CSC 615 Embedded Linux

Team 0xFF PI

FALL 2022

Github Repository https://github.com/CSC615-2022-Fall/csc615-term-project-faiyazc	
Members	**Github Usernames**
Hector M. Lua	hectormagallanes
Faiyaz Chaudhury	faiyazc
Briget Soriano	b-ts-o
Emanuel Francis	Emanuelf-sfsu

Table of Contents

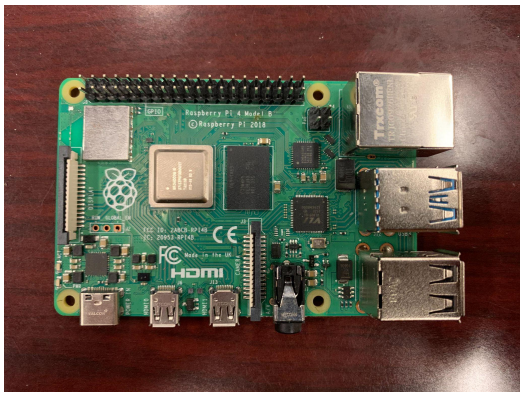
Task Description

The aim of this project was to create a robot/car using a Raspberry Pi 4 and a range of different sensors, wheels, and motors. Once the vehicle is assembled it must traverse through an obstacle course. The robot/car will follow a solid black line and navigate around objects on the path.

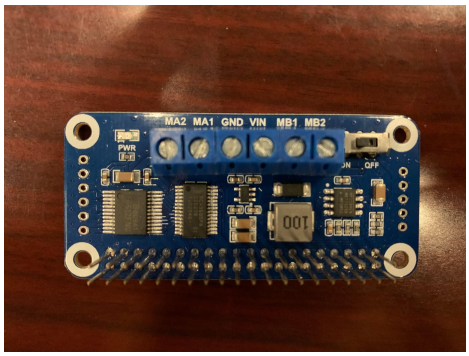
Building the Robot

Parts / Sensors

- Raspberry Pi 4



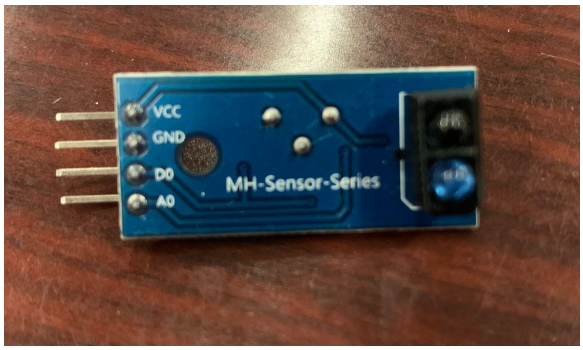
    - This part is the main “brains” of the robot. The Pi is able to gather all of the data from the sensors. This allows the robot to make its own decision based on its surroundings and the requirements of the project.
    - Specifications
        - Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
        - 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE Gigabit Ethernet.
        - More information found here
- Waveshare Motor Driver HAT for Raspberry Pi (2)



    - By stacking the two Motor Driver HATs the Raspberry Pi is able to control all 4 of the motors.
    - Each Motor Driver HAT had its own unique I2C address.
    - Specifications

        - Standard Raspberry Pi 40 PIN GPIO extension header
        - Onboard PCA9685 chip, provides 12-bit hardware PWM to adjust motor speed
        - More information found here.
- Lidar - Slamtech RPLIDAR A1

    - The robot depends on the Lidar for obstacle avoidance. The information recorded from the Lindar helps the robot navigate around objects.
    - Specifications
        - 360 Degree Omnidirectional Laser Range Scanning
        - Configurable Scan Rate from 2-10Hz
        - More information found here.
- TCRT5000 Infrared Reflective Sensor IR (2)



    - Specifications
        - Detecting the reflected distance: 1mm ~ 25mm applicable
        - Working voltage of 3.3V-5V
        - More information found here.

