Project 1 Report

CSC 460

Group #2

Shreyas Devalapurkar - V00827994

Rickus Senekal - V00826364

Table of Contents

List of Figures and Tables

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. All major terms to be used throughout the document will also be defined in this section of the report.

1.1. Purpose

The contents of this report will discuss our group's implementation of phase one and two of project one. 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 base station that sent, and a remote station that accepted commands via Bluetooth with an analog joystick controlling a pan-tilt kit and a laser. The project had two phases; phase one and phase two.

Phase one included creating a single base station that could perform the following tasks:

- Control two servo motors to perform a pan-tilt operation using a joystick
- Fire a laser on/off with the push of a joystick
- Detect the presence or absence of a light source with a light sensor
- Show the state of a light sensor and joystick with an LCD display

Phase two included splitting up the base station into a base and remote station. The base station interacted with a joystick, a light sensor, and an LCD screen. The remote station interacted with the servo motors in the pan-tilt kit and a laser mounted on top of the pan-tilt kit. The communication between the base and remote station occurred via bluetooth using the protocol we developed. In order to minimize CPU utilization of our system, we developed and utilized a time-triggered scheduler.

The scope of this project was well outlined by Dr. Mantis Cheng (https://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p1.html) [1] and this outline was used throughout the development process to ensure completeness and correctness of our project.

1.3. Glossary of Terms

Term	**Description**
LCD	Liquid crystal display used for displaying some content on a screen
LDR	Light dependent resistor, also known as photoresistor
Photoresistor	Light sensitive devices most often used to indicate the presence or absence of light, or to measure the light intensity
Servomotor (servo)	A rotary actuator that allows for precise control of angular or linear position, velocity and acceleration
TTA	Time triggered architecture

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. Phase One

The objective of phase one of the project was to familiarize ourselves with the hardware and electrical components that would be used for subsequent projects; namely project two and three, and to build a base station. The components we used for this phase of the project were:

- One ATMega2560 board
- One analog joystick
- One pan-tilt kit with two servo motors
- Two breadboards
- One LDR light sensor (photoresistor)
- One DFRobot LCD shield
- One laser pointer
- Miscellaneous wires
- One resistor

2.1.1. System Architecture Overview

The architecture for the system we developed in phase one consisted of many components as previously outlined, and the block diagram detailing how all these components were connected with each other is presented in Figure 1.0.

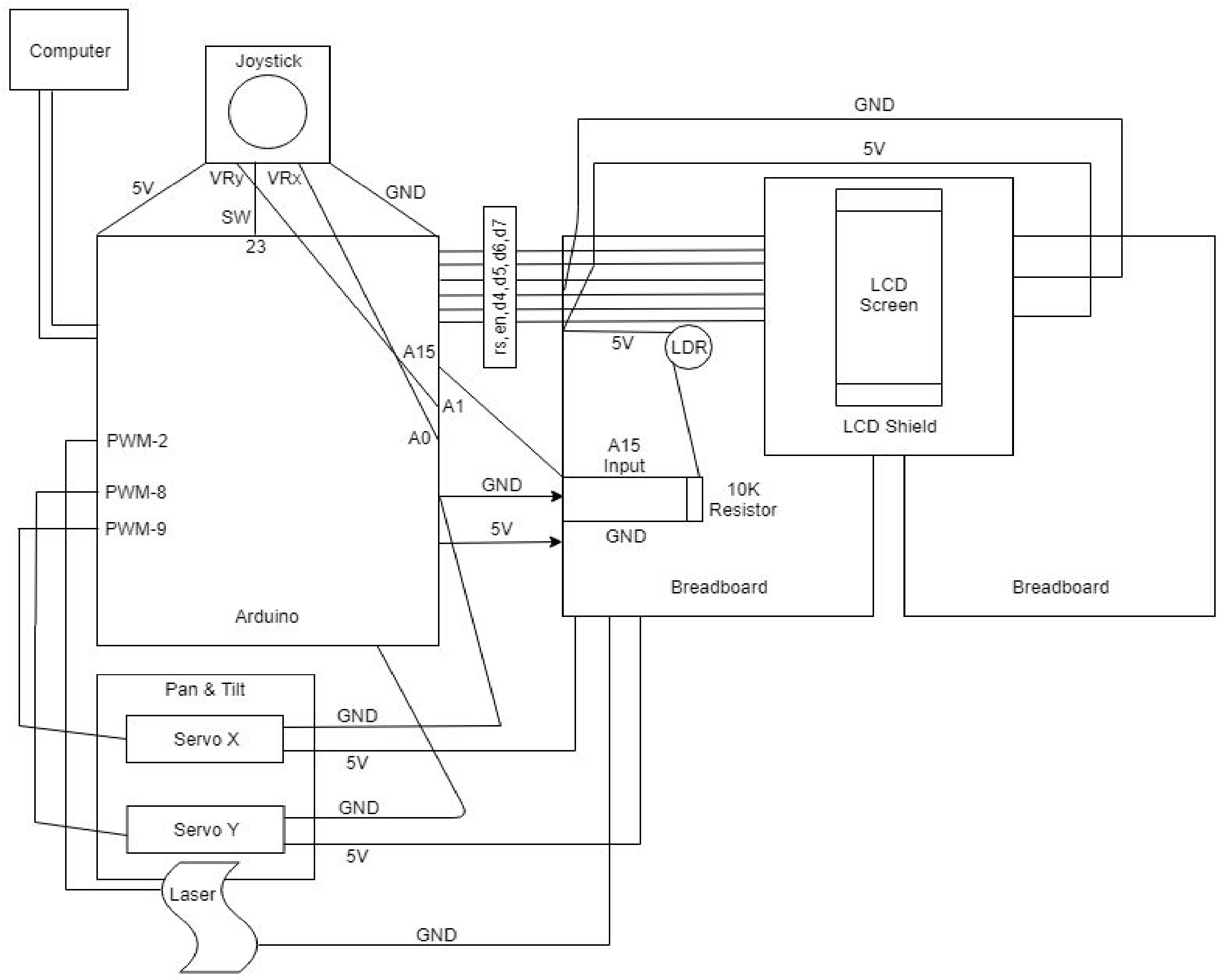


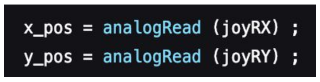
Figure 1.0 - Block diagram of phase one system components

