ECE 1188

Group Gundam

Group members: Cameron Kirsch(Wall-Follow Controller, MQTT), Yuheng Lin(Bluetooth, IoT Dashboard), Jiahao Li(bump)

Autonomous Racing Report

Overview

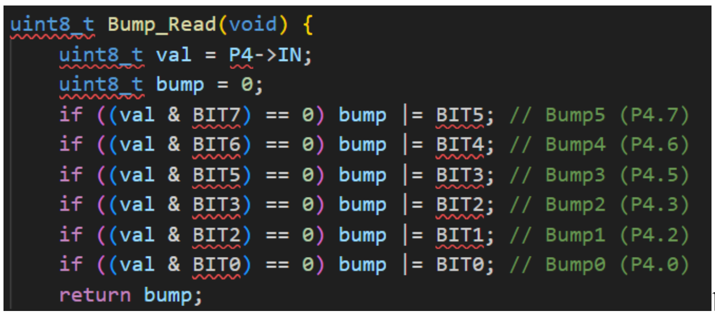
The design of our robot is to compromise speed for stability and visibility. The robot is programmed with two different wall-following logic that can be used interchangeably depending on which one would be the most performant for the portion of the race. The metrics of the robot could be seen on an IoT dashboard communicated through the Wi-Fi module on the robot connected to an MQTT broker.

Bump Sensor Integration and Collision Recovery Logic

This work extends the original Line Following Race firmware by integrating six bump sensors as a secondary safety mechanism. While the robot primarily relies on a front-facing time-of-flight (ToF) distance sensor for obstacle avoidance, certain environmental conditions—such as non-reflective surfaces or lateral impacts from other robots—can cause the ToF sensor to fail. In such cases, bump sensors provide a reliable fallback, allowing the robot to detect and respond to physical collisions that optical sensing cannot resolve.

Hardware Configuration

Six active-low bump sensors are connected to GPIO Port 4, with internal pull-up resistors maintaining a HIGH level (3.3 V) when not pressed. Each pin is configured for falling-edge interrupt triggering, which detects contact as the signal transitions from HIGH to LOW. Upon detection, the pin's interrupt edge is switched to rising to capture the release event. Once released, the pin is reconfigured to falling-edge to detect the next collision. This toggle mechanism ensures accurate registration of both press and release events.



```c
uint8_t Bump_Read(void) {
    uint8_t val = P4->IN;
    uint8_t bump = 0;
    if ((val & BIT7) == 0) bump |= BIT5; // Bump5 (P4.7)
    if ((val & BIT6) == 0) bump |= BIT4; // Bump4 (P4.6)
    if ((val & BIT5) == 0) bump |= BIT3; // Bump3 (P4.5)
    if ((val & BIT3) == 0) bump |= BIT2; // Bump2 (P4.3)
    if ((val & BIT2) == 0) bump |= BIT1; // Bump1 (P4.2)
    if ((val & BIT0) == 0) bump |= BIT0; // Bump0 (P4.0)
    return bump;
```

bump data read

Edge-Triggered Interrupt Implementation

The corresponding interrupt is configured to respond to falling edges. Pressing any bump sensor results in an immediate interrupt, which is handled in the ISR. Within the interrupt service routine, the system identifies which sensor or sensors were pressed, clears the interrupt flags, updates the collision tracking variables, and sets a global notification flag to inform the main control loop that a physical collision has occurred. The pressed pin is then temporarily set to trigger on rising edge to detect the release event. Upon release, it is switched back to falling edge detection.

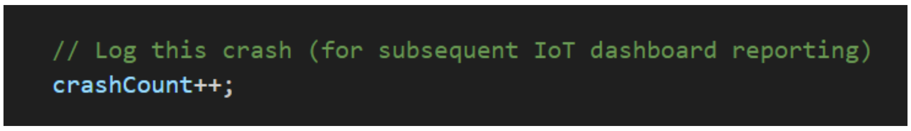
Collision Handling and Recovery

When a collision is detected via bump sensors, the interrupt routine invokes a collision response mechanism. First, the robot halts its current action and waits briefly to avoid reacting too early to sensor noise or bounce. Next, the system determines which sensors were activated, categorizing them into left and right groups: Bump0 to Bump2 represent the left side, while Bump3 to Bump5 represent the right. Based on the count of activated sensors on each side, the robot performs an appropriate maneuver.