- Brushed DC Motor Gearmotor (4)

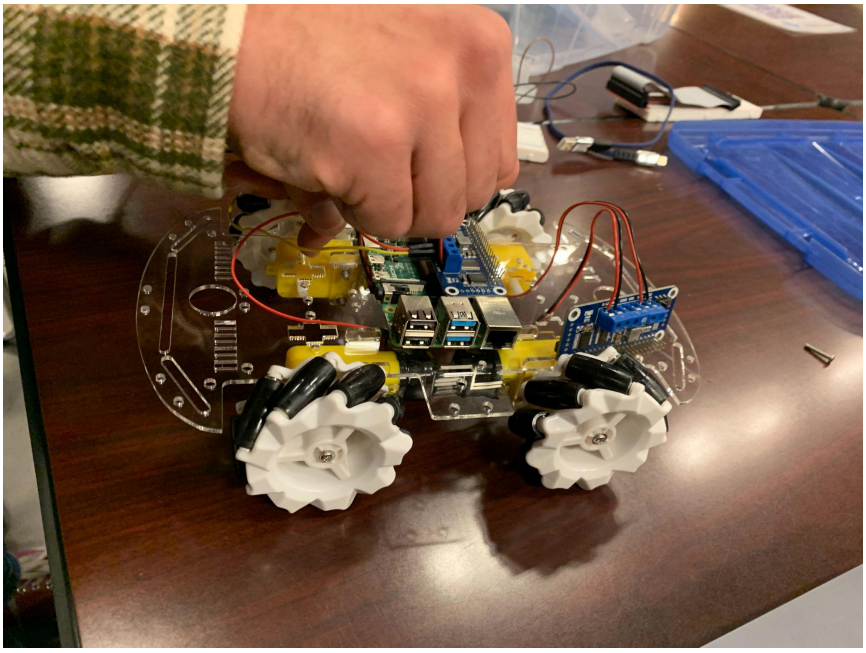
  - Specifications
    - 200 RPM
    - 3 ~ 6VDC
    - More information found here.

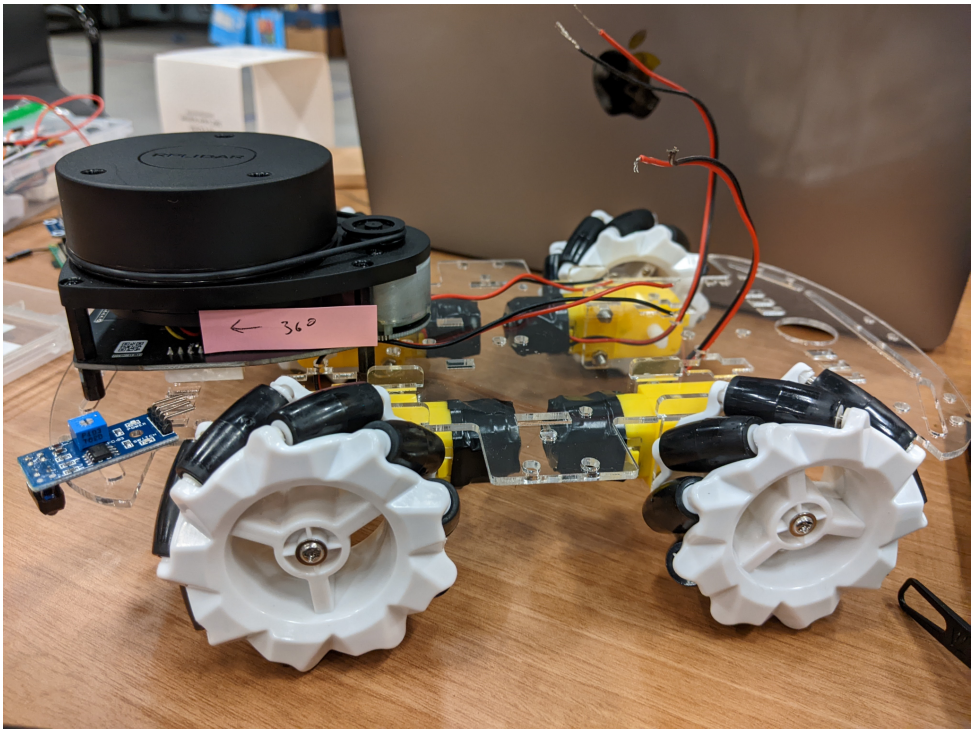
- Button (1)
- Small Breadboard (1)
- Phone battery charger (1)
- Batteries (3)
- Acrylic Chassis kit & Mounts
  - More information found here

How was bot built?

