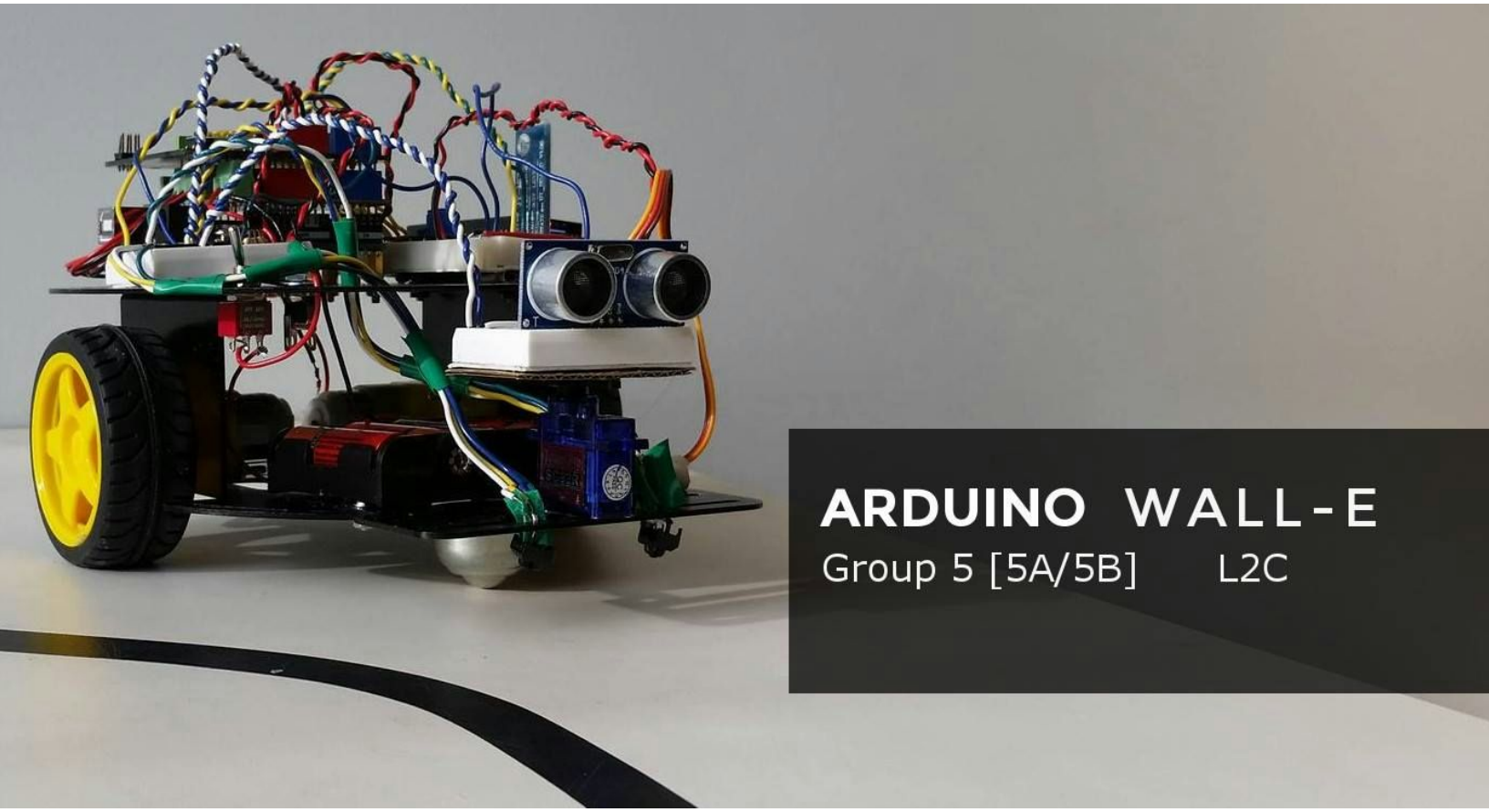



# ELEC 291

## Project 1

---

**GROUP MEMBERS**

Simon Jinaphant

Nicholas Leung

Robyn Castro

Valerian Ratu

Saif Sajid

Paul Kim

# TABLE OF CONTENTS

# ELEC 291: ARDUINO WALL-E

Lab section: L2C Group: 5 Bench: 5A & 5B

| Student Name | Student Number | Contribution Percentage |
|---|---|---|
| Simon Jinaphant | 26271149 | 16.6% |
| Nicholas Leung | 30252143 | 16.6% |
| Robyn Castro | 37283141 | 16.6% |
| Valerian Ratu | 18420142 | 16.6% |
| Saif Sajid | 25125148 | 16.6% |
| Paul Kim | 21959144 | 16.6% |

## Contribution Summary:

Simon Jinaphant:

- ➢ Software architecture, Code refactoring, Bluetooth control code, Report Formatting

Nicholas Leung:

- ➢ Robot Assembly, Ultrasonic Range Finder, Temperature Sensor, Servos

Robyn Castro:

- ➢ LCD display logic, Wire organization, Hardware optimization, Report organization

Valerian Ratu:

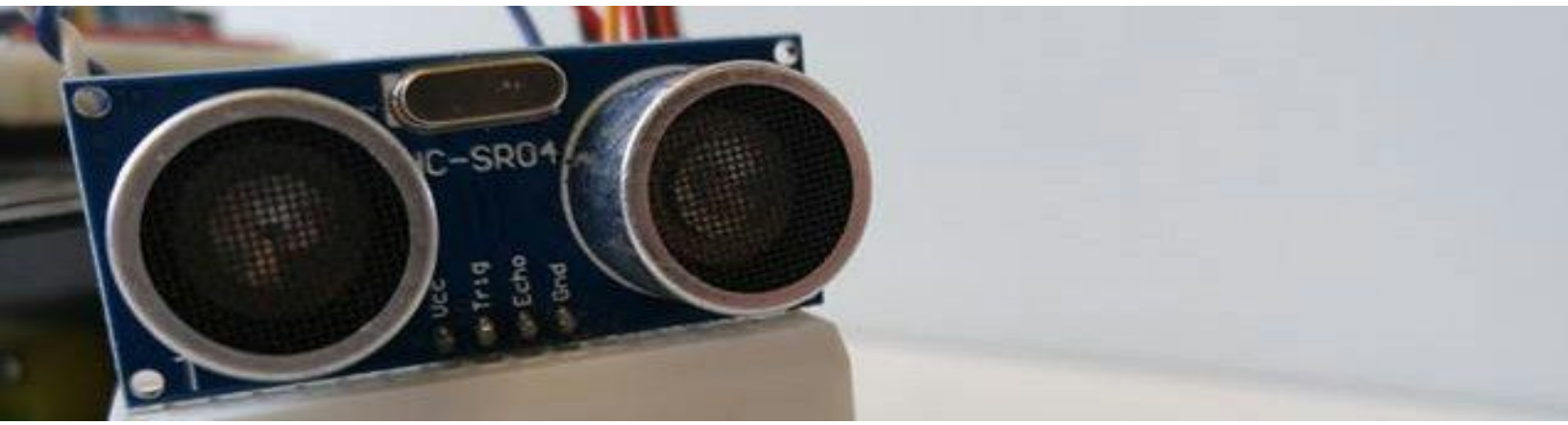
- ➢ Motor circuit and driver, Autonomous mode code, General robot assembly, Report Formatting

Saif Sajid:

- ➢ Bluetooth circuit, Bluetooth mobile app controller (Remote User Interface)

Paul Kim:

- ➢ Reflective optical sensor logic and circuit, Wire organization, Hardware optimization




# B. INTRODUCTION & MOTIVATION

This project is a showcase of the various skills we learned in the labs for ELEC 291. The objective is to implement a multi functional robot using an Arduino, a 2 wheel drive mobile platform, and various sensors. The robot must have two principle functions defined by the instructor and one additional function left to be defined and implemented by the students. The specifications are as follows:

## Principle Function 1

*Autonomous Mode*

- The robot operates autonomously
- The robot must move at the maximum speed forward
- If an object is detected in front of the robot, it slows down gradually
- The robot must stop within a close distance of the object
- An ultrasonic sensor attached to a servo is then used to scan the area around it
- The robot turns 90 degrees left or right depending on the farthest distance received by the sensor

## Principle Function 2

*Line Following Mode*

- The robot operates autonomously
- The robot uses reflective optical sensors to guide itself along a path on the ground
- The path is a line that is darker than the background

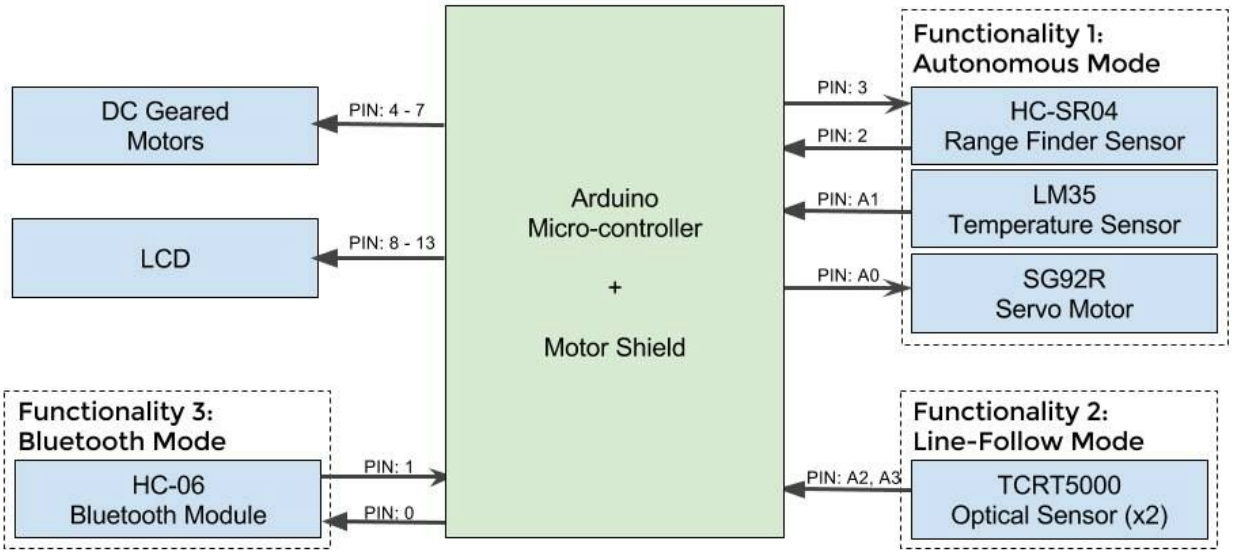
## Additional Function

*Bluetooth Mode*

- The robot operates through human inputs from a Bluetooth enabled controller
- The robot moves forward and backward through input
- The robot rotates left and right through input
- The robot can also do special patterns from a specific input

# C. PROJECT DESCRIPTION

This project integrates multiple modules that together implement the three different specifications required for this robot. The different components are organized by the diagram below.