If both left and right sensors are triggered, the robot interprets this as a central collision and reverses to create space. It then turns toward the side with fewer active bump sensors. If only one side is triggered, the robot turns in the opposite direction to avoid the obstacle. In ambiguous or lightly triggered cases, the robot simply backs up slightly to disentangle itself. After the robot moves away from the obstacle and the bump sensor is released, the edge-detection settings are reset for the next press.

IoT Dashboard Integration

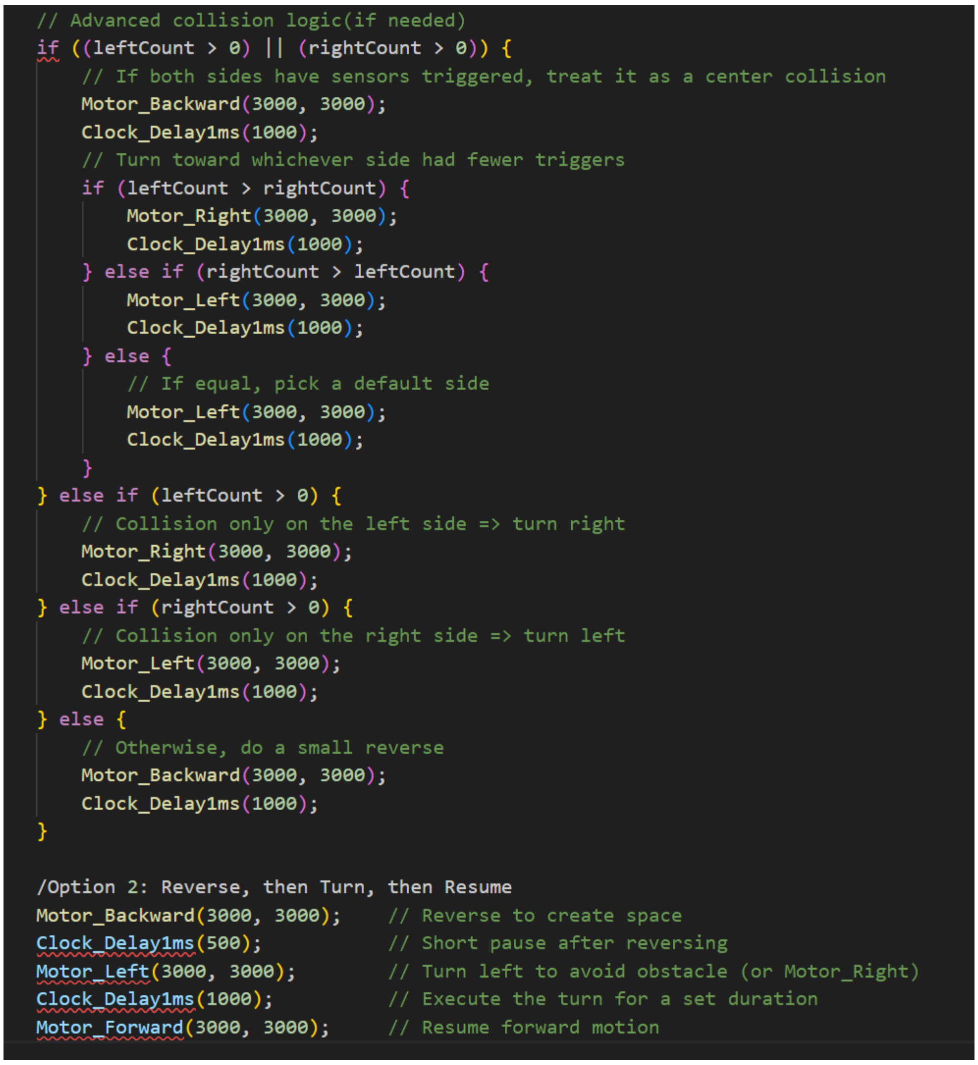
To support remote monitoring and diagnostics, a global counter has been added to track the number of collision events. This allows the robot to communicate the total number of physical contacts to an IoT dashboard for performance analysis or debugging purposes. The counter is incremented within the interrupt handler and defined as follows:



```
// Log this crash (for subsequent IoT dashboard reporting)
crashCount++;
```

