ECE 1188

Autonomous Racing Report

Team Daredevil: JJ Oswald, Adam Brower, Troy Weaver

1 Overview of Design

Before the race starts our robot will establish a connection to the MQTT broker. Once this connection is made the bot will receive Bluetooth communication when the race starts and will publish to a topic to record the start time of the race. The robot will then enter the main loop where the PID will handle race conditions, wall following, and object avoidance. Throughout the race metrics such as max motor speeds and distance sensor readings are logged and stored. Once the robot either receives communication via Bluetooth to stop or crosses the finish line it will trigger one final publish to the server and will send all remaining metrics such as stop time, number of crashes, distance sensor logs, and max motor speeds.

2 Description of features design and contributions

2.1 JJ

I handled the MQTT stuff on the robot side as well as the robot crashing logic. This included figuring out how to log all of the relevant MQTT data and sending it to the MQTT broker and Adam's Node Red server in a suitable format. A large portion of this was piecing together components from previous projects into a single place.

2.2 Adam

I was mainly in charge of getting the node-red dashboard up and ready to receive telemetry from the robot via MQTT. After that was implemented, I handled the Bluetooth communication as well as the line sensing to tell the robot to stop once the finish line is reached.

2.3 Troy

I was tasked with handling the racing functionality of the robot. I implemented the PID controller used for the race as well as helped integrate the bump sensors into the final robot program. The PID controller needed to be tuned to ensure optimal performance, and the logic was altered before the beginning of the race to help tailor the PID for the first track.

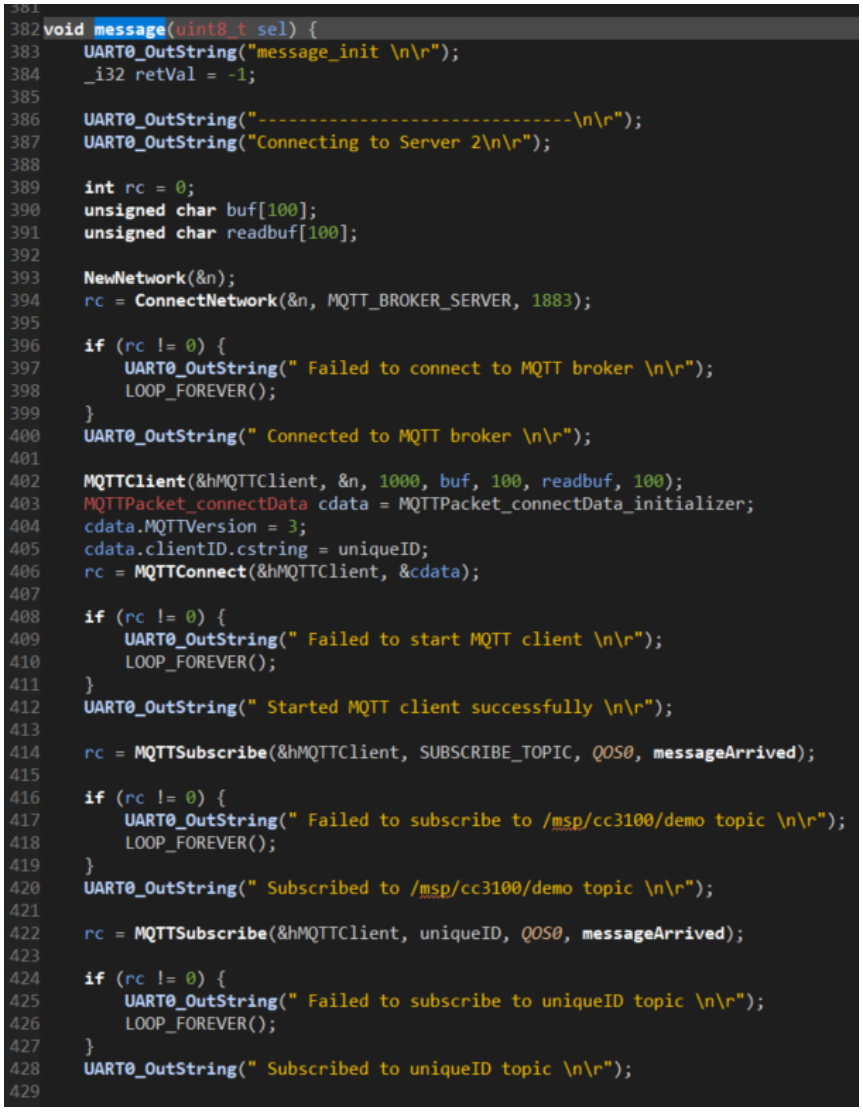
3 Complete Description of Code implemented.

3.1 JJ

I started the project by creating a fork off of Team Olimar's line follower repository. This gave the team a good jumping off point as that project already had numerous components integrated such as the hardware timer motors and reflectance sensor. Then I added in the demo code Lab21_Solutions from this project, the Tachometer files from lab5, and the MQTT demos from lab6.

Now that the project was creating it was time to start implementing the MQTT stuff. The first step was converting the mqtt_main main() function into an MQTT initialization function for this project. That was essentially just removing some extra initialization stuff and the while(1) from the function.

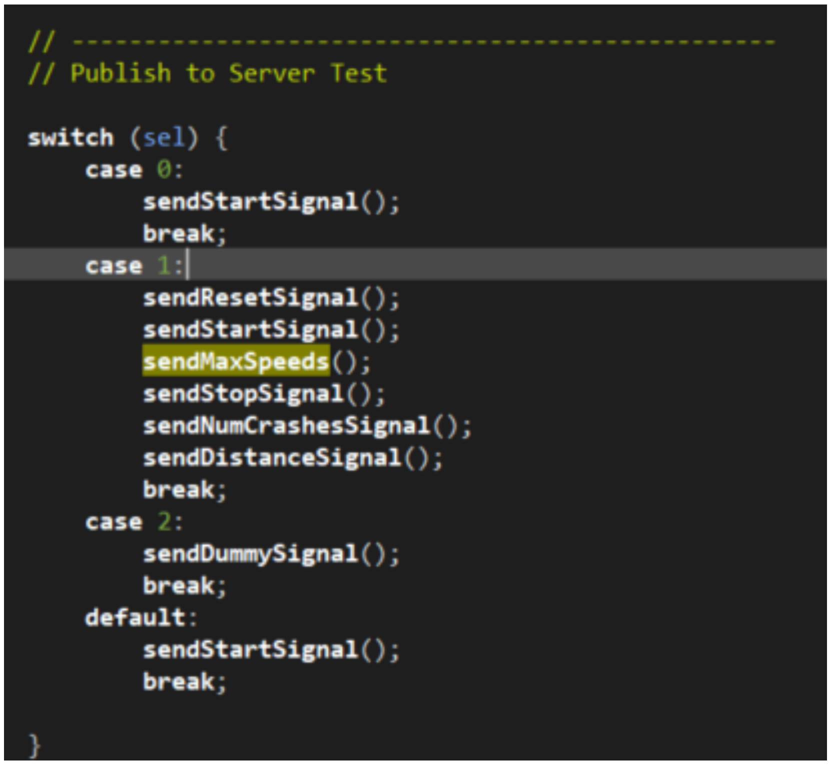
The next step in the process is figuring out how to send additional messages to the MQTT broker after connecting. This ended up being a difficult problem to figure out as the MQTT code would fail pretty much every time it wen to send a message after the first initialization. The solution to this ended up being remembering to reconnect to the MQTT broker every time we wanted to send a new message. This code was housed in a function named message, which handled all of the additional MQTT messages after initialization for the project.



```
void message(uint8_t sel) {
    UART0_OutString("message_init \n\r");
    _i32 retVal = -1;

    UART0_OutString("-----------------------------\n\r");
    UART0_OutString("Connecting to Server 2\n\r");

    int rc = 0;
    unsigned char buf[100];
    unsigned char readbuf[100];

    NewNetwork(&n);
    rc = ConnectNetwork(&n, MQTT_BROKER_SERVER, 1883);

    if (rc != 0) {
        UART0_OutString(" Failed to connect to MQTT broker \n\r");
        LOOP_FOREVER();
    }
    UART0_OutString(" Connected to MQTT broker \n\r");

    MQTTClient(&hMQTTClient, &n, 1000, buf, 100, readbuf, 100);
    MQTTPacket_connectData cdata = MQTTPacket_connectData_initializer;
    cdata.MQTTVersion = 3;
    cdata.clientID.cstring = uniqueID;
    rc = MQTTConnect(&hMQTTClient, &cdata);

    if (rc != 0) {
        UART0_OutString(" Failed to start MQTT client \n\r");
        LOOP_FOREVER();
    }
    UART0_OutString(" Started MQTT client successfully \n\r");

    rc = MQTTSubscribe(&hMQTTClient, SUBSCRIBE_TOPIC, QOS0, messageArrived);

    if (rc != 0) {
        UART0_OutString(" Failed to subscribe to /msp/cc3100/demo topic \n\r");
        LOOP_FOREVER();
    }
    UART0_OutString(" Subscribed to /msp/cc3100/demo topic \n\r");

    rc = MQTTSubscribe(&hMQTTClient, uniqueID, QOS0, messageArrived);

    if (rc != 0) {
        UART0_OutString(" Failed to subscribe to uniqueID topic \n\r");
        LOOP_FOREVER();
    }
    UART0_OutString(" Subscribed to uniqueID topic \n\r");
```