The following sections describe how the different modules were built with the listed components. Design choices, testing patterns, improvements, and final results were also highlighted for each section.

# C1. Robot Assembly

## *Summary:*

This part of the lab included assembling the DFRobot Turtle 2WD Mobile Platform. This kit includes: two geared DC motors, a chassis, a battery holder, two wheels, Arduino standoffs, and a ball bearing. It also included optional attachments: a chassis extension, a switch, and three rings. We also used a 2A Motor Shield for Arduino to interface the microcontroller with the motors. This part of the lab also included giving the Arduino and the motors an independent source of power.

## *Design Procedure:*

Questions that we had to take into account included:

- Which side will be the front of the robot?
- Which optional attachments should we include?

We decided on making the side with the ball bearing the front of the robot, with the wheels at the back for rear wheel drive. By having the ball bearing in front, it gave the optical sensors sufficient distance to sense the black tape (It was lower compared to being near the wheels instead).

We included all of the attachments besides the three rings. We soldered the switch to the power source to simplify turning the robot on or off. The chassis extension was required to support the different sensors used by the robot. In the instruction manual, the chassis extension was supposed to be put at the back, but we decided to move it to the front of the robot as a platform for the servo and range finder sensor. The grill spacing on the chassis extension was also a perfect fit to thread the two optical sensors. We did not include the three rings because it was unnecessary and cluttered up the robot, especially considering the area required to build our circuits.

After these decisions were made, the assembly of the robot was done using the manual that comes with the robot kit which is under references. The assembly was done with screws with minimal amount of soldering required for both the switch and the motor.

To allow the robot to be fully independent, the Arduino and the motor can be powered through five 1.5V batteries that are connected in series in the battery holder. To implement this, the ground and positive terminal of the battery were connected to the GND and Vin pin on the Arduino respectively. As well, the wires from the battery is also connected to the motor shield's external power supply pins. To ensure that the motors draw power from the battery, the jumpers on the shield must be set to PWRIN.

There were two modes to control the motors through the shield, PWM and PLL. Both modes use pins 4-7 on the Arduino to specify speed and direction of each motor. The PWM pins on the Arduino (5 & 6) control the speed and the digital pins (4 & 7) control the direction of spin. We chose to go with PWM mode since both modes seems to operate in the same manner and we were more comfortable with PWM logic. Using this logic, we implemented various high-level functions that controls the robot's movement that can be called by other modules.

### *Testing Procedure:*

Testing for structural integrity was done by applying force to the robot by hand (pushing, pulling, etc.) to ensure that the robot holds together when it's performing its various functionalities.

We tested the motors by giving a wide variety of inputs for its driver functions. We tested the correctness for both direction and speed by observing how the wheels move as the Arduino runs through the driver functions.

### *Results/Problems:*

Both connectors to the left motor broke off after we soldered the wires. Fortunately, we were able to fix it by soldering thin strands of wires to replace the connectors. Furthermore, the screws given to us had too big of heads, and would not screw into the Arduino perfectly. We were not able to get better screws, and decided that two fully tightened screw and one half tightened screw were sufficient to support the Arduino.

Besides these two complications, the rest of the robot was structurally sound, and worked well with the three functionalities that we had to implement.

## C2. Autonomous Movement

### *Summary:*

This part of the lab included integrating the temperature sensor, ultrasonic range finder (HC-SR04), and servo with the robot built in the previous section. It was set up so that the robot moved forward until it detected an object with the ultrasonic range finder and gradually slows down in order to avoid collision. Once it stops, the servo would rotate the ultrasonic range finder so it could take in data about its surroundings. With this data, the robot would then decide whether to turn left or right based on which side returned a longer distance. The temperature sensor was added to make the ultrasonic range finder more accurate by better approximating the speed of sound.

### *Design Procedure:*

Questions that we had to take into account included:

- How will we mount the range finder onto the servo?
- How will it slow down?
- How do we keep movement and turning smooth?
- How will we wire each of the components?

We decided to put the range finder on one of the small breadboard and use hot glue to mount the breadboard onto the servo. To prevent the breadboard metallic connections from being exposed, we glued on a sturdy piece of cardboard onto the adhesive and then attached the cardboard onto the servo. The wiring of the ultrasonic range finder and the servo was placed on this breadboard for convenience and close proximity as well as to reduce the amount of wires that are moved by the servo's rotations.

To slow the robot down, we adjust the speed based on the distance received from the range finder. When the robot reaches a certain distance away from an object, it will slow down from its maximum speed and reach a full stop 10cm away from the object. Initially, the slowdown was a linear relation between the voltage given to the motors to the distance given by the range finder. We found that this model doesn't exhibit the gradual slowdown explicitly stated in the specifications and would often exceed the minimum distance due to momentum. We moved to a model with a parabolic relation between distance and speed. Not only does this allow for a visible speed decay it also solved the momentum issue pushing the robot further than it should.

Another issue we had with the motor is that when the robot is stationary, it needs a certain threshold voltage to start moving due to friction. However, this issue also appears when the robot is moving slowly. Even though the robot is not yet at the minimum distance required from the object, the speed value given by our model is too low for the robot to move forward so it would be stuck without ever reaching the required distance. To solve this we implemented a counter that increments every time the same distance reading is obtained from the sensor while the robot is in its slowdown period. If the counter exceeds a certain value, we force the robot to find a new path. We tested this implementation by holding an object a certain distance from the robot such that the voltage given to the motors does not exceed the threshold and observes if the robot 'gives up' after a few moments before searching for a new route.

Another problem we had was that the wheels were unstable, and the motors were not moving the same speed even though voltage given was the same. This caused the robot not to go straight when it needed to. To fix this we attached a magnet and hung a hall effect sensor near each of the wheels. These hall effect sensors would detect whenever the magnet would pass it (it would detect whenever a full rotation happened); the hall effect sensors acted as rotary encoders. If a wheel was detected to be faster than the other wheel, the software would adjust the wheel speeds accordingly. Unfortunately, one of the hall effect sensors broke the before the demo and due to a lack of time we abandoned this idea. Instead, we calibrated the wheels every time we turned on the robot; if one wheel was going too fast, we adjusted the input voltage to that specific wheel.

The wiring of the temperature sensor and the motors were simple. We already knew how to wire the temperature sensor, and it could be put on the breadboard near the Arduino. The wires soldered to the motors were threaded through the holes in the robot and then connected to the Arduino. The wiring for the servo analog wire and the ultrasonic range finder however was more complicated because they had to be at the front of the robot and away from the Arduino. Therefore, there were long wires connected to the rotating mini-breadboard. To keep the moving wires wired into the breadboard, we braided them to keep them from deforming. We also made the wires longer they would not disconnect from the breadboard as it rotates.

In an attempt to make a more sophisticated turning algorithm, instead of a binary left/right, there exists a function, findEscapeAngle(), which scans the entire 180 degrees in front of the robot and returns an angle of the longest distance. The robot would then turn to this specific angle and move forward. To enter this mode, replace the similarly named 'findEscapeRoute()' function in the autonomous loop. We were not able to demo this due to time constraint.

### *Testing Procedure:*

We first tested to see if the robot could go straight, turn 90 degrees, slow down and stop. Once we were satisfied with how the robot moved, we then moved onto the next stage of testing.

We then focused on the ultrasonic range finder and the servo. We printed the values given by the ultrasonic range finder to the serial monitor and used a ruler to see if the values are as expected. Then we rotated the servo, which rotated the ultrasonic range finder to see if correct distances were given to surrounding blockades.

Once both movement and sensors were tested, we tested to see if they could work concurrently. We used books to create obstacles for our robot and observed if it chose the path with a further blockade (the one without a book). The books were arranged perpendicularly and placed so that the robot would be parallel to one of the books.

### *Results/Problems:*

Since our hall effect sensor broke we had no way to calibrate wheel speeds while the robot was running. This had a significant effect on how straight the robot would move after running for an extended period of time.

There were also problems with the wiring on the rotating breadboard. Even though we had implemented some preventative measures for wire disconnections (as described above), it did not work 100% of the time. If we were to do it again, we would get female - female connector so we could directly wire the ultrasonic range finder to the Arduino.

Besides these two complications, the ultrasonic range finder detected proper values the majority of the time, and the robot would turn accordingly. However, it still had problems when it encountered a slanted blockade.

## C3. Path Detection

### *Summary:*

This part of the lab relied on reflective optical sensors as our primary sensors; in this mode the robot can detect the reflectivity of a surface and adjust the wheels to turn accordingly in order to follow the black line.

### *Design Procedure:*

Questions that we had to take into account included:

- Where will we attach the reflective optical sensors?
- How many optical sensors should we attach?
- How fast should the robot go when on the path and when turning?
- How will we wire each of the components?

The reflective optical sensors were attached to the front of the robot so data was transmitted to the robot before the wheels hit that part of the track. This gave the wheels some time to process the data, and made the delay more manageable.

We knew we needed at least two optical sensors, so we could tell the robot when to turn left and when to turn right; if there was only one, we could only detect if it was on the path. We decided to use the minimal amount of optical sensors, because we were already running out of pins. We were running out of pins because we decided to implement an LCD to make the robot more user-friendly (this took 6 of our digital pins).

To implement the path finding functionality on the Arduino, we decided to use two optical reflective sensors. We began by initializing the reflectivity value of the "white" surface by reading values from the left and the right optical sensors using analogRead() in the setup. We then read the values of the optical sensors twice, this time inside the loop, and used the second value read. This was done in order to reduce the possibility of noise interfering with our logic. To determine if the robot should turn or go straight, we used a conditional "if" statement. If the left sensor read a value greater than the value initialized in the setup, the robot should turn left. Similarly, if the right sensor read a value greater than the initial value, the robot should turn right. If neither of the sensors detected anything, then the robot should go straight at maximum speed. More detail on how we determined the best logic for turning is included in the "Testing Procedure" section.