Advanced Collision Logic

To ensure robust recovery under all circumstances, an advanced collision-handling strategy is prepared to handle collisions from different angles or unexpected scenarios. This allows the robot to reliably escape even when traditional maneuvers fail. The implementation is as follows:



```c
// Advanced collision logic(if needed)
if ((leftCount > 0) || (rightCount > 0)) {
    // If both sides have sensors triggered, treat it as a center collision
    Motor_Backward(3000, 3000);
    Clock_Delay1ms(1000);
    // Turn toward whichever side had fewer triggers
    if (leftCount > rightCount) {
        Motor_Right(3000, 3000);
        Clock_Delay1ms(1000);
    } else if (rightCount > leftCount) {
        Motor_Left(3000, 3000);
        Clock_Delay1ms(1000);
    } else {
        // If equal, pick a default side
        Motor_Left(3000, 3000);
        Clock_Delay1ms(1000);
    }
} else if (leftCount > 0) {
    // Collision only on the left side => turn right
    Motor_Right(3000, 3000);
    Clock_Delay1ms(1000);
} else if (rightCount > 0) {
    // Collision only on the right side => turn left
    Motor_Left(3000, 3000);
    Clock_Delay1ms(1000);
} else {
    // Otherwise, do a small reverse
    Motor_Backward(3000, 3000);
    Clock_Delay1ms(1000);
}

/Option 2: Reverse, then Turn, then Resume
Motor_Backward(3000, 3000);    // Reverse to create space
Clock_Delay1ms(500);           // Short pause after reversing
Motor_Left(3000, 3000);        // Turn left to avoid obstacle (or Motor_Right)
Clock_Delay1ms(1000);          // Execute the turn for a set duration
Motor_Forward(3000, 3000);     // Resume forward motion
```

Observed Motor Behavior and Proposed Safety Feature

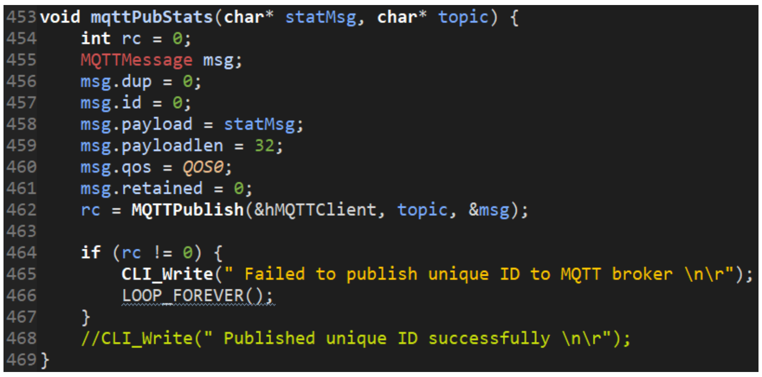
Through testing, we observed that after prolonged operation, the motors may become unresponsive or behave irregularly—especially when the robot gets stuck in a corner and the wheels continue spinning without actual movement. This issue typically occurs after extended runtime and appears to resolve itself after a short rest period (around three minutes) followed by a system reboot, suggesting a possible thermal or driver-related issue.

To address this, a future enhancement could include a safety mechanism: if all distance sensor readings remain constant while the motors continue running for more than one second, the robot should automatically halt and enter a temporary pause state. This would prevent unnecessary strain on the motors and give team members a chance to manually intervene and reposition the robot.