Figure 1: MQTT Reconnecting

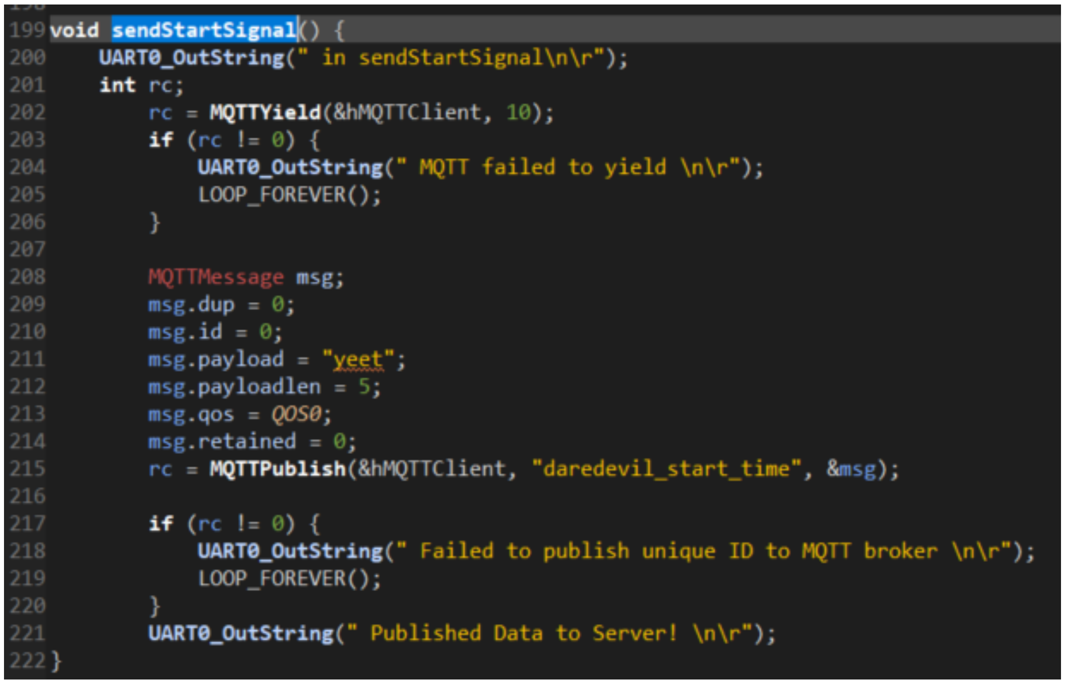
Message then also head a switch statement that allowed the team to send various different signals to the MQTT broker depending on the parameter to message. The two main scenarios where we had to publish data to the MQTT broker are at the start and end of the race. We chose to send a start signal to the broker because this allowed the Node Red server to log a time stamp of the start time and ultimately calculate the race duration when it got the end time stamp. The following figure highlights the switch statement. There is a lot more data being sent at the end, which makes sense because we need to send all of the race statistics instead of just a start signal.



```
// ------------------------------------------------
// Publish to Server Test

switch (sel) {
    case 0:
        sendStartSignal();
        break;
    case 1:
        sendResetSignal();
        sendStartSignal();
        sendMaxSpeeds();
        sendStopSignal();
        sendNumCrashesSignal();
        sendDistanceSignal();
        break;
    case 2:
        sendDummySignal();
        break;
    default:
        sendStartSignal();
        break;

}
```

Figure 2: Message Switch Logic

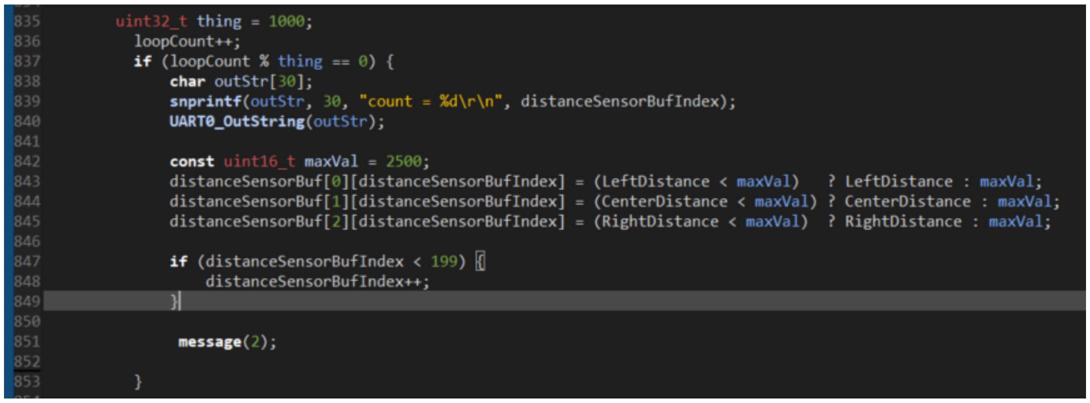
The team pretty much used a different mqtt topic for all of the different data points being logged. This ended up being very efficient and easy to work with as certain signals like the start time have a dummy payload, and the Node red server only needs to know if a signal was sent.



```
void sendStartSignal() {
    UART0_OutString(" in sendStartSignal\n\r");
    int rc;
        rc = MQTTYield(&hMQTTClient, 10);
        if (rc != 0) {
            UART0_OutString(" MQTT failed to yield \n\r");
            LOOP_FOREVER();
        }

        MQTTMessage msg;
        msg.dup = 0;
        msg.id = 0;
        msg.payload = "yeet";
        msg.payloadlen = 5;
        msg.qos = QOS0;
        msg.retained = 0;
        rc = MQTTPublish(&hMQTTClient, "daredevil_start_time", &msg);

        if (rc != 0) {
            UART0_OutString(" Failed to publish unique ID to MQTT broker \n\r");
            LOOP_FOREVER();
        }
        UART0_OutString(" Published Data to Server! \n\r");
}
```

