IDC Final Design Report

ECE 110L-08 Fundamentals of Electrical and Computer Engineering

Spring 2019

Chaser Group 2

Rachel VanRyzin and Isabella Knox

Contents

Part I

Report

1 Abstract (201 words)

The Integrated Design Challenge is comprised of several tasks that needed to be completed by each individual bot within the lab section. Each bot's task requires the utilization of line following, communication, and different sensing capabilities. Successful sensing of the bot's objects and communication between bots would result in the accurate calculation and display of the score acquired between the bot pair and the lab section according to the set up of objects in the IDC arena. Throughout the IDC, several different sensors were used to complete the tasks given to each bot. Three QTI sensors placed at the bottom of each bot were used for line following. For communication between bots, an XBee module was used. The specific sensing technique differed between bots depending on the object it was given to sense. For Chaser Bot 2, the sensing task assigned was to detect the number of large, red quaffles placed at each hash along its line. A whisker wire was used to sense these quaffles. During the final demonstration, all five trials were perfectly executed. The individual bot, Chaser-Beater pair, and final team scores were all displayed accurately on their own LCD display after each bot perfectly sensed their objects.

2 Introduction

For this semester's Integrated Design Challenge, all the bots within a lab section worked together to play the ECE110L version of robot Quidditch. The team was comprised of five autonomous robots. A half-pitch Quidditch field was used, along with three goal posts at one end of the field and five black lines, one for each robot to go down. In this game, each bot is either a Chaser, a Beater, or the Seeker. The Chaser goes down its line detecting the "quaffles" present at each hash. Each detected quaffle is worth 10 points; however, if a bludger hits a Chaser in possession of a quaffle, the Chaser misses the hoop. This results in no added points. The Chaser is only able to score if its partnered Beater blocks any bludgers successfully. The Seeker is on the field in order to find the "golden snitch". If the golden snitch is caught by the Seeker, 150 points are added to the team score. No outside interference with bots is allowed once the trial has begun without adding a "foul".

The goal of the project is to produce a Quidditch team of five robots which will sense their objects and communicate with one another to determine and display the score that the team would have scored from quaffle goals and possibly catching the golden snitch. To achieve this goal, the Chasers must detect ramps or quaffles, and communicate with their paired Beater, which must sense RFID tags or red and green signs, respectively. The Chasers and Beaters must also communicate their scores to the Seeker bot, which detects the number of mirrors on its line and drives to the corresponding line in an attempt to catch the snitch (150 additional points). All bots must be able to sense and follow lines so that they can stay in their lanes, as well as communicate with every other bot so that each bot can display the team's final score. Specifically, Chaser 2's bot must sense the presence of red quaffles and communicate with Beater 2 (which senses red and green signs) to determine whether or not the team would score 10 points at each of five hash marks. Each bot's task (sensing and score computation) is important for the team's overall ability to calculate the correct score displayed on the mat.

To solve this challenge, Chaser Bot 2 was wired to utilize three QTI sensors, a whisker wire, an Xbee module, and an LCD display. Along with coding skills, circuitry abilities were extensively developed. As a team, troubleshooting and problem-solving skills were honed throughout every week of the IDC as problems with coding, bot parts, and bot communication occurred. The need for all bots to work together to complete and calculate the final score also fostered teamwork, communication, and reliability within the lab section.

In order to fulfill the problems presented in the IDC challenge, the Chaser 2 bot needed to achieve a series of steps/tasks. First, Chaser 2 had to closely follow its assigned black line using its QTI sensors. The bot stopped and sensed whether a quaffle was present at each hash using a whisker wire. The Chaser bot then stopped at the end of its line and relayed an array containing information on which quaffles were sensed and at what location to Beater 2. Beater 2 computed the team score and send it both to the Seeker bot and back to Chaser Bot 2, which was displayed as the "paired score". The Seeker bot then computed the final score of the lab section after it had found (or not found) the snitch and received both scores from the two Chaser-Beater pairs. This final score was sent out from the Seeker bot to every other bot to display on their own LCD screen. Chaser 2 displayed the number of quaffles found, the paired score from the Beater, and the final team score at the end of each trial.

3 Experimental Procedure and Results

In order to create a Chaser bot that would successfully sense quaffles and lead Quidditch Team 8 to victory, several considerations were made. First and foremost, the Chaser needed to be accurate with respect to line following, communication, and quaffle sensing. This triad was crucial to the correct calculation of an individual score, a paired score with the Beater 2 bot, and ultimately, the calculation of a team-wide score. Additionally, the bot needed to perform its tasks in a reasonable time frame, meaning that the code and quaffle-sensing technique (whisker wires) had to be efficient. To achieve these goals, each week a specific task associated with the desired final function of the bot was tackled, and ultimately integrated into one cohesive code. Likewise, the code was revised and re-structured as new tasks were added in order to ensure that the code was well organized and easy to troubleshoot.

Week 1: Xbee Communication

In order to ensure the success of a five-bot quidditch team, one of the key tasks was to ensure that each bot could communicate with the rest of the team. To achieve this goal, an Xbee (see Figure 1) was implemented, which allowed for wireless communication between the bots, and with sentry monitor.

To begin the process of generating communication between the bots, the goal for this week was to successfully send a single character via Xbee, which could be displayed on the sentry monitor, as well as displayed in the serial monitor of each of the other bots. Thus, Serial 2 was chosen, as it runs at a frequency (9600 bits/s) that can be read by all of the bots. First, a wireless Xbee communicator was added to pins Rx16 and Tx17 on the breadboard of the bot. This allowed the bot to both send and receive communication. Then, two colored LEDs were added, one to indicate that the bot was receiving communication, and one to show that it was successfully sending out characters (Figure 7). A green LED was chosen to represent that the bot was sending out a character, while a red LED to indicate that the bot was receiving communication. To trigger a character to be sent, a push-button was added, which, when pressed, completed a circuit and from 5V to ground. Pin 10 was used to monitor the state of this circuit and functioned as an input to trigger a character to be printed to the serial monitor, as well as for the "sending" LED to be turned on (See Figure 6). The setup code for Xbee communication is shown below:

```
#define Rx 17 //DOUT to pin 17
#define Tx 16 //DIN to pin 16
const int button = 10; //ties the pushbutton to pin 10
const int transmitting = 8; // ties pin 8 to transmission Green LED (talking!)
const int receiving = 6; // ties pin 6 to reception Red LED(listening!)

void setup() {
  // put your setup code here, to run once:
  pinMode(button, INPUT); //sets up the pushbutton as input
  pinMode(transmitting, OUTPUT); //sets up LED as output for transmitting
  pinMode(receiving, OUTPUT); //sets up LED as output for receiving
  Serial.begin(9600); // sets Serial (our bot) to 9600 bits/sec
  Serial2.begin(9600); //sets Serial.2 to 9600 bits/sec
  delay(500); //delay half a sec
  Serial.print('q'); //prints "q" to make sure setup code happened
}
```

While the pushbutton allowed for the time in which the bot was sending out a communication to be discreetly chosen, the bot needed to be prepared to receive communication at all times. Thus, the code called for continuous checking for available data from Serial 2. If a character was read, the code turned the red "receiving" LED on, and printed the character to the serial monitor. While communication could have been proven using the serial monitors of each bot alone, the LEDs provided an easy way of telling if one or more bots were not properly receiving communication. Additionally, different characters were chosen for each bot on the team, allowing for an easy way of determining whose bots were successfully communicating and receiving information. The code associated with using the push button to trigger communication, as well as printing any characters that are received, can be seen below:

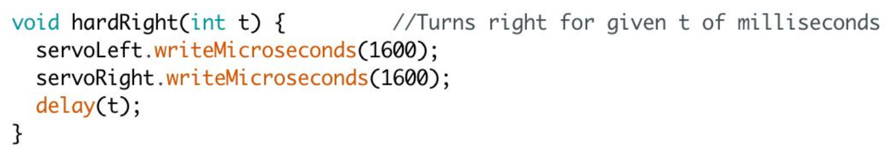
```
int buttonState = digitalRead(button); //reads if the button is pushed, pushed is HIGH
if(buttonState == HIGH){ //if the button is pushed
  //if(Serial.available()>0){
     digitalWrite(transmitting, HIGH); //turn the transmitting LED on
     Serial2.print('m'); //send out a 't' to be printed by other people
     delay(500); //wait half a sec
     digitalWrite(transmitting, LOW); //turn the transmitting LED off
 // }
}

if(Serial2.available()>0){  //if there is infor someone is trying to send to our bot
  digitalWrite(receiving,HIGH); //turn the recieving LED on
  Serial.print(char(Serial2.read())); //have our bot's moniter print out whatever character serial.2 is trying to tell us
  delay(500); //wait half a sec
  digitalWrite(receiving,LOW); // turn the receiving LED off
}
```

During the demonstration of communication, each bot (including Chaser 2) successfully communicated its character to every other member of the team, as indicated by the LEDs. A perfect score was obtained for this demonstration.