2.1.2. System Functionality Overview

Once the wiring of all the components was complete, the software had to be written in order to provide functionality and perform a multitude of tasks with the various system components.

2.1.2.1. Joystick and Controlling the Servo Motors

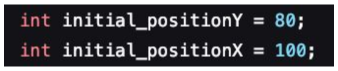
Our first task was to connect the joystick and pan-tilt kit to the arduino board and create a communication channel between them. The wiring of these components up to the arduino is shown in Figure 1.0. In order to control the servo motors, we read the x and y joystick directional values from the specified analog pins. This can be seen in Figure 2.0.



```
x_pos = analogRead (joyRX) ;
y_pos = analogRead (joyRY) ;
```

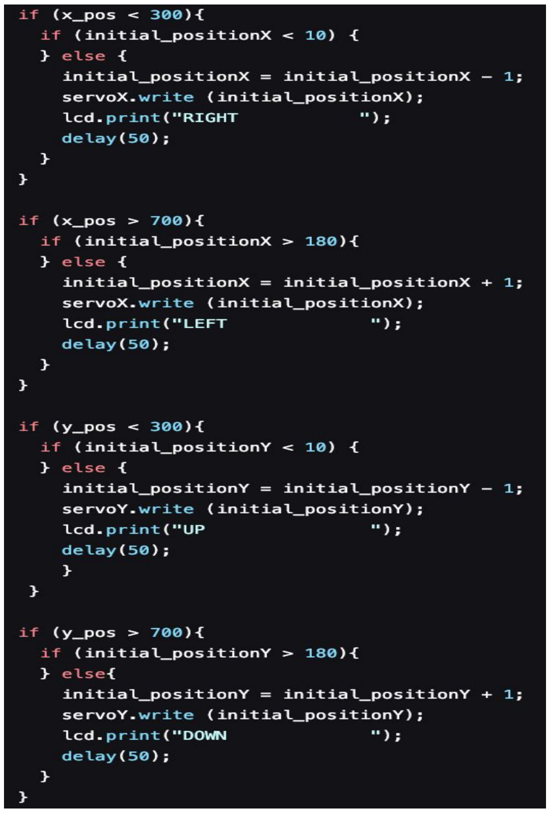
Figure 2.0 - Using analogRead() to read in joystick directional values

Once we had these values, we then used them to determine when to adjust the movement of the servos. Initial values of x and y positions for the servo motors were defined as shown in Figure 3.0 and modified accordingly in order to actually move the servo motors. This logic is shown in Figure 4.0.



```
int initial_positionY = 80;
int initial_positionX = 100;
```

Figure 3.0 - Defining initial values for initial x and y positions of servo motors



```
if (x_pos < 300){
  if (initial_positionX < 10) {
  } else {
    initial_positionX = initial_positionX - 1;
    servoX.write (initial_positionX);
    lcd.print("RIGHT           ");
    delay(50);
  }
}

if (x_pos > 700){
  if (initial_positionX > 180){
  } else {
    initial_positionX = initial_positionX + 1;
    servoX.write (initial_positionX);
    lcd.print("LEFT              ");
    delay(50);
  }
}

if (y_pos < 300){
  if (initial_positionY < 10) {
  } else {
    initial_positionY = initial_positionY - 1;
    servoY.write (initial_positionY);
    lcd.print("UP                ");
    delay(50);
    }
 }

if (y_pos > 700){
  if (initial_positionY > 180){
  } else{
    initial_positionY = initial_positionY + 1;
    servoY.write (initial_positionY);
    lcd.print("DOWN              ");
    delay(50);
  }
}
```

Figure 4.0 - Logic for controlling movement of servo motors

Due to a slight offset in the natural position of the servo motors, the initial positions of the pan-tilt kit were determined through physical examination of the servo positions and set to a desired, centered location. In order to move the pan-tilt kit with a joystick, the values of the joystick were taken into account and sent to the respective horizontal or vertical servo motor. The minimum value input when moving the joystick was 0, with a maximum of 1024 in both the x and y axes. The range between 300-700, known as the dead zone, where no physical movement of the servos are achieved was excluded to increase feedback in the resulting movement. As detailed in Figure 4.0, with physical orientation of the joystick, movement to the left was produced by detecting when the x-axis direction was greater than 700, as well as less than 180 to avoid extending the servo motor past its 180 degree limit. Similarly, movements in the right, up, and down directions were achieved by checking for the respective boundary and activation limits.

Due to the general simplicity of the servo code, the only challenge faced in its implementation was finding the appropriate delay and incremental values of the positions to achieve smooth movement. We identified that if the values are not aligned properly, the control will result in very fast, jittery transitions. The desired values were achieved through trial and error and fine tuning until satisfied with the result.

