CSC 615 UNIX Programming

Term Project, A Mini Autonomous Vehicle

Team Mushroom

SFSU, Spring 2021, Prof. Robert Bierman

Developed by:

Erik Chacon, Mark Jovero, Tim Wells, & Katie Kennedy

Date Submitted:

May 21, 2021

CONTENTS

Task Description

This term project required a team of four of us to develop a fully-functional autonomous vehicle controlled by a Raspberry Pi unit. The requirements provided by the professor were slim, which allowed for a lot of liberty in teams' decision making. The only given requirements were the AV was to follow a track (of which the shape was unknown to the students), and an obstacle must be detected and avoided when encountered on the track. We were told that the path could contain sharp corners and curved segments.

The robot's logic was required to be coded exclusively in C; the only exception to this rule was for the LiDAR unit, which was permitted to include code from a C++ library. The only other requirement was that the vehicle must not fly due to safety and liability issues.

Team Members

Due to COVID19-related restrictions, the main hardware kit provided by the professor went to one person in each team. Our hardware manager was Erik Chacon, who was responsible for a majority of the code testing until mid May when we met in person as a (vaccinated) team. After this meeting, Erik then took the assembled vehicle home to continue obstacle avoidance testing. The following table outlines each team member's information:

Name	Student ID	GitHub Username
Erik Chacon (Manager)	920768287	SaintGemini
Mark Jovero	916691664	Mark-Jovero
Timothy Wells	913806522	twells96
Katie Kennedy	920628472	katie-develops-things*

*account used for project repository

GitHub Repository

https://github.com/CSC615-Spring2021/csc615-group-term-project-Mark-Jovero

Parts & Sensors Used

The following table lists each piece of hardware that was used to build and operate our vehicle. Because the vehicle is autonomous, a variety of sensors were employed to make sure the AV remained on the track (a black line about 3 inches in width) and avoided any possible obstacles that could be encountered on the track. We also included extra obstacle sensors as a backup option in the case that the LiDAR unit integration wouldn't perform correctly.

Part Num.	Part Name	Model Number	Notes on Use
1	Raspberry Pi 4 (2GB)	SC15184	
2	Long breadboard		
3	DIY Smart Car Chassis Kit		
4	Slamtec RPLIDAR	A1M8	Single unit used.
5	Line sensor	TCRT5000	3 units used.
6	DC Electric motor		4 units used.
7	WaveShare motor driver HAT		Single unit used for all 4 motors.
8	MicroUSB Rechargeable battery		Single unit used.
9	REXQualis breadboard jumper wires	8541692640	~30 units used.
10	Basic art-supply pipe cleaners		3 units used to position & support line sensors.

Software Used

The code implementation for the vehicle's logic was written with the aid of the WiringPi library in C, which was developed specifically for Raspberry Pi units running the 32-bit Rasperian operating system. WiringPi also provides a specific library for I2C functionality. Though it is no longer maintained or distributed by the original developer, we were able to get a copy of the source code from the professor. The official documentation can be found here.

Our implementation for the LiDAR unit we used required its own library provided by the manufacturer, Slamtec. Because this library is written in C++, we needed to pipe

the data from the C++ read function to be able to use that information with the rest of our code. The documentation for the Slamtec RPLidar Library can be found here.

Pin Assignments

The following table lists the pin number assigned to each sensor component. These pin values refer to Broadcom's Raspberry Pi 4 numbering and must be initialized with the corresponding WiringPi setup function.

Pin Number	Device	Pin Mode
22	Left line sensor	Input
27	Middle line sensor	Input
17	Right line sensor	Input

Building the Bot & Design Development

Assembling the Robot

The hardware manager, Erik, was responsible for assembling the vehicle chassis provided by the instructor. The team initially wanted to implement a three-wheel design with two regular wheels in the front and one omnidirectional wheel in the middle of the back of the chassis. However, this design was not feasible because it would cause the car to drive in random directions. Once we switched to a classic, four wheel design, the car would drive reliably straight.