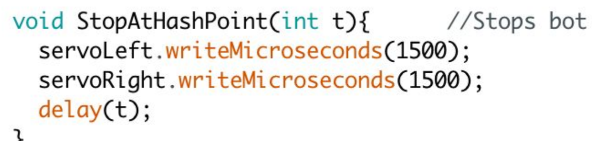
Week 2: Line Following

To complete the task of sensing quaffles, the bot needed to be able to smoothly navigate along a line and stop at hash marks, where eventually the quaffles would be placed. To detect where the lines on the field were drawn, three QTI sensors were implemented (See Figure 2). These sensors were chosen because of the stark contrast between the black line against the white background of the field. The QTI sensors detect the difference between black and white by using the reflected light from the object at which they are pointed to charge a capacitor. Then, the decay time to discharge the capacitor is measured and stored. This value was then compared to a threshold value, which was found via trial and error to be 280. Above this threshold, the ground beneath the QTI was determined to be black, i.e. a line. Below the threshold, the field was considered white, indicating that the QTI was not over a line.

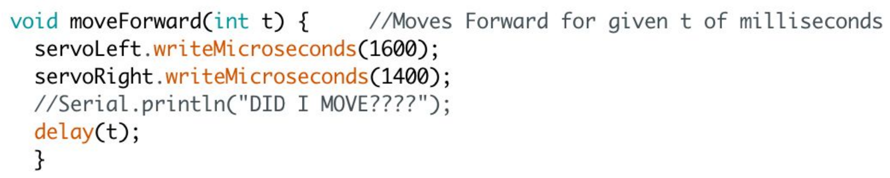
To integrate the QTIs into the bot, three QTI sensors were mounted to the underside of the chassis with standoffs and connected to pins 49, 51, and 53 using three pin cables (see Figures 2 and 6). Using the standoffs allowed the QTI sensors to come very close to the surface of the field, making accurate sensing much easier. Then, code was written to take the decay time for each of the QTIs and compare it to the established threshold value to determine whether that QTI sensed black or white. Functions were also written to tell the bot how to respond to the determined state of the QTI's. If the center QTI read black, while the left and right QTIs read white, the bot was centered on the line, and the "moveForward" function was called to write the left and right servos to 1600 and 1400, respectively. If the left QTI read black while the other two read white, a function was called to write the servo values such that the bot would turn right. These same techniques were applied to write instructions for seven of the eight possible combinations of servo readings. Notably, a reading of three blacks indicated that the bot had found a hash, and called for the bot to stop, while a reading of three white indicated that the bot had lost the line completely, and called for it to reverse in an attempt to relocate the line. Examples of several the line following functions can be seen below, and the full line following code can be found in the appendix.



```
void hardRight(int t) {          //Turns right for given t of milliseconds
  servoLeft.writeMicroseconds(1600);
  servoRight.writeMicroseconds(1600);
  delay(t);
}
```

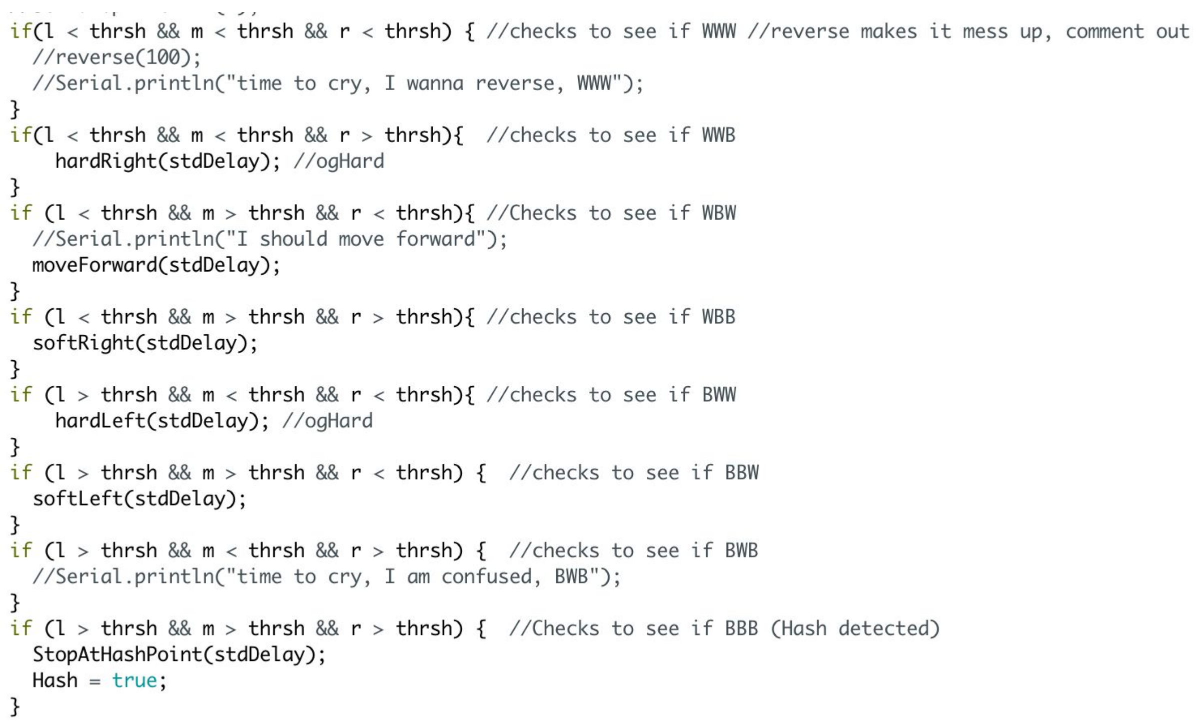


```
void StopAtHashPoint(int t){      //Stops bot
  servoLeft.writeMicroseconds(1500);
  servoRight.writeMicroseconds(1500);
  delay(t);
}
```



```
void moveForward(int t) {     //Moves Forward for given t of milliseconds
  servoLeft.writeMicroseconds(1600);
  servoRight.writeMicroseconds(1400);
  //Serial.println("DID I MOVE????");
  delay(t);
  }
```

The eighth possible case of QTI readings (black, white, black) was left out because the only possible way of obtaining this reading was if the bot had backed itself into a 90^{o} corner. Due to the fact that the bot actually encountering this scenario was less likely than the possibility of shadows on the board causing the bot to erroneously read this QTI pattern, this particular permutation was emitted from the code. The if-tree that calls the line following functions can be seen below:



```
if(l < thrsh && m < thrsh && r < thrsh) { //checks to see if WWW //reverse makes it mess up, comment out
  //reverse(100);
  //Serial.println("time to cry, I wanna reverse, WWW");
}
if(l < thrsh && m < thrsh && r > thrsh){  //checks to see if WWB
    hardRight(stdDelay); //ogHard
}
if (l < thrsh && m > thrsh && r < thrsh){ //Checks to see if WBW
  //Serial.println("I should move forward");
  moveForward(stdDelay);
}
if (l < thrsh && m > thrsh && r > thrsh){ //checks to see if WBB
  softRight(stdDelay);
}
if (l > thrsh && m < thrsh && r < thrsh){ //checks to see if BWW
    hardLeft(stdDelay); //ogHard
}
if (l > thrsh && m > thrsh && r < thrsh) {  //checks to see if BBW
  softLeft(stdDelay);
}
if (l > thrsh && m < thrsh && r > thrsh) {  //checks to see if BWB
  //Serial.println("time to cry, I am confused, BWB");
}
if (l > thrsh && m > thrsh && r > thrsh) {  //Checks to see if BBB (Hash detected)
  StopAtHashPoint(stdDelay);
  Hash = true;
}
```

In implementing the line following code, several issues needed to be troubleshooted. First and foremost, the threshold value for the QTIs had to be calibrated so that the bot could properly distinguish between black and white. This was achieved by printing the values read by each QTI to the serial monitor, and then observing the values as the bot was placed over different colored portions of the field. Additionally, the bot seemed to have trouble with the "reverse" function, which was likely erroneously called when external light sources interfered with the QTI, causing it to read white-white-white. Because this function was created as a last resort, to be called only when the bot had completely wandered off of the line, it was removed from the code entirely to avoid seizure-like spasms that occurred when the function was accidentally called.