Finding the right speed was done by a software test, we had the robot start off at a base Motor PWM of 150/255 and automatically increments itself by units of 10 per every 10 seconds. The LCD was also used for outputting the current PWM speed for us to observe and record the ideal number; this finds the right speed the robot can follow the line before it goes too fast and runs off course without having to manually stop and calibrate the motor speed manually.

We decided to wire the components using a small breadboard. We connected a 5V power source to the collector and the anode through a 10k and a 100 ohm resistor respectively and the emitter and the cathode to the ground. The collector was also connected to one of the analog pins. The optical sensor was attached to the chassis extension, and wires were soldered onto them. These wires were then threaded through the robot and attached to a small breadboard near the Arduino.

### *Testing Procedure:*

We first tested to see if the optical sensors were detecting proper values given a black background and a white background. Once we saw a noticeable difference in the values printed to the serial monitor with two different backgrounds, we decided that the reflective optical sensors were working.

We then tested the wheel logic to find the most optimal speed at which the robot should turn. To turn in a direction, we tried setting wheel located in the direction we wanted to turn to be slightly slower than the other wheel, to stop, and to move backwards. After testing multiple times on the track, we found that stopping the wheel (in the program) made the robot move the fastest and wobble the least on the track. From this testing, we also realized that even if

we set the speed of the wheel to be zero on the software side, on the hardware side, the wheels needed time to slow down and stop completely.

To test for functionality, we used different tracks made up of black electrical tape on the ground and saw if it could follow the path. Different paths had different maximum speeds before the robot fell off the track, so we first used the software test mentioned above to get an approximate speed to test (motor speed of 160 for the smaller track which had sharp turns and 200 for the larger track which had smooth curves). We then ran the tests again with the approximated speed to determine which part of the track was causing the robot to go off course and made calibrations to the speed accordingly. From this testing, we also noticed that the distance between the two optical sensors made a significant difference on the path finding functionality. If the sensors were placed too close to each other, then both the sensors were detecting the path while turning which made it go off track. If the sensors were placed too far apart from each other, the robot ended up turning too soon which also made it go off the track. As a result, we had to make adjustments to the position of the sensors while testing.

### *Results/Problems:*

During our initial stage of testing, we had the robot connected to a serial monitor to see if the optical sensors were reading the expected values. Although we were able to confirm that the optical sensors were functioning correctly, we did not realize that the serial.print() was slowing down the program significantly and as a result, the robot did not turn properly at corners of the track. In our final version of the robot there was no serial monitor, so the robot could follow the path at a relatively quickly.

Another problem we encountered was that the values read from the optical sensors were fluctuating slightly because the movement of the robot was not perfectly stable. As a result, the robot was turning when it was not supposed to. To fix this problem, we added a threshold such that the robot only turned if the value read was greater than the initial value plus two hundred. This was possible because the values read from white and black surface differed by more than five hundred.

## C4. Bluetooth Mode

### *Summary:*

This part of the lab was the additional functionality left for the team to freely implement; for this feature we chose to implement remote user interface/control functionality over Bluetooth. Smartphones are ubiquitous these days and so is bluetooth. The combination provides the best possible user interface to the robot. We also added Voice Recognition to increase accessibility and the coolness factor.

Using an HC-06 bluetooth module, we connected the robot/Arduino to an Android mobile app that has a GUI for changing between Line Following, autonomous, and manual control model. We also used the Google Speech API for predefined voice commands, but dropped the feature because the professor told us not to include any internet enabled applications.

Instead, we leveraged the phone's internal gyroscope sensor to add tilt to move functionality and shake to wiggle functionality.

To continue with the Rapid Prototyping spirit of the Arduino platform, MIT App Inventor (http://appinventor.mit.edu) was used to develop the Android app that served as the remote user interface to control the robot. This tool allowed for incredibly fast development without wasting time diving into the nitty gritty details of Android app Development. New features and functionality were added in seconds, therefore development time wasn't a limiting factor. Though the interface/creation process may seem simplistic, one can create complex and powerful apps with minimal limitations.

The Bluetooth module allowed for two-way serial communication over Pins 0 and 1. Command were sent over Serial input from phone to Arduino, and status data was sent from Arduino to phone.

### *Design Procedure:*

Connecting the bluetooth module to the board was pretty simple. VCC to power, GND to ground. The TX pin on the module was connected to pin 0 (RX) on the Arduino. The RX pin on the module was first connected to a voltage divider to bring down the voltage to around 3 volts and then to pin 1 (TX) on the arduino.

The app was created using MIT App inventor. The IDE was entirely web based and required a Google Account to get started. Furthermore, the entire process was free.

First the GUI of the app was created. Then the GUI elements were given life by designing the app logic in the Scratch like programming language used by App Inventor.

### *Testing Procedure:*

Testing was also a fairly simple process, since the Bluetooth Module acted simply as a Serial input/output port. All we had to do was to open Serial monitor and test for expected inputs and outputs.

One powerful feature of App Inventor was its Live Companion App. This allowed for live updates to the app on the phone as changes were made to the interface or logic. This meant we could add a button and a corresponding serial command and live test it within seconds.

### *Results/Problems:*

One problem we ran into was trying to implement continuous movement with the control button. Originally, when a button was clicked, the robot would go in the specified direction for half a second and halt. To go in a specified direction for a long time, multiple successive button presses was required. Instead, we wanted to simply press and hold the button and have the robot go in the specified direction until the button was let go. To implement this, we did the following:

- When a button is pressed down, send the corresponding serial command
- When the button is let go, send a halt command

We used single character commands to overcome any latency issues that might have happened. A related problem we encountered was that sometimes when the Arduino sends a

status message to the phone, parts of the message wouldn't appear. We suspect this might be because the app checks for status updates every 300ms, and this interval may not sync up with when the Arduino/BT module sends its update. To fix this, we could reduce this interval. The LCD also serves as a backup status display.

Only one button works at a time to prevent any command conflicts with simultaneous multiple button presses.

A problem that we didn't encounter but still considered was sudden loss of bluetooth connection. This might be a problem for when the robot is waiting for a halt command while in manual control mode. There are two ways to gracefully fail in this situation. One is to have the phone check in with the Arduino through the bluetooth module on a set interval to reassure the robot that Bluetooth is still connected. The second option is to use the unsoldered 'State' pin on the BT module to see if it is connected to a phone. If any disconnection is detected a graceful fail would be halting after a set time interval and displaying a relevant message on the LCD such as "BT disconnected".

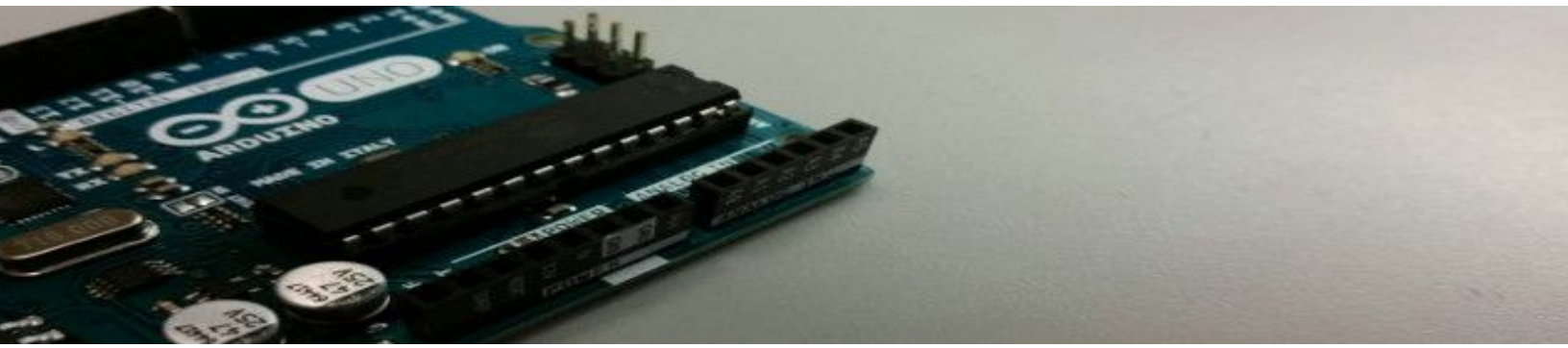
## C5. Further Additions

### *LCD*

We decided to implement the LCD to display the relevant information about the robot depending on its mode. For line-follow mode it displays whether it is turning right or left or moving forward. During autonomous mode the LCD displays the distance to the nearest object, and the escape angle once the robot finds one. It also displays the current mode when mode changes occurs.

### *Mode Changing*

We initially had a push button wired up to digital pin 1 to cycle through the different states of the robot on every rising or falling edge of the button. However, when the Bluetooth mode was implemented we changed mode switching to be handled by Bluetooth instead. Through this, we can change modes easily and remotely, making the robot more user friendly.




# D. CONCLUSION

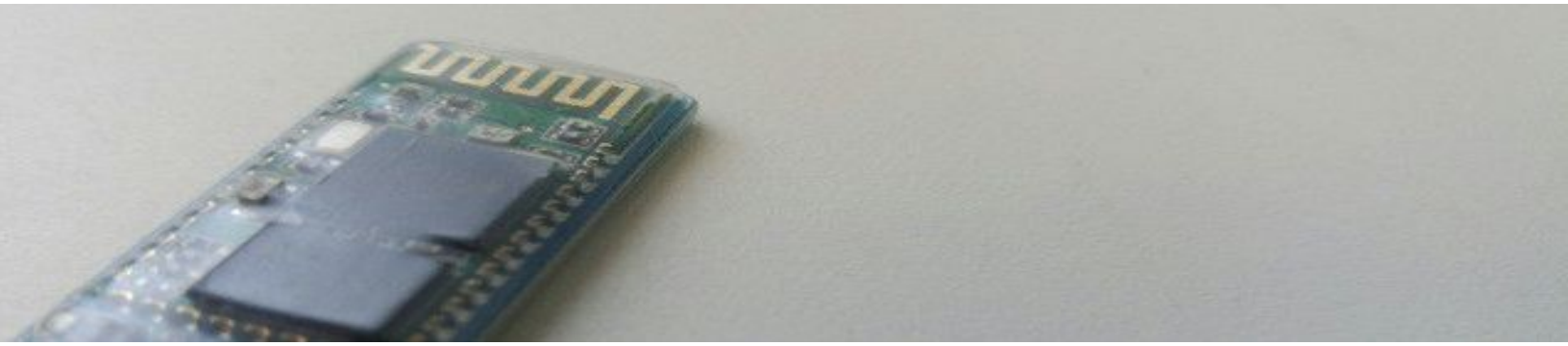
In this project we were able to culminate all our knowledge from the previous labs to create a multi-functional robot; we were able to use our knowledge of displays, various sensors, wiring practices, and soldering. We also learned how to use hall effect sensors, reflective optical sensors, and how to wire an Arduino to an external power source. Although there were some complications regarding the hall effect sensors, our robot was able to perform reasonably well most of the time.