Wall Following integration & MQTT connectivity

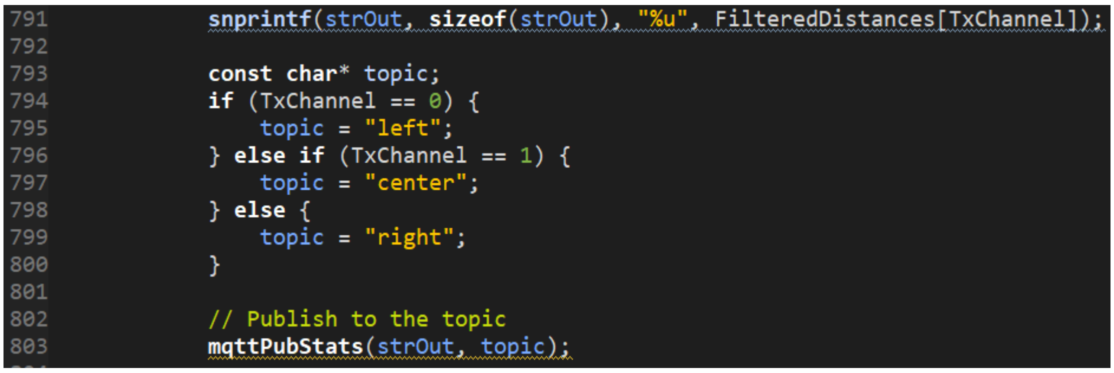
Due to the inherent complexity of the MQTT communication logic, it was prioritized early in the development process. While this decision allowed for a functioning messaging system, it introduced several integration challenges when merging the competition demo codebase with Lab 5's MQTT initialization code. This integration was non-trivial and led to numerous compatibility issues with other system components. Given the limited time and extensive debugging efforts required, certain features—namely, the tachometer and total track time tracking—were ultimately omitted.

In their place, a real-time MQTT-based telemetry system was developed to publish live distance readings as the robot navigated the track. A generic function was implemented to publish any given string to a specified MQTT topic. This function can be seen below.