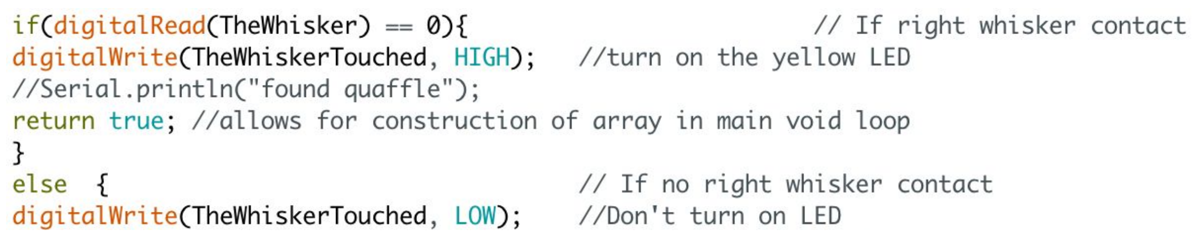
In addition to coding issues, several hardware issues had to be troubleshooted. In particular, calibrating the servos proved particularly challenging. Using code designed to set the rotation of each servo to zero, full stop was calibrated using a screwdriver to tweak the servos to be motionless. However, when the function written to stop the servos at hash marks was called, which set the servo rotations to be identical to the values used to calibrate it to a full stop, the bot was observed to slowly turn, due to the right wheel slowly drifting backward. Because there was no code written that ever called for this behavior, a proposed reason for the drift was charge buildup within the servo causing the residual rotation. Likewise, the bot tended to drift to the right while calling the "moveForward" function, further indicating a calibration issue with the servos. Because the bot had a tendency to erroneously turn in the same direction (right), another proposed issue was a malfunction of the QTIs. However, when replacing the QTIs did not solve the issue, it became clear that the servos were the source of the problem. In an attempt to overcome the calibration issues, the line following code was altered so that the rotation of the right wheel would be slower than it should be if the servo were calibrated correctly. This temporarily solved the issue.

During the line following demonstration, despite changes to the code to account for the right servo rotating too quickly, the servo fell back out of calibration in the time between calibration and the demonstration. As a result, the bot unfortunately displayed the same right wheel rollback issue as it did before the right servo was calibrated. The demonstration included following a line in a circle, around a square, on a squiggle, and on a line with three hashes. The bot managed to accomplish all of these tasks successfully, if somewhat jerkily, except the three hash line. Moving forward, it became clear that more work would have to be done to solve the calibration issues.

Week 3: Line Following and Quaffle Sensing

In addition to following lines and stopping at hash marks, the bot needed to be able to sense if quaffles were present at each of the hashes. In the Conceptual Design Report, two methods of quaffle sensing seemed worthy of investigation: whisker wires and IR sensing. Because using IR sensing involved precisely aiming the IR sensor and receiver at the quaffle, this option was quickly ruled out. Since the bot was encountering so many issues with line following, achieving the correct angle to properly sense quaffles seemed unreliable at best, and impossible at worst. Likewise, as the bot was handled and manipulated, it seemed likely that IR sensors mounted on nothing more than wires would become deflected, compromising the sensing capabilities of the bot. Thus, whisker wires were chosen to detect the quaffles.

To install the whisker wire, a one inch standoff was used to mount the whisker wire to the front right corner of the board. A three pin header was attached to the breadboard such that when the wire was deflected, it would make contact with the three pin header. A series circuit was constructed from the 5V input, through a 10kΩ resistor, the node containing the three pin header, and a 220Ω resistor, and to input pin 7 (See Figures 3 and 4). Thus, when the wire was not deflected (no quaffles), input pin 7 registered high. However, when the whisker was deflected (a quaffle was detected), an alternate circuit of less resistance was completed, with current running through the three pin header and to the grounded whisker instead of through the 220Ω resistor and the digital input pin. Thus, the input pin detects registers low. Using an if-tree depending on the state of pin 7, a yellow LED was added to the bot that illuminated when a quaffle was detected (See Figure 7). This allowed for a visual representation of the data that the bot was receiving regarding the presence of a quaffle. Due to the relatively simple sensing technique, no major issues were encountered in the wiring or the coding for quaffle sensing with whisker wires. The code used to illuminate the LED can be seen below:



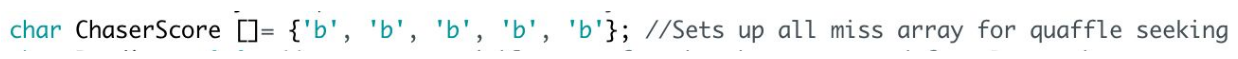
```
if(digitalRead(TheWhisker) == 0){                        // If right whisker contact
digitalWrite(TheWhiskerTouched, HIGH);   //turn on the yellow LED
//Serial.println("found quaffle");
return true; //allows for construction of array in main void loop
}
else  {                                  // If no right whisker contact
digitalWrite(TheWhiskerTouched, LOW);    //Don't turn on LED
```

While the whisker wire sensing was simple and effective, it also posed a slight problem: the bot will continuously sense a quaffle for as long as the wire is depressed, which leads to repeated readings of the same quaffle. To combat this problem, it was proposed that the bot only sense for the presence of a quaffle once, when it was stopped at a hash. While integrating line following and sensing together was not a requirement in this week's tasks, this sensing strategy highlighted the importance of achieving reliable line following. To improve upon the line following of the previous week, the QTI sensors, which were originally oriented with the sensor forward, were rotated 180^{o} so that they faced the rear of the bot. This change allowed for better sensing because originally, the point about which the robot pivoted was on the line through the sensors, meaning that even a small rotation could drastically change the readings of the QTIs. Flipping the QTIs geometrically situated the pivot point off of the line of sensors, which made line following less jerky. Additionally, the speed at which the bot performed turns was decreased so that more subtle turns could be achieved. Finally, in an effort to prevent the aforementioned charge buildup between trials, the servos were attached, detached, and reattached in the void setup of the code. Together, these changes were enough to allow the bot to follow the line and stop at hash marks relatively fluidly.

The demonstration for Week 3 required that the bot be able to follow lines and detect quaffles, although not simultaneously. The bot successfully achieved both of these goals, and a perfect demonstration score was achieved.

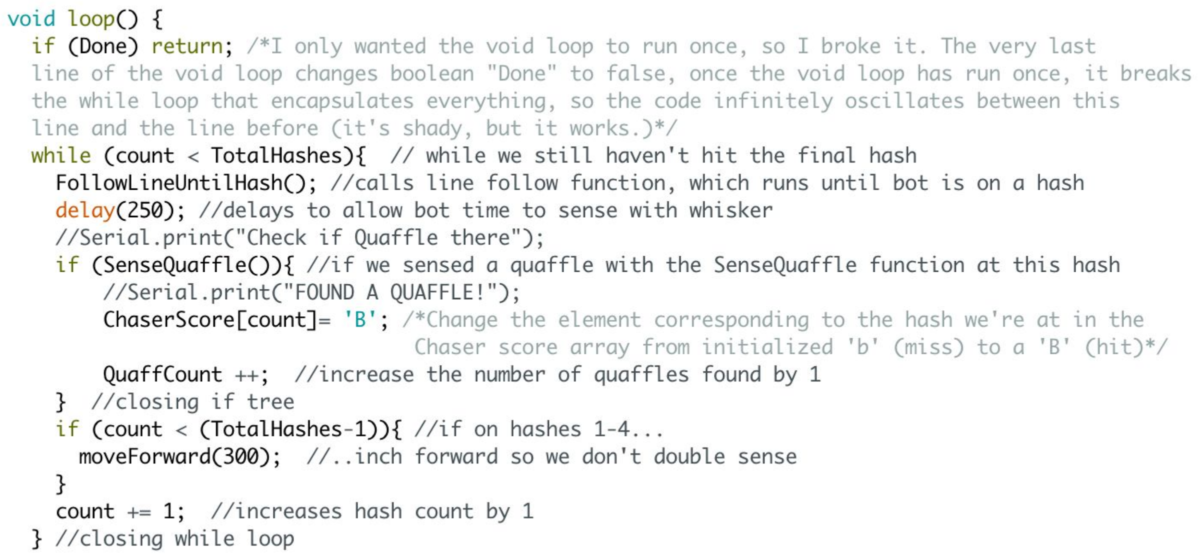
Week 4: Integrated Sensing, Processing, and Navigation

