Course Project
RoboCupJunior Rescue Line

ENGR-UH-3530: Embedded Systems

Alexander Avdeev

17th December 2019

Brandon Chin Loy (bcl320)
Lujain Ibrahim (li431)
Sohail Bagheri (swb300)

Introduction

This report is a technical documentation of our term project for the fall 2019 Embedded Systems class. This project is inspired by and based on the RoboCupJunior Rescue Line challenge to build an autonomous robot that follows a black line while overcoming different problems in a modular field formed by tiles with different patterns.

Timeline and Planning

Project conceptualization began in mid October where possible platforms, sensors, robot size, and microprocessors were considered. In our search for a chassis for our robot, one of our main constraints was the size of the robot considering the size of the tiles was set to 30 cm x 30 cm in the rules. In addition to that, since the robot also needs to avoid obstacles and navigate short turns, we quickly realized that the smaller the robot and the lower its centre of mass, the better it will perform in these situations. As a result, we chose the Zumo Chassis and shield whose design and functionality are explained in more detail in the next sections.

Following that decision, we searched for sensors to detect the black line, the green squares, and the distance. For line following, the Zumo Chassis has an expansion area in the front designed for the Zumo reflectance sensor array to be mounted which can be used for line-following and edge-detection capabilities. As for detection of green, we first considered the TCS230 TCS3200 color sensor from the lab as well as the TCS34725 RGB Light Color Sensor from the Engineering Design Studio. After performing a couple of tests, we realized that the latter was more suited for our project and performs better at green detection. Finally, we considered IR distance sensors as well as ultrasonic distance sensors to measure distance for obstacle detection. We ultimately chose to use the ultrasonic HC-SR04 distance sensors.

Following these preliminary hardware decisions, the different sections of the challenge were addressed in the following order:

1. Line Following
2. Turning on Green (Left/Right)
3. Turning on Green (U-Turn)
4. Obstacle Detection and Avoidance
5. Ramps
6. Speed Bumps
7. Gaps
8. Integration and Fine Tuning

Circuit Design and Implementation

The circuit implemented in the project is shown below:

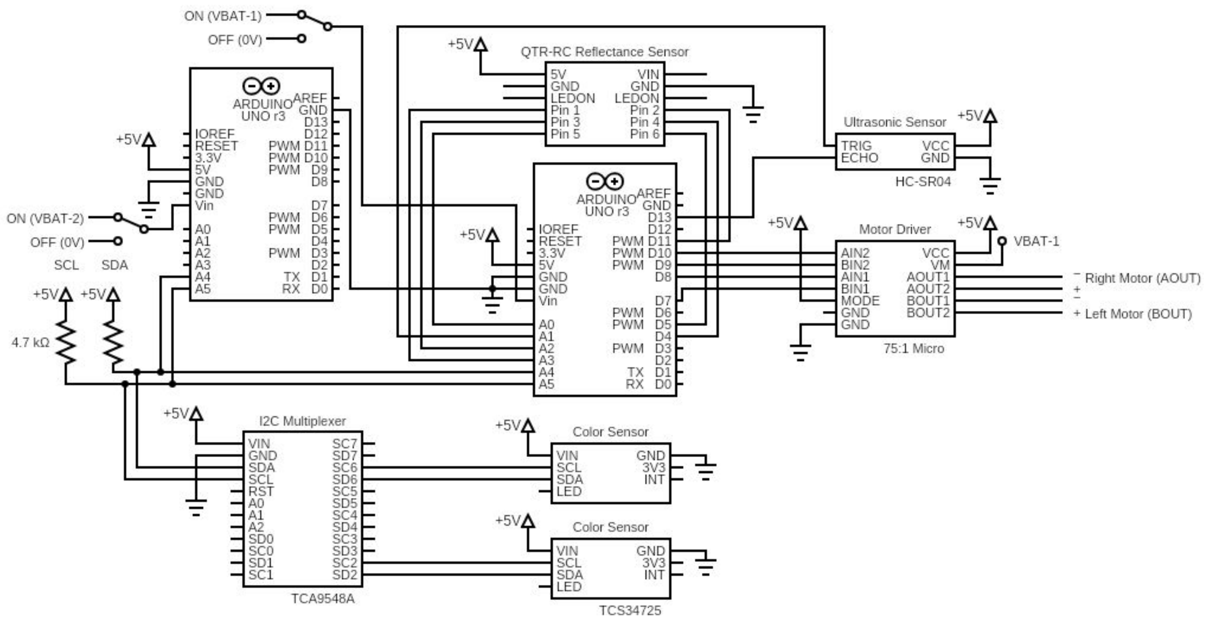

Figure 1: *Circuit Diagram for Robot*

Hardware Descriptions

Figure 1 illustrates that the robot was commanded by two Arduino Unos. The primary Arduino was directly attached to the shield and served as the processing unit for the robot's operations as specified in the competition manual. As was stated earlier, the shield was attached to a chassis that contained the power source and the movement-related hardware in the interior. The Arduino on the shield is powered by four AA batteries, which also powers the integrated DRV8835 H-bridge dual motor drivers that control the two 75:1 HP micro metal gearmotors. These gearmotors were attached to two wheels at the front of the vehicle. There were also two wheels at the back of the vehicle, although these were not attached to any motors. Instead, they were connected to the front wheels using two silicone tracks. As a result, the left back wheel replicated the motion of the left front wheel (and the same for the right wheels). This provided additional stability and mobility. The second Arduino, seen to the left in Figure 1, was used as a supporting processor to carry out intensive calculations. This was important in ensuring that the primary Arduino was able to perform other time-sensitive tasks, such as the PID control system. This second Arduino was powered using a 9V battery inside a LAMPVPATH 9V Battery Holder, which provided an on/off switch. Furthermore, the tasks laid out in the competition manual were achieving using the following additional sensors:

1. **Line Following, Speed Bumps and Gaps:** 1 x Zumo Reflectance Sensor Array
2. **Turning on Green:** 2 x Adafruit TCS34725 Color Sensors and 1 x Adafruit TCA9548A 1-to-8 I2C Multiplexer Breakout
3. **Obstacle Detection:** 1 x HC-SR04 Ultrasonic Distance Sensor
4. **Ramps:** 1 x LSM303D Compass and Accelerometer. Note that this IMU was already built into the shield and did not need to be added separately.

Hardware Arrangement

The location of the primary Arduino was predetermined by the shield. The placement of the various sensors was determined by considerations concerning space, size and weight distribution. The culminating hardware arrangement is shown in the pictures below:

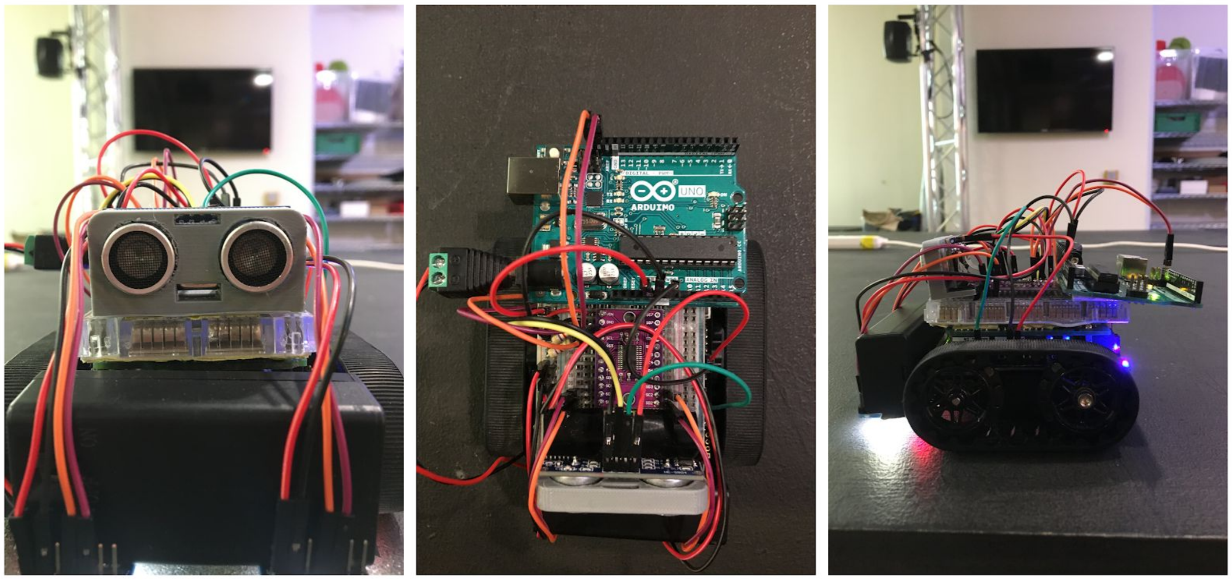

Figure 2: *Hardware Arrangement on Robot*

The middle photo above shows that a breadboard was attached directly on top of the shield to provide additional points of connection, which was especially needed for the serial clock (SDC) and serial data (SDA) lines for I2C. This was a deliberate choice as we knew that numerous devices had to share SDA and SCL, specifically the two Arduinos, the accelerometer, the I2C multiplexer and the two color sensors. Note that each line was attached to a 4.7kΩ pull-up resistor to ensure that all devices had the same resting state. In addition, all devices on the I2C lines had a common ground to prevent communication issues. The distance sensor was attached toward the front of the robot to ensure that the ultrasonic signal could be sent with minimal blockage from wires and the other sensors. For additional security, a 3D-printed case was attached to it. The two color sensors were placed in front of the reflectance sensor. This was also a deliberate choice because the color sensors did not fit behind the reflectance sensor, which was a preferable alternative. These sensors were oriented to face downward at no angle, providing the most correct readings during testing. Although ramps were not testing on the demo, this arrangement considered weight distribution. The battery for the second Arduino could not be placed in the middle or at the back of the vehicle as it would tip backward on an incline. As a result, placing the battery on the front of the vehicle was more suitable. This allowed room for the second Arduino on the back, an arrangement that kept the robot grounded.