We also learned some lessons from the complications we encountered. These lessons included:

- Double checking if wiring is correct in order to not short-circuit any components.
- Reinforcing any fragile connections or being more careful when handling them.
- Checking the data sheet before and while handling a new component.

Our additional functionality was the ability to control our robot via a bluetooth connection. Once integrated, it worked quite well with our robot, and effectively made the button obsolete. This freed up some pins, which we could have used for another component if we had some more time.

We would definitely recommend including the HC-06 bluetooth module in lab kits for future courses. It's relatively cheap ($6) and offers a lot of freedom through its wireless capability. Coupled with this, we also recommend MIT App inventor to make Android apps that interact with the module. The learning curve is minimal and the possibilities are endless. MIT App Inventor is analogous to Processing or the Arduino platform where the focus is on creativity and not on the nitty gritty details that go into making a full fledged application.



# E. REFERENCES


**Project1 document posted on Connect**

**Motor Shield wiki:**
**http://www.dfrobot.com/wiki/index.php?title=Arduino_Motor_Shield_(L298N)_(SKU:DRI0009)**

**Motor Shield schematic: ArduinoL298ShieldSch.pdf**

**Mobile Platform Assembly Manual: 2WDTurtleAssemblyManual.pdf**

**Motor Shield's motor controller datasheet: L298N_datasheet.pdf**

**Arduino Reference: http://arduino.cc/en/Reference/HomePage**

**Temperature sensor IC datasheet: LM35_datasheet.pdf**

**HC-SR04 datasheet 1: HC_SR04_1.pdf**

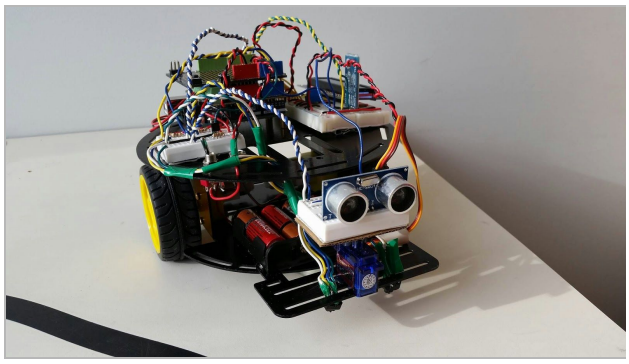
**HC-SR04 datasheet 2: HC-SR04_Manual.pdf**

**Reflective Optical Sensor datasheet: tcrt5000_datasheet.pdf**

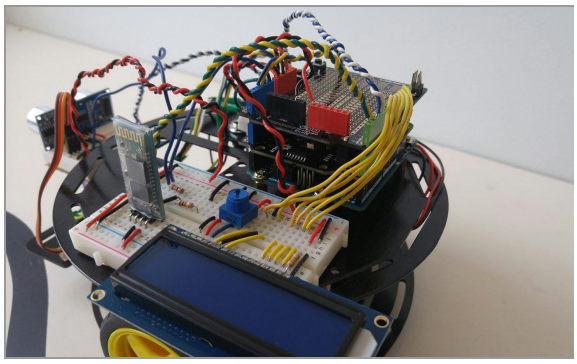
**Hall Effect Sensor datasheet: HallEffect_datasheet.pdf**