To begin preparing the bot to compete in the final IDC demonstration, the formerly achieved goals of line following, quaffle sensing, and communication needed to be integrated into one code. While this seems like a trivial "copy/paste" ordeal, in reality, creating a functional integrated code proved to be much more challenging. The bot needed to travel down the line, stop at each hash mark, sense for a quaffle, and communicate to the sentry bot at which hashes quaffles were detected. With the eventual goal of being able to communicate with the Beater 2 bot where quaffles were detected, it was decided that information about where quaffles were found should be stored in an array of five elements, called “ChaserScore”. This array was initialized as five lowercase “b”s, and when a quaffle was found, the element corresponding to the hash at which the quaffle was sensed was changed to an uppercase “B.” Using this method, when it came time to communicate with the Beater 2 bot, the bots would not have to wait at each hash until both bots had arrived, making the overall trial time faster. Likewise, if one bot malfunctioned, the other bot could still complete its line and display its individual findings. The initialization of the ChaserScore array is shown below:



```
char ChaserScore []= {'b', 'b', 'b', 'b', 'b'}; //Sets up all miss array for quaffle seeking
```

While no new wiring was added to the bot, the code was drastically changed and re-organized. The primary issue with integration was that while sensing and communication worked beautifully, line following once again broke down when it was rolled into the rest of the code. To make the code easier to troubleshoot, the line following functions were transferred from the main void loop into a large function called “FollowLineUntilHash.” Then, quaffle sensing code was moved to a function called “SenseQuaffle.” The main void loop called the FollowLineUntilHash function first, which made the bot follow the line until it hit a hash. At this point, the “SenseQuaffle” function was called, which allowed the bot to detect if the hash had a quaffle. The presence or absence of a quaffle was recorded in the ChaserScore array. Finally, the ChaserScoreArray was printed to the sentry by using a for loop to print each element of the array. The code below shows how the FollowLineUnitilHash function and the SenseQuaffle function are called in the main void loop, as well as how the ChaserScore array gets modified:



```
void loop() {
  if (Done) return; /*I only wanted the void loop to run once, so I broke it. The very last
  line of the void loop changes boolean "Done" to false, once the void loop has run once, it breaks
  the while loop that encapsulates everything, so the code infinitely oscillates between this
  line and the line before (it's shady, but it works.)*/
  while (count < TotalHashes){  // while we still haven't hit the final hash
    FollowLineUntilHash(); //calls line follow function, which runs until bot is on a hash
    delay(250); //delays to allow bot time to sense with whisker
    //Serial.print("Check if Quaffle there");
    if (SenseQuaffle()){ //if we sensed a quaffle with the SenseQuaffle function at this hash
        //Serial.print("FOUND A QUAFFLE!");
        ChaserScore[count]= 'B'; /*Change the element corresponding to the hash we're at in the
                                  Chaser score array from initialized 'b' (miss) to a 'B' (hit)*/
        QuaffCount ++;  //increase the number of quaffles found by 1
    }  //closing if tree
    if (count < (TotalHashes-1)){ //if on hashes 1-4...
      moveForward(300);  //..inch forward so we don't double sense
    }
    count += 1;  //increases hash count by 1
  } //closing while loop
```

To address the recurring problem of jerky line following, several of the line following functions were altered. Namely, a distinction was made between hard and soft turns. Previously, QTI readings of black-white-white and black-black-white were both treated the same, as a left turn about a pivot point in the center of the bot (i.e., the right wheel moved forward while the left wheel moved back). This logic forced the bot to make relatively harsh corrections, regardless if it was slightly drifting or about to completely veer off of the line. To make the line following more sensitive, the black-black-white case was altered to call a new function, which turned left softly by stopping the left servo, and letting the right wheel slowly move forward. Thus, the pivot point was changed to be the left wheel, and the bot stopped overcorrecting when just a minor adjustment was needed to keep it on the line. Additionally, by creating distinct functions for each task, the line following code loops until the bot is at a hash, meaning that while the bot is in motion, only one small portion of the code is running. This cuts out parsing time that may cause too long to pass between QTI readings, leading to jerky, spastic line following. The code below demonstrates the difference between what are considered hard and soft turns:

```
void hardLeft(int t) {        //Turns left for given t of milliseconds
  servoLeft.writeMicroseconds(1400);
  servoRight.writeMicroseconds(1400);
  delay(t);
}
void softLeft(int t) {        //Turns left for given t of milliseconds
  servoLeft.writeMicroseconds(1500);
  servoRight.writeMicroseconds(1400);
  delay(t);
}
```

In addition to changing the code, one small but crucial adjustment was made to the bot. On the whisker wire, a small piece of additional wire was added. This additional piece was shaped so that the whisker would deflect back into the three-pin-header, but then skim up and over the quaffle. The skimming motion was extremely important to the success of our bot and out team: by skimming over the top, the risk of knocking the quaffle over and having it roll into another bot was nullified. More importantly, the skimming motion prevented the whisker from deflecting so much that it oscillated hard enough to hit the three pin header (which could cause a faulty reading at an empty hash following a hash with a quaffle) (See Figure 6).

After the code was reorganized, serial monitor print statements were used throughout the code to make sure each function was being called at the appropriate time. Using those statements to troubleshoot, the code was successfully debugged. In the demonstration, the bot followed the line, stopped at each hash, sensed for a quaffle and lit up a yellow LED if it found a quaffle. At the fifth hash, the bot printed an array of upper and lowercase b's to the sentry bot, indicating which hashes had quaffles. The bot perfectly executed these tasks in all five trials, and a perfect demonstration score was achieved. The code below shows the for loop used to print the ChaserScore array:

```
int element; //Names a variable for the counter of the for loop that prints the array
for (element = 0; element <=4; element++){ //forloop, count up by 1 from 0 to 4 (array index starts at 0)
//Serial.println(ChaserScore[element]);
Serial2.print(ChaserScore[element]);  //print to serial2 personal score array
```

Week 5: Team Sensing, Processing, and Navigation

In Week 5, the Chaser and Beater bots had to work together, deciding at which hashes the Chaser had found a quaffle and the Beater had successfully blocked a bludger, allowing the pair of them to score 10 points. Additionally, the paired score had to be displayed on an LCD screen.

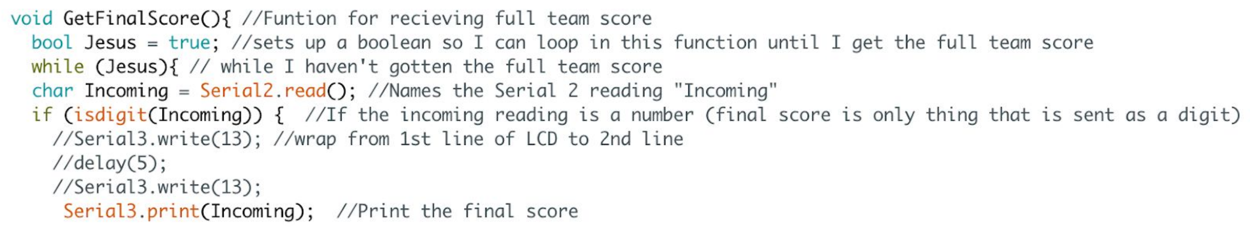
With four bots trying to communicate at the same time, working as a team was crucial to ensure that no erroneous Xbee readings. It was established that just Chaser 2 and Beater 2 would use the characters 'Z,' 'B,' 'b,' 'u,' 'v,' 'w,' 'x,' 'y,' and 'z.' To calculate a paired score, the idea was to wait until both bots had reached the end of the line, at which point Beater 2 would send a 'Z' to indicate that it was ready to receive. The Chaser 2 bot would wait until it received the 'Z,' at which point, it would use a for loop to print its array of upper and lowercase b's. The Beater bot recorded and stored the array from the Chaser, compared the Chaser's array to its own, and generated a score based upon the number of hashes at which the Chaser and Beater found a quaffle and blocked the bludger. This score was then communicated back to the Chaser, who selectively listened for characters 'u,' 'v,' 'w,' 'x,' 'y,' or 'z.' These characters were then translated back to a paired score (0, 10, 20, 30 , 40, or 50 points, respectively) using an if tree. By communicating the scores through letters rather than numbers, the paired score could be sent with just one character. This prevented the confusion that would ensue if Chaser 1, Chaser 2, Beater 1, and Beater 2 were all trying to send digits at the same time. Because each pairing has its own set of letters it is permitted to use, there is no chance of interpreting information meant for the other pair of bots.

In terms of coding, to communicate with Beater 2, relatively few changes had to be made. Instead of immediately printing the score array to Serial 2, the bot waits until the Beater 2 bot is at the end of the line and sends the ready ('Z') signal. Additionally, code was added that scans Serial 2 until the bot reads a letter 'u' though 'z,' which is translated into a decimal score. The code for this communication can be seen in the "CommunicateWithBeater" function in the code in the Appendix.