Microprocessor Choice

In this project, we use two Arduino Unos which communicate together over I2C. This will be explained later on in more detail in the I2C section. One Uno is attached to the Zumo Shield, whereas the other one is used for reading and processing the color sensor readings.

The Zumo Shield is only compatible with the Arduino Uno and Arduino Leonardo. This informed and restricted our choice of the Arduino on the Zumo Shield. While we ran some tests using the Arduino Leonardo on the shield, for debugging purposes, the Arduino Leonardo did not perform as well as the Uno with fast sensor readings and serial prints. This is because in the Leonardo, which only uses one processor (the Atmega32u4), serial goes through only one processor which fills up the computer's serial buffer faster than the Uno and other boards. This leads the Arduino IDE to run much more slowly and was not effective for debugging.

As for the choice of the other Arduino, we also chose an Uno. This is because we attempted I2C communication with the Shield Uno using a Leonardo and a Nano (which is more optimal for space), but the Uno to Uno I2C communication proved to be the most reliable and consistent.

One issue we experienced was the number of pins available for use, which also pushed us to use two Arduinos even during the tests we conducted with the TCS230 TCS3200 Color Sensor which do not use I2C or require as much computation as the TCS34725 RGB Light Color Sensors (our final color sensor choice). We attempted to use an Arduino Mega on the Zumo Shield to have access to more pins, but due to incompatibility of the Mega with the Shield, we experienced issues with PID control and decided to revert back to the Uno.

Line Following

The line following capability was achieved with a QTR reflectance sensor array:

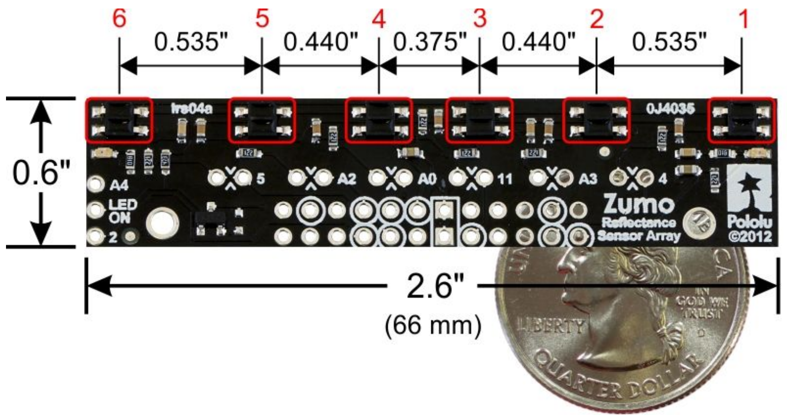

Figure 3: *Zumo Shield Reflectance Sensor Array* (Source)

Figure 3 illustrates that this array has six IR emitter/phototransistor pair modules that enable the detection of black on the surface beneath the vehicle based on the amount of light reflected from the ground to the phototransistor, and hence a lower reflection corresponds to black due to its high absorption property. Furthermore, the readLine function in the library returns an estimated position of the line by calculating the weighted average of the sensor indices multiplied by 1000 (Source). The position formula for six modules is shown below:

$$\text{Position} = (0*S_1 + 1000*S_2 + \ldots + 5000*S_6) / (S_1 + S_2 + \ldots + S_6)$$

If the position is exactly equal to 0, 1000, 2000, et cetera, then the black line is directly under S_1, S_2, S_3, et cetera, respectively. That being said, when the black line is under two or more sensors, the position will read intermediate values. Here, the range of position is [0, 5000]. Since this array provides continuous tracking of the black line on the track, then the position value can be used to calculate the speeds of the motors needed to redirect the vehicle to the center in cases where deviation occurs. This is achieved with PID. Before this can be done, the position must be used to provide an error term. If the black line is exactly at the center of the array, then the position will be equal to 2500. By subtracting 2500 from the position value, then 0 will correspond to the black line at the center of the vehicle, thereby suggesting that the motors should not change speed. As a result, the error range is [0-2500, 5000-2500] = [-2500, 2500]. The application of this error to PID is explained below.

PID Control

We initially set out to use a PID controller, but noticed that values, especially for the integral term, were causing jitters in our robot due to the calculations on the Arduino. As a result, we set to utilize a PD controller using the proportional and derivative terms:

Values:

- Kp = ¼
- Kd = 6

In each loop, the value of *error* and *lastError* is measured, which corresponds to the derivative term as a high change corresponds to a high rate of change between the error values. *lastError* is set to *error* after each PD control calculation.

