Rick Brophy
ECE1188 Cyber Physical Systems
Dr. Dickerson
Due: 4/28/24

Magneto - Wall Follower

System Overview

For this project, our robot was made to complete a maze in Benedum Hall as fast as possible while dynamically avoiding obstacles. Our work was split up into three modules, the wall follower, reflectance and collision handling, and bluetooth control and IoT dashboard. The wall follower module used the distance sensors on the robot to avoid obstacles and traverse through a course with walls as guidance. This was implemented using FSM logic and PID control. When the bump sensors were triggered, the robot would stop momentarily, back up slowly, then resume FSM logic. Using the reflectance/line sensors, the robot would come to a complete stop when it crossed the finish line. Bluetooth control was implemented to manually start and stop the robot. After the robot finished the course, data that was collected throughout the race would be sent to a webapp made using Node-Red. The code and readme files are shared with Dr. Dickerson from my other teammates. The zip file is too large to submit oin gradescope.

Brief Description of Contribution

I oversaw the IoT dashboard and Bluetooth control sections of the wall follower project. This submodule incorporates connecting to a local network, subscribing to multiple topics, collecting data to send via Wi-Fi, and BLE start / stop commands. I also created my robot digital twin through node red to accept, parse, and display race data to the UI.

Code Development

I had a lot of success in the last lab where I collected RPM and distance sensor data to send via Wi-Fi. Therefore, I didn't encounter many bugs for my submodule. Before integration, I used previously given source files such as the tach demo and OPT3101 as templates. These files guided me on writing barebone functions that would collect RPM and distance sensor data to send back to main. During this process, I had to familiarize myself with returning pointers. Since all data was stored in arrays, returning pointers was in my best interest. Once that was complete, my major debug errors were getting the distance sensor data to display on the UI. During the lab, I used a text box to display distance data which was very straight forward. The graph node would display the entire array with a single time point. To create a work around, I used a delay node to send at a rate of 1 sample per second. Although it updated slowly, this was a solid solution to display all data correctly.

Contribution

Before integration, I had my own main.c file that called other functions to collect RPM and distance sensor data. As mentioned, I used previously given code as templates to understand how the distance sensor and tachometer were initialized and used. After this data is collected, I call the MQTT function, passing all the data, which connects the robot to a local network, and publishes all data via Wi-Fi. The MQTT function is the same source file given for lab 6. I deleted a lot of fluff in the original file to make it easier to read and understand. As I explain my portion of the code, I am using the final integrated code as the reference.

Our main.c file is the wall follower code provided by Dr. Dickerson. Most of that explanation will be done by Ed Loveday, who was responsible for the FSM and PID control. In the while loop, we constantly ask the robot the state of this variable "command." The command variable is a single character which stores the BLE input. The first line in the while(1) reads in the BLE command via UART. That message is stored in "command" where the while(1) is broken into if-statements. Dependent on the BLE message, the robot will enter either a start or stop loop. Our code is only trained for an "s" or a "p" transmission which translates to start or stop respectively. Seen in *Figure 1,* if the command is start, the robot enters that loop and will stay there until a "p" is received from BLE.

```
241
242     UART0_OutString("\n\nEnter 's' to start\n");
243     while(1) {
244         command = UART0_InChar(); // accept BLE input ('s' for start, 'p'; for stop)
245
246         if(command == 's') {
247             test_reflectance();
248             command = UART0_InChar();
249
250             if(command == 'p') {
251                 Motor_Stop();
252                 UART0_OutString("\nRace Complete!\n\n"); // ensure we enter trapped loop
253                 bump_count = bump_count / 2;    // half bump count bc interrupt is called twice
254                 //sprintf(message, "Max RPM = %d\nTime = %d seconds\nBump Count = %d\n\n", rpm_max, time_count, bump_count);
255                 //UART0_OutString(message);       // display what was measured to compare what was received by MQTT
256
257                 int none = mqtt(rpm_max[0], rpm_max[1], time_count, bump_count, dist_left, dist_mid, dist_right, ldsize, mdsize, rdsize);
258                 StatePtr = Stop;
259                 while(1);
260             }
```

Figure 1: Beginning of while(1)

In the stop loop, there is a debug message printed via UART and all data is passed into the MQTT function. As you will see in the code files, the rest of the start loop is collects distance sensor data and having the robot enter different motor states. At the end of this loop, I also call a function called "get_RPM." This is one of the files I wrote, based off Tachdemo, to collect RPM data. This function uses the modulo operator to collect RPM data every 50 start loop iterations. Although that seems high, this would collect RPM data from both wheels every second or so. The value is compared to a max value which then is stored into the max only if the new RPM value is greater. Seen in *Figure 2,* the only difference from this function to Tachdemo is that I am returning RPM data without having any PID control. This is due to the fact that the robot is changing motor states in main.c.

```
int *get_RPM(void) {
    int i = 0;
    t = 0;
    Tachometer_Init();
//    EnableInterrupts();

//    //Start motors with nominal values
//    UR = RightDuty;
//    UL = LeftDuty;

    while(1) {
        Tachometer_Get(&LeftTach[i], &LeftDir, &LeftSteps, &RightTach[i], &RightDir, &RightSteps);
        i = i + 1;

        if(i >= TACHBUFF) {
            //This section of the code checks the wheel state every second (10*100ms)
            i = 0;

            //Sense state of wheels (in RPM) and take the average of the last n values
            // (1/tach step/cycles) * (12,000,000 cycles/sec) * (60 sec/min) * (1/360 rotation/step)
            ActualL = 2000000/avg(LeftTach, TACHBUFF);
            ActualR = 2000000/avg(RightTach, TACHBUFF);
//            // hold values for the previous error value

            rpm_data[0] = ActualL;
            rpm_data[1] = ActualR;
            return rpm_data;

        }

        Clock_Delay1ms(100); // delay ~0.1 sec at 48 MHz
        t = t + 100; // keep track of timer
    }
}
```