2.1.2.2. Joystick and Firing the Laser

The next task was to enable the push button on the joystick to turn our laser on and off. The physical wiring and connections of the laser to the arduino is shown in Figure 1.0. In order to be able to easily control the location of the laser pointer based on the movement of the pan-tilt kit, we mounted our laser on the kit as shown in Figure 5.0.

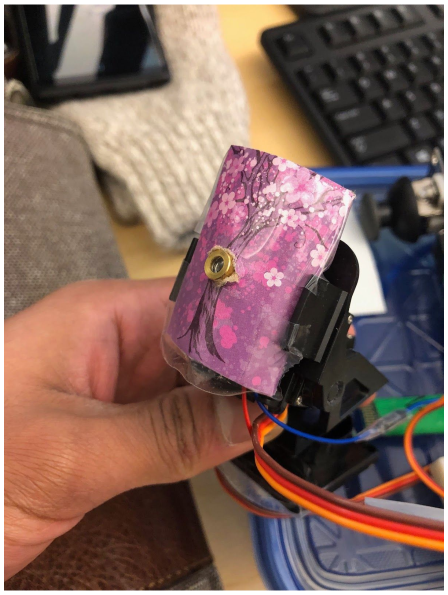
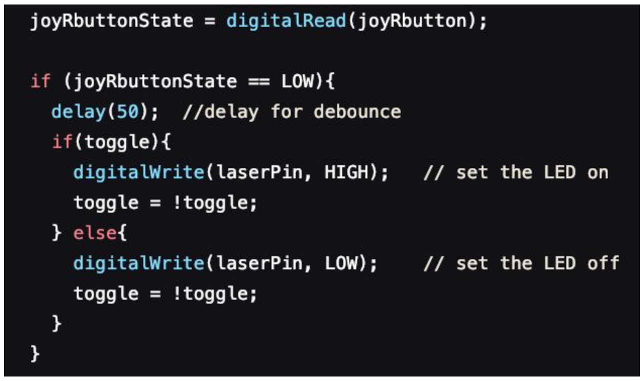


Figure 5.0 - Laser mounted on pan-tilt kit

In order to read in the joystick button push value, we used the Arduino *digitalRead()* function. This allowed us to read in the HIGH or LOW digital value from our joystick. Once we had this value, we were able to use this button state to toggle the laser on and off as shown in Figure 6.0.



```
joyRbuttonState = digitalRead(joyRbutton);

if (joyRbuttonState == LOW){
  delay(50);  //delay for debounce
  if(toggle){
    digitalWrite(laserPin, HIGH);   // set the LED on
    toggle = !toggle;
  } else{
    digitalWrite(laserPin, LOW);    // set the LED off
    toggle = !toggle;
  }
}
```

Figure 6.0 - Toggling laser based on joystick button push

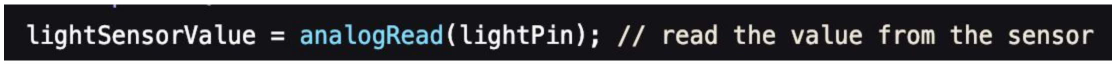
The pin the joystick switch was initialized into the INPUT_PULLUP mode to invert the behaviour of the input, where HIGH meant that the switch was open, and LOW meant that the switch was pressed. This was done using the built-in 20K pullup resistors in the ATMega chip. When a button is pressed, there is usually an oscillation in the signal as the switch returns to its resting state. This is known as bounce. In order to manage the bouncing, a short 50 millisecond delay was added to avoid the writing of any signals sent during this bounce period. A toggle variable was initialized as true, and alternated upon every press of the button. Toggling enables the writing of the connected laser pin from HIGH to LOW and vice versa using the Arduino *digitalWrite()* function.

An overlying issue in regards to a clean on and off switch of the laser remained despite the added delay to handle bouncing of the switch. This was due to timing of the function call and the delay value. In order to have a precise toggle on the switch, the timeframe in which the digitalWrite is written to has to account for multiple values of LOW during the actual button press. If a write command is sent for every single cycle, the laser will flash on and off as it's being written to. This was solved by taking a set of the input values within a period of time and processing them all simultaneously.

2.1.2.3. Detecting a Light Source and LCD Feedback

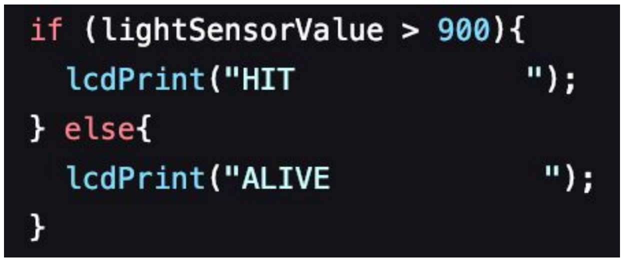
The next task was to have our light sensor or photoresistor be able to detect the presence or absence of a light source. This would be useful for our subsequent projects because our robots would be shooting each other with lasers and we would need a way to determine whether we have been hit or not. The goal was to use a photoresistor to measure the amount of light that was hitting the resistor. If it was under direct light from a laser the Arduino would update the LCD display, saying that it was detecting light.

In order to accomplish this, we first read in the light sensor value from the appropriate pin using the Arduino *analogRead()* function. Once we had this value, we compared it to a specific threshold value to determine whether we were detecting light or not; whether we had been hit or not. This logic and approach is shown through figures 7.0 and 8.0.



```
lightSensorValue = analogRead(lightPin); // read the value from the sensor
```

Figure 7.0 - Reading in light sensor value from pin



```
if (lightSensorValue > 900){
  lcdPrint("HIT             ");
} else{
  lcdPrint("ALIVE           ");
}
```

Figure 8.0 - Logic to detect light source

As shown in figure 8.0, the *lcd.print()* function was used to output text or messages to the LCD screen. This function was used in other places as well such as while controlling the servo motors as shown in figure 4.0. This allowed us to clearly display and indicate system state changes; whether we had been hit or not and which direction the servo motors were moving in.