The PD control is used to calculate the *speedDifference*, which is then directly used to change the values of the left (*m1Speed*) and right (*m2Speed*) motors.

m1Speed = MAX_SPEED + speedDifference
m2Speed = MAX_SPEED - speedDifference

If, after the calculation using *speedDifference*, the values for *m1Speed* or *m2Speed* are either below MIN_SPEED or larger than *MAX_SPEED*, then they are limited to those values. *MIN_SPEED* and *MAX_SPEED* are set by us to be - 100 and 130 respectively. This allows one motor to actually reverse if needed in order to better help with sharp turns.

A higher value for Kp (= 1) improved our performance significantly around sharp turns and zigzags. However, this came at the drawback of not continuing straight during T-junctions. The reflectance sensors would detect a short black line protruding from the side and want to turn towards it. A lower value for Kp (= ⅙) solved this problem, as the momentum of the robot helped it go straight. However, this came at the cost of poor performance during zigzags, where the robot was unable to turn as agile as cases with high Kp. We decided to settle on a value in between, with higher priority given to the momentum of the robot (Kp = ¼). The Kd value did not have as much of an effect as Kp, but the robot started having very jittery motion when Kd increased very high. The best performance we measured was with Kd = 6.

Green Turns

To detect green, two TCS34725 RGB Light Color Sensors were used along with a TCA9548A 1-to-8 I2C Multiplexer. These color sensors have only one fixed address and no back up address (resulting in an address clash if the two sensors need to be read in the same circuit). And so, a multiplexer was used which enables the use of up to 8 I2C modules with the same address. However, due to a delay in the shipment of the I2C multiplexer, initial tests with 2 color sensors were conducted using the TCS230 TCS3200 color sensor available in the lab. However, the turning logic remained the same. The only difference was in calibration and detection accuracy and sensitivity. Color detection occurs on the second Arduino Uno where the color values are read over I2C. The results are then sent to the Zumo Arduino on request, also over I2C.

Calibration

1. **TCS230 TCS3200 Color Sensor:** to calibrate these color sensors, they were first attached in what was believed to be an ideal position on the robot for color detection: in front of the reflectance sensors with an approximately 1 cm gap between them to account for the black line (that way the left and right color sensors would fall over their respective green turn squares when they appear). This sensor senses color light using an 8x8 array of photodiodes and a Current-to-Frequency Converter to convert the readings to a square wave with a frequency proportional to the light intensity. The photodiodes have three different color filters: green, red, and blue. To select which color to read, the two control pins S2 and S3 are used (S2=0, S3=0: red. S2=0, S3=1: blue. S2=1, S3=0: clear. S2=1, S3=1: green). The two other control pins S0 and S1 are used to set the output frequency.

   A script that takes 500 readings from the sensor for different colors (by changing the S2 and S3 control pins) and returns the minimum and maximum RGB values from the range was used in calibration. This was run over 5 colors: red, green, blue, black, and white. After getting the minimum and maximum RGB values for each of these, the range that best detects green (which will be the condition of the detection if statement) was found using an elimination process (eliminate black and white first, then red and blue). However, it quickly became clear that these sensors are extremely sensitive to lighting conditions since the calibration had to be repeated often. As some tests were done to check turning when green is detected, it was observed that the sensors pick up green in random instances, more specifically when the color sensors are half over black and half over white, which led to erroneous turns. In order to combat this, we used cardboard to separate the color sensors from each other and minimize interference from ambient light as seen in Figure 5 below. Even though this improved performance slightly, we were still compelled to pursue the TCS34725 sensors even further due to their much better performance and higher resistance to interference.

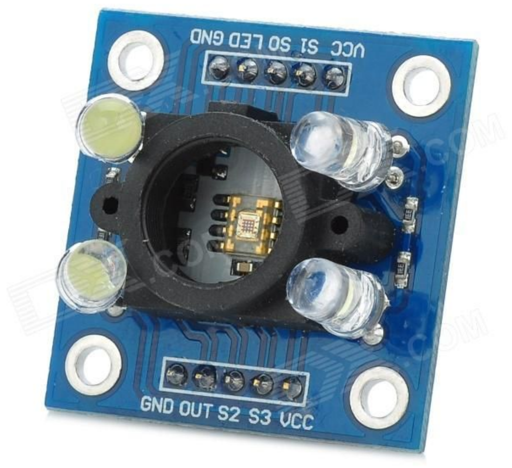

Figure 4: *TCS230 TCS3200 Color Sensors* (Source)