Figure 3: start signal

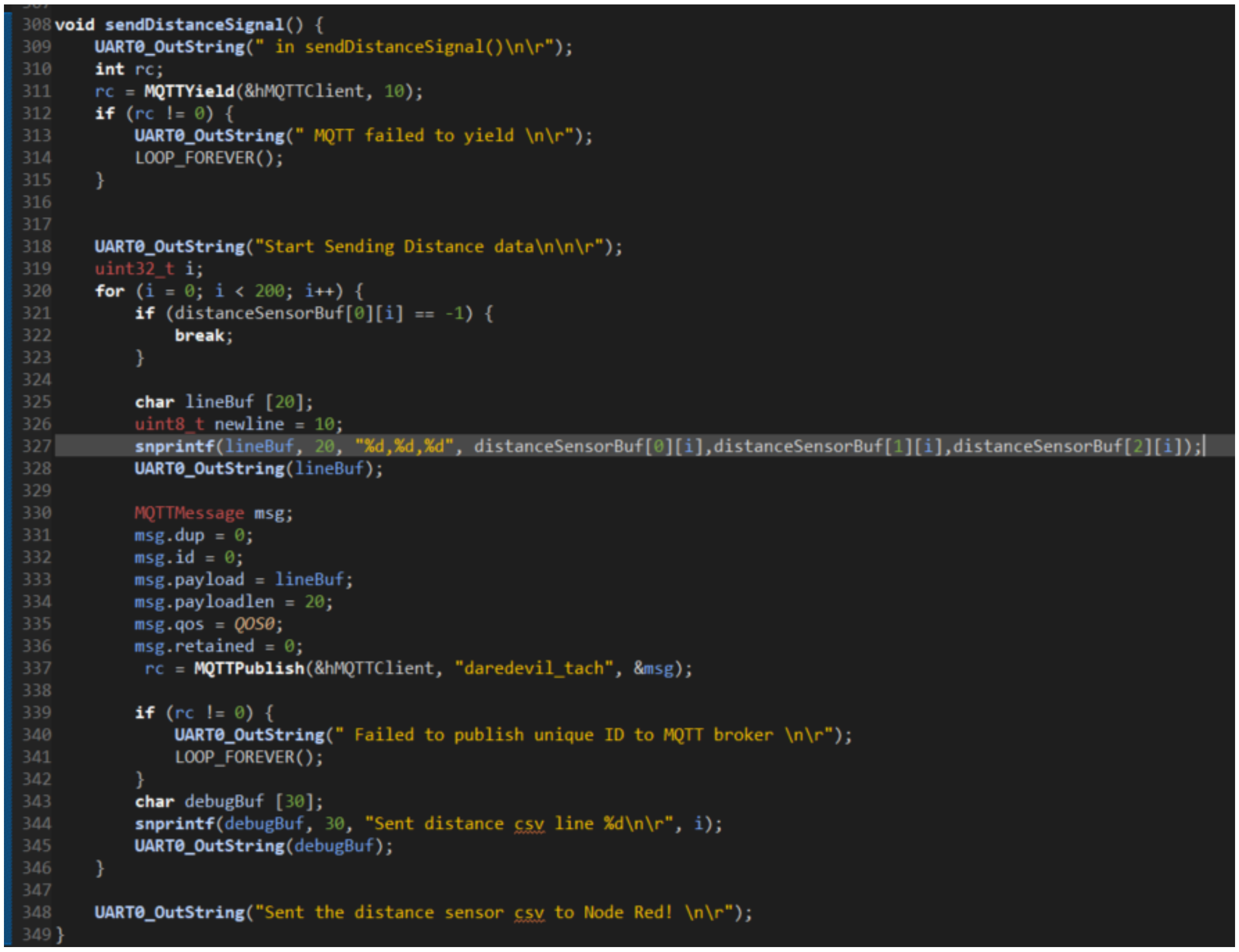
Some race statistic information was more challenging to log and send. The most challenging statistic ended up being managing the distance sensor changing overtime. I created a 2d array with the rows being the left, middle, and right distance sensor readings and the columns being up to 200 data points that can be logged. We logged this distance sensor data every 1000 main loop iterations, which ended up being roughly once a second. This led to us being able to log just over 3 minutes of distance sensor data before running out of space.



```
        uint32_t thing = 1000;
          loopCount++;
          if (loopCount % thing == 0) {
              char outStr[30];
              snprintf(outStr, 30, "count = %d\r\n", distanceSensorBufIndex);
              UART0_OutString(outStr);

              const uint16_t maxVal = 2500;
              distanceSensorBuf[0][distanceSensorBufIndex] = (LeftDistance < maxVal)   ? LeftDistance : maxVal;
              distanceSensorBuf[1][distanceSensorBufIndex] = (CenterDistance < maxVal) ? CenterDistance : maxVal;
              distanceSensorBuf[2][distanceSensorBufIndex] = (RightDistance < maxVal)  ? RightDistance : maxVal;

              if (distanceSensorBufIndex < 199) {
                  distanceSensorBufIndex++;
              }

               message(2);

          }
```

Figure 4: Distance Sensor Logging

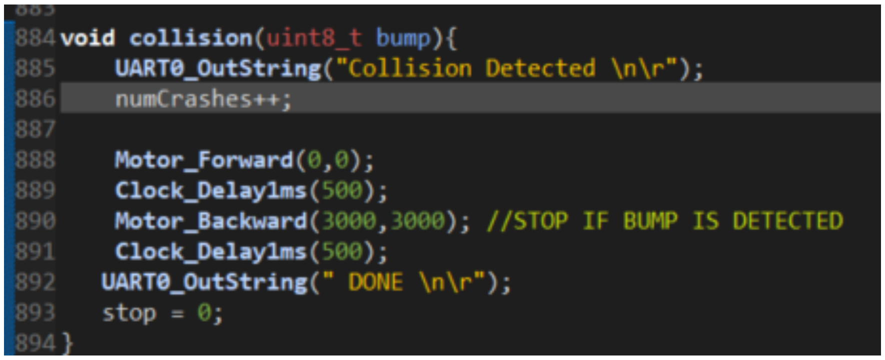
Sending the distance sensor data was also challenging. The team liked the idea of sending a CSV file of the distance sensor array as it would be easy to parse and plot on the Node red side. Unfortunately, there was issues sending a CSV file that was up to 200 lines long all at once, so the team opted to publish the CSV file one line at a time to the MQTT broker, and then add extra logic on the node red side to handle plotting the distance data over time. This ended up being very effective and worked on both ends of the project.



```c
void sendDistanceSignal() {
    UART0_OutString(" in sendDistanceSignal()\n\r");
    int rc;
    rc = MQTTYield(&hMQTTClient, 10);
    if (rc != 0) {
        UART0_OutString(" MQTT failed to yield \n\r");
        LOOP_FOREVER();
    }


    UART0_OutString("Start Sending Distance data\n\n\r");
    uint32_t i;
    for (i = 0; i < 200; i++) {
        if (distanceSensorBuf[0][i] == -1) {
            break;
        }

        char lineBuf [20];
        uint8_t newline = 10;
        snprintf(lineBuf, 20, "%d,%d,%d", distanceSensorBuf[0][i],distanceSensorBuf[1][i],distanceSensorBuf[2][i]);
        UART0_OutString(lineBuf);

        MQTTMessage msg;
        msg.dup = 0;
        msg.id = 0;
        msg.payload = lineBuf;
        msg.payloadlen = 20;
        msg.qos = QOS0;
        msg.retained = 0;
         rc = MQTTPublish(&hMQTTClient, "daredevil_tach", &msg);

        if (rc != 0) {
            UART0_OutString(" Failed to publish unique ID to MQTT broker \n\r");
            LOOP_FOREVER();
        }
        char debugBuf [30];
        snprintf(debugBuf, 30, "Sent distance csv line %d\n\r", i);
        UART0_OutString(debugBuf);
    }

    UART0_OutString("Sent the distance sensor csv to Node Red! \n\r");
}
```