Printing to the wired 16-bit LCD display required a small amount of finesse despite its simplicity. When writing values in the form of strings, each character is sent to the display and resides within the LCD memory until cleared. Within our code, each movement of the joystick causes a print to the LCD in the form of the direction moved. When the longer word “Alive” is written into the LCD memory followed by a shorter word, such as “Hit”, the display will show the characters “Hitve.” This lead to a small challenge printing only the values desired. Writing the full 16-bit capacity of the display each time movement was made solved this problem.

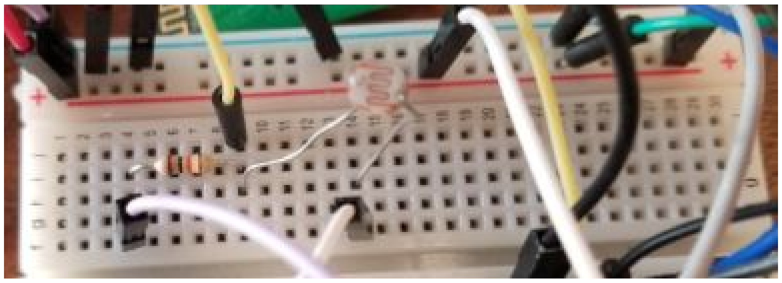


Figure 9.0 - Wiring of photoresistor to detect light source

To receive a desired range of values between 0 and over 1000, a resistor was wired in between the signal pin and the photoresistor. As displayed in Figure 9.0, the sequence of the wired connections goes from a 5-volt input into the photoresistor, and through a 10K resistor to the input pin to read from the grounded sensor. Without the use of the resistor, the values detected by the photoresistor lie in a very small range between roughly 20 and 120. Once in contact with direct light from the laser, the value would drop below 80. In the implementation including the 10K resistor, the range allows for more accurate readings of light intensity and absence. When the laser light is shone onto the photoresistor, the input value increases to a value above 900.

In the subsequent projects where the robots will be shooting each other with lasers, a recalibration of the detected values will be required while the photoresistor is placed in a darkly sealed container. When a laser is shone onto the cup, it will illuminate and trigger signifying that we have been hit. There were no challenges presented in both the connection of the photoresistor as per Figure 1.0 and the programming of its input values.

2.2. Phase Two

The objective of phase two of the project was to build upon phase one and create a communication channel between a base station and remote station via bluetooth. In order to accomplish this, the following components were used:

- Two ATMega2560 boards
- Two bluetooth modules (one master and one slave)
- One analog joystick
- One pan-tilt mechanism with two servo motors
- Two breadboards
- One LDR light sensor (photoresistor)
- One DFRobot LCD shield
- One laser pointer

- Miscellaneous wires
- One resistor

2.2.1. System Architecture Overview

The overall design of phase two required us to make the switch from a delay-based architecture to a TTA-based model in order to accomplish smooth movement for our hardware components. The benefit of a TTA-based model over a delay-based architecture was that we were no longer committed to maintaining a single delay for all our components, and we were able to update certain components more frequently than others when necessary. The TTA we used was a periodic cooperative multitasking system with no operating system to provide oversight and error handling. As such, tasks did not have internal delays; rather task functions returned as soon as possible to give other tasks the chance to run [2]. Phase two of the project consisted of two parts, the base station and the remote station.

2.2.1.1. Base Station

Presented below in figure 10.0 is the system architectural block diagram for our base station, which outlines all the components used and their connections. Each component's wiring was done in reference to the course resources, including schematics and other wiring diagrams [1]. Majority of the components used for the base station were similar to that of phase one, with the addition of a HC-05 bluetooth module. There was also a shift in partial functionality from the base station to the remote station with movement of the pan-tilt kit.