**Mit App Inventor: http://appinventor.mit.edu/explore/index-2.html**

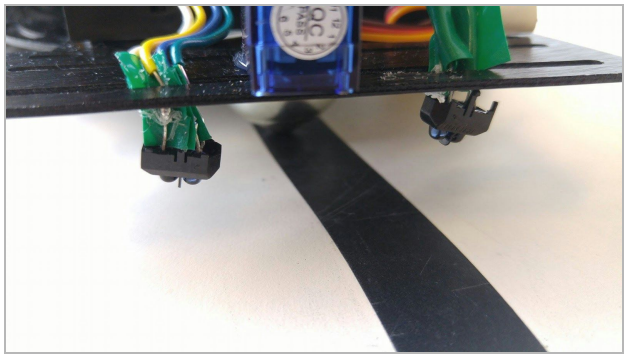
# APPENDIX A. Robot Pictures



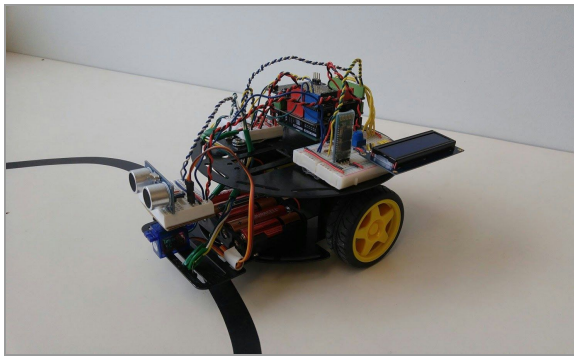
Front side view of Robot



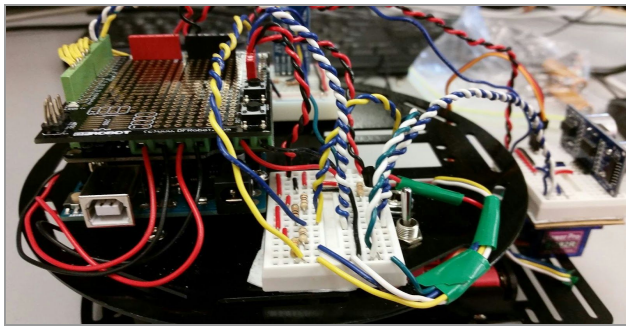
Back view of Wirings



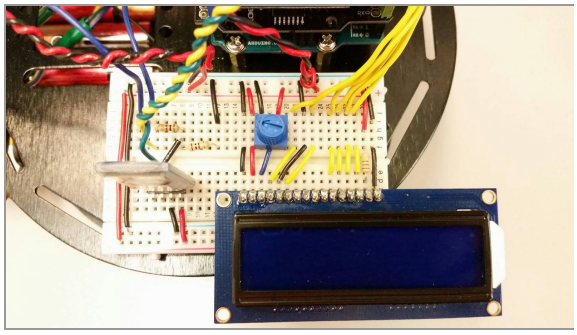
Optical Sensors on bottom front



Side view of robot



Closeup of optical sensor and range finding circuit



Closeup of the LCD and Bluetooth circuit

# **APPENDIX B.** Code

```
#include <LiquidCrystal.h>
#include <Servo.h>

//--------PIN DEFINITION--------
LiquidCrystal lcd(8, 9, 10, 11, 12, 13); //RS, EN, D4, D5, D6, D7

const int ULTRASONIC_TRIG_PIN = 3;
const int ULTRASONIC_ECHO_PIN = 2;
//BLUETOOTH ON PINS 0 AND 1 FOR 2-WAY SERIAL COMMUNICATION

const int LEFT_OPTIC_PIN = A3;
const int RIGHT_OPTIC_PIN = A2;
const int TEMP_SENSOR_PIN = A1;
Servo myservo; //SERVO WILL BE CONNECTED ON A0

const int MOTOR_E1_PIN = 5;
const int MOTOR_M1_PIN = 4;
const int MOTOR_E2_PIN = 6;
const int MOTOR_M2_PIN = 7;

//---------Motor Driver Constants---------
const double LEFT_MOTOR_ADJUSTMENT = 0.97;
const double RIGHT_MOTOR_ADJUSTMENT = 1.0;
const double MAX_SPEED = 255;
const int TURN_TIME = 800;

//Motor Direction Constants
const boolean RIGHT = true;
const boolean LEFT = false;
const boolean FORWARD = true;
const boolean BACK = false;

//Speed Modelling for Motor Acceleration
double A;
double B;

//--------LCD Constants--------
const int FIRST_COL = 0;
const int TOP_ROW = 1;
const int BOTTOM_ROW = 1;

//----Functionality State Changing----
int currentState = 2;   //Start with Bluetooth mode by default

//--------Autonomous Mode--------
const double D_MAX = 40; //Distance robot slows down
const double D_MIN = 10; //Distance robot should stop

//Variables for checking whether robot is stuck
const int D_STALL = 3;
const int STUCK_THRESHOLD = 60;
int isStopped = 0;
```

```
//For Ultrasonic calculation
double temperature;

//--------Line Follow Mode--------
const int LIGHT_THRESHOLD = 200;
const int LINE_SPEED = 150;

int leftWhite = 0;
int rightWhite = 0;
int leftSensor;
int rightSensor;

//Robot States for Line Follow Mode
int robotState;
int lastRobotState = 0;
const int FULL_SPEED = 0;
const int TURN_LEFT = 1;
const int TURN_RIGHT = 2;

//--------------Bluetooth-----------
byte byteRead;
char const BT_HALT = '0';
char const BT_FORWARD = '1';
char const BT_BACK = '2';
char const BT_LEFT = '3';
char const BT_RIGHT = '4';
char const BT_WIGGLE_WIGGLE = 'w';
char const MODE_AUTO = 'a';
char const MODE_LINE = 'l';
char const MODE_BT = 'c';

void setup() {
  //Ultrasonic setup
  pinMode(ULTRASONIC_ECHO_PIN, INPUT);
  pinMode(ULTRASONIC_TRIG_PIN, OUTPUT);

  //Servo setup
  myservo.attach(A0);

  //Light Sensor Initialization
  leftWhite = analogRead(LEFT_OPTIC_PIN);
  rightWhite = analogRead(RIGHT_OPTIC_PIN);

  //Motor Setup
  pinMode(MOTOR_M1_PIN, OUTPUT);
  pinMode(MOTOR_M2_PIN, OUTPUT);

  //LCD Setup
  lcd.begin(16, 2);

  //Setting up the modelling variables
  A = MAX_SPEED / (pow(D_MAX, 2.0) - pow(D_MIN, 2.0));
  B = (MAX_SPEED * pow(D_MIN, 2.0) / (pow(D_MIN, 2.0) - pow(D_MAX, 2.0)));
```

```
  //Temperature initialization
  temperature = (( analogRead(TEMP_SENSOR_PIN) / 1024.0) * 5000) / 10;

  // Initialize serial for bluetooth communication
  Serial.begin(9600);
}

void loop() {
  if (Serial.available()) {
    //Read from Bluetooth to determine state change
    byteRead = Serial.read();
    lcd.clear();

    switch (byteRead) {
      case MODE_AUTO:
        lcd.print("Autonomous Mode");
        currentState = 0;
        break;

      case MODE_LINE:
        lcd.print("Line Follow Mode");
        currentState = 1;
        break;

      case MODE_BT:
        lcd.print("Bluetooth Mode");
        currentState = 2;
        halt();
        break;
    }
  }


  switch (currentState) {
    case 0: autonomousLoop(false); break;
    case 1: lineFollowLoop(); break;
    case 2: bluetooth_loop();  break;
  }

}

/**
   Perform Autonomous navigation functionality
*/
void autonomousLoop(boolean useAngular) {
  myservo.write(90);

  //get rid of interference
  long distance = max(max(get_distance(), get_distance()), get_distance());

  //Retry when the distance isn't within the data-sheet expected values
  while (distance < 2 || distance > 400) {
    distance = get_distance();
  }
```

```
  //Determines whether the robot is stalled
  if ((distance < D_MAX ) && (D_STALL > abs(distance - get_distance()))) {
    isStopped++;
  }

  //Adjusts the speed of robot based on distance of object in front
  int robotSpeed = adjustSpeed(distance);
  moveInDirection(FORWARD, robotSpeed);

  //Outputs status on LCD
  lcd.clear();
  lcd.print("Distance: " + String(distance));
  lcd.setCursor(FIRST_COL, BOTTOM_ROW);
  lcd.print("Speed Ratio: " + String(robotSpeed));

  //Turning Logic
  if (distance < D_MIN || isStopped > STUCK_THRESHOLD) {
    lcd.clear();
    if (isStopped > STUCK_THRESHOLD) {
      lcd.print("Stuck at: " + String(distance));
    } else {
      lcd.clear();
      lcd.print("Object detected at:");
      lcd.setCursor(FIRST_COL, BOTTOM_ROW);
      lcd.print("Dist: " + String(distance));
    }
    halt();

    //Determines the direction the robot should turn
    int escapeAngle = (useAngular ? findEscapeAngle() : findEscapeRoute());

    lcd.setCursor(FIRST_COL, BOTTOM_ROW);
    lcd.print("Turn: " + String(escapeAngle));
    rotateAngle(escapeAngle);
    isStopped = 0;
  }
}

/**
   Perform Line Following functionality
*/
void lineFollowLoop() {
  //Multiple reads to reduce interference
  analogRead(LEFT_OPTIC_PIN);
  leftSensor = analogRead(LEFT_OPTIC_PIN);
  analogRead(RIGHT_OPTIC_PIN);
  rightSensor = analogRead(RIGHT_OPTIC_PIN);

  if (leftSensor > leftWhite + LIGHT_THRESHOLD) {
    moveLeftWheel(FORWARD, 0);
    moveRightWheel(FORWARD, MAX_SPEED);
    robotState = TURN_LEFT;

  } else if (rightSensor > rightWhite + LIGHT_THRESHOLD) {
    // if right sensor detects path, move right
```

```
    moveRightWheel(FORWARD, 0);
    moveLeftWheel(FORWARD, MAX_SPEED);
    robotState = TURN_RIGHT;

  } else {
    // if nothing is detected, move straight
    moveInDirection(FORWARD, LINE_SPEED);
    robotState = FULL_SPEED;
  }

  if (robotState != lastRobotState) {
    lcd.clear();
    switch (robotState) {
      case FULL_SPEED: lcd.print("FULL SPEED"); break;
      case TURN_LEFT: lcd.print("TURN LEFT"); break;
      case TURN_RIGHT: lcd.print("TURN RIGHT"); break;
      default: lcd.print("Invalid");
    }
    lastRobotState = robotState;
  }

}

/**
  Perform bluetooth control functionality; commands from
  the bluetooth daemon will be sent as a char byte depending
  on the required actions.
*/
void bluetooth_loop() {
  switch (byteRead) {
    case BT_HALT:
      Serial.println("Halting");
      halt();
      break;

    case BT_FORWARD:
      Serial.println("Going forward");
      moveInDirection(FORWARD, MAX_SPEED);
      break;

    case BT_BACK:
      Serial.println("Going backward");
      moveInDirection(BACK, MAX_SPEED);
      break;

    case BT_LEFT:
      Serial.println("Turning left");
      rotateAngle(105);
      break;

    case BT_RIGHT:
      Serial.println("Turning right");
      rotateAngle(85);
      break;
```

```
    case BT_WIGGLE_WIGGLE:
      Serial.println("Wiggle in progress");
      lcd.clear();
      lcd.print("Wiggle in");
      lcd.setCursor(FIRST_COL, BOTTOM_ROW);
      lcd.print("progress");

      for (int i = 0; i < 2; i++) {
        wiggle(3);
        moveInDirection(FORWARD, MAX_SPEED);
        delay(1500);
        turnLeft();
        turnLeft();
      }

      wiggle(5);
      lcd.clear();
      lcd.print("Wiggle complete!");
      Serial.println("Wiggle complete!");
      delay(2000);

      byteRead = '0';
      break;
  }
}

/**
   Make the robot do a little dance; to be used in
   bluetooth mode for more additional "features"
*/
void wiggle(int amount) {
  for (int i = 0; i < amount; i++) {
    rotateAngle(45);
    rotateAngle(135);
  }
}

/**
   Moves the robot in a specified direction at a specified speed
   @param dir - the direction the robot is heading
                true for forward, false for backwards
   @param robotSpeed - the speed the robot is moving
*/

void moveInDirection(boolean dir, int robotSpeed) {
  moveLeftWheel(dir, robotSpeed);
  moveRightWheel(dir, robotSpeed);
}


/**
   Stops motors of the robot
*/
void halt() {
  analogWrite(MOTOR_E1_PIN, 0);
```

```
  analogWrite(MOTOR_E2_PIN, 0);
}

/**
   Rotates the robot in place
   @param dir - the direction the robot is turning
                true for right, false for left
   @param duration - the duration the robot is turning
*/
void rotate(boolean dir, int duration) {
  moveRightWheel(!dir, MAX_SPEED * 0.5);
  moveLeftWheel(dir, MAX_SPEED * 0.5);
  delay(duration);
  halt();
}

/**
   Turns the robot left
*/
void turnLeft() {
  rotateAngle(180);
}

/**
   Turns the robot right
*/
void turnRight() {
  rotateAngle(0);
}

/**
   Moves the right wheel in a direction with a speed
   @param dir - the direction the wheel is moving
                true for forward, false for backward
   @param robotSpeed - the speed the wheel is moving
*/
void moveRightWheel(boolean dir, int robotSpeed) {
  digitalWrite(MOTOR_M1_PIN, dir);
  analogWrite(MOTOR_E1_PIN, robotSpeed * RIGHT_MOTOR_ADJUSTMENT);
}

/**
   Moves the left wheel in a direction with a speed
   @param dir - the direction the wheel is moving
                true for forward, false for backward
   @param robotSpeed - the speed the wheel is moving
*/
void moveLeftWheel(boolean dir, int robotSpeed) {
  digitalWrite(MOTOR_M2_PIN, dir);
  analogWrite(MOTOR_E2_PIN, robotSpeed * LEFT_MOTOR_ADJUSTMENT);
}

/**
   Rotates the robot a specific angle in a direction.
   @Param angle - the angle it turns to
```

```
      - 180 for left, 0 for right
*/
void rotateAngle(int angle) {
  if (angle < 0 || angle > 180) return;
  if (angle > 90) {
    rotate(LEFT, (1.0 / 90 * angle - 1) * TURN_TIME);
  } else {
    rotate(RIGHT, (-1.0 / 90 * angle + 1) * TURN_TIME);
  }
}

/**
   Adjust the speed of the robot depending on the distance obtained
   @param distance - the distance of the object in front of robot
   @return speed - the speed of the robot
*/
int adjustSpeed(int distance) {
  int robotSpeed;
  if (distance > D_MAX) {
    robotSpeed = MAX_SPEED;
  }
  else if (distance < D_MIN) {
    robotSpeed = 0;
  } else {
    robotSpeed = A * pow(distance, 2.0) + B;
  }
  return robotSpeed;
}

/**
   Get the distance in centimeters to the closest detected object
   from the ultrasonic sensor
*/
unsigned long get_distance() {

  float speed_of_sound = 331.5 + (0.6 * temperature);
  float denom = (20000.0 / speed_of_sound);

  //Pulse the TRIG on the ULTRASONIC sensor
  digitalWrite(ULTRASONIC_TRIG_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(ULTRASONIC_TRIG_PIN, LOW);

  //Read the ECHO signal from the ULTRASONIC sensor
  unsigned long duration = pulseIn(ULTRASONIC_ECHO_PIN, HIGH);
  unsigned long distance = duration / denom;
  return distance;
}

/**
  Turns the robot strictly left or right,
  depending on in which direction it can go further
*/
int findEscapeRoute() {
  int distance;
```

