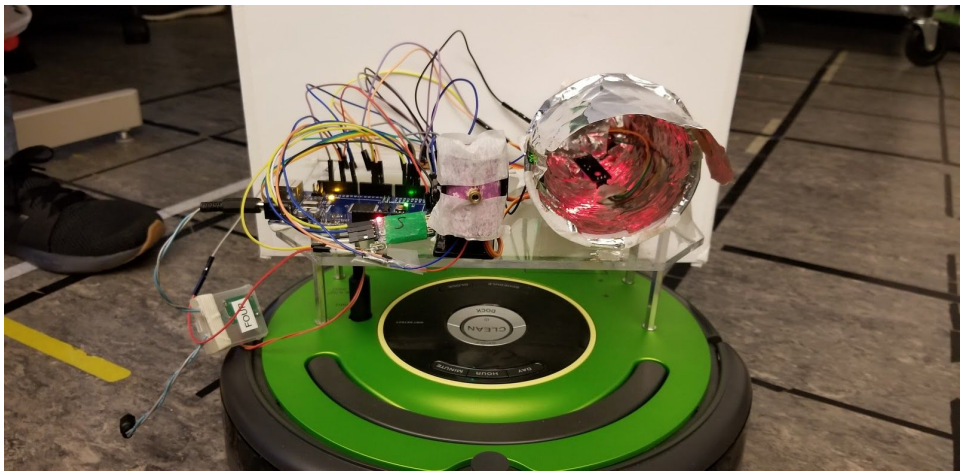

Figure 14.0 - Frontal view of the light sensing mechanism

To implement the feature of dying after an enemy laser hit our roomba's sensor for at least 2 seconds, a sporadic task with a delay of 2 seconds was added every time light was sensed, but removed every time light was not sensed. Thus, if light was continuously active on our sensor, the sporadic task would execute and the roomba's state would switch to **dead**, pronouncing it motionless and unavailable.

2.2.3.5. Receiving Joystick Input

Once the joystick positions and button status had been sent via *uart_putchar*, it had to be retrieved at the same rate on the base station. As can be seen in Figure 15.0, each byte was sent and received in order of the high and then low byte of the integer values initially sent. The low byte was then appended to the high byte using the bitwise "shift" and "or" operations. Once all 10 desired bytes were received, the buffer was cleared to avoid buffer overflow errors.

```
if (uart_bytes_received(CH_2) >= 10) {
        speedHIGH = uart_get_byte(0, CH_2);
        speedLOW = uart_get_byte(1, CH_2);
        speedRoomba = (speedHIGH << 8) | speedLOW;
        radiusHIGH = uart_get_byte(2, CH_2);
        radiusLOW = uart_get_byte(3, CH_2);
        radiusRoomba = (radiusHIGH << 8) | radiusLOW;
        xJoyHIGH = uart_get_byte(4, CH_2);
        xJoyLOW = uart_get_byte(5, CH_2);
        xJoy = (xJoyHIGH << 8) | xJoyLOW;
        yJoyHIGH = uart_get_byte(6, CH_2);
        yJoyLOW = uart_get_byte(7, CH_2);
        yJoy = (yJoyHIGH << 8) | yJoyLOW;
        LeftJoyButtonHIGH = uart_get_byte(8, CH_2);
        LeftJoyButtonLOW = uart_get_byte(9, CH_2);
        LeftJoyButton = (LeftJoyButtonHIGH << 8) | LeftJoyButtonLOW;
        uart_reset_receive(CH_2);
}
```

Figure 15.0 - Receiving and assignment of movement variables on base station

As per Figure 16.0, the speed and rotation of the roomba was done using a threshold value, along with various checks. The turning of the roomba, known as radius, and the speed, both had an offset of 20 to create a stable deadzone. Once the input of either speed or radius exceeded the offset in either positive or negative deviations, the roomba would start moving in the designated

direction. To normalize the speed input, its value was divided by 0.25. This allowed for control of the speed in regards to the roomba's physical maximum value of 500. The radius was also multiplied by 15.87 for this very reason. This increased its range, resulting in -2000 to 2000. The radius movements of the roomba were made in increments of 150 to enable smooth turning capabilities.

```
//Roomba Radius
int radiusOffset = 20;

if(radiusRoomba < radiusOffset && radiusRoomba > -radiusOffset){
        radiusRoomba = 0;
        }else{
        radiusRoomba = radiusRoomba*15.87;

        if(radiusRoomba >  2000)radiusRoomba =  1999;
        if(radiusRoomba < -2000)radiusRoomba = -1999;

        if(radiusRoomba <  2000 && radiusRoomba >=  radiusOffset)radiusRoomba = 2000-radiusRoomba+150;
        if(radiusRoomba > -2000 && radiusRoomba <= -radiusOffset)radiusRoomba = -2000-radiusRoomba-150;
}

//Roomba Speed
int speedOffset = 20;
float tempspeed=0.000;

if(speedRoomba < speedOffset && speedRoomba > -speedOffset){
        speedRoomba = 0;
        if (radiusRoomba > radiusOffset){
                speedRoomba = 350;
                radiusRoomba = 1;
        } else if (radiusRoomba < -radiusOffset){
                speedRoomba = 350;
                radiusRoomba = -1;
        }
}else{
        speedRoomba = speedRoomba / .25; //(normal max is 500)
        if(speedRoomba >  500)speedRoomba =  500;
        if(speedRoomba < -500)speedRoomba = -500;
}
Roomba_Drive(speedRoomba, radiusRoomba);
```