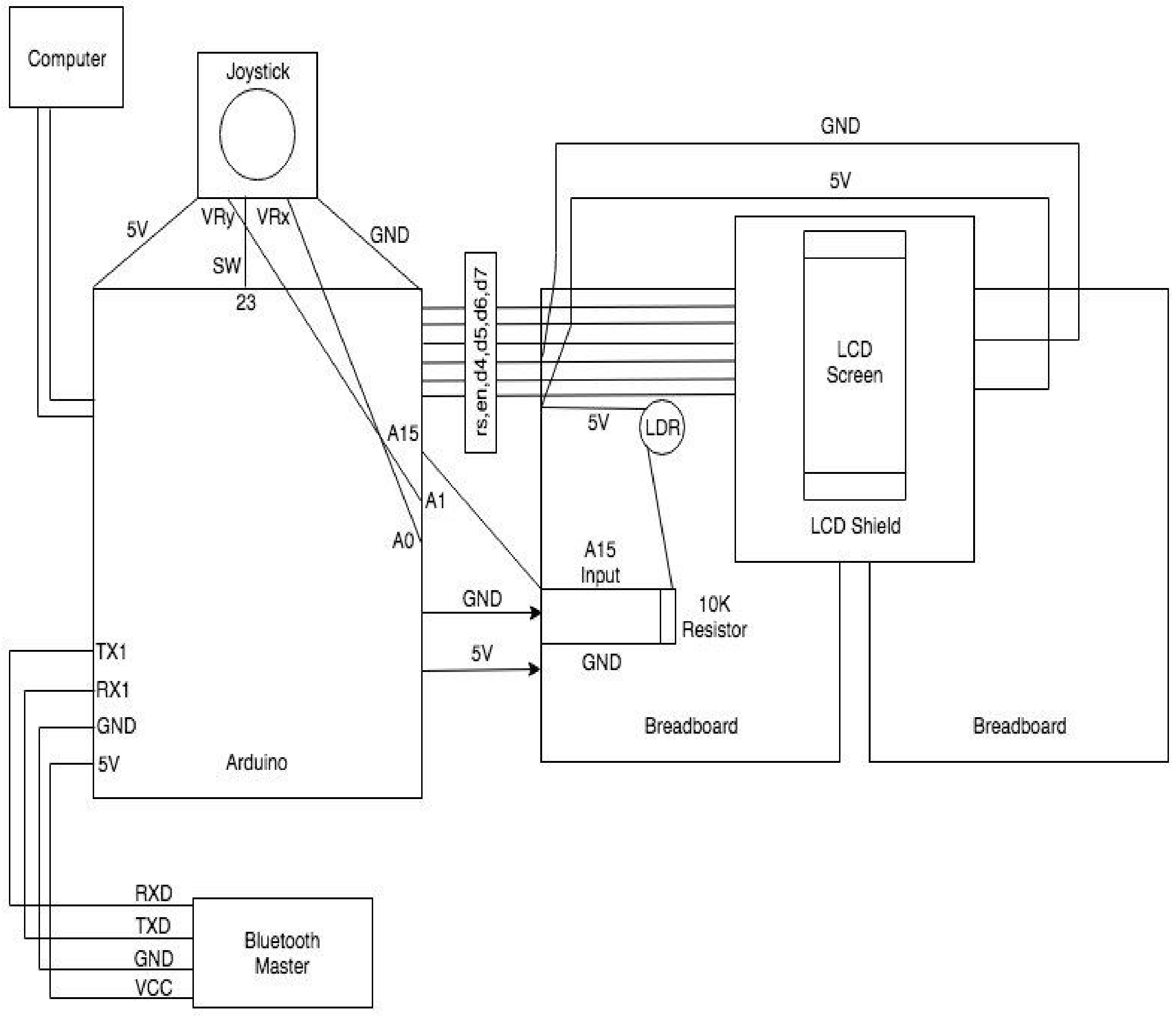


Figure 10.0 - System architecture of the base station

2.2.1.2. Remote Station

Presented below in figure 11.0 is the architectural system block diagram for our remote station and all the necessary wiring. Each component's wiring was done in reference to the course resources, including schematics and other wiring diagrams [1].

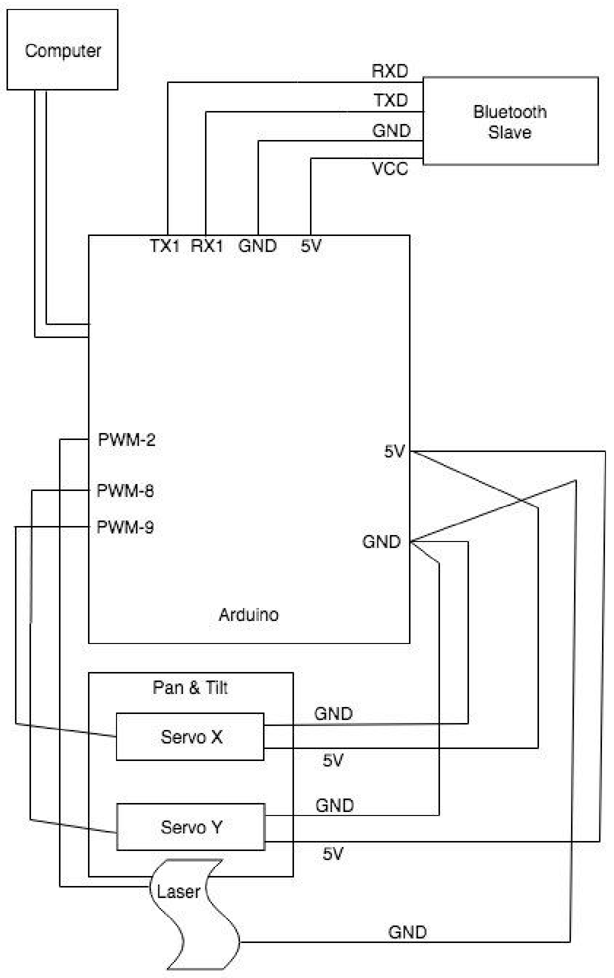


Figure 11.0 - System architecture of the remote station

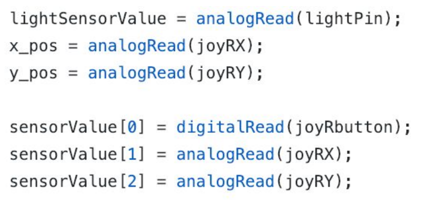
2.2.2. System Functionality Overview

Once the wiring of all the components was complete, the software had to be written in order to provide functionality and perform a multitude of tasks with the various system components. Most of the functionality was ported over from phase one; however, the logic had to be split between the two stations and a communication channel had to be established using bluetooth. Also, data had to be sent over this channel and interpreted before performing the necessary actions.

2.2.2.1. Base Station Functionality

The main functionality of the base station included reading the various sensor values such as movement of the joystick or button push, sending this data over bluetooth to the remote station, checking the light sensor for light detection, and outputting the joystick movement values to the LCD display.