Figure 5: Publishing Distance Data

Finally, I also handled the crashing logic. This was very easy to implemented as all of the pieces were already setup for it work. I simply reused the BumpInt interrupt from the line follower code and changed the handler function to meet the requirements. They increased the number of crashes by one, stopping the robot for 500 ms, and then reversing for 500 ms to hopefully recover from the crash.



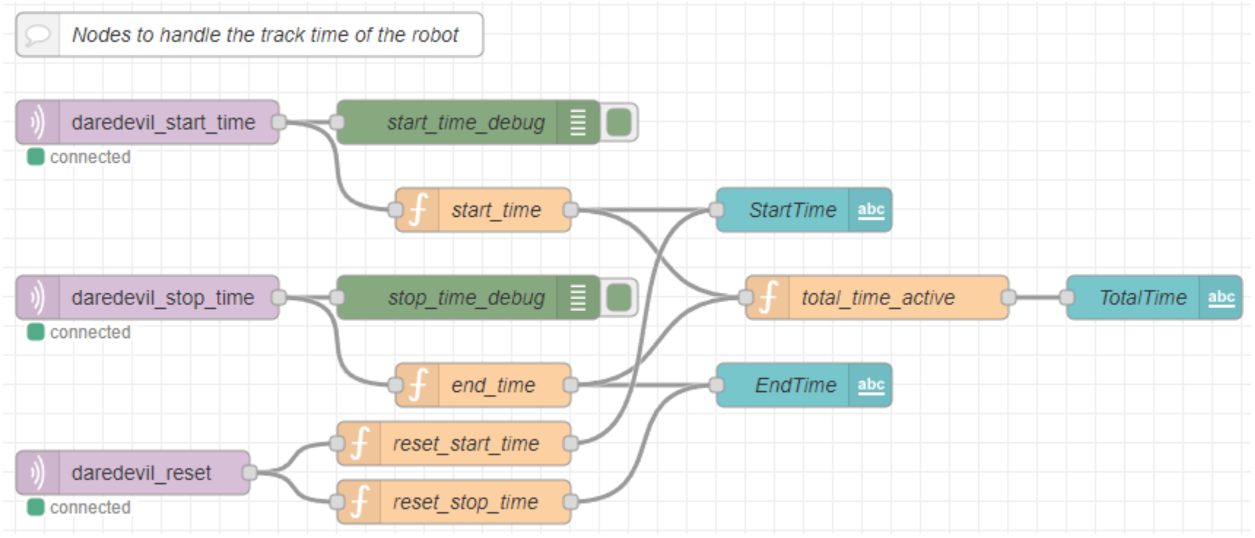
```c
void collision(uint8_t bump){
    UART0_OutString("Collision Detected \n\r");
    numCrashes++;

    Motor_Forward(0,0);
    Clock_Delay1ms(500);
    Motor_Backward(3000,3000); //STOP IF BUMP IS DETECTED
    Clock_Delay1ms(500);
   UART0_OutString(" DONE \n\r");
   stop = 0;
}
```

Figure 6: Collision

Section 4: Video Demonstrations of tests performed to validate code. Contains video proof of all the code explained above working properly

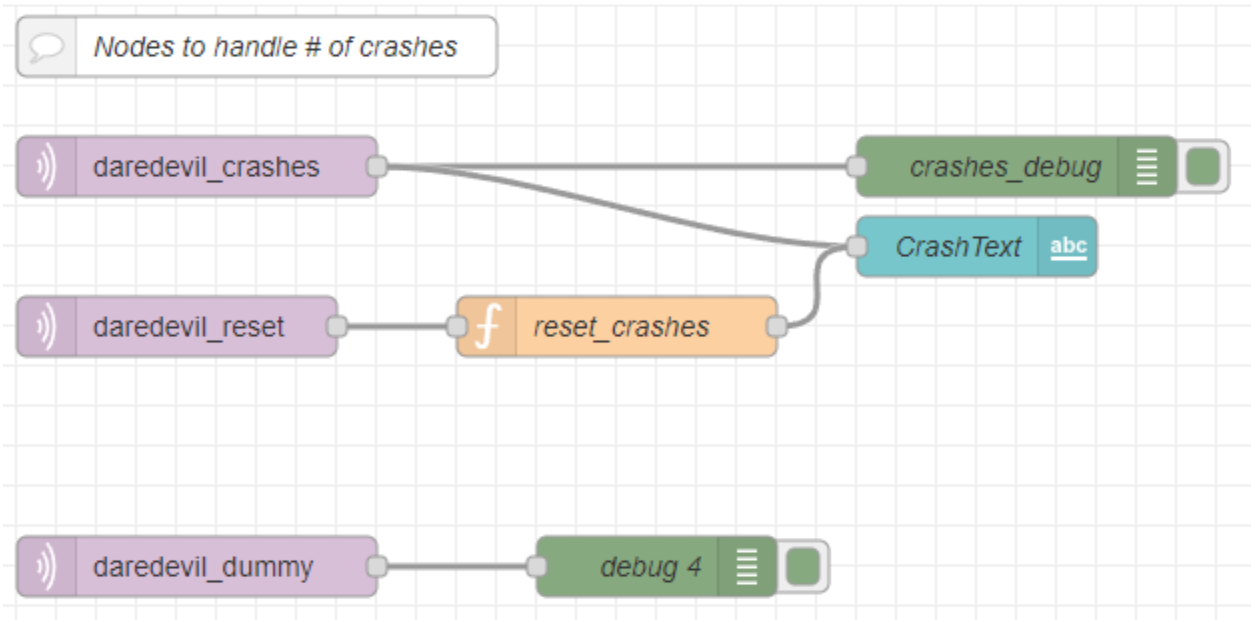
3.2 Adam

The node-red server is subscribed to unique topics for each data metric we were sharing, making it easier to receive and eliminates the need for parsing a long json file. The bot will publish to the start time topic at the beginning of the race to log the start time, then after the race is over it will publish to all other topics. A reset topic is also in place for all sections in the node-red server which will undo any published previously made. This is called upon each robot restart which allows only the current race telemetry to be viewed.