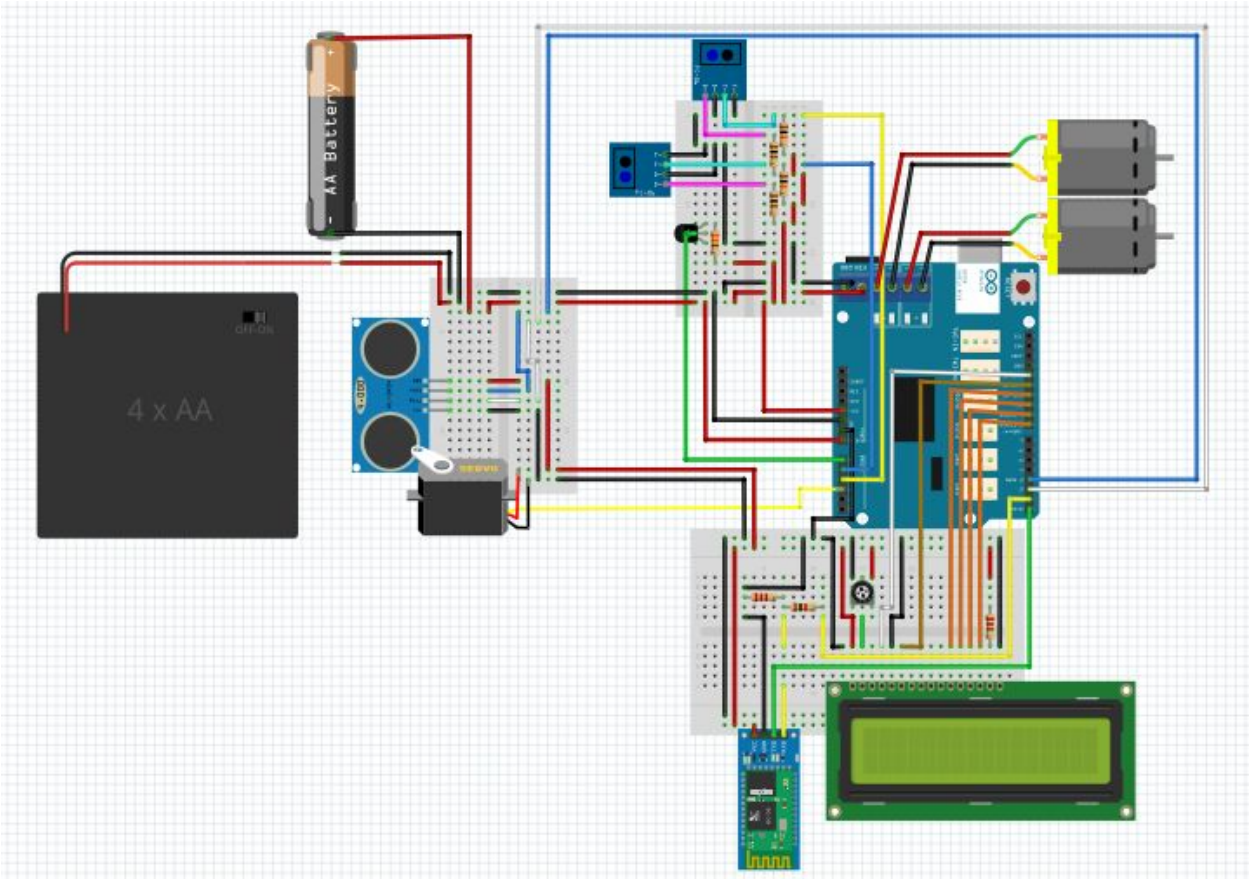
```
  int maxDistance = 0;
  int maxAngle = 0;
  myservo.write(0);
  delay(1000);
  distance = get_distance();
  myservo.write(180);
  delay(1000);
  if (get_distance() > distance) {
    maxAngle = 180;
  }
  myservo.write(90);
  return maxAngle;
}

/**
  Turns the robot in any angle
  depending on in which direction it can go further
*/
int findEscapeAngle() {
  int distance;
  int maxDistance = 0;
  int maxAngle = 0;

  for (int p = 0; p <= 180; p++) {
    myservo.write(p);
    delay(10);
    if (p == 0) {
      delay(20); //Timing issue in initial servo pan?
    }
    distance = get_distance();
    if (distance > maxDistance) {
      maxDistance = distance;
      maxAngle = p;
    }

  }
  return maxAngle;
}
```

# **APPENDIX C.** Fritzing Schematics



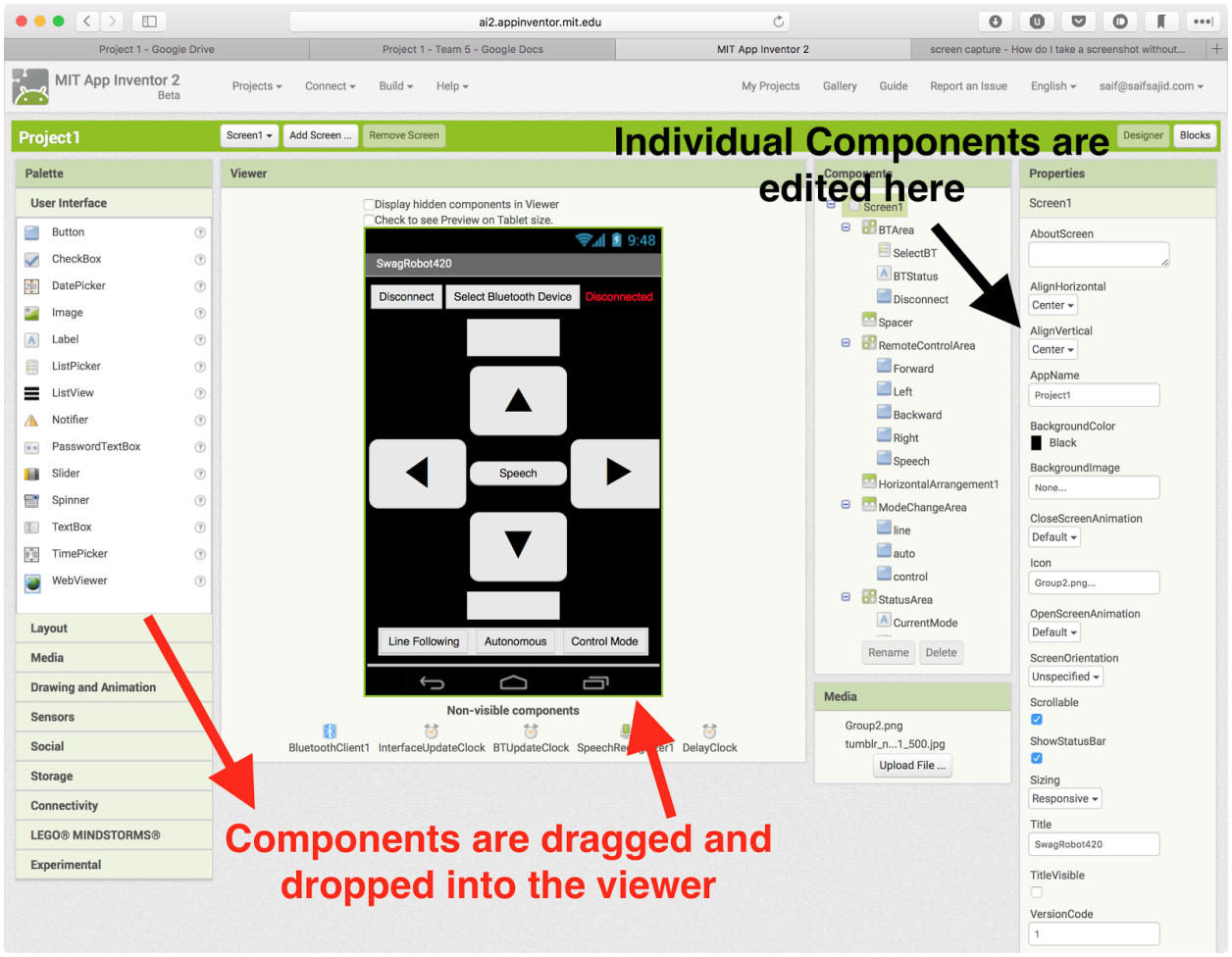
# APPENDIX D. Bluetooth Mobile App Controler

## A-D1. CONTROLLER INTRODUCTION

There are two parts to MIT App Inventor: The 'Designer' editor and the 'Blocks' editor.

The Designer editor is used to add all the interface elements such as Buttons, labels, lists, and non-visible components such as Interval Timers, Bluetooth Connectivity, and Speech Recognition support. Components are dragged from a list on the left on to the Viewer.

Each component can be individually edited on the right hand side to customize attributes such as Name, Colour, Dimensions, etc. Table layouts are used to organize elements easily. Take for example the Forward/Backward/Left/Right Control buttons. They're each placed in a cell in a 3x3 table.

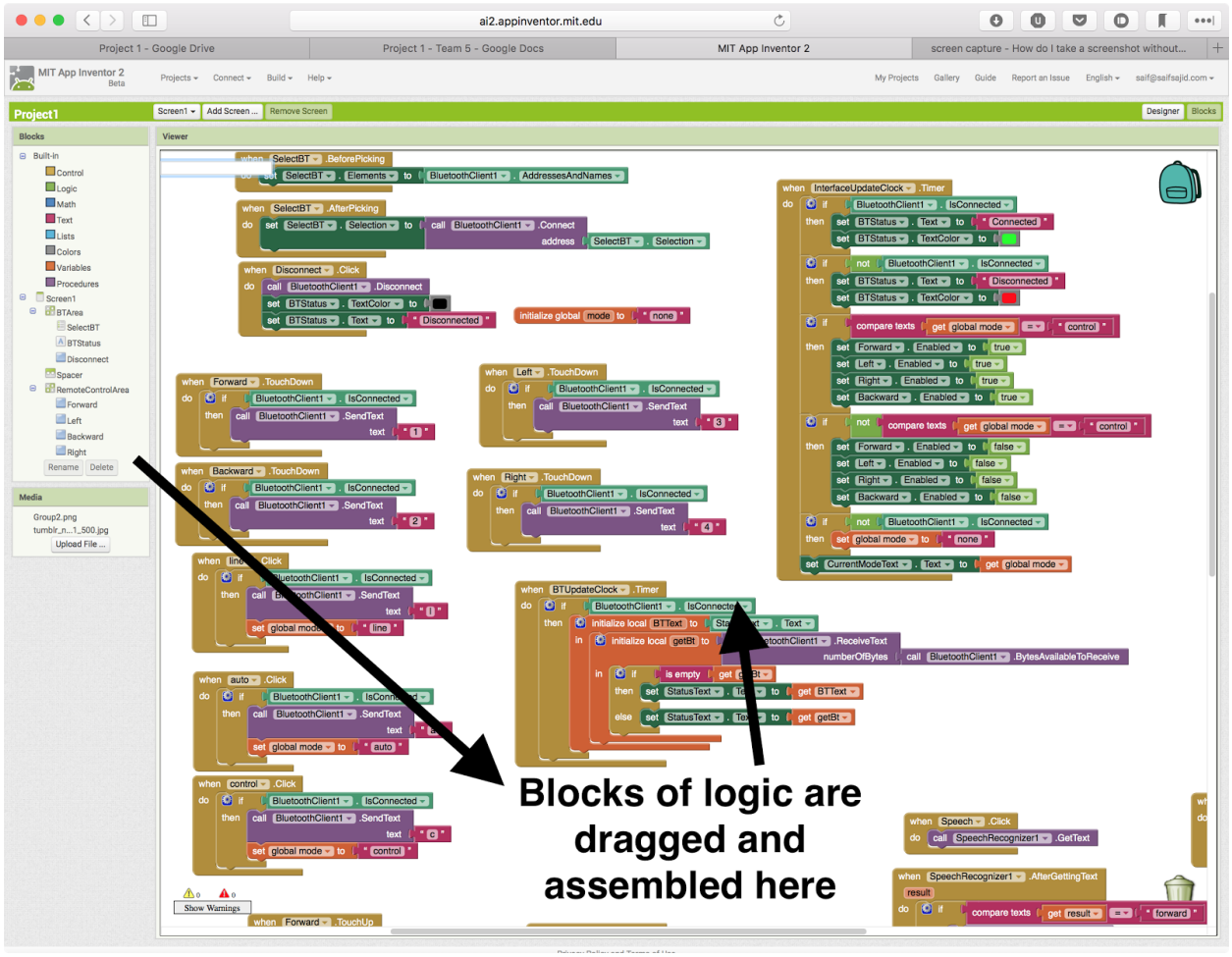



*The 'Designer' Editor*

The Blocks editor is used to define the logic behind the app itself. It uses the same design as MIT Scratch. All the components of a regular program: While loops, if/else blocks, etc. are all treated as

connectable blocks. The only difference between regular programming and this style is that instead of typing everything out, it's all drag and dropping, visually connecting Procedures, variables, and tests.