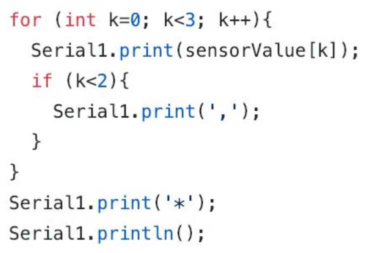
Most of this functionality was identical to that of phase one but the major differences were in the reading data and sending data parts, because now there was communication between two ends and not all the functionality was occurring at a single location. Along with this, another big difference was that our architecture was TTA-based and not delay-based. The code for reading in the sensor values and sending those values over to the remote station is shown in figures 12.0 and 13.0 respectively.



```
lightSensorValue = analogRead(lightPin);
x_pos = analogRead(joyRX);
y_pos = analogRead(joyRY);

sensorValue[0] = digitalRead(joyRbutton);
sensorValue[1] = analogRead(joyRX);
sensorValue[2] = analogRead(joyRY);
```

Figure 12.0 - Reading in sensor values from hardware



```
for (int k=0; k<3; k++){
  Serial1.print(sensorValue[k]);
  if (k<2){
    Serial1.print(',');
  }
}
Serial1.print('*');
Serial1.println();
```

Figure 13.0 - Sending sensor values in a predefined format to the remote station

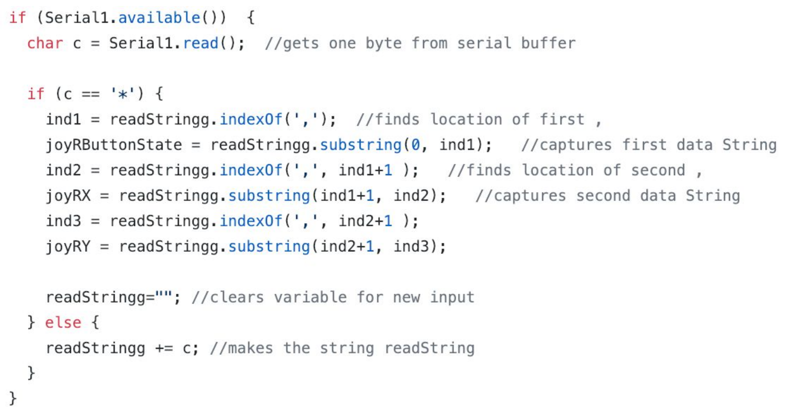
The packet that we sent from the base station to the remote station had a specific format in order to allow the remote station to decipher it and store all the sensor values that it received. The format we decided to use was as follows:

- Joystick button push value (HIGH or LOW) followed by a comma (,)
- Joystick right or left movement value followed by a comma (,)
- Joystick up or down movement value followed by an asterisk (*)

2.2.2.2. Remote Station Functionality

The main functionality of the remote station included reading in the packets sent to it via the base station, deciphering the values present in the packets, and then controlling the pan-tilt kit servo motors and laser. Once again all this was done using a TTA-based approach and the analysis/discussion of this will be shown in a later section of this report.

The trickiest part of implementing the remote station was reading in the values from each packet being sent and getting the values out of the packet as necessary. In order to do this, we read the input byte by byte and stored the characters into a string. Once we knew that we have reached the end of the string; seen the * character, we separated the string by the index of the commas and got our 3 resulting values. This logic is shown in figure 14.0.



```
if (Serial1.available())  {
  char c = Serial1.read();  //gets one byte from serial buffer

  if (c == '*') {
    ind1 = readStringg.indexOf(',');  //finds location of first ,
    joyRButtonState = readStringg.substring(0, ind1);   //captures first data String
    ind2 = readStringg.indexOf(',', ind1+1 );   //finds location of second ,
    joyRX = readStringg.substring(ind1+1, ind2);   //captures second data String
    ind3 = readStringg.indexOf(',', ind2+1 );
    joyRY = readStringg.substring(ind2+1, ind3);

    readStringg=""; //clears variable for new input
  } else {
    readStringg += c; //makes the string readString
  }
}
```

Figure 14.0 - Getting necessary values from packet

Once this part was done, the rest of the functionality was simple because it was a copy of what we had already accomplished in phase one of the project.

2.2.3. Timing and CPU Utilization

To measure the actual runtime of each task that was running on either the base or remote station, we used digital logic analyzers. At the beginning of each task that was being run, we set a digital output pin to HIGH, and then once the task was complete we set this same pin to LOW. Setting the digital pins takes virtually no time, so this was able to provide us with a very accurate measurement of the runtime of our tasks. The following two sections will go into more detail about the measurements taken from the base station and the remote station, as well as calculations of the CPU utilization of each program.

In order to calculate the total CPU utilization of our program, we were able to use the following formula:

CPU Utilization = computation time/period

where the total CPU utilization equals the sum of all individual task utilizations.

2.2.3.1. Base Station Calculations

As mentioned above, the base station had specific pieces of functionality that it needed to perform. This functionality was broken down into 6 specific time-triggered functions. These functions ran at certain periods as shown in figure 15.0 and we were able to analyze this data to determine what the measured runtime was for each task and this is shown in table 1.0.