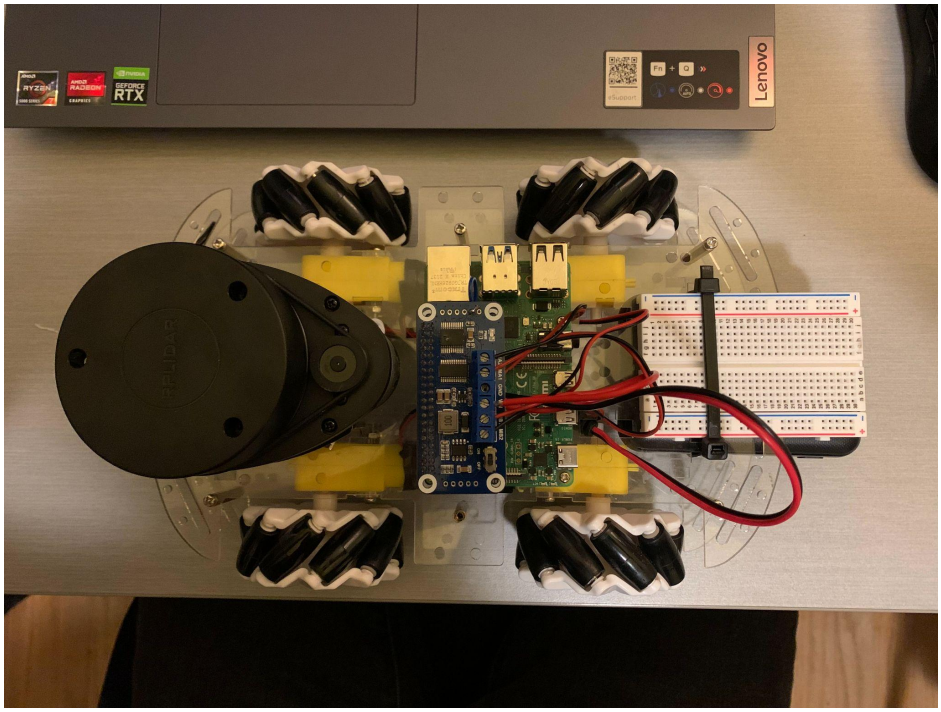
The robot was built in 3 stages.

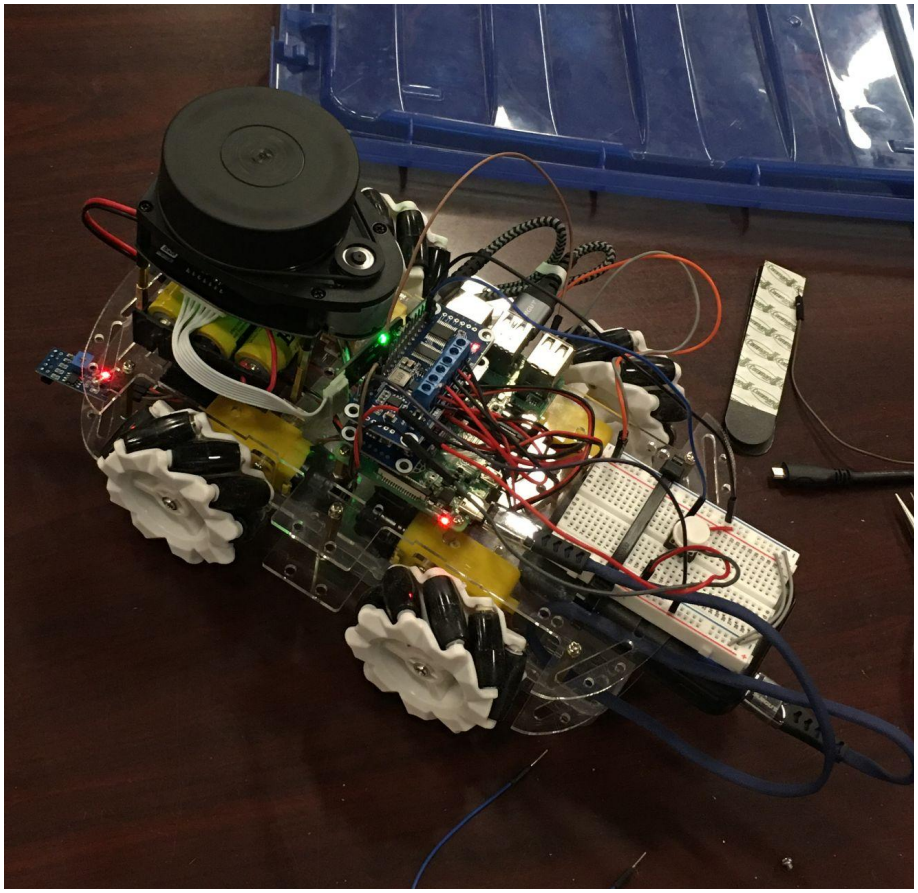
- Mechanical assembly
  a. This included assembling the chassis and attaching motors, sensors, and wheels. We also had to drill holes into the acrylic plate to secure the sensors.