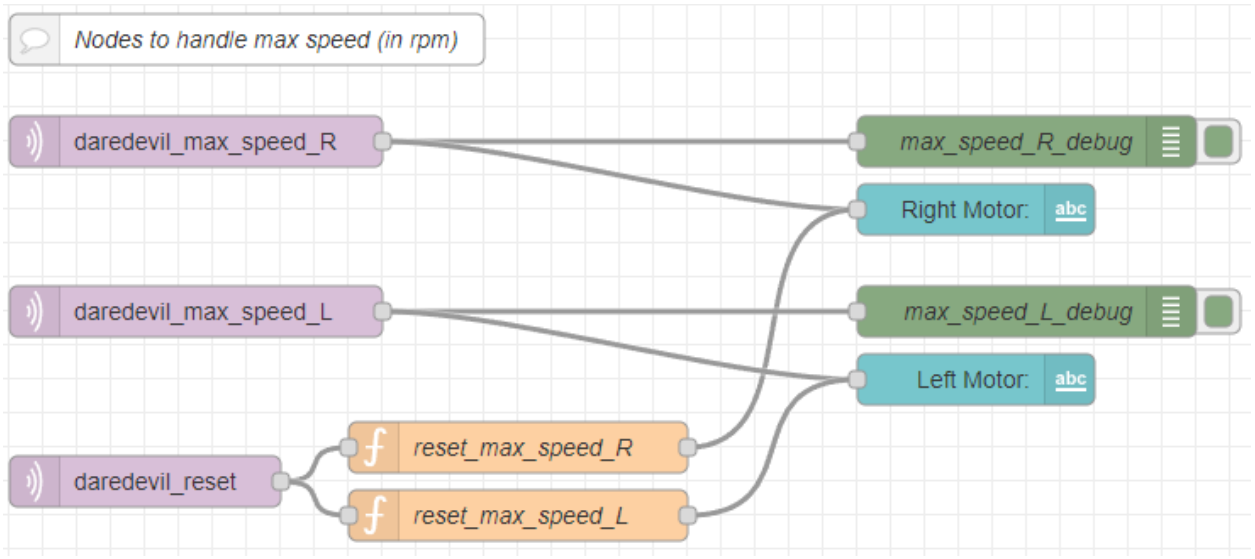
Node-Red Nodes to Handle Robot Track Time

Whenever the robot receives a bluetooth command to begin moving the bot will publish to the 'daredevil_start_time' topic. This triggers the function 'start_time' which records the system time of the computer and logs that to a flow variable as well as displays that time through the text dashboard node. The 'daredevil_stop_time' node works in the exact same fashion, whenever the bot senses the finish line or is told to stop by the user via bluetooth the topic will trigger a function to record the time that the bot stopped and update a flow variable. Once both start and stop time flow variables are defined the total_time_active function will read the times for both and calculate the difference to display the total time in seconds that the bot was active on the course.



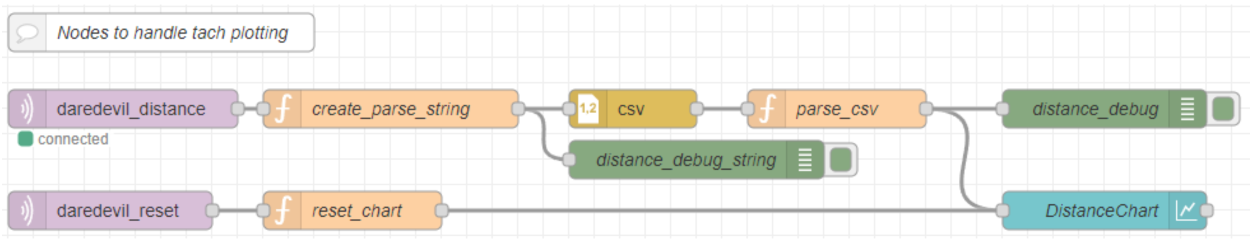
Node-Red Nodes to Handle Robot Crashes

Every time the bot's bump sensors are triggered the collision handler increments the total bumps by 1, then continues into its race bump logic. Once the race is over the bot will receive the published crashes and display that to the text dashboard node. The dummy topic was mainly used for debugging purposes to see if the bot was staying connected to the MQTT broker throughout the entire race.



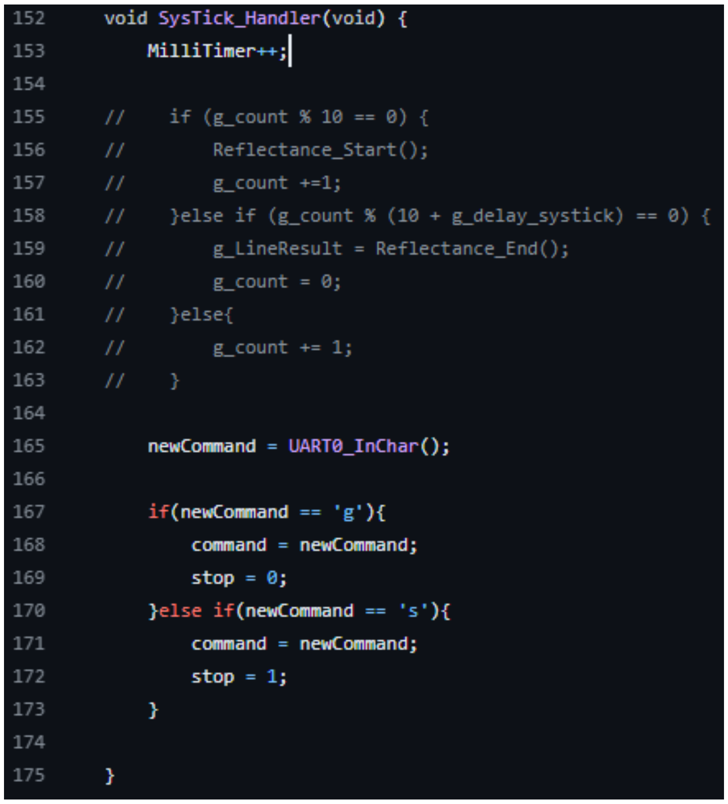
Node-Red Nodes to Handle Robot's Max Speed

The max speed nodes work similarly to the crash nodes, the bot will keep track of the highest speed each wheel reached during the race and hold that until the end once they are published to the speed topics. These are then also displayed on the dashboard through text nodes.



Node-Red Nodes to Handle Robot's Distance Sensor Charting

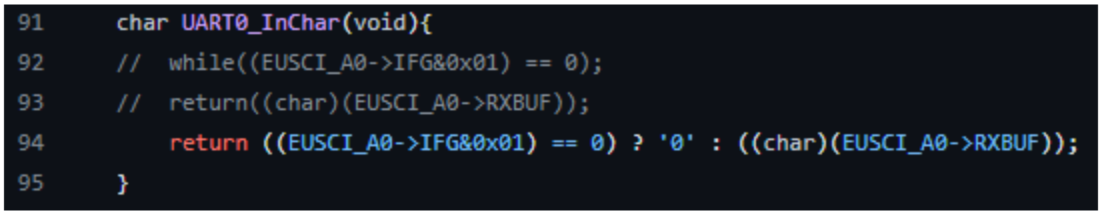
Once again, when the bot finishes the race or is told to stop it will publish all of the distance sensor logging to the topic. What the bot will do is publish each sampled distance sensor reading line by line in csv format, the create_parse_string function reads the message payload and appends that to a string to create a long string formatted in csv. This is then passed through through the csv node to produce a single array message that is then parsed in the parse_csv function. This reads each column and puts the respective data into an object that is then passed to the chart node to be displayed on the dashboard.



```
void SysTick_Handler(void) {
    MilliTimer++;

//    if (g_count % 10 == 0) {
//        Reflectance_Start();
//        g_count +=1;
//    }else if (g_count % (10 + g_delay_systick) == 0) {
//        g_LineResult = Reflectance_End();
//        g_count = 0;
//    }else{
//        g_count += 1;
//    }

    newCommand = UART0_InChar();

    if(newCommand == 'g'){
        command = newCommand;
        stop = 0;
    }else if(newCommand == 's'){
        command = newCommand;
        stop = 1;
    }

}
```