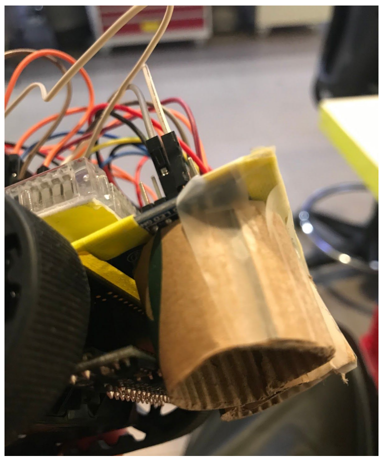

Figure 5: *Robot with TCS230 TCS3200 Color Sensors*

2. **TCS34725 RGB Light Color Sensors:** these sensors, which were our choice for the final design, use an IR blocking filter which minimizes the IR spectral component of the incoming light which leads to more accurate color detection. For calibrating this sensor, the raw readings of the sensor are first read, the average of the readings is calculated by adding the RGB values and dividing by 3, and this average was then used to normalize the readings by dividing each reading (R, G, and B) by the average. This makes the readings smaller and amplifies differences to the human eye which was crucial since our calibration was manual. The RGB values were once again read over red, green, blue, black, and white and the RGB range over which the color sensor should detect green was then found through observation and elimination. The process of detecting black was similarly carried out using manual trial-and-error methodologies. However, it was observed that white and black produce similar RGB values at the conditions under which

the calibration was taking place. As a result, to accurately differentiate between black and white, the illuminance associated with each color reading had to be calculated. This is a measure of the intensity of light that is received by the color sensor upon reflection. It was observed that a white surface resulted in a higher intensity due to higher reflection when compared to a black surface. Therefore, when checking the conditions for positively identifying a black surface, the illuminance was checked in tandem with the RGB values to prevent confusion between black and white.

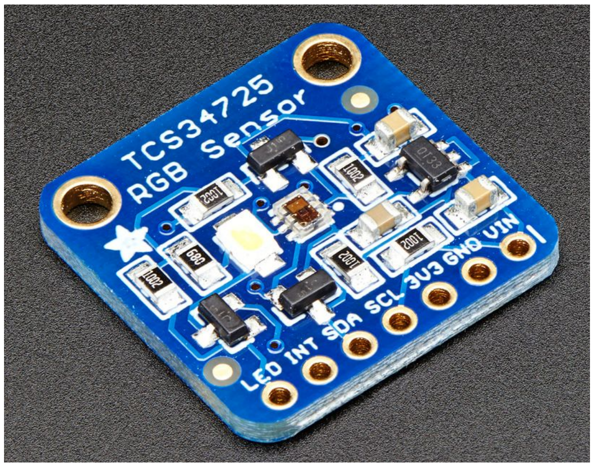

Figure 6: *TCS34725 Color Sensor* [(Source)]

I2C Integration

The communication of color values from the second Arduino to the primary Arduino on the shield was achieved with I2C. RGB values were continuously taken from the color sensors on the second Arduino. This was due to the fact that the normalization calculations mentioned above were computationally intensive, considering the inclusion of multiple floating point arithmetic calculations, which would adversely affect the timeliness of the PID control and the smooth movements of the robot. As a result, the primary Arduino was equipped with a function that allowed it to request the color detected by a specific sensor at that particular point in time. This function is called *readSensor* and implements functions from the Arduino Wire library:

```
    int value = 0;
(1) Wire.beginTransmission(ADDR_2);
(2) Wire.write(command);
(3) int status = Wire.endTransmission(0);
(4) Wire.requestFrom(ADDR_2, responseSize, 1);
(5) if (Wire.available() != 0) {
        (6)  value = Wire.read();
    }
```

The primary Arduino initiates communication as a master writer to the second Arduino (1). Here, an important question arises. How will the primary Arduino differentiate between a color value from the left sensor or the right sensor? This was overcome by implementing a command system. First, the primary Arduino indicates to the second Arduino the specific sensor that it wants to read the color from, which is represented by COLOR_L or COLOR_R (2). Afterward, it does not release the bus, but rather sends a restart message after transmission by including a false boolean in the *endTransmission* command (3). This was a critical parameter because by default the boolean is true, which means that another device on the I2C line could acquire the bus before the primary Arduino is able to request the color from the specified sensor (4). This ruined the communication flow, and accounted for numerous errors on the I2C lines during initial testing before changing the boolean to false. If the second Arduino had a value for the specified sensor (5), then the value was read by the primary Arduino (6). Otherwise, no action was taken.

The second Arduino handled the commands from the primary Arduino using an *receiveEvent* and *requestEvent* interrupt functions:

receiveEvent

```
(1) command = Wire.read();
```

requestEvent

```
switch (command) {
        (2) case COLOR_L: Wire.write(value_L); break;
        (3) case COLOR_R: Wire.write(value_R); break;
```

The specific sensor that the primary Arduino wants to read the color from is received in (1), again represented by either COLOR_L or COLOR_R. As a result, when the secondary Arduino receives the request from the primary Arduino for the color value, then the correct value is sent (2 and 3). The I2C communication flow between the two Arduinos is summarized below:

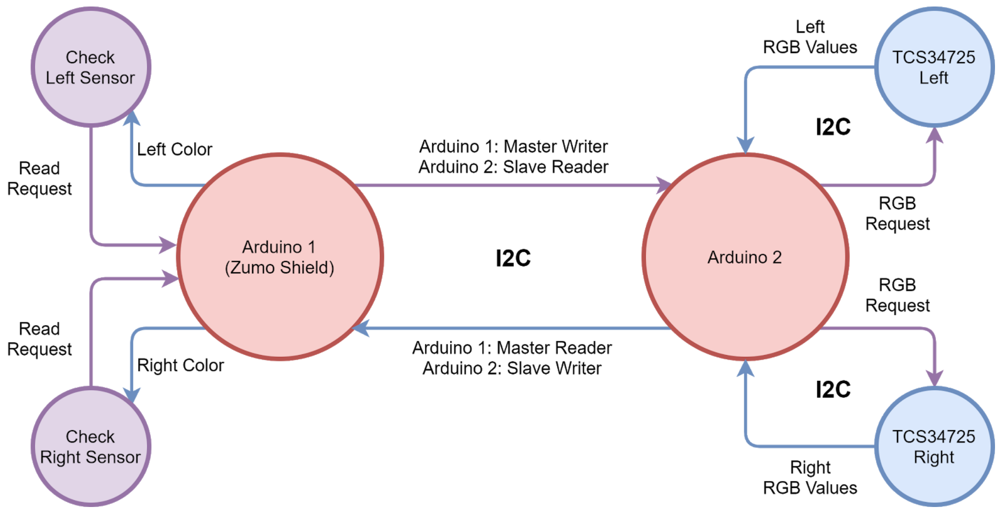

Figure 7: *Flowchart for I2C Communication for Color Detection*

The purple colors represent the first half of communication, whereas the blue colors represent the second half. This allowed for efficient interfacing between the two Arduinos.

Turn Logic

There were three cases to consider: a left turn, a right turn and a U turn:

1. If the left sensor detected green **and** the right sensor did **not** detect green, then the robot was instructed to make a left turn. This was hard-coded. (*Note that the compass was initially used to make accurate 90° turns, but proved to be unreliable over time due to magnetic interference from the surrounding environment.)*
2. If the right sensor detected green **and** the left sensor did **not** detect green, then the robot was instructed to make a right turn.
3. There were three possible time-dependent combinations that produced a U-turn:
    a. The left color sensor detected green **before** the right color sensor detected green.
    b. The right color sensor detected green **before** the left color sensor detected green.
    c. Both the left and right color sensors detected green at the same time.

   (a) and (b) were specifically accounted for. If the left color sensor detected green, then the right color sensor was read five times. If these readings did not detect green, then it was ensured that there was no green square on the right and the robot turns left. If it did detect green, then there was a green square on the right and the robot makes a U turn. This same logic was applied for the right color sensor.

Obstacle Avoidance

For obstacle detection and avoidance, an Ultrasonic HC-SR04 sensor was used.

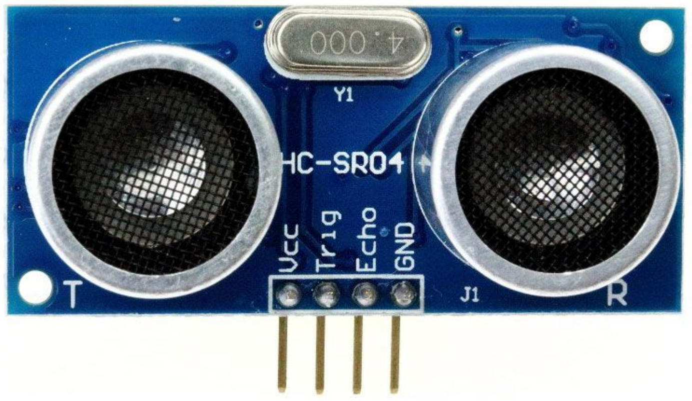

Figure 8: *HC-SR04 Ultrasonic Distance Sensor* (Source)

To begin detecting using the ultrasonic sensor, the sensor is first triggered by a HIGH pulse of 10 or more microseconds. The signal, which is a HIGH pulse whose duration is the time (in microseconds) from the sending of the ping to the reception of its echo off of an object, is then read. The time is converted into a distance using a microseconds to centimeters function. The distance is then checked to see if it's between 4 and 12 cm. If it is within that range:

1. The current millis value is assigned to the previous millis variable and the millis function is called again and stored in the current millis variable
2. The code checks if *currentMillis_obstacle - previousMillis_obstacle* is less than 3 seconds. If it is, it begins to implement the avoidance logic. This is to ensure that the sensor does not redetect the same obstacle it just detected as it is trying to avoid it. Based on trial and error, it was seen that not detecting for 3 seconds works best to avoid this problem.
3. The obstacle detection counter is then incremented. This counter is used later in the randomization of the avoidance direction (where it's a left or right turn).
4. If *(counter % 2)* is 0, it avoids the obstacle from the left. Otherwise, it avoids it from the right. That way it will alternate between avoiding from the left and the right. To actually avoid the obstacle, two functions are called depending on the direction: *obstacleFromLeft()* and *obstacleFromRight()*.
5. *obstacleFromLeft()*: move left and move forward to move away from the object, move right and move forward to become parallel with the object and facing the forward direction to pass the obstacle, then finally move right and forward again to get back on the black line and continue line following. When it moves forward and parallel to the obstacle, it moves for a delay of 1400 ms for a short obstacle and 1700 ms for a long obstacle. *obstacleFromRight()* works similarly but in the other direction. These values can be altered depending on the length of the object. The time turning 90 degrees to avoid the obstacle is calculated based on the RoboCupJunior rules that there is always at least a 10 cm clearance to the edge of the board.