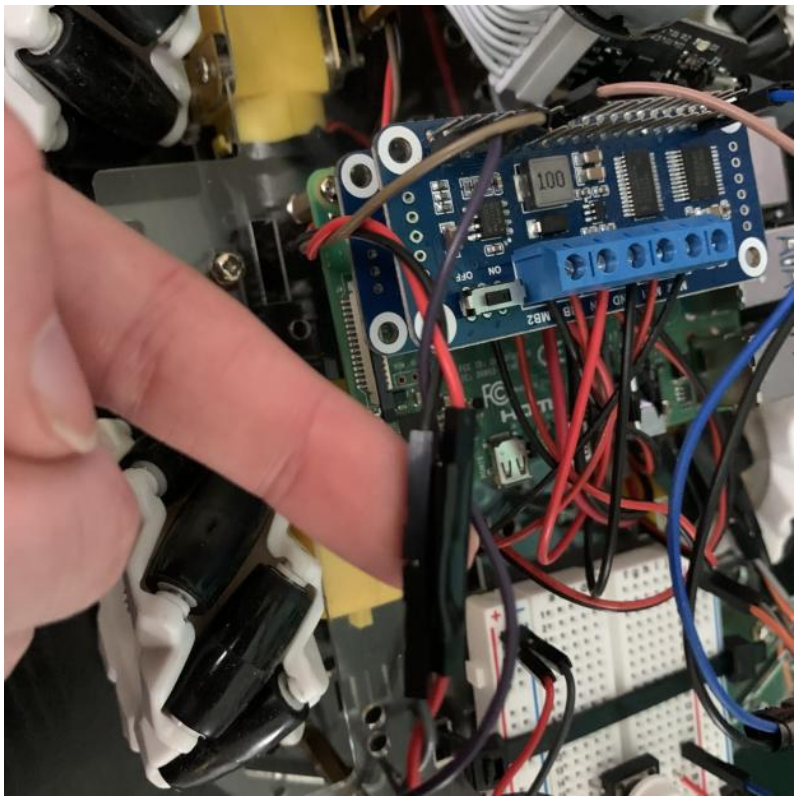
(This picture was taken on Thanksgiving break.)

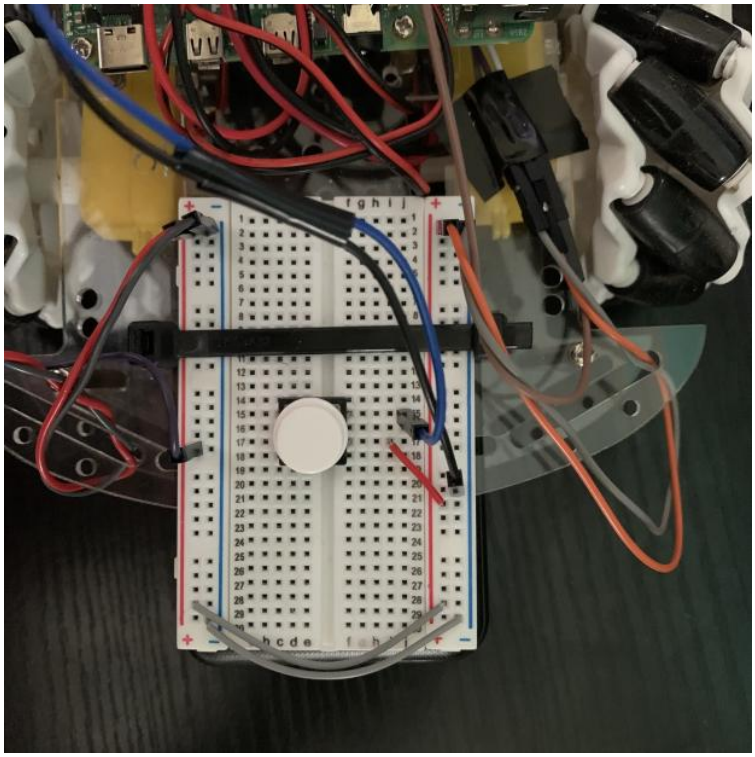




Final look of the car

- Electrical assembly
    a. The robot uses two separate battery packs to power the motors, sensors, and raspberry pi.