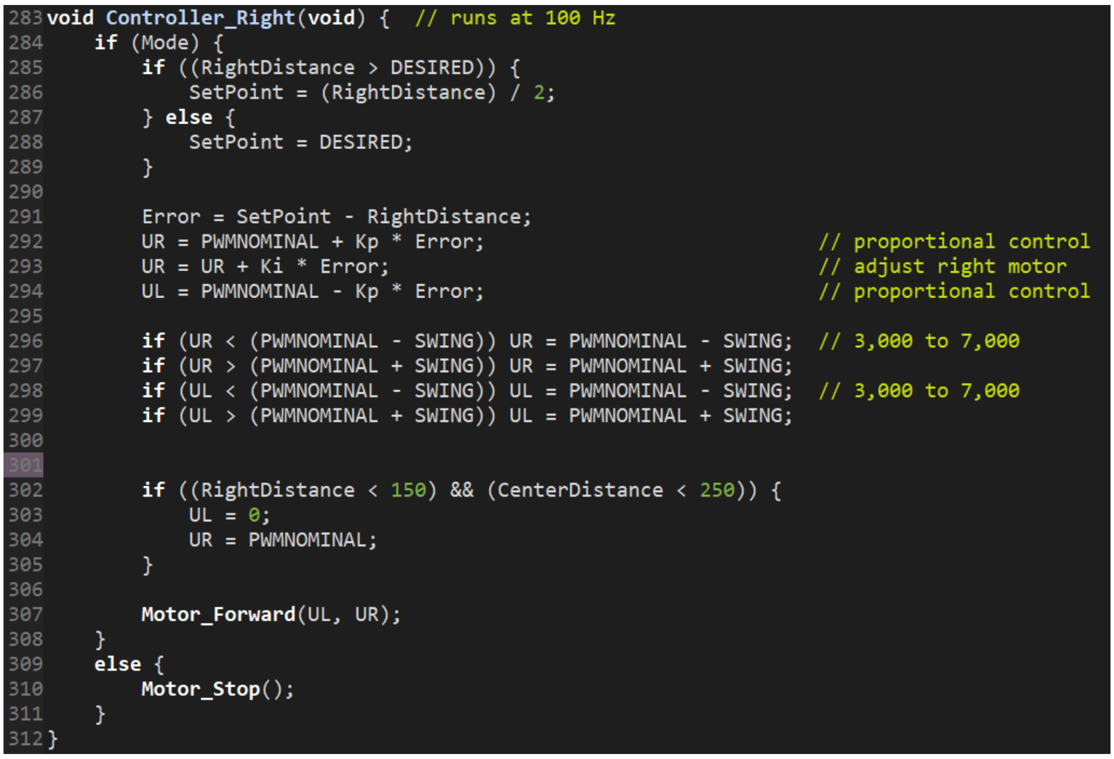
```c
void mqttPubStats(char* statMsg, char* topic) {
    int rc = 0;
    MQTTMessage msg;
    msg.dup = 0;
    msg.id = 0;
    msg.payload = statMsg;
    msg.payloadlen = 32;
    msg.qos = QOS0;
    msg.retained = 0;
    rc = MQTTPublish(&hMQTTClient, topic, &msg);

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

The integer values returned by the distance sensors were converted into strings and published to one of three topics—**"left"**, **"right"**, and **"center"**—based on the currently selected sensor channel. Additionally, a fourth topic was dedicated to reporting crash events, with the crash count being incremented and published in real time.

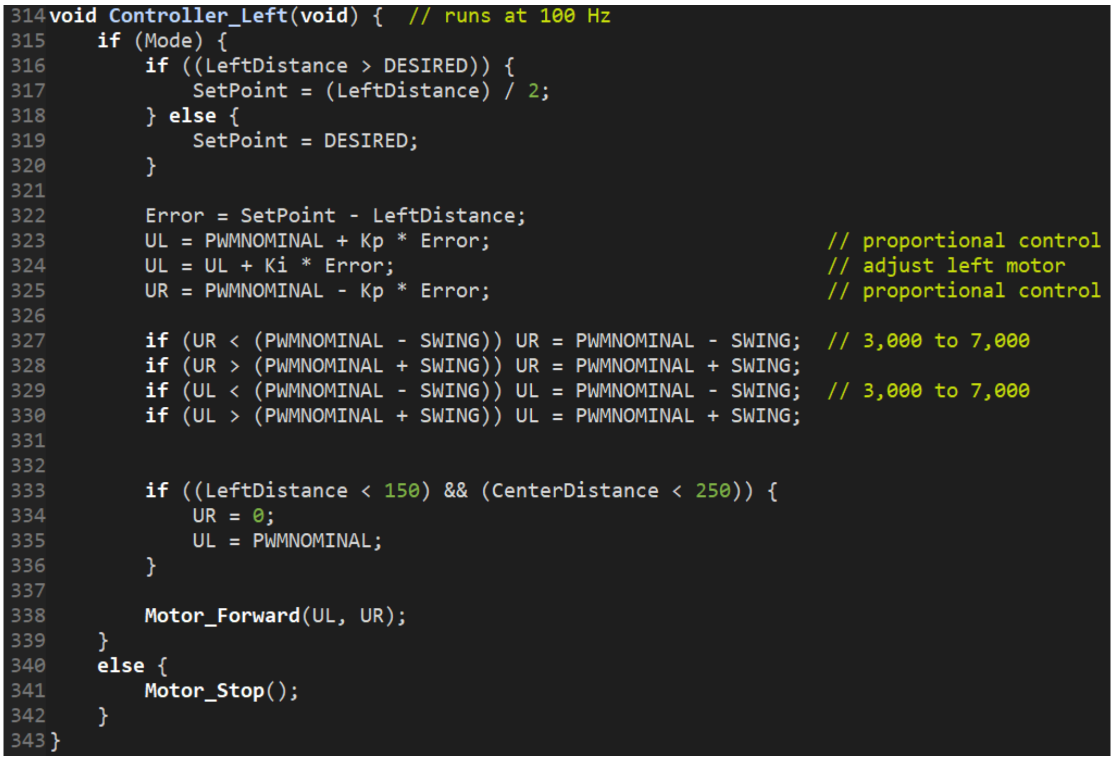