To display the pair score, code was added to start Serial 3 for the LCD screen, as well as access the LCD library. Serial3.print statements were added to print the paired score to the LCD. However, due to the fact that the LCD screen originally provided was broken, the paired score was also printed to the Serial monitor to prove that it had been properly received. For the demonstration, both bots successfully sensed quaffles and bludgers and communicated to generate a paired score. A perfect demonstration score was achieved.

Week 6: Final IDC

For the final IDC, the paired communication of Week 5 was expanded so that all five bots could communicate to generate a final team score. To do so, Beaters 1 and 2 both calculated a paired score. When the Seeker bot detected whether or not it had found the snitch, it sent a ‘p’ to Serial 2, indicating to the Beaters that it was ready to receive a score. Both Beaters used the same letter that they used to communicate the pair score (in the case of Chaser 2 and Beater 2, letters ‘u’ through ‘v’) to communicate with the Seeker. Once the Seeker received a character from both pairs, it translated those letters back to decimal scores, and added them to zero (if it had not found the snitch) or 150 (if it had found the snitch). This final score was then sent out in the form of three digits. Because printing the final score was the only time the team ever used digits to communicate, it was simple to filter for digits and receive a final score. This final score was then printed to the LCD screens of each bot. The code used to receive the final score is shown below. Note that an infinite loop was intentionally created to ensure that all of the digits of the final score are received (the boolean “Jesus” will never be false).



```
void GetFinalScore(){ //Funtion for recieving full team score
  bool Jesus = true; //sets up a boolean so I can loop in this function until I get the full team score
  while (Jesus){ // while I haven't gotten the full team score
  char Incoming = Serial2.read(); //Names the Serial 2 reading "Incoming"
  if (isdigit(Incoming)) {  //If the incoming reading is a number (final score is only thing that is sent as a digit)
    //Serial3.write(13); //wrap from 1st line of LCD to 2nd line
    //delay(5);
    //Serial3.write(13);
     Serial3.print(Incoming);  //Print the final score
```

While the concept of team-wide communication was not difficult, the application of the idea required a lot of troubleshooting. This was challenging due to the fact that if a final score was not received, it could have been due to any of the five robots. To overcome this challenge, the team worked together closely to debug the code, ultimately using print statements to the sentry bot to track which bot had failed to communicate. Once the code was debugged, the sentry bot became unnecessary.

Once a replacement LCD was received, it was wired to the arduino using a three pin cable and mounted to the side the bot using two half inch standoffs and two brackets (See Figure 5). Using the Serial3.print command, the number of quaffles sensed, the paired score between Chaser 2 and Beater 2, and the final score were all displayed on the LCD.

During the final IDC, all five bots had to successfully sense their objects, communicate paired scores, and receive and display the final score. During the IDC, every bot worked perfectly, and for the first time in Quidditch history (and the history of the IDC), a lab section achieved a flawless IDC final demonstration.

4 Analysis and Discussion

The starting cost of the bot (the Robot Shield with Arduino) was $199.00, and the added cost to the bot (with sensors, wiring, etc.) came out to a total of $96.31. The final cost of the bot at the end of the IDC totaled to $295.31. This could have been slightly lowered by trying to streamline the circuit and the placement of the sensors in order to lower the amount of extra wiring used in the circuit board and lower the number of brackets and other parts used for mounting the sensors. The quaffle sensor itself (the whisker wire) was a fairly cheap sensing option (totaling around $3.00 for the sensor itself).

The overall design of the bot was shown to be very reliable during testing and proved to be an optimal design for the task presented to Chaser Bot 2. This proved evident in the five perfect trials completed during the final IDC demonstration. The main decisions that had to be made throughout the IDC challenge for the design of the bot were the type of sensor that would be used to find the quaffles, the number of QTI sensors that would be used for line following, and the way in which the bots would relay information between themselves. For deciding the quaffle sensor, a whisker wire was used; but an IR LED with cover and IR receiver, a phototransistor, and a ColorPAL sensor were also considered in the original conceptual design report. The whisker wire was chosen as the final sensor because of its many advantages. It is an easily added/removed sensor allowing for changes in the placement of the sensor to be made with ease. When the quaffle was sensed/not sensed, it was easily seen by looking at whether the whisker wire came in contact with the three pin header. If a quaffle was missed during a trial throughout the final stages of the IDC, the only thing that needed to be altered for the bot to sense the quaffle was the position of the quaffle in relation to the line and the hash. The number of QTI that would be used for line following was decided upon as a lab section after realizing three QTI allowed for high quality line sensing and a less complicated code. The way in which communication occurred between bots was also decided on as a lab section. It was found that the most reliable way to relay information efficiently was through sending arrays once each bot had reached the end of its lines and sensed all of their own objects. First, the two pairs of Beaters and Chasers would relay arrays to each other and compute paired scores. Then, the paired scores were sent to the Seeker bot. The Seeker bot would compute the lab section's final team score and send it back to each of the other bots.This proved to be both an accurate and efficient technique for team communication and score calculation.

During all five of the trials during the final demonstration, the bot was able to accurately sense all quaffles, receive and send scores, and display the number of quaffles sensed, the calculated pair score, and the lab section's final score. In order for Chaser Bot 2 to be more reliable during each trial, the area in which the quaffle is placed in order for the whisker wire to sense it can be widened. This would allow for fewer misses when trying to sense a quaffle. This could possibly be done by lengthening the reach of the sensing wire by extending the sensor out more with extra wiring or by moving it simply further to side to expand its sensing area ability. Another way to ensure a quaffle is sensed is by double checking the whisker wire's sensing with a second sensor (for instance, an IR sensor could be placed along with the whisker wire to reduce sensing misses). However, adding a second sensor could greatly overcomplicate and already highly-successful robot, in terms of both circuitry and code, and thus was not attempted in the original IDC.

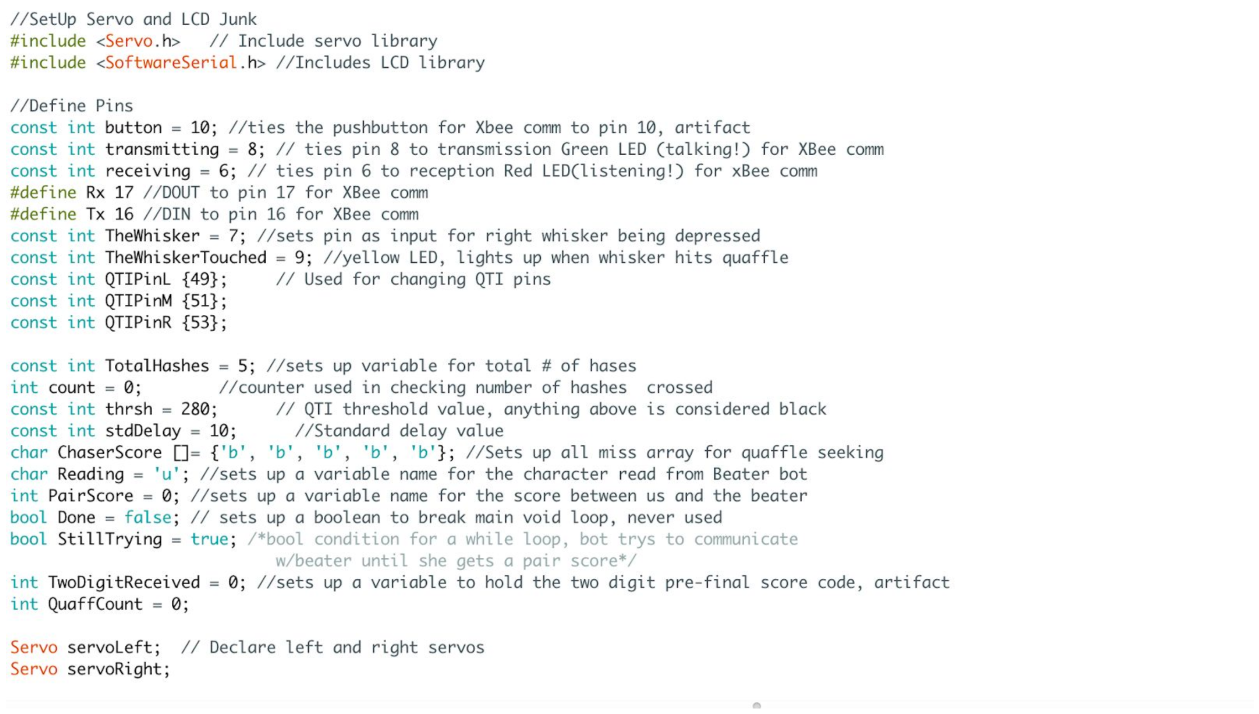
5 Conclusion

The main objective of the Integrated Design Challenge was to utilize and learn line following, bot communication, and object sensing in order to calculate the correct score based on object arrangements. Chaser Bot 2 was able to complete all expected tasks, both when working as an individual bot and when working with other bots. Chaser Bot 2 was able to line follow smoothly to the end of its line, stop at each hash, detect each quaffle, communicate with the other bots, and display the correct scores. Despite have a sensing that at times proved to be very specific for quaffle placement, the code and overall physical design of the bot showed to be consistently efficient and reliable at a relatively low cost (compared to the other options available). Overall, the final bot design represents the collaborative effort of the entire Section 8 Quidditch Team, and was refined and debugged over several weeks to create a cost effective and efficient final design.