During this phase, the bot's mobility was tested on a relatively smooth hardwood floor. However, during the team meeting, we tested the vehicle on black tape placed directly on paved concrete, which proved to be much rougher than the wood at Erik's house. The added friction caused the vehicle to lose so much traction that its wheels would spin, but the car would not move. Adding a two-pound weight to the top of the car fixed this problem, but we had to remove the LiDAR sensor to accommodate the mini-dumbbell. Eventually, we abandoned testing on the concrete and went and bought a rough, waterproof bed sheet to simulate the canvas track we would be using for the final demonstration. Luckily, testing on this surface alleviated the need for the extra weight.

Motors

We use WaveShare motorhat to attach 4 motors. The left and right-hand motors are paired and share a single slot on the motorhat. In order to maneuver, we change the speed or direction.

Driving on the Line

Before testing our vehicle on a track, our team initially designed our vehicle with five line sensors, as we were not yet sure how the exact logic for turning would work. For right angle turns, we thought we would need two line sensors on either side of the bot. However, after our hardware manager's initial testing, the team decided that we would only need three line sensors placed at the front of the vehicle: one in the center and two on either end of the front. All three faced the front.

The logic for directing the vehicle is based on the combination of data from all three line sensors during a single read cycle, which is performed every 150 milliseconds. Note that the motors must be stopped before turning at a right (90°) angle to prevent a burnout. The following table illustrates the directional output based on what each sensor reads:

Left Line Sensor	Middle Line Sensor	Right Line Sensor	Vehicle Instruction Output
1	1	1	Drive straight forward.
1	1	0	Stop then turn left at a 90° angle.
1	0	0	Adjust direction to the right.
1	0	1	[invalid data]
0	1	1	Stop then turn left at a 90° angle.
0	0	1	Adjust direction to the left.
0	1	0	Drive straight forward.
0	0	0	Stop vehicle (stop both left and right motors).

Integrating LiDAR Data for Obstacle Avoidance

After we were successful in getting our vehicle to follow a line, we moved on to integrating data from the LiDAR sensor to begin writing the logic for the obstacle detection and avoidance.

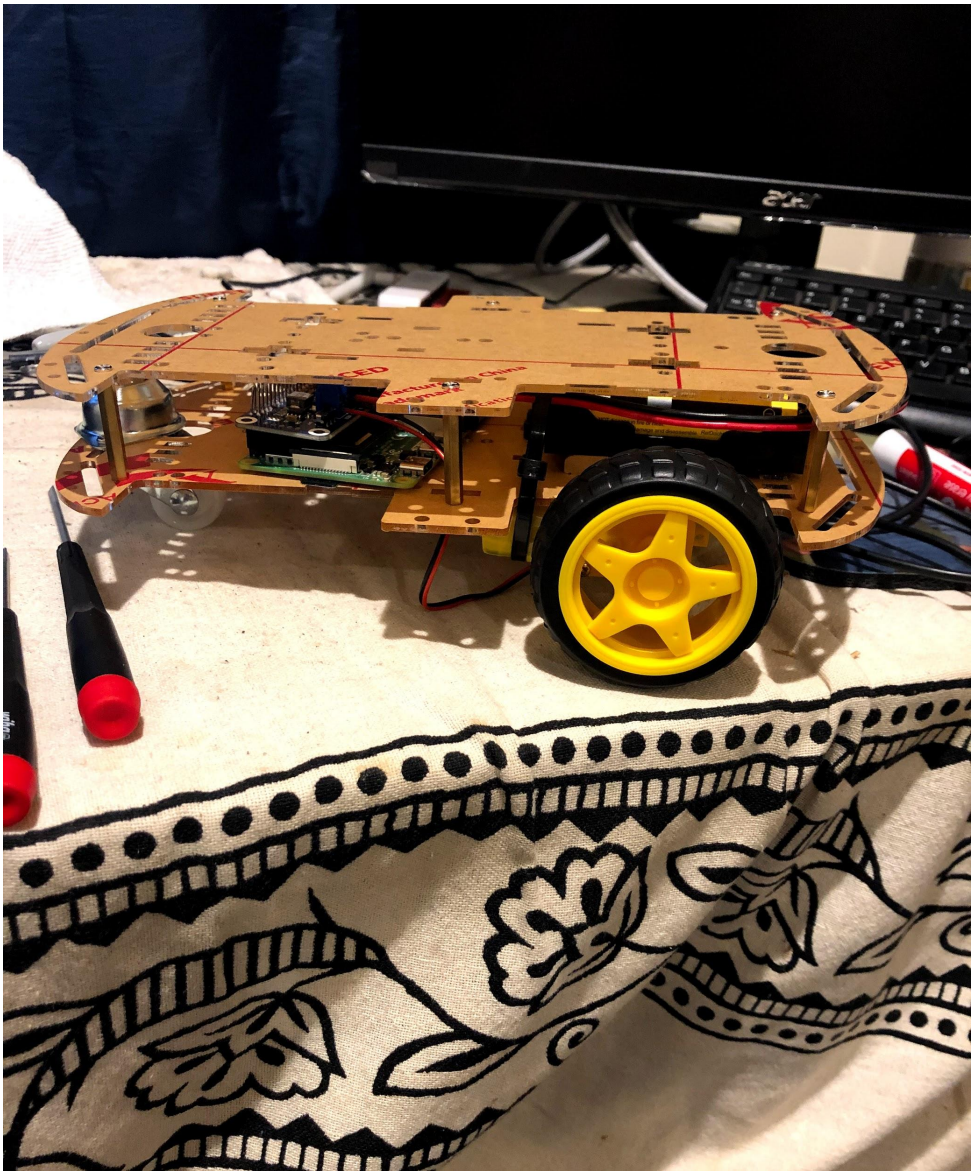
The challenging part of integrating major hardware in a small time frame is that we have to work with two different languages to quickly get it working. Slamtech, the LiDAR manufacturer, included sample C++ files to run and display data. Since our car's main thread is separate from the LiDAR thread, we used named pipes to transmit data between the threads. Since pipes use FIFO, data must be pushed into it in a specific, structured order. To do this, we push theta, distance, and quality over and over again. This data is piped/written into a temporary file.