It also abstracts away the system level API details and does much of the scaffolding for you, leaving only the essential logic up to you. It's analogous to the Arduino platform, where much of the microcontroller level details are abstracted away and you simply interact with the pins and do some basic logic. This appendix will further go into detail about the 'source code' behind the app.

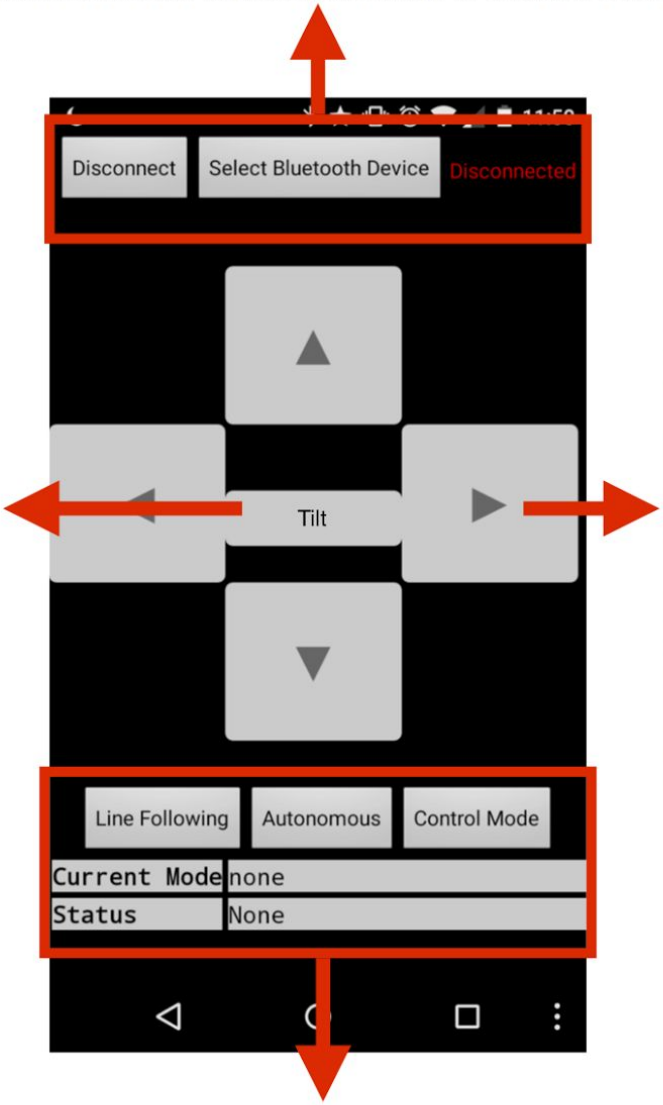



*The Blocks Editor*

## A-D2. USER INTERFACE

**Bluetooth Connectivity**

The *Disconnect* button disconnects from the connected bluetooth module. The *Select Bluetooth Device* button opens a list of paired and available devices. This is where you can select the HC-06 bluetooth module used and connect to it. (Note: Pairing must be done beforehand in Android's Bluetooth Settings. The pairing code is 1234 by default). The last component is the *Disconnected/Connected* label that updates based on connection status.




**Tilt Button**

When the tilt button is held down, the app starts recording the pitch and roll angles of the orientation of the phone. This is calculated by internal software of the phone using its gyroscope and accelerometer. Using these angles we send commands to the robot to move in the direction of the tilt (Forward/ backward/left/right). Also, when shaking is detected, the robot does a little wiggle

**Control Buttons**

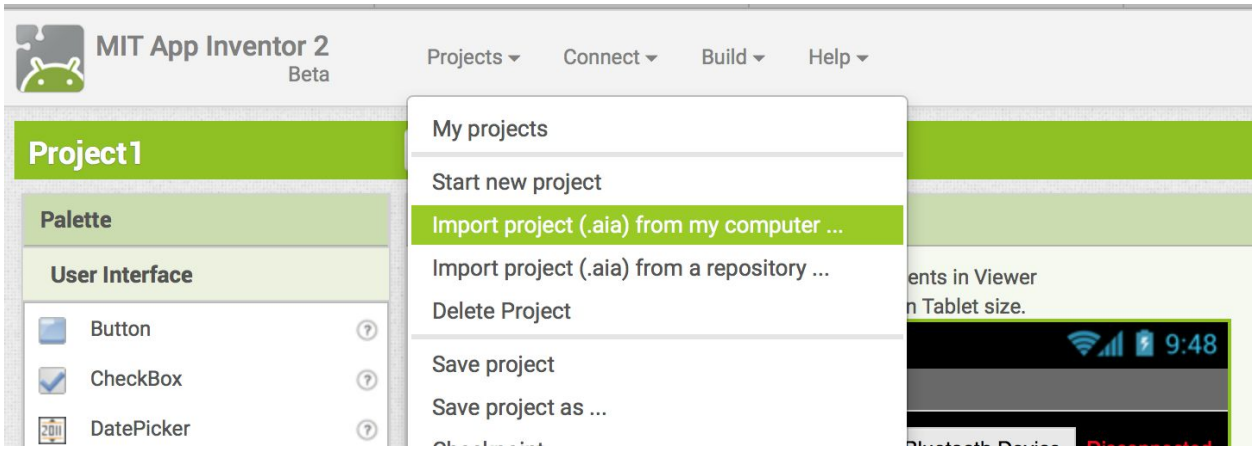
The control buttons are simply buttons with Unicode characters corresponding to their directions. Their large size was designed with ease of use in mind, as it is easier to find and tap a large button rather than a smaller one. Another usability feature is that the buttons are greyed out and disabled when the robot is not in 'Control Mode'. This serves as visual feedback of the current mode and prevents unintentional commands from going through.

**Mode and Status**

The three buttons change the mode of the robot. They only send a command if a bluetooth device is connected, to prevent error dialogs. The '*Current Mode*' label is set by the app. By default, it is 'none'. The '*Status*' label is where the Robot/Arduino can send text to the phone. This is a usability feature where one doesn't need to read the LCD to know what's going on with the robot and allows for complete remote functionality

**A-D3. SOFTWARE LOGIC**

You can explore the logic/source code in MIT App Inventor using the uploaded 'Project1.aia' file. Simply go to the website: http://appinventor.mit.edu/explore/, login using your Google Account, start a new project. On the upper menu, click 'Projects' and then 'Import Project from Computer' and upload the aia file:

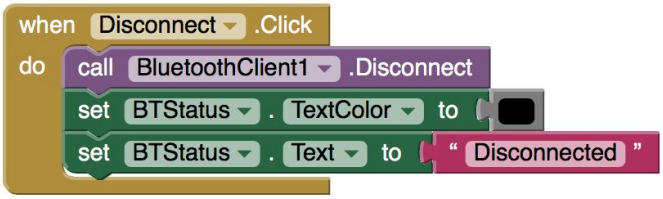



We've included all the source code behind the app as screenshots with comments below:

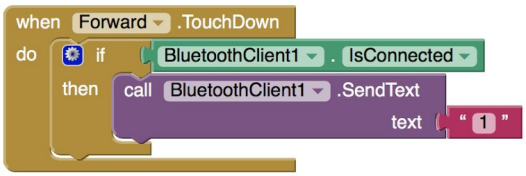
**Populate 'Select Bluetooth Device' list with previously paired devices**

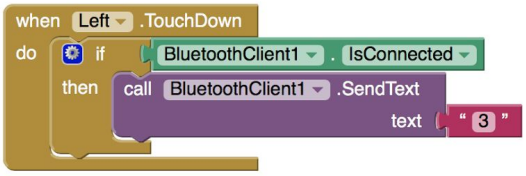

**Once a selection has been made, connect to the selected device**

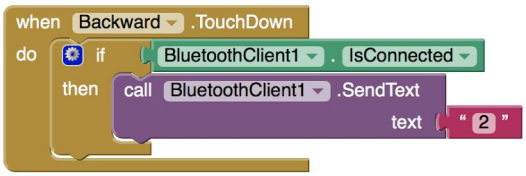

**If Disconnect button is pressed, disconnect device and update Disconnect label**

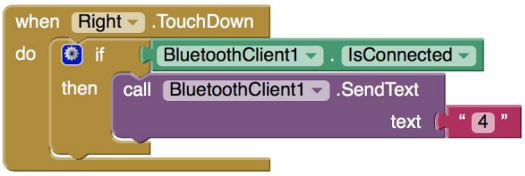


**If any of the control buttons (Forward/Backward/Left/Right) are pressed, then send a corresponding character through Bluetooth. The robot will go in direction continuously until a halt command is received when the button is let go**