```c
            snprintf(strOut, sizeof(strOut), "%u", FilteredDistances[TxChannel]);

            const char* topic;
            if (TxChannel == 0) {
                topic = "left";
            } else if (TxChannel == 1) {
                topic = "center";
            } else {
                topic = "right";
            }

            // Publish to the topic
            mqttPubStats(strOut, topic);
```

The wall-following logic was adapted from the provided competition example code. Initial efforts focused on using both left and right distance readings to guide the robot, but this approach proved overly complex. The strategy was simplified to follow either the left or the right wall exclusively. This was implemented via two separate control functions—one for right wall following and another for left wall following. These functions can be seen in the two images below.

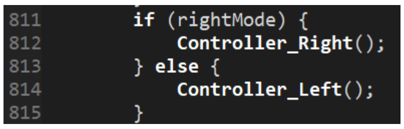


```
void Controller_Right(void) {  // runs at 100 Hz
    if (Mode) {
        if ((RightDistance > DESIRED)) {
            SetPoint = (RightDistance) / 2;
        } else {
            SetPoint = DESIRED;
        }

        Error = SetPoint - RightDistance;
        UR = PWMNOMINAL + Kp * Error;                              // proportional control
        UR = UR + Ki * Error;                                      // adjust right motor
        UL = PWMNOMINAL - Kp * Error;                              // proportional control

        if (UR < (PWMNOMINAL - SWING)) UR = PWMNOMINAL - SWING;   // 3,000 to 7,000
        if (UR > (PWMNOMINAL + SWING)) UR = PWMNOMINAL + SWING;
        if (UL < (PWMNOMINAL - SWING)) UL = PWMNOMINAL - SWING;   // 3,000 to 7,000
        if (UL > (PWMNOMINAL + SWING)) UL = PWMNOMINAL + SWING;


        if ((RightDistance < 150) && (CenterDistance < 250)) {
            UL = 0;
            UR = PWMNOMINAL;
        }

        Motor_Forward(UL, UR);
    }
    else {
        Motor_Stop();
    }
}
```

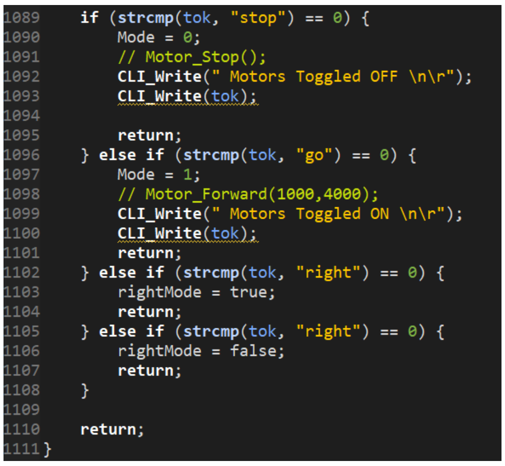


```
void Controller_Left(void) {  // runs at 100 Hz
    if (Mode) {
        if ((LeftDistance > DESIRED)) {
            SetPoint = (LeftDistance) / 2;
        } else {
            SetPoint = DESIRED;
        }

        Error = SetPoint - LeftDistance;
        UL = PWMNOMINAL + Kp * Error;                              // proportional control
        UL = UL + Ki * Error;                                      // adjust left motor
        UR = PWMNOMINAL - Kp * Error;                              // proportional control

        if (UR < (PWMNOMINAL - SWING)) UR = PWMNOMINAL - SWING;   // 3,000 to 7,000
        if (UR > (PWMNOMINAL + SWING)) UR = PWMNOMINAL + SWING;
        if (UL < (PWMNOMINAL - SWING)) UL = PWMNOMINAL - SWING;   // 3,000 to 7,000
        if (UL > (PWMNOMINAL + SWING)) UL = PWMNOMINAL + SWING;


        if ((LeftDistance < 150) && (CenterDistance < 250)) {
            UR = 0;
            UL = PWMNOMINAL;
        }

        Motor_Forward(UL, UR);
    }
    else {
        Motor_Stop();
    }
}
```

The active control mode was determined by a boolean flag, which toggled the robot's behavior based on commands received via another MQTT topic. This same topic was also used to start and stop the robot remotely.