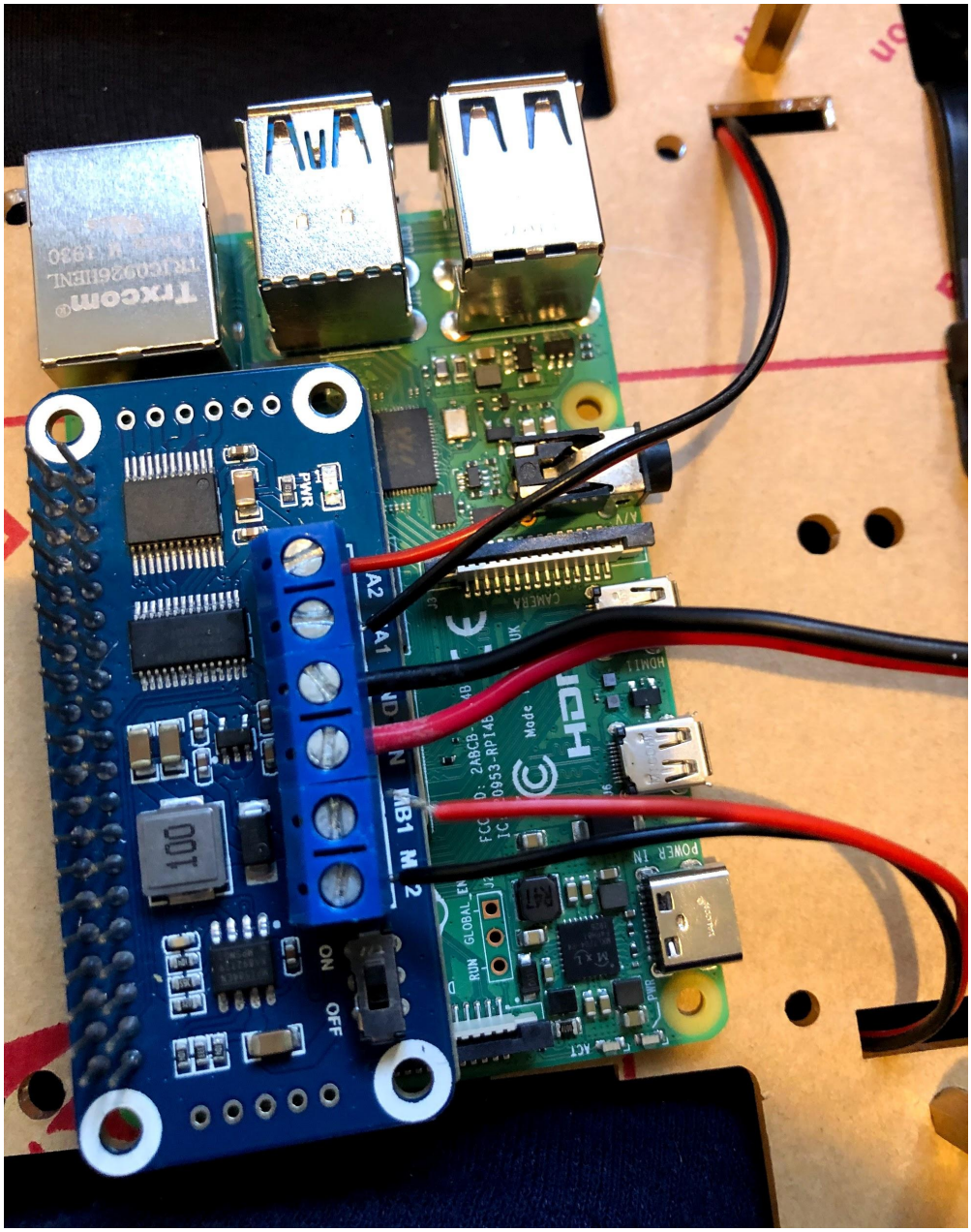
In our C program, we read from the temporary file. Since the order of which data is transmitted is known, we are able to read data successfully. In order to prevent a clogged pipe, we use a thread that will continuously read from the pipe and store data into our data structure. This structure is then utilized by the car to identify and navigate around objects.