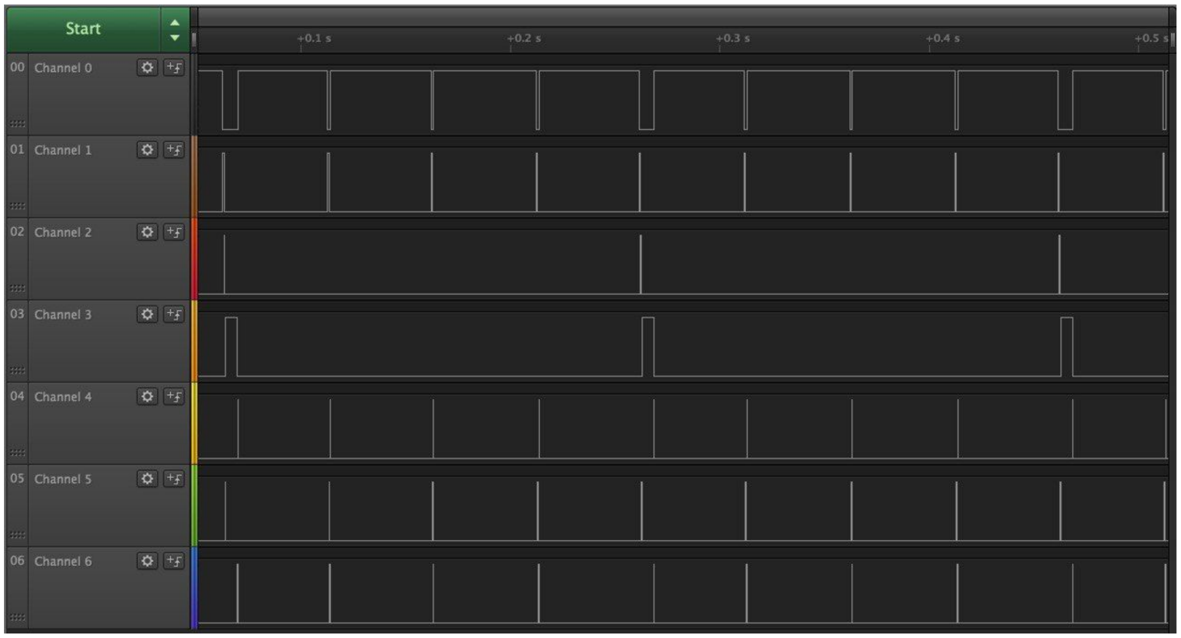


Figure 15.0 - Logic analyzer results for the 6 main functions of the base station

Channel	Task	Offset (ms)	Period (ms)	Measured Runtime (ms)
0	idle	-	-	-
1	readSensors	0	50	0.56
2	sendValues	0	200	0.38
3	checkLightSensor	0	200	5.38
4	checkJoyRAxis	0	50	0.0066
5	setCursorZero	0	50	0.32
6	setCursorOne	0	50	0.32

Table 1.0 - Measured runtime for base station tasks

Using the formula presented above, we were able to determine that our net CPU utilization for the base station was:

(0.56/50) + (0.38/200) + (5.38/200) + (0.0066/50) + (0.32/50) + (0.32/50) = **0.0529** or **5.29%**
In this case, the CPU idle time would be the remainder of the time, or **94.71%** of the time.

2.2.3.2. Remote Station Calculations

As mentioned above, the remote station had specific pieces of functionality that it needed to perform. This functionality was broken down into 3 specific time-triggered functions. These functions ran at certain periods as shown in figure 16.0 and we were able to analyze this data to determine what the measured runtime was for each task and this is shown in table 2.0.

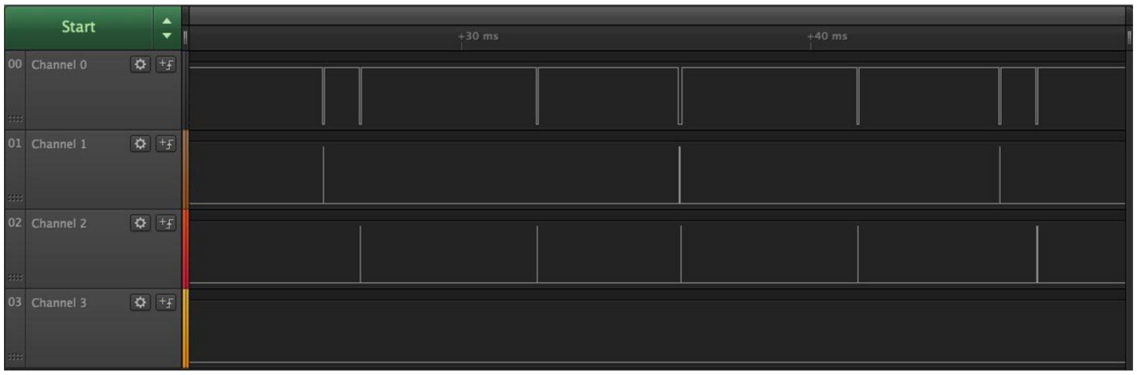


Figure 16.0 - Logic analyzer results for the 3 main functions of the remote station

As you can see in table 2.0, channel zero shows that the system is idle for the majority of the time, and that the rest of the tasks that need to be run are nicely distributed, avoiding conflicts.

Channel	Task	Offset (ms)	Period (ms)	Measured Runtime (ms)
0	idle	-	-	-
1	readValues	0	10	0.0056
2	sendLaser	1	5	0.011
3	moveServo	0	50	0.047

Table 2.0 - Measured runtime for remote station tasks

Using the formula presented above, we were able to determine that our net CPU utilization for the base station was:

(0.0056/10) + (0.011/5) + (0.047/50) = **0.0037** or **0.37%**
In this case, the CPU idle time would be the remainder of the time, or **99.63%** of the time.

3. Conclusion

Through this project, we were able to learn how to connect and wire various different hardware components together and use software to control them. In phase one we built a base station that controlled the servo motors within a pan-tilt mechanism, controlled a laser and also handled sensing the presence of a light source using a photoresistor. In phase two, we built a remote station and created a communication channel between our base and remote stations via bluetooth. We also used a time triggered scheduler to handle performing all the operations using our various components. The final build result is shown in figures 17.0 and 18.0.