Part II

Appendix

6 Code



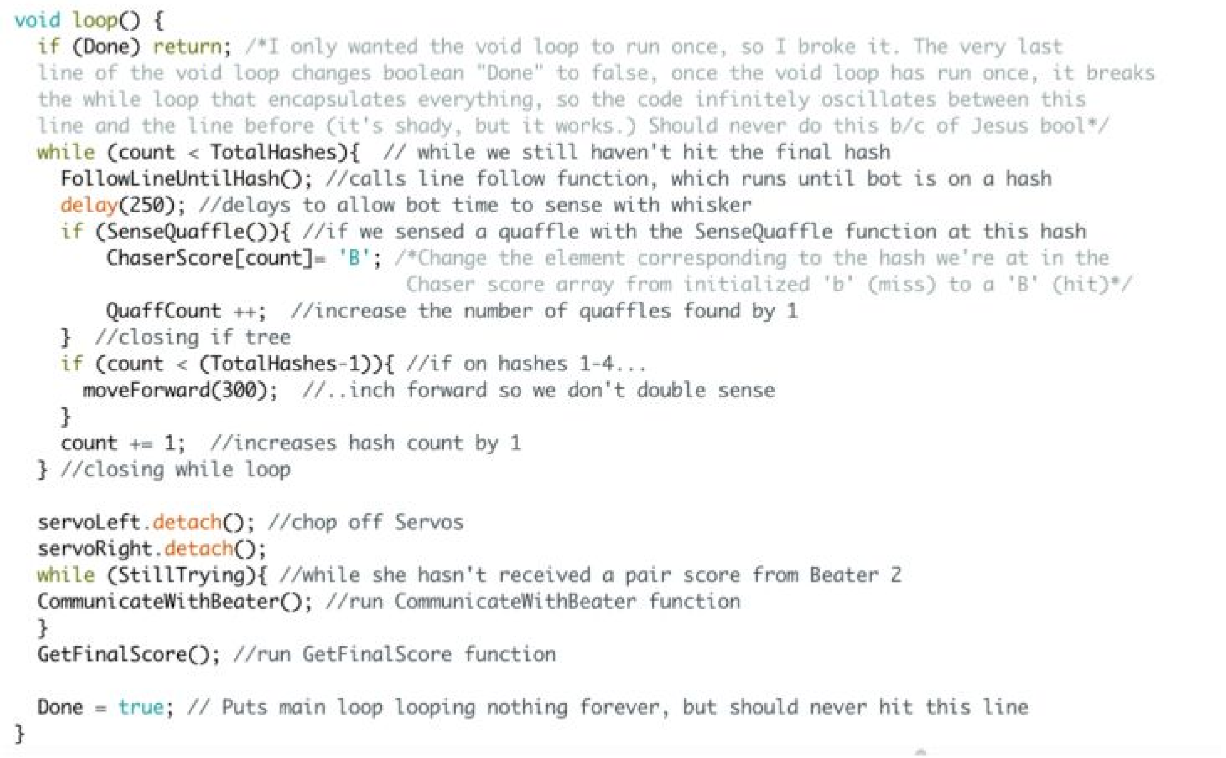
```
//SetUp Servo and LCD Junk
#include <Servo.h>   // Include servo library
#include <SoftwareSerial.h> //Includes LCD library

//Define Pins
const int button = 10; //ties the pushbutton for Xbee comm to pin 10, artifact
const int transmitting = 8; // ties pin 8 to transmission Green LED (talking!) for XBee comm
const int receiving = 6; // ties pin 6 to reception Red LED(listening!) for xBee comm
#define Rx 17 //DOUT to pin 17 for XBee comm
#define Tx 16 //DIN to pin 16 for XBee comm
const int TheWhisker = 7; //sets pin as input for right whisker being depressed
const int TheWhiskerTouched = 9; //yellow LED, lights up when whisker hits quaffle
const int QTIPinL {49};     // Used for changing QTI pins
const int QTIPinM {51};
const int QTIPinR {53};

const int TotalHashes = 5; //sets up variable for total # of hases
int count = 0;        //counter used in checking number of hashes  crossed
const int thrsh = 280;      // QTI threshold value, anything above is considered black
const int stdDelay = 10;      //Standard delay value
char ChaserScore []= {'b', 'b', 'b', 'b', 'b'}; //Sets up all miss array for quaffle seeking
char Reading = 'u'; //sets up a variable name for the character read from Beater bot
int PairScore = 0; //sets up a variable name for the score between us and the beater
bool Done = false; // sets up a boolean to break main void loop, never used
bool StillTrying = true; /*bool condition for a while loop, bot trys to communicate
                            w/beater until she gets a pair score*/
int TwoDigitReceived = 0; //sets up a variable to hold the two digit pre-final score code, artifact
int QuaffCount = 0;

Servo servoLeft;  // Declare left and right servos
Servo servoRight;
```

```
void setup() {
  //SET UP XBEE COMM
  pinMode(button, INPUT); //sets up the pushbutton as input for Xbee Comm ARTIFACT
  pinMode(transmitting, OUTPUT); //sets up LED as output for transmitting XBee Comm
  pinMode(receiving, OUTPUT); //sets up LED as output for receiving XBee comm
  pinMode(TheWhiskerTouched, OUTPUT); //sets up yellow LED as output for when the whisker is touched
  pinMode(TheWhisker, INPUT); //sets up the pin as an input for when the pin is pressed
  Serial.begin(9600); // sets Serial (our bot, serial moniter) to 9600 bits/sec
  Serial2.begin(9600); //sets Serial2 (the cloudish thing) to 9600 bits/sec
  Serial3.begin(9600); //sets Serial3 (LCD) to 9600 bits/sec
  //END SETUP XBEE COMM

  //SET UP LINE FOLLOW
  servoLeft.attach(12);                       // Attach left signal to P11
  servoRight.attach(11);  // Attach right signal to P12 (had to flip them because my bot is a demon)
  servoLeft.detach();
  servoRight.detach();
  servoLeft.attach(12);
  servoRight.attach(11);   /*all this detaching an reattaching is because my bot is a butthead,
                              she used to reverse at hashes even with no reverse code in the
                              program, theory was that she had charge buildup*/
  //END SETUP LINE FOLLOW

}


//LET'S WRITE SOME FUNCTIONS
long rcTime(int pin) {      //Takes value of QTI sensor
    pinMode(pin, OUTPUT);
    digitalWrite(pin, HIGH);
    delayMicroseconds(230);
    pinMode(pin, INPUT);
    digitalWrite(pin, LOW); // do not put earlier than previous line
    long time  = micros();
    while (digitalRead(pin));
    time = micros() - time;
    return time; //returns decay time necessary to discharge capacitor
}
```

```
void hardLeft(int t) {        //Turns left for given t of milliseconds
  servoLeft.writeMicroseconds(1400);
  servoRight.writeMicroseconds(1400);
  delay(t);
}
void softLeft(int t) {        //Turns left for given t of milliseconds
  servoLeft.writeMicroseconds(1500);
  servoRight.writeMicroseconds(1400);
  delay(t);
}

void hardRight(int t) {         //Turns right for given t of milliseconds
  servoLeft.writeMicroseconds(1600);
  servoRight.writeMicroseconds(1600);
  delay(t);
}

void softRight(int t) {        //Turns left for given t of milliseconds
  servoLeft.writeMicroseconds(1600);
  servoRight.writeMicroseconds(1500);
  delay(t);
}

void moveForward(int t) {     //Moves Forward for given t of milliseconds
  servoLeft.writeMicroseconds(1600);
  servoRight.writeMicroseconds(1400);
  //Serial.println("DID I MOVE????");
  delay(t);
  }

void FollowLineUntilHash(){ //Line Follow Funtion calling other functions
  bool Hash = false;  //while you're not on a hash
  while (! Hash) {
      int l = rcTime(QTIPinL);     //Takes QTI values and assigns to corresponding letter
      int m = rcTime(QTIPinM);     //for every loop
      int r = rcTime(QTIPinR);
      //Serial.println(l);                  //printlns QTI sensor values from left to right in neat format
      //Serial.println(", ");
      //Serial.println(m);
      //Serial.println(", ");
      //Serial.printlnln(r);
      if(l < thrsh && m < thrsh && r < thrsh) { //checks to see if WWW //reverse makes it mess up, comment out
        //reverse(100);
        //Serial.println("time to cry, I wanna reverse, WWW");
      }
      if(l < thrsh && m < thrsh && r > thrsh){  //checks to see if WWB
          hardRight(stdDelay); //ogHard
      }
      if (l < thrsh && m > thrsh && r < thrsh){ //Checks to see if WBW
        //Serial.println("I should move forward");
        moveForward(stdDelay);
      }
      if (l < thrsh && m > thrsh && r > thrsh){ //checks to see if WBB
        softRight(stdDelay);
      }
      if (l > thrsh && m < thrsh && r < thrsh){ //checks to see if BWW
          hardLeft(stdDelay); //ogHard
      }
      if (l > thrsh && m > thrsh && r < thrsh) {  //checks to see if BBW
        softLeft(stdDelay);
      }
      if (l > thrsh && m < thrsh && r > thrsh) {  //checks to see if BWB
        //Serial.println("time to cry, I am confused, BWB");
      }
      if (l > thrsh && m > thrsh && r > thrsh) {  //Checks to see if BBB (Hash detected)
        StopAtHashPoint(stdDelay);
        Hash = true;
      }
  }
}


bool SenseQuaffle(){
  if(digitalRead(TheWhisker) == 0){                         // If right whisker contact
  digitalWrite(TheWhiskerTouched, HIGH);   //turn on the yellow LED
  //Serial.println("found quaffle");
  return true; //allows for construction of array in main void loop
  }
  else  {                                 // If no right whisker contact
  digitalWrite(TheWhiskerTouched, LOW);    //Don't turn on LED
  return false;  //allows for construction of array in main void loop
  }
}


void CommunicateWithBeater(){
  char Reading =Serial2.read();  //names a variable "Reading" for whatever it reads from Serial 2
  if (Reading == 'Z'){  //if Nick is sending the 'Z' ready signal
    Serial.print("Nick Sent the Z");
    int element; //Names a variable for the counter of the for loop that prints the array
    for (element = 0; element <=4; element++){ //forloop, count up by 1 from 0 to 4 (array index starts at 0)
    //Serial.println(ChaserScore[element]);
    Serial2.print(ChaserScore[element]);  //print to serial2 personal score array
  }
  }
  /* All of the following if trees do the same thing, Nick's bot does math, sends a character indicating
  what our paired score is, translates the character to the pair score, clears the LCD, turns on the backlight,
 prints the number of quaffles, then a space, then the pair score, then a space. This function is called in the
 main void loop in a while loop that breaks once "StillTrying" is false (which is turned from the initialized
 true to false once my bot get's the beater's letter, ie, the bot will still try to communicate with the beater
 until she receives a pair score. */
  digitalWrite(transmitting, LOW); //turn the transmitting LED off
  //Reading = Serial2.read();
  if (Reading == 'u'){
    PairScore = 0;
    StillTrying= false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
  }
```

```
    else if (Reading == 'v'){
    PairScore = 10;
    StillTrying= false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
    }
    else if (Reading == 'w'){
    PairScore = 20;
    StillTrying= false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
    }

    else if (Reading == 'x'){
    PairScore = 30;
    StillTrying= false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
    }
    else if (Reading == 'y'){
    PairScore = 40;
    StillTrying= false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
    }
    else if (Reading == 'z'){
    PairScore = 50;
    StillTrying = false;
    Serial3.write(12);
    Serial3.write(17);
    Serial3.print(QuaffCount);
    Serial3.print(" ");
    Serial3.print(PairScore);
    Serial.print(PairScore);
    Serial3.print(" ");
    }

}
```

```
void GetFinalScore(){ //Funtion for recieving full team score
  bool Jesus = true; //sets up a boolean, originally supposed to break this fxn, but lets it loop forever
  while (Jesus){ // while I haven't gotten the full team score
  char Incoming = Serial2.read(); //Names the Serial 2 reading "Incoming"
  if (isdigit(Incoming)) {  //If the incoming reading is a number (final score is only thing that is sent as a digit)
     Serial3.print(Incoming);  //Print the final score
     //Jesus = false; // should have broken this while loop, but wasn't getting all 3 digits of score, so let it loop inf

  }

}
}

//DONE WITH FXNS
```



```
void loop() {
  if (Done) return; /*I only wanted the void loop to run once, so I broke it. The very last
  line of the void loop changes boolean "Done" to false, once the void loop has run once, it breaks
  the while loop that encapsulates everything, so the code infinitely oscillates between this
  line and the line before (it's shady, but it works.) Should never do this b/c of Jesus bool*/
  while (count < TotalHashes){  // while we still haven't hit the final hash
    FollowLineUntilHash(); //calls line follow function, which runs until bot is on a hash
    delay(250); //delays to allow bot time to sense with whisker
    if (SenseQuaffle()){ //if we sensed a quaffle with the SenseQuaffle function at this hash
        ChaserScore[count]= 'B'; /*Change the element corresponding to the hash we're at in the
                                      Chaser score array from initialized 'b' (miss) to a 'B' (hit)*/
        QuaffCount ++;  //increase the number of quaffles found by 1
    }  //closing if tree
    if (count < (TotalHashes-1)){ //if on hashes 1-4...
      moveForward(300);  //..inch forward so we don't double sense
    }
    count += 1;  //increases hash count by 1
  } //closing while loop

  servoLeft.detach(); //chop off Servos
  servoRight.detach();
  while (StillTrying){ //while she hasn't received a pair score from Beater 2
  CommunicateWithBeater(); //run CommunicateWithBeater function
  }
  GetFinalScore(); //run GetFinalScore function

  Done = true; // Puts main loop looping nothing forever, but should never hit this line
}
```