BLE Communication

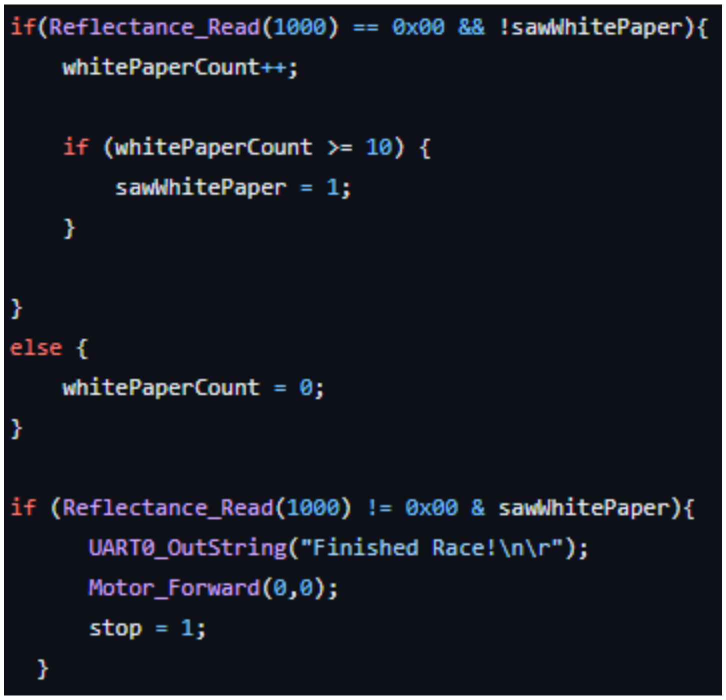
The main driver for the BLE communication is the SysTick Handler. Every time the handler is called a new char is read in via the Bluetooth module connected to the robot. Depending on if the char is 'g' for go or 's' for stop it will be assigned to command variable and then used in the main loop to determine whether the bot is able to drive or if it is stationary.



```
char UART0_InChar(void){
//  while((EUSCI_A0->IFG&0x01) == 0);
//  return((char)(EUSCI_A0->RXBUF));
    return ((EUSCI_A0->IFG&0x01) == 0) ? '0' : ((char)(EUSCI_A0->RXBUF));
}
```

New UART0_InChar Function

One main change that needed to be made in the UART library was in the InChar function. Typically the function will hang until a character is received and then it will proceed. However, if the code hangs in the Bluetooth communication the PID will never be allowed to update and the bot will be stuck in a single state and not be able to make corrections. To combat this, I created a ternary to see if a character was read, if so it will return that character and if not return a base character '0'. This gets rid of blocking code and allows the PID to continue running even if no Bluetooth communication is active.



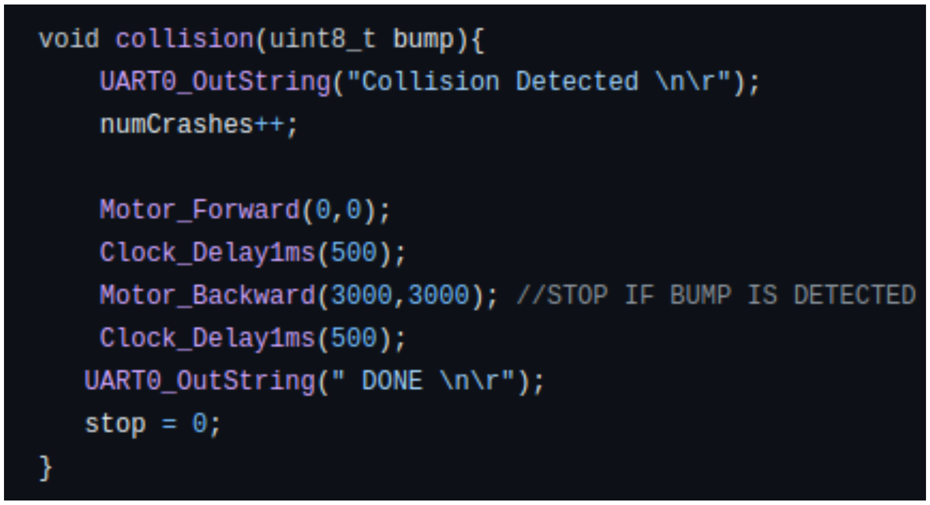
```
if(Reflectance_Read(1000) == 0x00 && !sawWhitePaper){
    whitePaperCount++;

    if (whitePaperCount >= 10) {
        sawWhitePaper = 1;
    }

}
else {
    whitePaperCount = 0;
}

if (Reflectance_Read(1000) != 0x00 & sawWhitePaper){
      UART0_OutString("Finished Race!\n\r");
      Motor_Forward(0,0);
      stop = 1;
  }
```

Line Sensing Logic

Within our main loop, if the bot is moving (received a 'g' character via bluetooth) the reflectance sensors are checked at the beginning of each loop. At a high level, the reflectance sensors will read the white paper to sense that the finish line is near and once the black tape is read the bot will stop. To get rid of erroneous readings we implemented a counter to ensure the reflectance sensors are reading the white paper. This counter will increment to 10 only when the bot reads the white paper, after that a 'sawWhitePaper' flag is set and the next time the sensors read something that is not white the bot will stop and upload any telemetry to the IOT dashboard.