Using delays for turning can be extremely unreliable as they are highly dependent on motor speed and uncontrollable factors such as battery degradation over time. As a result, they are hard to make reproducible and constantly need to be adjusted.

We initially set out to use the magnetometer built into the *LSM303 Triple-axis Accelerometer + Magnetometer (Compass)* sensor to calculate 90 degree turns accurately. This was achieved by determining the current heading of the robot, and indicating a target heading. In the case of turning to the right, the target heading would be the current heading plus 90 degrees, after which the speed of the motors was adjusted successively to turn the robot until the current heading fell within a specific margin of the target heading. However, as was mentioned earlier, we noticed that the magnetic interferences present underneath the circuit hindered the performance of the magnetometers significantly which made us return to using delays.

Ramps

For ramp detection, the LSM303D sensor was used:

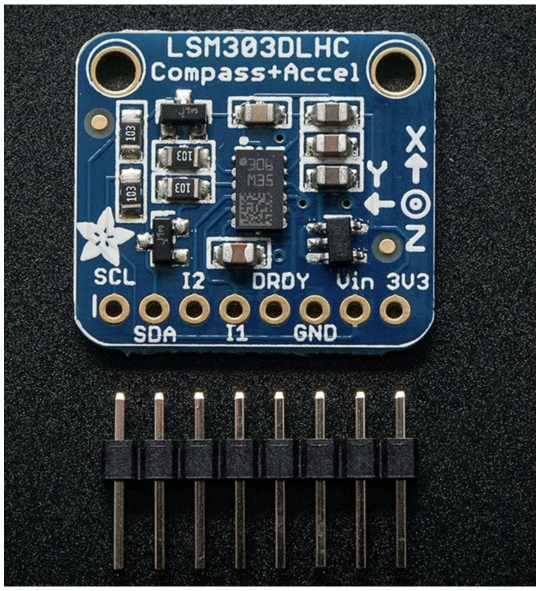

Figure 9: *LSM303D Accelerometer and Magnetometer* (Source)

The rules for the RoboCup Junior Rescue Line specify the possibility of including a ramp in the design of the circuit. To prepare for this, we designed an implementation in code to detect inclines, and change the motor speeds based on the value of the incline.

Regularly reading values from the compass, we stored the accelerometer values in the x-axis direction in a variable for incline. This was then used to alter the *MAX_SPEED* of the motors. If the incline is positive and over a set limit, the *MAX_SPEED* was increased to account for the extra power needed to traverse the ramp. Alternatively, if the incline is negative and below a set limit, the *MAX_SPEED* was decreased in order for the robot to carefully maneuver down the ramp.

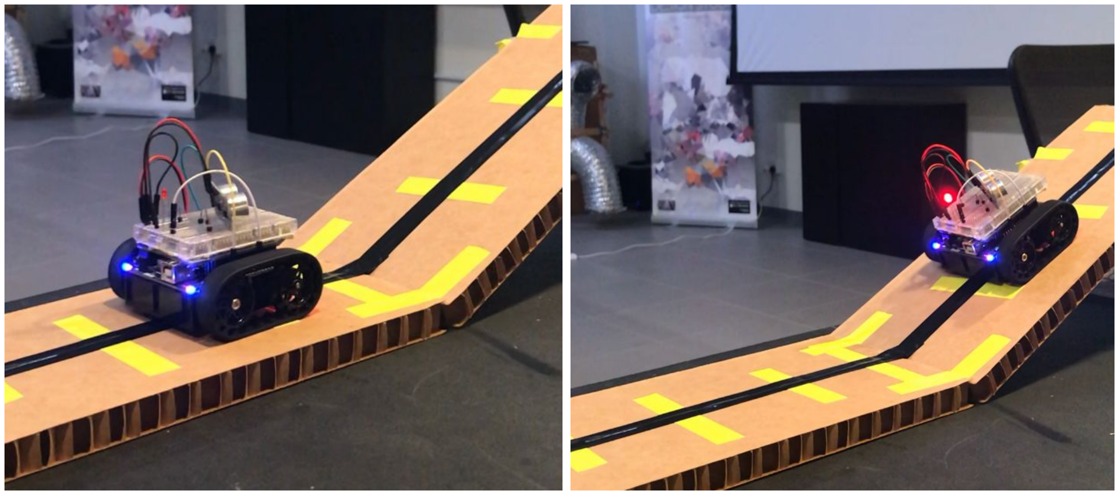

Figure 10: *Robot Tackling a Ramp*

In addition to changing the speed based on the incline, the small frame of our robot made it subject to getting stuck on ramps either at the start on the way up or at the end on the way down. To combat this, we created a function rampCheck() to detect if the robot has been stuck on the ramp either on the way up or on the way down.