7 Figures

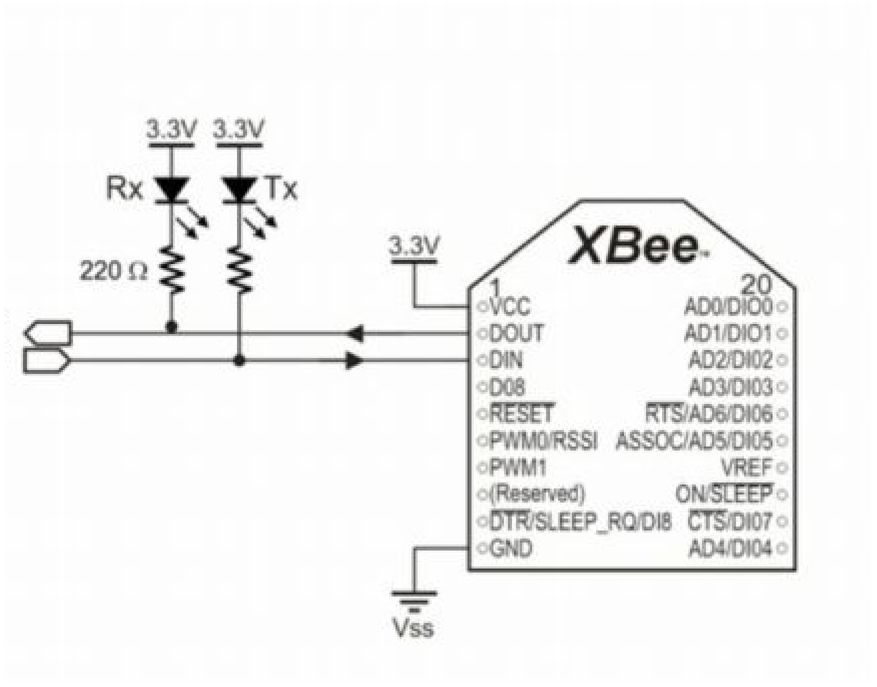


Figure 1. XBee wiring (3)

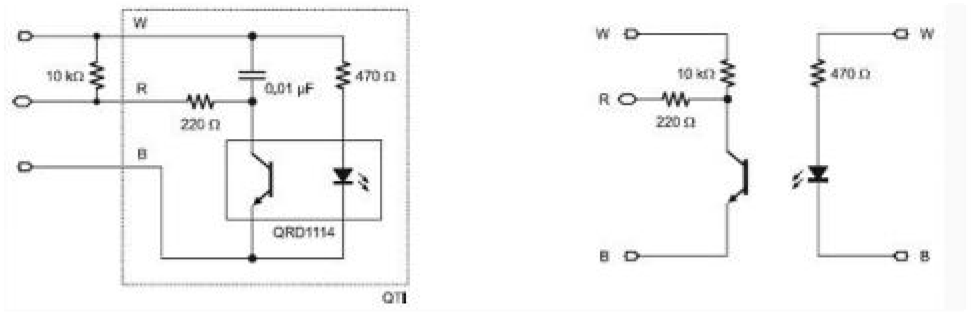


Figure 2. QTI general design (3)