3.3 Troy

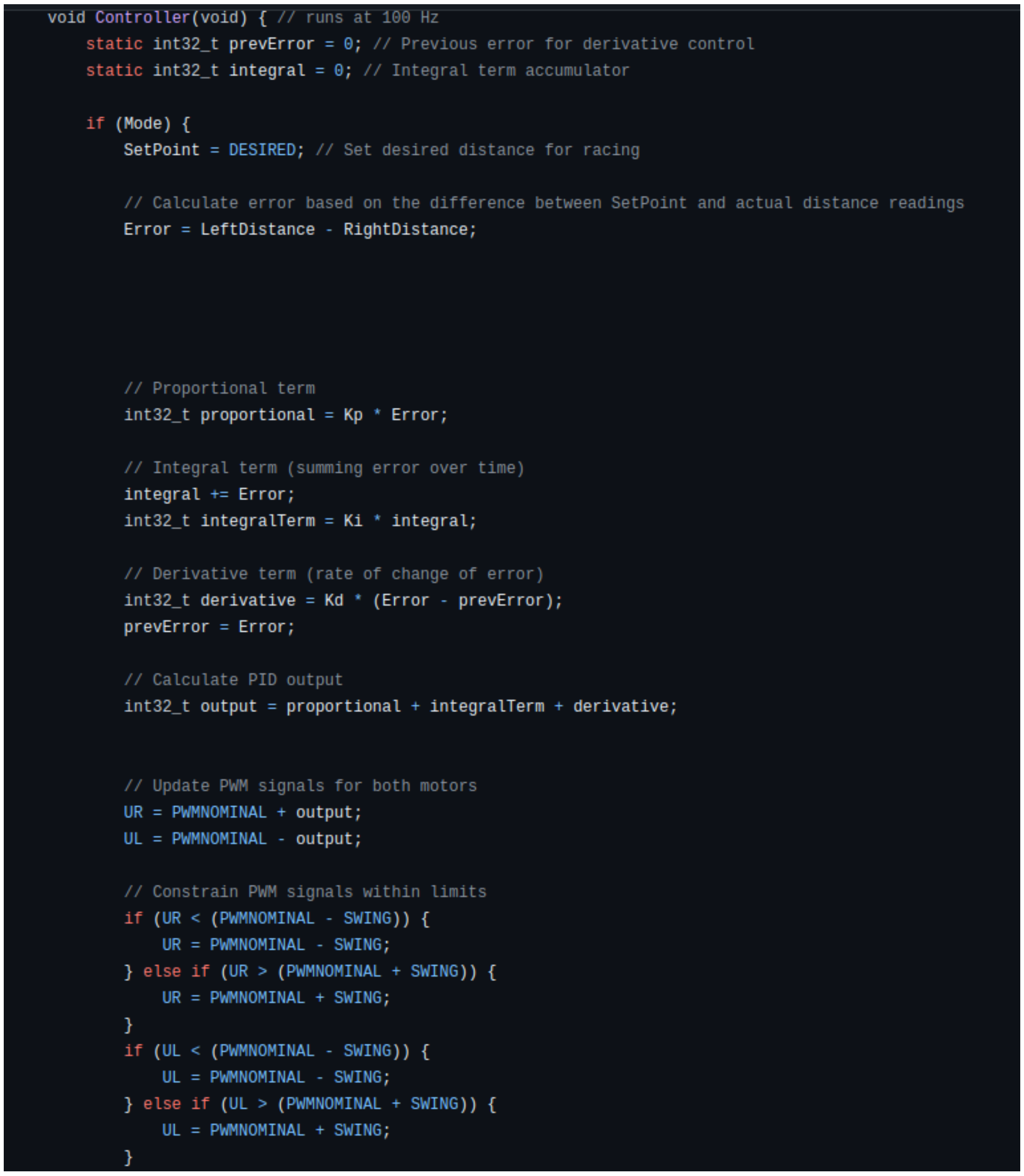
I was tasked with fully implementing a PID controller and creating a robot that raced well, so I only wrote code during pre-integration. Rapid testing called for ease of use which meant it was much easier to test the robot manually without having to use Bluetooth. Before creating the PID controller, I implemented a collision function to allow for the bump sensors to work.



```
void collision(uint8_t bump){
    UART0_OutString("Collision Detected \n\r");
    numCrashes++;

    Motor_Forward(0,0);
    Clock_Delay1ms(500);
    Motor_Backward(3000,3000); //STOP IF BUMP IS DETECTED
    Clock_Delay1ms(500);
   UART0_OutString(" DONE \n\r");
   stop = 0;
}
```

Collision Interrupt Function

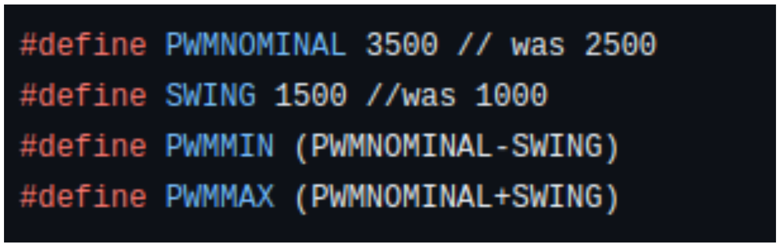
This collision function shows all of the post integration functionality which was mostly written by Adam and JJ. I just structured the files to allow for an interrupt to trigger when a bump sensor is pushed. Adam and JJ decided on the logic for what happened after a collision was detected. The next step in the process was creating a PID controller. To do this, I refactored the controller() function given to us in the example code. This code had already integrated the distance sensing aspect of the PID, which made it very easy to focus on the logic. Below is the initial PID controller function refactored for this project.



```
void Controller(void) { // runs at 100 Hz
    static int32_t prevError = 0; // Previous error for derivative control
    static int32_t integral = 0; // Integral term accumulator

    if (Mode) {
        SetPoint = DESIRED; // Set desired distance for racing

        // Calculate error based on the difference between SetPoint and actual distance readings
        Error = LeftDistance - RightDistance;





        // Proportional term
        int32_t proportional = Kp * Error;

        // Integral term (summing error over time)
        integral += Error;
        int32_t integralTerm = Ki * integral;

        // Derivative term (rate of change of error)
        int32_t derivative = Kd * (Error - prevError);
        prevError = Error;

        // Calculate PID output
        int32_t output = proportional + integralTerm + derivative;


        // Update PWM signals for both motors
        UR = PWMNOMINAL + output;
        UL = PWMNOMINAL - output;

        // Constrain PWM signals within limits
        if (UR < (PWMNOMINAL - SWING)) {
            UR = PWMNOMINAL - SWING;
        } else if (UR > (PWMNOMINAL + SWING)) {
            UR = PWMNOMINAL + SWING;
        }
        if (UL < (PWMNOMINAL - SWING)) {
            UL = PWMNOMINAL - SWING;
        } else if (UL > (PWMNOMINAL + SWING)) {
            UL = PWMNOMINAL + SWING;
        }
```

PID Controller Function

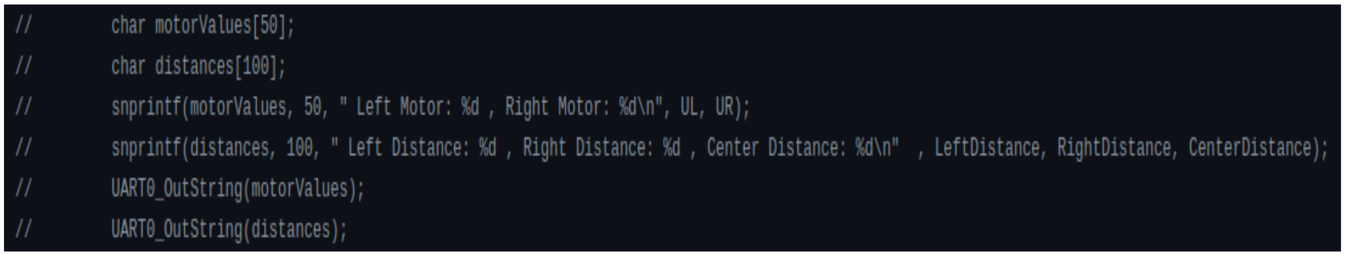
The initial PID controller calculates error by finding the difference between the distance sensor readings. Initially there was a Setpoint variable that was going to be used in logic created for maintaining a specific distance from the wall. This was carryover from the wall follower function in the sample code. However, it was decided that better performance was observed when the error was calculated using the distance sensor readings. The proportional, integral, and derivative terms were calculated using gain values and the error. The error value was modified correctly to account for the accumulating error and rate of error change in the integral and derivative terms. The PWM values for the motors were then updated based on the PID, however, they were constrained within specific limits to make sure the motors were not asked to do anything too extreme. The variable values used in constraining are shown below.



```
#define PWMNOMINAL 3500 // was 2500
#define SWING 1500 //was 1000
#define PWMMIN (PWMNOMINAL-SWING)
#define PWMMAX (PWMNOMINAL+SWING)
```