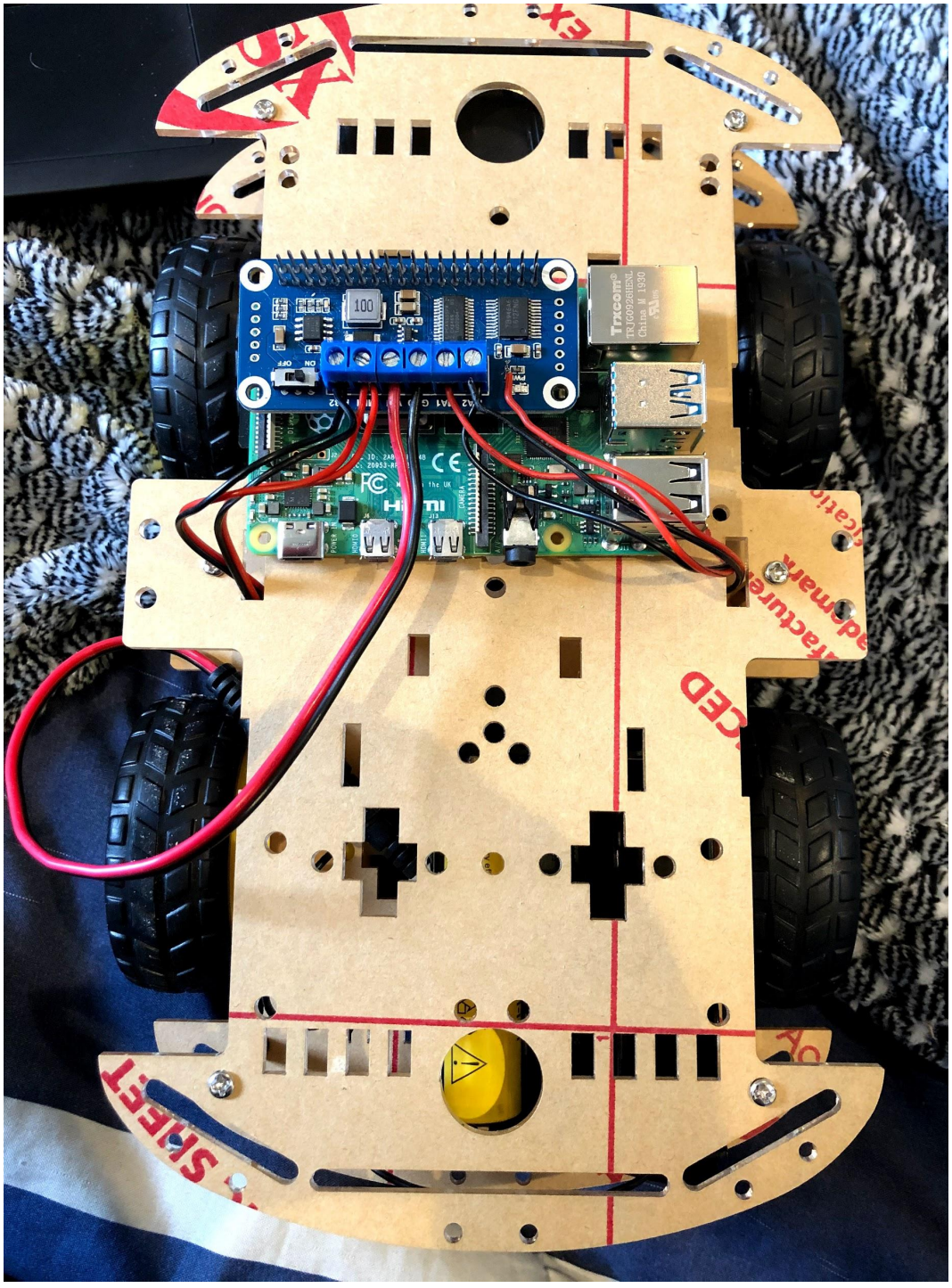
Once we were successful in accurately reading data from the LiDAR unit, we began planning out the logic for detecting and driving around the obstacle on paper. Because the LiDAR sensor spins, we could detect objects at any angle within a distance range of 0.15 and 6 meters, which we used to our advantage in planning the avoidance route. The LiDAR unit's reference point of 0° is directly in front of the vehicle, so when it registered an obstacle at 0° and within a distance of 12 inches (~308 millimeters), we paused reliance on the line sensors and began to perform our avoidance maneuver. Going around the obstacle required a total of four turns. The table below describes the avoidance system:

Obstacle location angle θ (approx.)	**Obstacle distance from AV (in mm)**	**Vehicle Instruction**	**Driving direction relative to line (after instruction)**
0°	305	Turn left 90°	Perpendicular, away
231°	330 - 488	Turn right 90°	Parallel
237°	300 - 457	Turn right 90°	Perpendicular, towards
180°	N/A	Turn left 90°	Parallel, on top of line

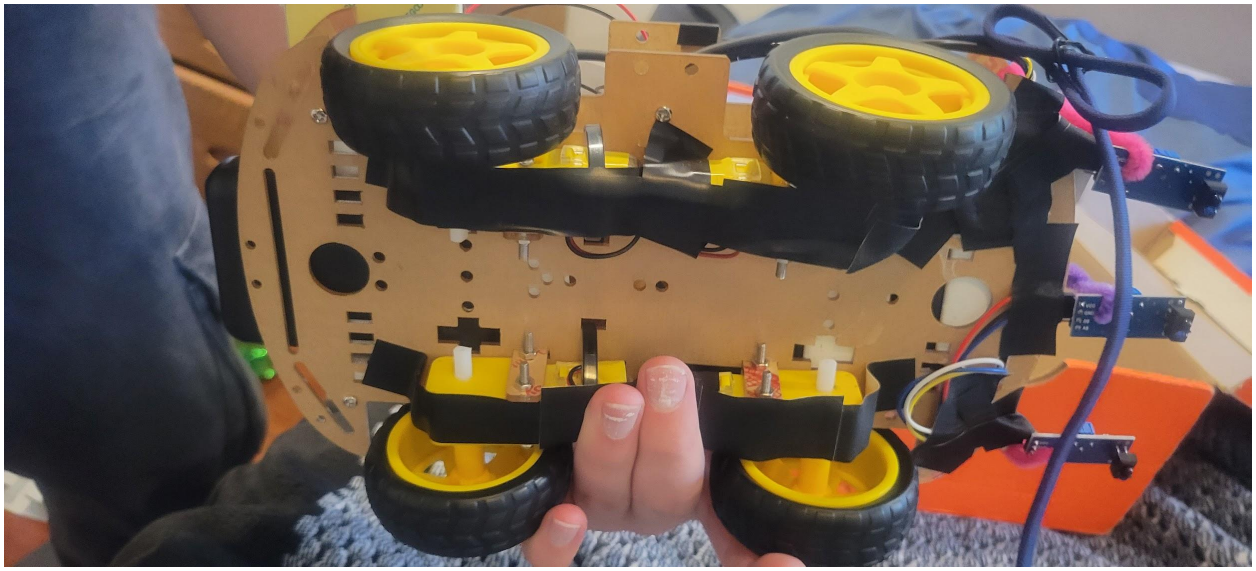
Original 3 wheel design




Trxcom
PWR
A2
A1
MB1
ON
OFF
RUN
GLOBAL_EN
POWER IN


100
Trxcom
TRJG0926HENL
China M 1930

New 4 wheel set up



Program Instructions to Compile & Execute

LiDAR

It is important to run LiDAR file first. That way, there is data available in the pipe once the car runs. To run LiDAR, connect it to one of the RaspberryPi USB ports. Then, locate the serial port by running the following command:

```
ls /dev/tty*
```

Take note of the port containing USB. Usually, this will be `/dev/ttyUSB0`.

Then, navigate to:

```
lidar/rplidar/sdk/output/Linux/Release
```

Then, run the following to start up LiDAR piping:

```
./ultra_simple /dev/ttyUSB0
```

The Car

To start and run the robot, the two libraries mentioned above must first be added. The WiringPi library can be installed to the Raspberry Pi unit with following command:

```
sudo apt-get install wiringpi
```

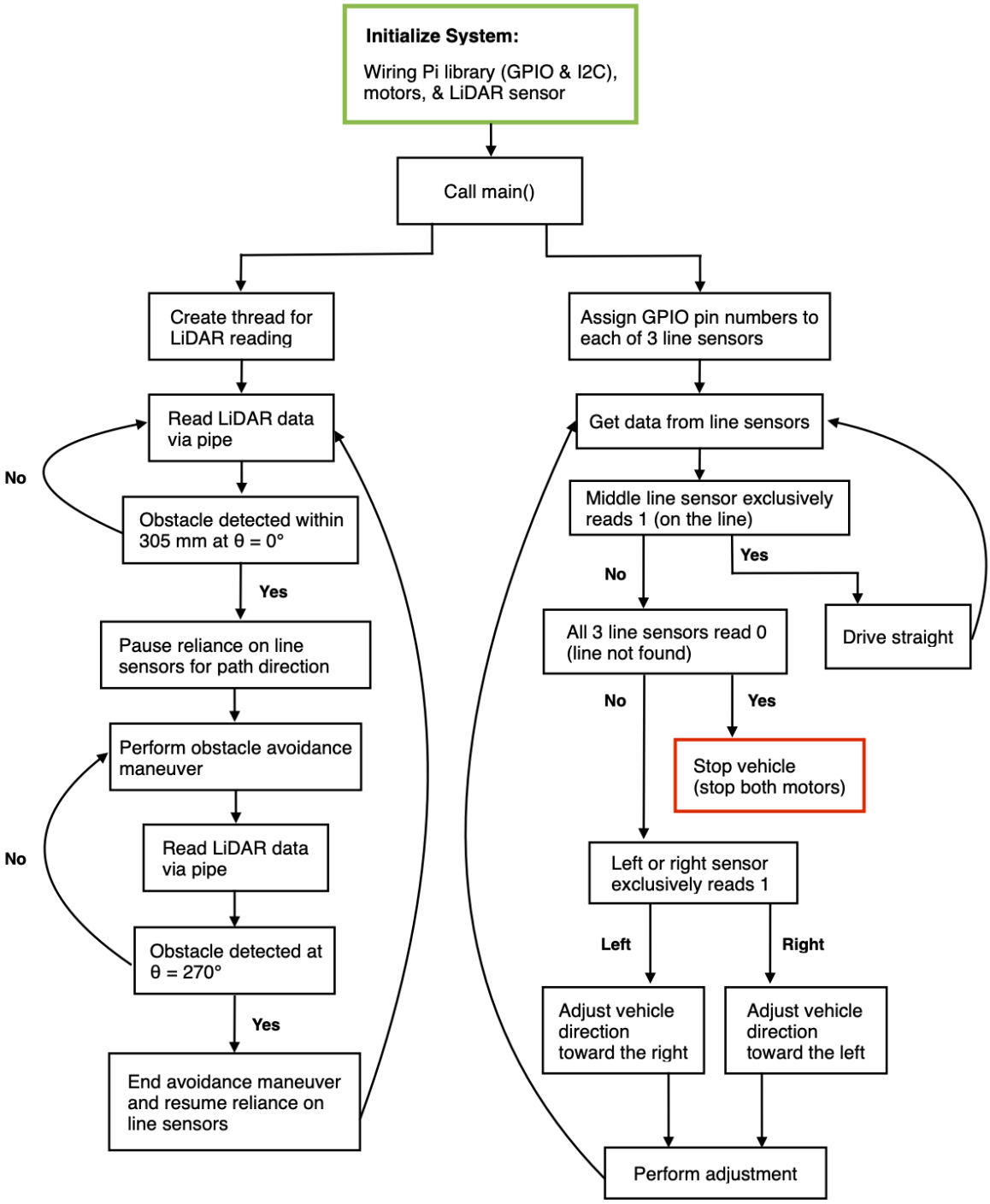
The development kit for the Slamtec RPLidar unit can be cloned from this GitHub repository and can be compiled with command: `make` (once the kit is added to the project directory.)