```c
        if (rightMode) {
            Controller_Right();
        } else {
            Controller_Left();
        }
```



```c
    if (strcmp(tok, "stop") == 0) {
        Mode = 0;
        // Motor_Stop();
        CLI_Write(" Motors Toggled OFF \n\r");
        CLI_Write(tok);

        return;
    } else if (strcmp(tok, "go") == 0) {
        Mode = 1;
        // Motor_Forward(1000,4000);
        CLI_Write(" Motors Toggled ON \n\r");
        CLI_Write(tok);
        return;
    } else if (strcmp(tok, "right") == 0) {
        rightMode = true;
        return;
    } else if (strcmp(tok, "right") == 0) {
        rightMode = false;
        return;
    }

    return;
}
```

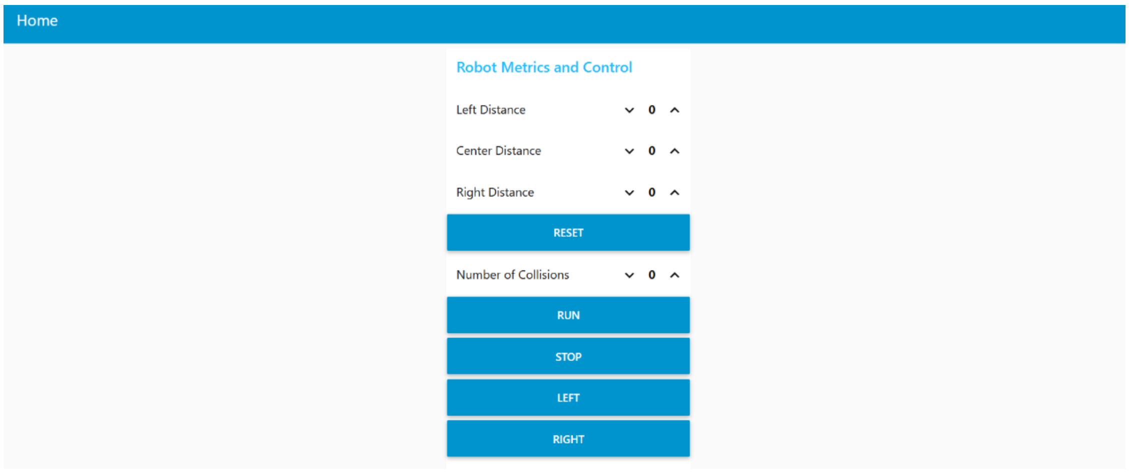
Minimal tuning was required for the proportional-integral control algorithm, with final values set at **Kp = 3** and **Ki = 1** to achieve stable performance.

One major issue encountered with the real-time MQTT updates was the latency introduced into the control loop. Because the distance sensor readings were being formatted, published, and transmitted over MQTT before being interpreted by the control logic, this added a noticeable delay to the robot's ability to respond to its environment. At higher speeds, this delay manifested as the robot failing to correct its trajectory in time, often resulting in direct collisions with the wall before any corrective maneuver could be executed. This behavior strongly indicated that the communication and processing overhead was effectively throttling the robot's responsiveness. To mitigate this, the robot's speed was deliberately reduced to allow additional time for the distance data to be received, interpreted, and acted upon. While this solution compromised overall performance speed, it significantly improved the robot's reliability and stability during wall-following, especially in narrow or curved track sections. Future iterations could potentially address this bottleneck by decoupling MQTT communication from the control loop through the use of a multi-threaded or asynchronous architecture, allowing for data logging and telemetry without impairing real-time control.

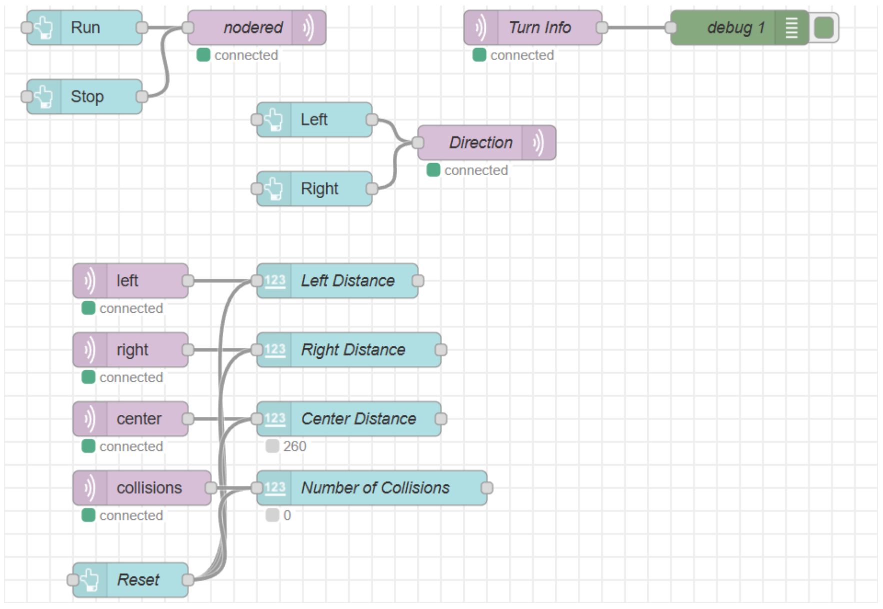
IoT Dashboard and Bluetooth Integration