PWM Constrained Values

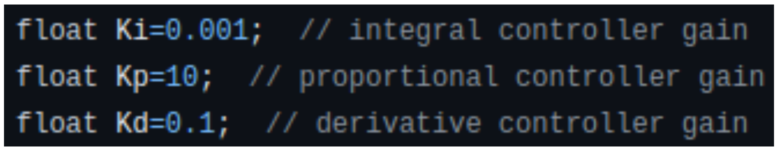
As the values show. The nominal PWM value for the motors is 3500. This means that with no error, the motors should use 3500 as the PWM values. The swing variable means that the PWM values will never be more or less than the nominal by 1500. The minimum and maximum PWM values are therefore 2000 and 5000 respectively. The next step in the process was to tune the PID gain variables. Since I am not super good at tuning, I decided that it would be best to take a systematic approach to tuning. So, instead of using the robot performance to tune, I printed the PWM motor values out to a serial terminal using UART. This allowed me to tune in the comfort of a chair as I could just use my hand to trigger different distance sensor values and see how the PWM values changed.



```
//        char motorValues[50];
//        char distances[100];
//        snprintf(motorValues, 50, " Left Motor: %d , Right Motor: %d\n", UL, UR);
//        snprintf(distances, 100, " Left Distance: %d , Right Distance: %d , Center Distance: %d\n"  , LeftDistance, RightDistance, CenterDistance);
//        UART0_OutString(motorValues);
//        UART0_OutString(distances);
```

UART Tuning Code

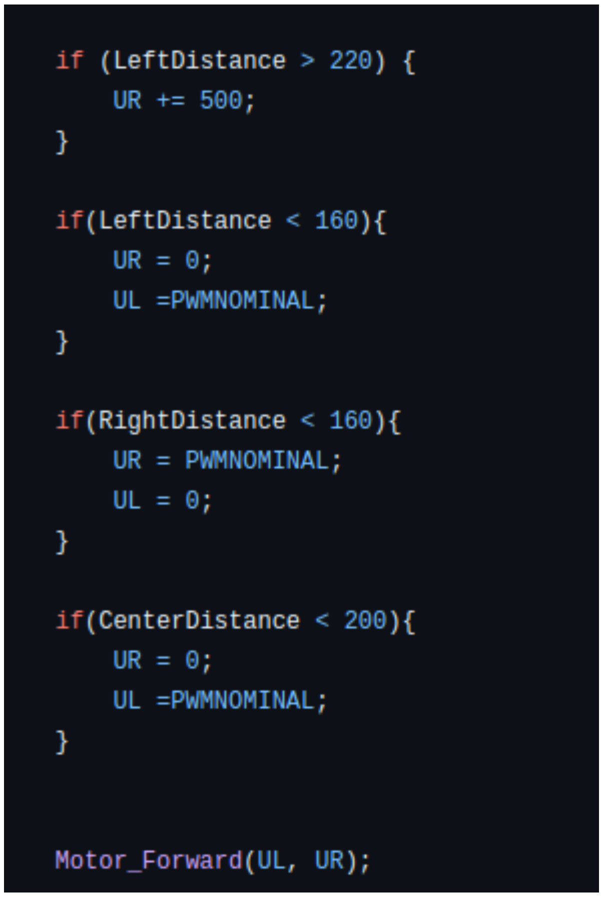
Shown above is the commented-out code to allow for this easy tuning process. By tuning this way, I was able to achieve the following gain values that resulted in decent performance.



```
float Ki=0.001;  // integral controller gain
float Kp=10;  // proportional controller gain
float Kd=0.1;  // derivative controller gain
```

Gain Values

A large proportional gain was helpful in ensuring that the PWM values quickly changed based on the distance sensor readings. A super low integral gain was used because too much caused steady state error (robot spinning in circles) and too little did not support the quickness of the PWM value changes. The derivative gain was less measurable, however a smaller value of 0.1 seemed to work the best. The last part of the PID controller development was creating logic to account for the first track of the race. The first track of the race included a sharp left turn, so I decided to add a bias to the left to keep the robot close to the left wall and make it turn sharp when it was not close.



```
if (LeftDistance > 220) {
    UR += 500;
}

if(LeftDistance < 160){
    UR = 0;
    UL =PWMNOMINAL;
}

if(RightDistance < 160){
    UR = PWMNOMINAL;
    UL = 0;
}

if(CenterDistance < 200){
    UR = 0;
    UL =PWMNOMINAL;
}


Motor_Forward(UL, UR);
```

Gain Values

The top if statement was used to create a sharp turn when the left wall was not close to the robot. The bottom 3 if statements are catch all cases that stop the robot and force it to turn sharply if it is extremely close to an object. With these implemented into the PID controller function, the code was ready to be integrated into the full program that allowed for Bluetooth operation and Wi-Fi communication.

4 Video Demonstrations of tests performed to validate code

Video of preliminary testing with BLE controls and PID here.

Video of BLE being used to start and stop the robot, bump sensors pausing then backing up before continuing forward, and after the bot is told to stop automatically uploading telemetry to the IOT dashboard here.

Video of racing robot stopping on the black line is here.

5 Summary of changes after integration

Through the first iteration of the node-red distance charting, we thought we would be able to send a large string and immediately parse it and send it to the chart node. However, issues arose when the char buffer we were using was over ~80, this allowed us to only read 3 or 4 samples which wasn't sufficient. To bypass this, we still logged all the data into a char buffer but instead sent one line at a time as to not overload the MQTT communication. This allowed the node-red server to create the csv string in the front end and then pass it to the csv node to chart.

There were some issues integrating everything together with the PID logic. One thing we did not account for after integration is tuning was no longer possible without using bluetooth to start the robot. Since tuning was still important to focus on as the race drew closer, we had to work with github to create a branch that allowed us to go back to a former pre-integration state so that the PID could be worked on right up to the race.

Another key change that the team had to make when integrating the IOT stuff with the PID logic was removing extra delays and blocking code. We had a reconnection loop within the main loop that ensured we were not losing connection to the MQTT broker. This was necessary as there was issues sending data at the end of the race it we were not consistently staying connected after the initialization. Unfortunately, this code takes a decent bit of time to run and it was running pretty much all the time. This led to the PID control loop breaking as the main loop was no longer running fast enough for the PID to make intelligent decisions. To solve this problem, we decided to simply redo all of the MQTT initialization stuff at the end of the race, instead of staying connected during it. This means that it would take ~15+ seconds to send the data at the end now, but that was fine for the scope of this project.

6 Video of how the robot performed on the track

Race Video: https://youtu.be/kbXcliuV7Cw

7 Reflection on the project outcomes

Overall, the team is satisfied with the project outcomes. The timeline felt significantly shorter than the line follower likely due to the nature of finals week chaos. The team was able to achieve all of the main requirements and pass the checkoffs before the race. We did not complete the first track of the race due to the robot getting stuck in a PID steady state where the robot would continuously turn in circles. Fortunately, we did have some good runs on the main tracks, they just did not happen on race day. It seemed like teams that used a wall follower as the basis of their code had a lot of success, so the team could either pivot to that or further tune the PID if we wanted to improve the race performance in the future.

8 Repository Link

GitHub: https://github.com/jjo527/ece1188-autonomous-racing

Node-Red: http://4.234.184.53:1880/#flow/cb01bb56cebeb30b

Node-Red Dashboard: http://4.234.184.53:1880/ui/#!/0?socketid=P-Jlev1PT6fXgVNZAAAL