To run the vehicle, change the working directory to “mushroom-lidar-car” and enter the following command:

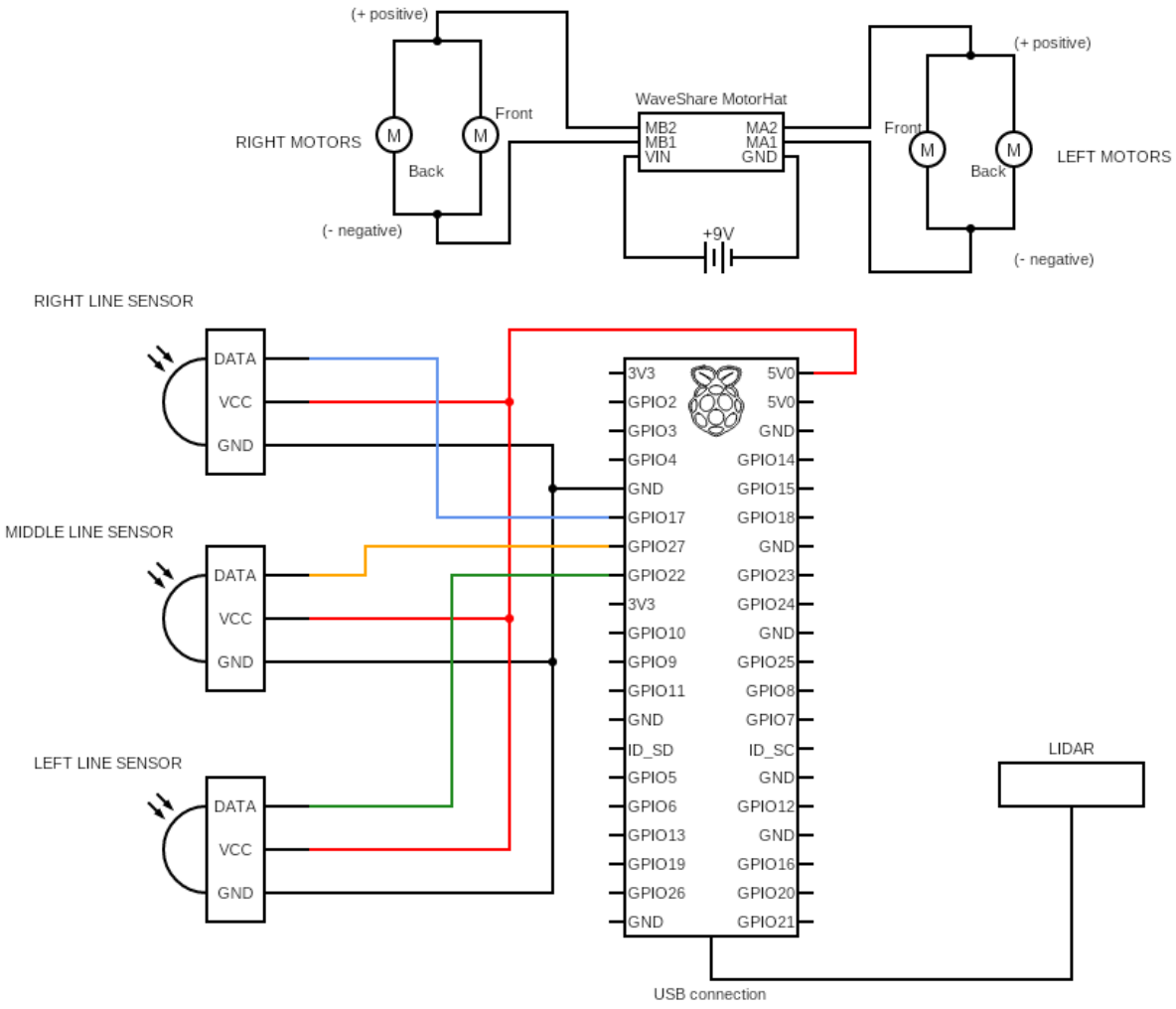
```
make run
```

To stop the vehicle, the user must press CNTL-C to exit the program, which will stop the motors.

Code Flowchart



Hardware Diagram



What Worked Well

Our team members initially decided to form a group with each other because we all remained in or around San Francisco during the semester, which was a wise decision, as we did get to meet up to work on the car in mid May before it was due. Forming a team based on location allowed our other three teammates to get some hands-on experience with the AV before the final run, as well as give Erik a break from doing all the testing. We were all vaccinated and met outside when we did meet as a group.

Our team also had great communication; we all checked our Discord chat regularly and everyone was very responsive in a timely manner. We met live on Zoom about once a week to check in and prepare checklists of what we needed to accomplish by the end of that week. Erik also provided our team with near-constant updates of our robot's assembly via videos and code testing results. All four of our teammates were extremely receptive to alternative ideas and possible implementations, which allowed for seamless collaboration amongst us.

As for the implementation of the car, we were lucky to receive a LiDAR sensor to integrate into our vehicle. The addition of the LiDAR sensor made obstacle avoidance quite simple to implement with basic trigonometry.

Issues & Challenges