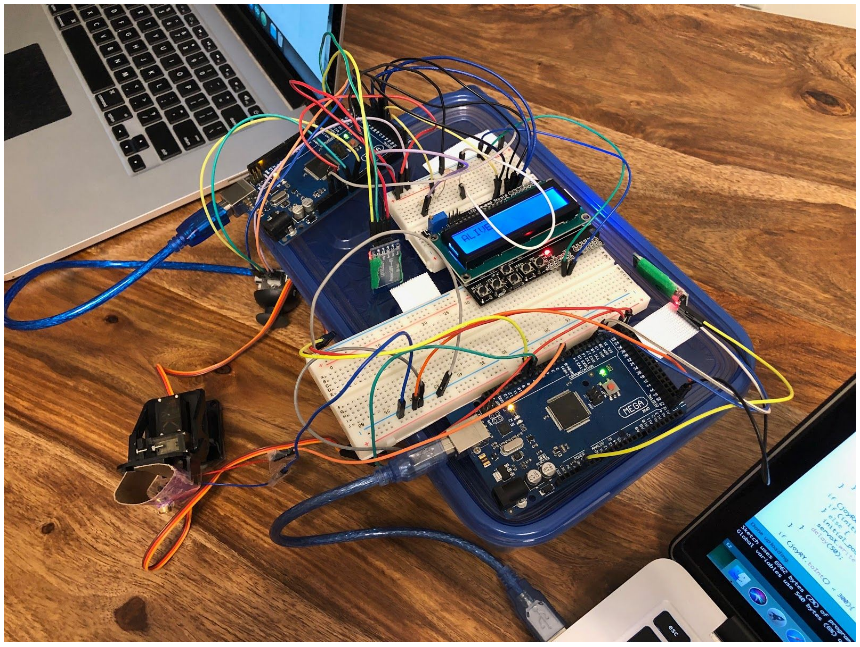


Figure 17.0 - Phase two final build

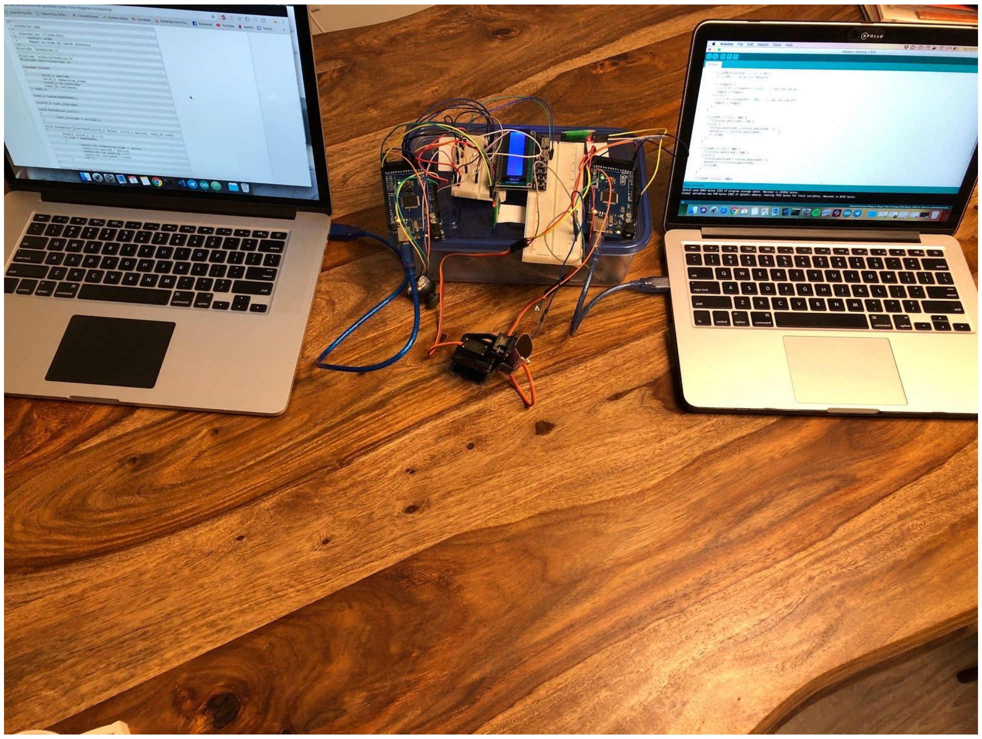


Figure 18.0 - Communication between base and remote station via bluetooth

We thoroughly enjoyed working on this project and are proud of what we were able to accomplish. We look forward to the subsequent projects wherein we will be able to further hone our skills and build even cooler mechanisms. We would like to acknowledge Prashanti Priya Angara, our lab 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 1,” Jan, 2019. [Online]. Available: https://webhome.csc.uvic.ca/~mcheng/460/spring.2019/p1.html. [Accessed Jan. 30, 2019].

[2] N. Macmillan, “Lab Guide: Time-Triggered Scheduling,” Feb, 2011. [Online]. Available: https://nrqm.ca/mechatronics-lab-guide/lab-guide-time-triggered-scheduling/. [Accessed Feb. 1, 2019].

A. Appendix

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

A.1. Source Code

Phase one code can be found at:

https://github.com/sdevalapurkar/legendary-roomba/tree/master/project1/phase1

Phase two code can be found at:

https://github.com/sdevalapurkar/legendary-roomba/tree/master/project1/phase2