IoT dashboard and Bluetooth was continued from the previous work Lab 4 and 5. One issue that was encountered was the usage of UART0 module by both the MQTT client and the Bluetooth module. This twofold usage of UART0 causes neither the MQTT client nor the Bluetooth module to function as it was intended. The first attempt to mitigate this issue was to refactor the Bluetooth module with the usage of UART1 instead. This proved to be time-consuming and unsuccessful due to the FIFO buffer of the UART1 module. The next approach was to remove the usage of CLI_Write by MQTT client that was using the UART0 module. This took away the client's ability to connect to the broker that was set up in Lab 5. Due to not finding a good fix to the Bluetooth issue, we decided to just use the IoT to start and stop the robot. After the race, I was able to find time and find a fix where the Bluetooth module worked correctly with UART1, adding methods that were like the ones in UART0.

The IoT dashboard was hosted using an Azure virtual machine. The link to the dashboard is http://20.84.115.15:1880/ui/#!/0?socketid=FxcXuwN6wr10-znqAACF. The GUI was built using node-RED and increased the productivity by removing the need to write the boilerplate code to set up a web app. The dashboard includes crucial information about the robot race conditions and initial settings such as real time distance measurements, buttons to reset the current number of collisions and distances, starting and stopping motor controls and choosing which logic the robot will use to follow the wall and avoid collisions.



Due to the time constraints, the tachometer and track time functionality was never implemented on the robot. None of the group members were familiar with the tachometer module and wasn't able to get the module integrated and functioning with the robot. Track time would have theoretically been easier to implement but due to the time spent attempting the integration of the tachometer module, we had to abandon the requirement be functional during the race.



Lessons Learned and Suggestions for Improvements

An important lesson that was learned was to keep track of the hardware on the robot that the software is using to make sure that there is no double usage of a hardware component. A suggestion to improve in future iterations is to modularize all of the code that the team uses especially the MQTT setup and make everything as barebones as possible to increase customization and make sure that everything that is not essential to the application could be removed.

Autonomous Racing Performance Video Link:

https://youtube.com/shorts/yBD9wCGNNWU