Figure 16.0 - Rotation and speed control implementation

2.2.3.6. Controlling the Pan-tilt Mechanism

Although precise control of the roomba's motors had been achieved, it was important to also focus on the mechanism to which our laser was mounted. To initialize the servo motors installed within the pan-tilt kit, two PWM pins were set as output and to low. Timer 3 was initialized and set to zero for the servos with the prescaler set to "fast pwm" mode. Default values for the rotational angles of the pan and tilt were set via visual inspection to desired positions using the OCR3C and OCR3B registers. These values controlled the angle of the pan and tilt respectively

and were adjusted upon receiving new joystick input. As can be observed from Figure 17.0, once the x-axis of the joystick, xJoy, surpassed its deadzone, the angle of the servo was adjusted at a speed range of between -5 and 5. Movement in the opposite direction, as well as in the y-axis, yJoy, occurred in the same manner. This speed determined the increment value to be added or subtracted from the current angle within the **adjust_pan_angle** and **adjust_tilt_angle** functions.

```
if (xJoy > 150){
        adjust_pan_angle(-3);
} else if (xJoy < 125){
        adjust_pan_angle(3);
}

if (yJoy < 125){
        adjust_tilt_angle(3);
} else if(yJoy > 150){
        adjust_tilt_angle(-3);
}
```

Figure 17.0 - Pan and tilt implementation

The functions to adjust the pan and tilt of the servos verified that the passed in incremental value was between -5 and 5, and then proceeded to either increment or decrement the current angles based on which direction the joystick was being pushed to. The OCR3C and OCR3B values were then updated with the new current angles.

3. Conclusion

Through the completion of this project, we were able to learn how to program and control a roomba robot! We were able to use our existing TTA scheduler to enable this functionality. Using uart and bluetooth, a connection was set up to allow communication between a remote station and base station; enabling the roomba to accept commands via user input. The full range of everything that went into the creation of this roomba can be seen in Figure 18.0; from the innovative light sensing mechanism, pan-tilt kit with mounted laser, and the various communication and wiring methods.

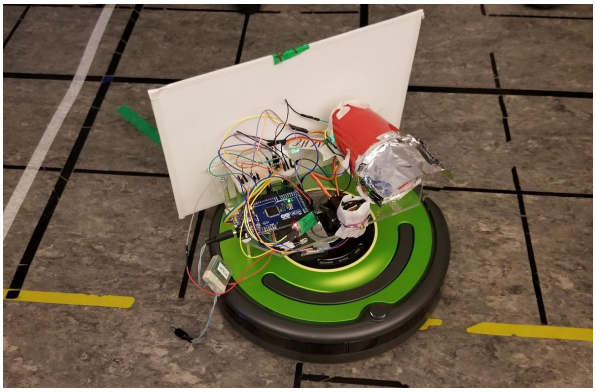

Figure 18.0 - Top-down view of the roomba's features

We thoroughly enjoyed working on this project despite its many tricky parts and we are proud of what we were able to accomplish; building a legendary robot that defended its castle and throne. We are proud to say that our robot was part of the winning team, as shown in Figure 19.0!

Figure 19.0 - A proud moment for Mr. Devalapurkar (right) and Mr. Senekal (left)

All of our hard work paid off in the end! We would like to acknowledge Prashanti Priya Angara, our lab instructor (TA) for all her help and guidance. We are also thankful to Dr. Mantis Cheng for all his teachings.

4. References

[1] M. Cheng, “CSC 460/560 Project 3,” Feb. 2019. [Online]. Available: http://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p3.html. [Accessed Mar. 22, 2019].

[2] N. Macmillan, “Hardware: Roomba,” Feb. 2011. [Online]. Available: https://nrqm.ca/roomba-report/hardware/. [Accessed Mar. 25, 2019].

A. Appendix

This section will outline any additional information or resources necessary to supplement the contents of the report.

A.1. Source Code

The code for our complete roomba implementation can be found at: https://github.com/sdevalapurkar/legendary-roomba/tree/master/project3