It utilizes *totalChange*, a variable which adds the absolute error of the changes in values of the reflectance sensor over a large number of loops. The number of loops was determined based on the longest stretch of straight black line on the circuit, in order to prevent false positives that the robot was stuck. If *totalChange* is less than a set value (corresponding to it being stuck) and the robot is on an incline, then a high PWM is sent to the motors for 325 milliseconds.

Speed Bumps

To detect speed bumps, the code first checks if the robot is stationary. It does this check every 4000 loops. It then calls *speedBumpCheck(),* a function which checks for a speed bump by checking the change in the reflectance sensor readings. When the *totalChange,* a variable which adds the absolute error of the changes in values of the reflectance sensor over a large number of loops, becomes less than 20, this means that the robot is stuck at a speed bump and needs a push to get on top of the speed bump. In order to do this, it reverses by setting the speeds of both of the motors to -100 and then moves forward with more force by setting the speeds of both of the motors to *MAX_SPEED*1.5.* This allows the robot to approach the speed bump at an angle allowing one of the sides (wheels) to get over the speed bump, which along with the line follower capabilities, allows the robot to overcome the speed bump. This speed is set for only a short duration of 400 milliseconds which allows the robot to return to its original speed once it resumes line following.

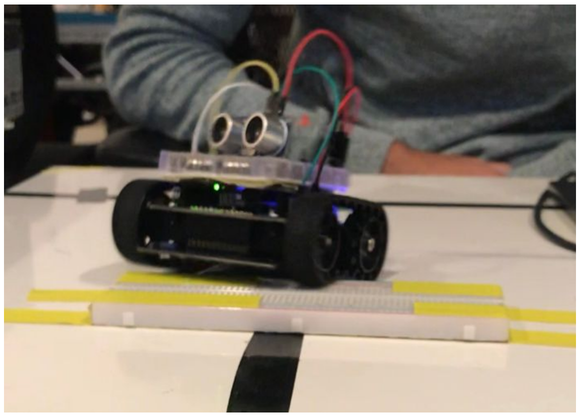

Figure 11: *Robot Tackling a Speed Bump*

Gaps

To detect gaps, we attempted various methods. The first "brute force" method was to simply continue straight when the end of a black line was detected until it was detected again. Although this worked for a majority of cases, there were times where it would detect false positives for gaps. To prevent these false positives, a few measures were taken:

1. When reflectance sensors measure that the robot veers off black line (absolute value of error = 2500), then increment a counter *gapCounter*. If a value other than 2500 is ever measured, then reset *gapCounter*. This means that the robot has found a black line and is back to line following.
2. If *gapCounter* is greater than a set *maxGap*, then return to initial position before it started veering off the edge of the black line.
    a. This is done by measuring the time taken (*elapsedTime* using *millis()* function) from when *gapCounter = 0* to when *gapCounter > maxGap*.
    b. Depending on whether the error was positive (2500) or negative (-2500), return to original position by appropriately setting the motor speeds to reverse the motion. Stay in the motion for time = *elapsedTime.*
    c. Instead of assuming we are at a gap, we still need to check the other side that the robot did not veer off to as there may be a black line to follow there. If the error was 2500, then the robot initially veered off to the right before we brought it back

to the center. Now, we would need to check the left side before asserting that we are located at a gap.

  i. This is done by calculating a *newTimeElapsed* and entering a while loop with *newTimeElapsed* < *timeElapsed* and the absolute value of the current error = 2500. The value of the error updates in every iteration of the while loop and the loop is exited if the time elapsed is equal to the initial time elapsed from where it veered off or the error != 2500 which means that it detected a black line.
    1. If a black line is detected, we set a flag *blackFound = true*
    2. Otherwise, we return it to the original point in a similar way to when we return it to the original position before it veered off in the original (opposite) direction.

d. If *blackFound* flag is true, then we resume line following

e. Otherwise, we set the motors to go straight, meanwhile measuring values from the reflectance sensors until a black line in detected, after which line following is resumed again.

Despite the following measures, there were significant drawbacks to our algorithm of detecting gaps. First, we continue moving straight from the reference of a single point on the preceding black line as opposed to a line, which makes it difficult to predict where the next black line will be. A better alternative would be to use two points on the previous black line in order to extend a more accurate line that the robot could follow towards the landing black strip.

Code

Aside from NYU Classes, the code and development history for this project can be seen here: https://github.com/lujainibrahim/embedded-systems-project