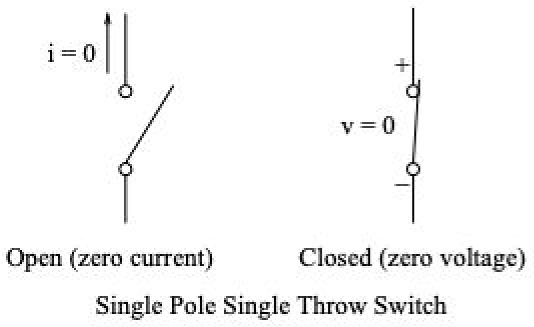


Figure 3. Whisker wire single pole single throw switch (2)

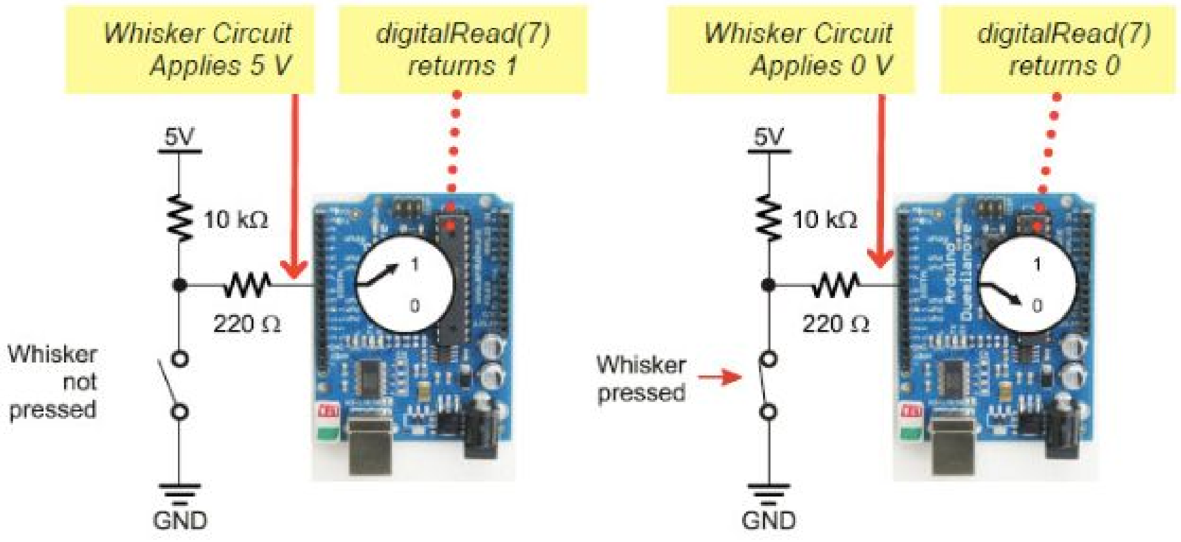


Figure 4. Whisker wire circuit for pressed and not pressed wires (1)

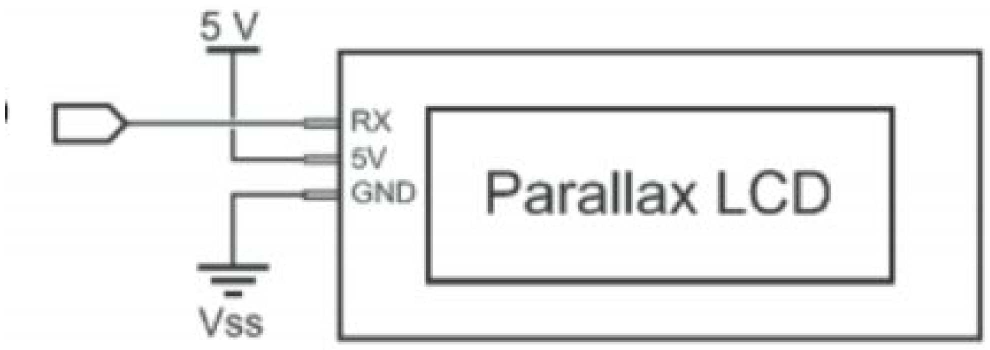


Figure 5. LCD wiring (3)

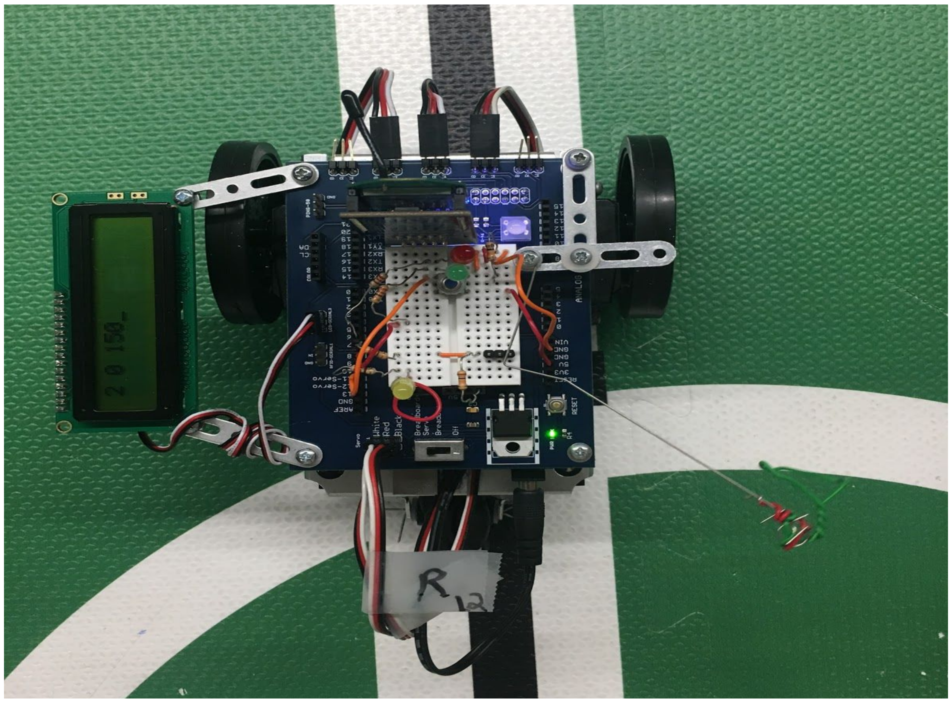


Figure 6. Final Bot

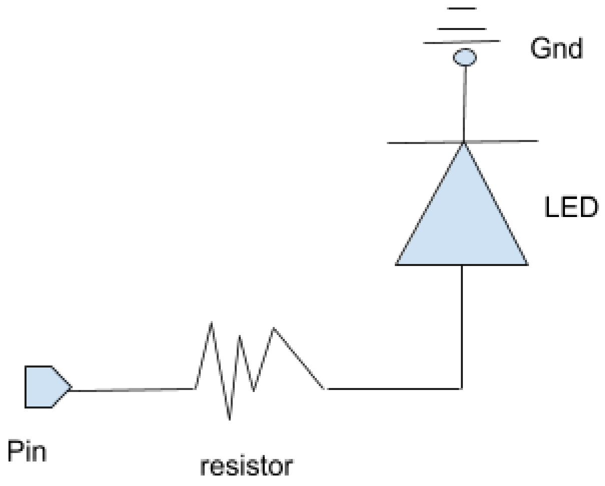


Figure 7. LED circuit schematic

Bibliography

1. "LEARN.PARALLAX.COM." *How Whisker Switches Work | LEARN.PARALLAX.COM*, learn.parallax.com/tutorials/robot/shield-bot/robotics-board-education-shield-arduino/chapter-5-t actile-navigation-9.

2. Brown, et al. *Laboratory V Basic Electrical Measurements II and Pressure Sensing*. 2019, sakai.duke.edu/access/content/group/4071a68e-dc02-4ae9-a77c-31a367645e3e/Labs/Lab%204%3A%20Temperature/Lab5_S19.pdf.

3. Mak, Grant. *ECE 110L - Fundamentals of Electrical and Computer Engineering IDC Final Report*. 2019, sakai.duke.edu/access/content/attachment/4071a68e-dc02-4ae9-a77c-31a367645e3e/Announcements/8da8b25d-543a-4d64-87e2-318e58712e29/ECE_110L___IDC_Final_Report.pdf.