Obviously, the greatest challenge with this project was not being able to regularly meet in person to build the robot and test code ourselves due to remote learning. This project involved intense use of physical computing, which ended up forcing our hardware manager, Erik, to do a vast majority of the testing and building of our AV.

In terms of building the car, one of the greater challenges we had was with our initial design. We originally planned to use a three wheel design that placed the omnidirectional wheel in the mid-back position of the chassis. However, after testing, we realized the use of this wheel was just not feasible, so we had to overhaul the initial design. Constructing the robot required a lot of trial and error which was challenging.

We also faced quite a few challenges while testing the robot. While Erik got the car to follow a line reliably on his hardwood floor at home, Mark, Tim, and Katie ran into trouble while testing it outside on a different floor material. We tested the car on a coarse waterproof bed sheet to simulate the painter's cloth that was going to be used for the official final run. Because the three of us tested it outside, we had trouble with the black tape used for the track reflecting sunlight, which would cause the line sensors to fail to interpret the black tape as the line it was supposed to follow. Adjusting the line sensor sensitivity fixed this problem in some cases, but not always. Casting a shadow over the line while testing outside also fixed this problem in many cases.

During the final, our car had trouble with right angle turns to the right. The problem may have been that the distance between the middle line sensor and the right light sensor were closer together than they should have been. The left line sensor was at a slightly greater distance away from the middle line sensor and our car handled right angle turns to the left very well.