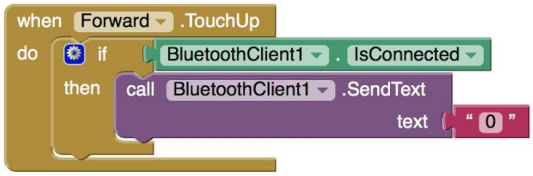


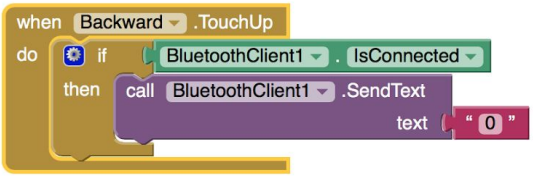



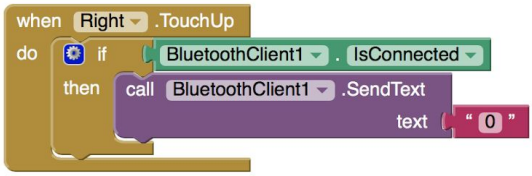



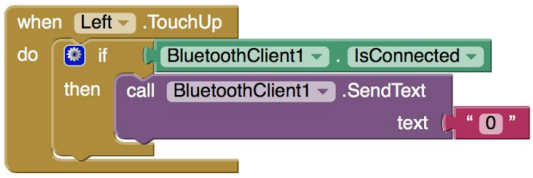




**Once any of the buttons is let go, send a character that corresponds to the 'Halt' command to stop the robot**

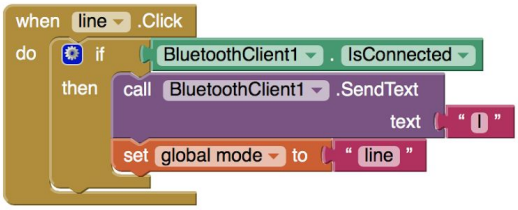


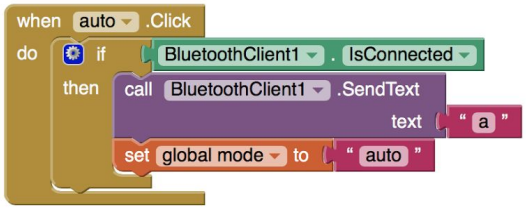



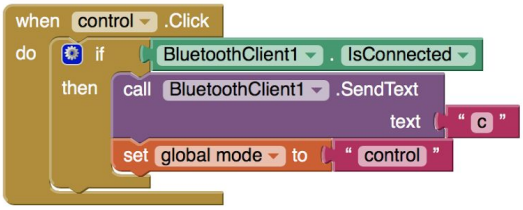




**Initialize global 'mode' var to 'none'. If any of the mode changing buttons is pressed, then send a corresponding character through bluetooth to change mode. Also update the 'mode' var to the changed mode**

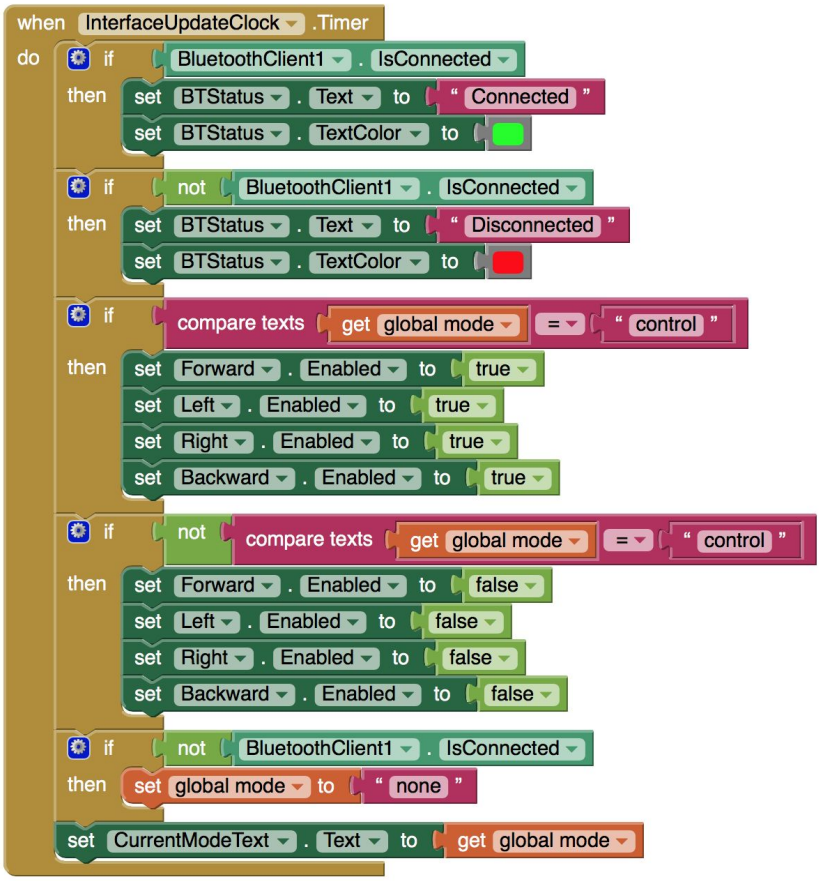




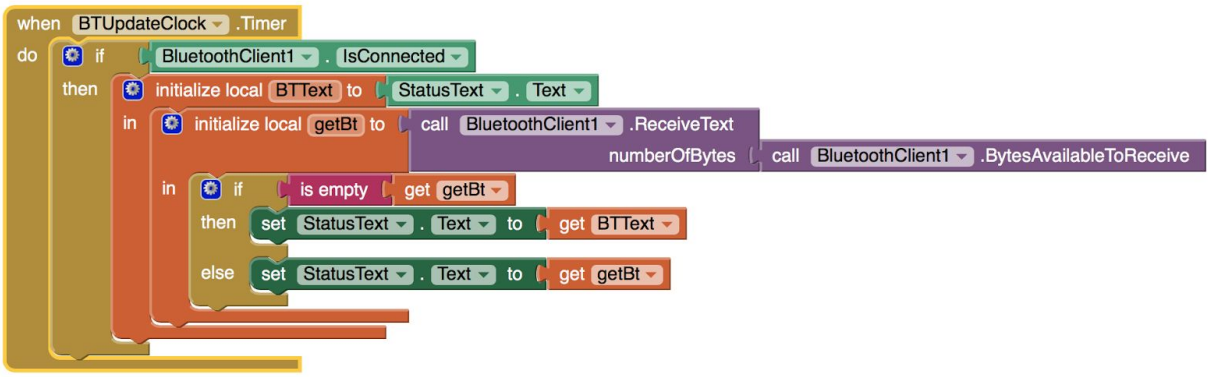


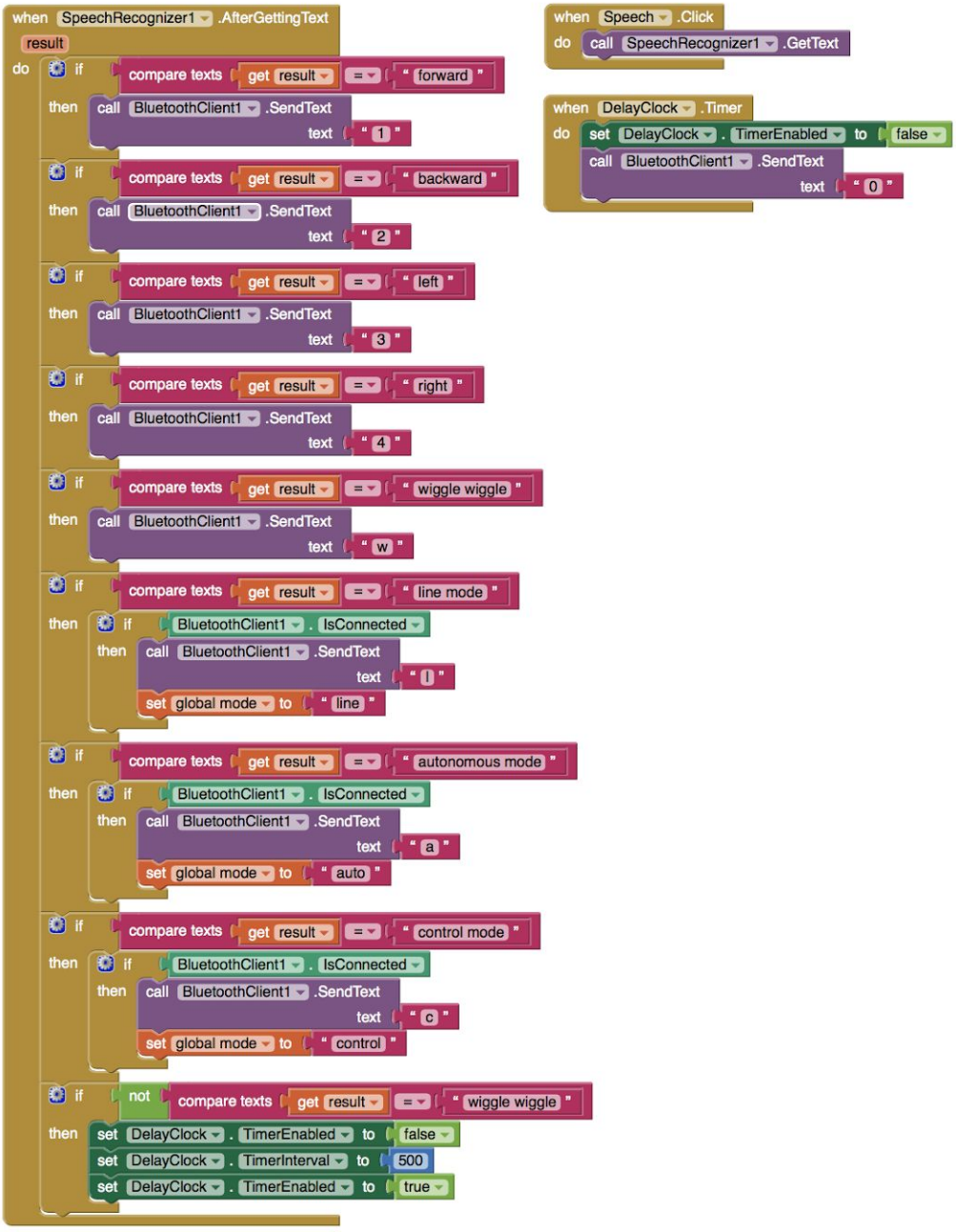
**Whenever the InterfaceUpdateClock fires (1000ms), update the BT connection status, current mode, disable/enable the control buttons based on mode**




**Every 300ms, check for Bluetooth data from Arduino, if none, show the same status text as before, otherwise, update the 'Status' text**



**When the Speech button is clicked, the Google Voice dialog opens and converts speech to text. After getting the text a few 'if' blocks compare the text. For the regular control commands, it sends the corresponding control character over bluetooth and starts a 500ms timer after which it sends a halt command. For the other commands it just sends the corresponding character**




This functionality was removed in the final version to comply with the 'no internet' rule

**When the tilt button is pressed down, the tilt boolean is true, false when let go. When the tilt boolean is true, orientation data is recorded. Commands are sent over bluetooth to move in the direction of the tilt of the phone, once it turns a certain threshold angle. A halt command is sent when the phone is level. When the phone shakes, a wiggle command is sent, and the robot wiggles.**