Figure 2: Function get_RPM.c

In the MQTT function, I have the robot publish to 4 different topics. The first topic is used to send RPM data, bump count, and race time. The other 3 topics are used to send data from the 3 different distance sensor arrays, one for each direction (left, middle, and right). Seen in *Figure 3,* I concatenate the array into a string, with spaces separating each value. This makes it easy for Node Red to understand and display correctly. This process is repeated 2 more times and sent to their respective topics.

```
    // -------------FOR WEB APP - DIST0 DATA------------
    char bufferLEFT[25] = "";
    char buffLEFT[25];
    int i = 0;
    for(i = 0; i < ldsize; i++) {
        sprintf(buffLEFT, "%d ", dist_l[i]);
        strcat(bufferLEFT, buffLEFT);
    }

    msg.dup = 0;
    msg.id = 0;
    msg.payload = bufferLEFT;
    msg.payloadlen = 28;
    msg.qos = QOS0;
    msg.retained = 0;
    rc = MQTTPublish(&hMQTTClient, PUBLISH_TOPIC_2, &msg);
    CLI_Write("Published dist0 data successfully\n");
```

Figure 3: Sending Distance Sensor Data

To display data correctly, I use the split and switch nodes to parse the incoming messages for the RPM, bump, and time data. Node Red is expecting this form: “left: #, right: #, bump: #, time: #.” I set the first delimiter to “, “ and the second one to “: “.For the distance data, I use 3 separate topics for the respective arrays. I use the split node again and set the delimiter to a space. The delay node was utilized as mentioned before to have a time point associated with each data point. *Figures 4 and 5* show my node-red flow and UI. As you will see in the UI, I spent some time in formatting to make the dashboard visually pleasing. As a quick additional feature, seen in *Figure 6,* I added a large graph to have all distance data displayed on top of each other with a legend. The node red video (link provided below) shows the dashboard in action.

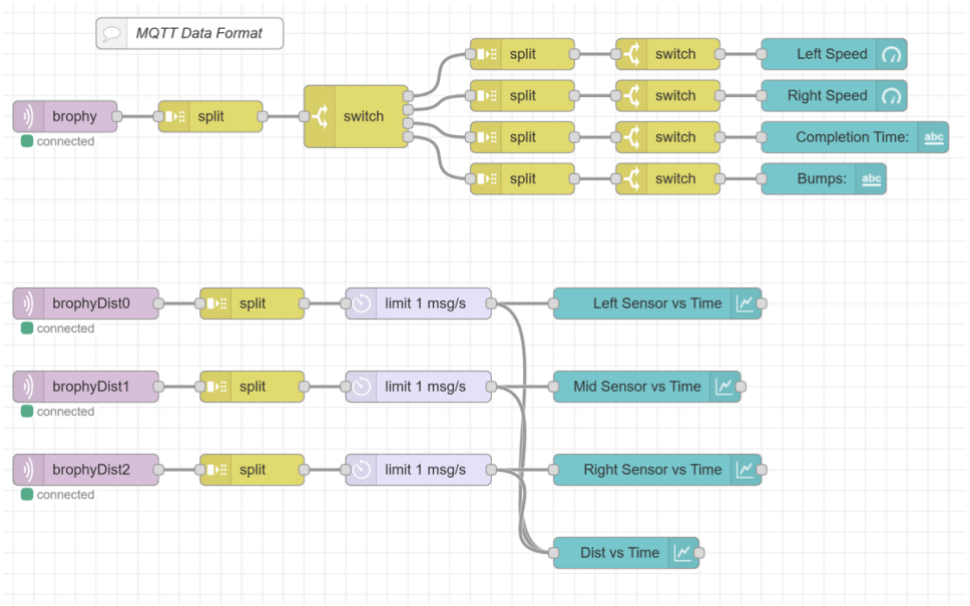

Figure 4: Node Red Flowchart

Figure 5: Node Red Dashboard

Figure 6: Node Red Dashboard – Large Graph

Video Demos
BLE Commands: https://youtube.com/shorts/XJCJWihaY8w?si=4iOzpT3rkeoXp3u4
Node Red: https://youtu.be/0lLSVJUm4i8?si=5m0dabBI2zlX4We6

Summary of Changes after integration
After integration, I did not have to edit much for my module. This was because I called additional functions to collect data and send to the main. I did it that way on purpose so that integrating the wall follower code was smooth. The only difference was that distance data is collected in the main instead of a separate function. After integration, most of our time was spent for tuning the PID and FSM.

Video of racetrack
https://youtu.be/mWzL0BQ1z80

Improvements
During race day, our robot was not able to efficiently traverse the wall course. We initially tuned the robot on wider wall margins by changing a set point distance that the finite state machine relied on. During the race, the first section had smaller margins than our robot could handle, which led to rapid state changes and essentially stalled the robot out. With more time, the team could have retuned our state machine and PID gains to better handle the specific course.