- Programming

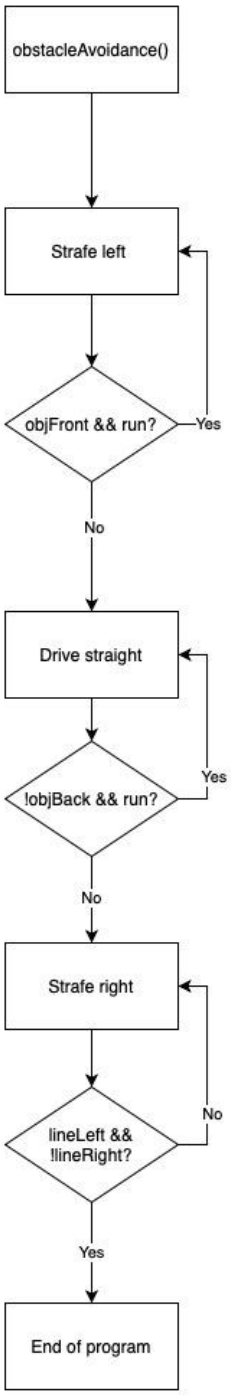
1. Write engine driver code: From the sample WaveShare code we are using the PCA9685, DEV_Config, dev_hardware_i2c, and sysfs_gpio files. Most were modified to support multiple motor HATs.
    a. Remove unnecessary code from DEV_Config files:
        i. Removed all logic related to the WiringPi and BCM2835 libraries
        ii. Removed DEV_Digital_Write(), DEV_Digital_Read(), DEV_SPI_WriteByte() and DEV_SPI_Write_nByte()
    b. Create new EngineDriver files to hold engine functionality
        i. Implement first draft of core functions:
            1. engine_init() → Initialize bottom HAT (addr 0x49)
            2. engine_exit() → close HAT's file descriptor
            3. engine_run() → run a specific motor at a specific speed and direction
            4. engine_stop() → stop a specific motor from running
    c. Modify engine functions and WaveShare libraries to support multiple Motor HATs:
        i. Replace the HARDWARE_I2C struct from the dev_hardware_i2c files with a new engine struct in EngineDriver files to hold a separate file descriptor for each Motor HAT.
        ii. PCA9685 modifications:
            1. PCA9685_Init() → returns the file descriptor of the Motor HAT being initialized
            2. The rest of the functions take an extra int parameter for the file descriptor to distinguish which Motor HAT to operate on
        iii. DEV_Config modifications:
            1. DEV_I2C_Init() → returns the file descriptor of the Motor HAT being initialized
            2. DEV_ModuleInit() → returns the file descriptor for the system
            3. I2C_Write_Byte(), I2C_Read_Byte() and DEV_ModuleExit()→ take an extra int parameter for the file descriptor to distinguish which Motor HAT to operate on
        iv. dev_hardware_i2c modifications:
            1. DEV_HARDWARE_I2C_begin() → returns the file descriptor of the Motor HAT being initialized
            2. The rest of the functions take an extra int parameter for the file descriptor to distinguish which Motor HAT to operate on
        v. EngineDriver modifications:
            1. engine_init() → initialize the file descriptor values of the engine struct. Calls DEV_ModuleInit() for the system file descriptor. Initialize each Motor HAT and set their PWM frequency to 100
            2. engine_exit() → close all file descriptors in the engine struct
            3. engine_run() → run a specific motor from a specific Motor HAT
            4. engine_stop() → stop a specific motor from a specific Motor HAT from running
    d. Implement car movement functions in EngineDriver.c:
        i. runFL(), runFR(), runRL(), runRR() functions to control each individual motor
        ii. engine_strafe_left(), engine_strafe_right(), engine_run_cw() and engine_run_ccw()
2. LiDAR Handling
    a. Modify LiDAR driver code
        i. Remove unnecessary print statements in main.cpp for ultra_simple
    b. Write handler functions

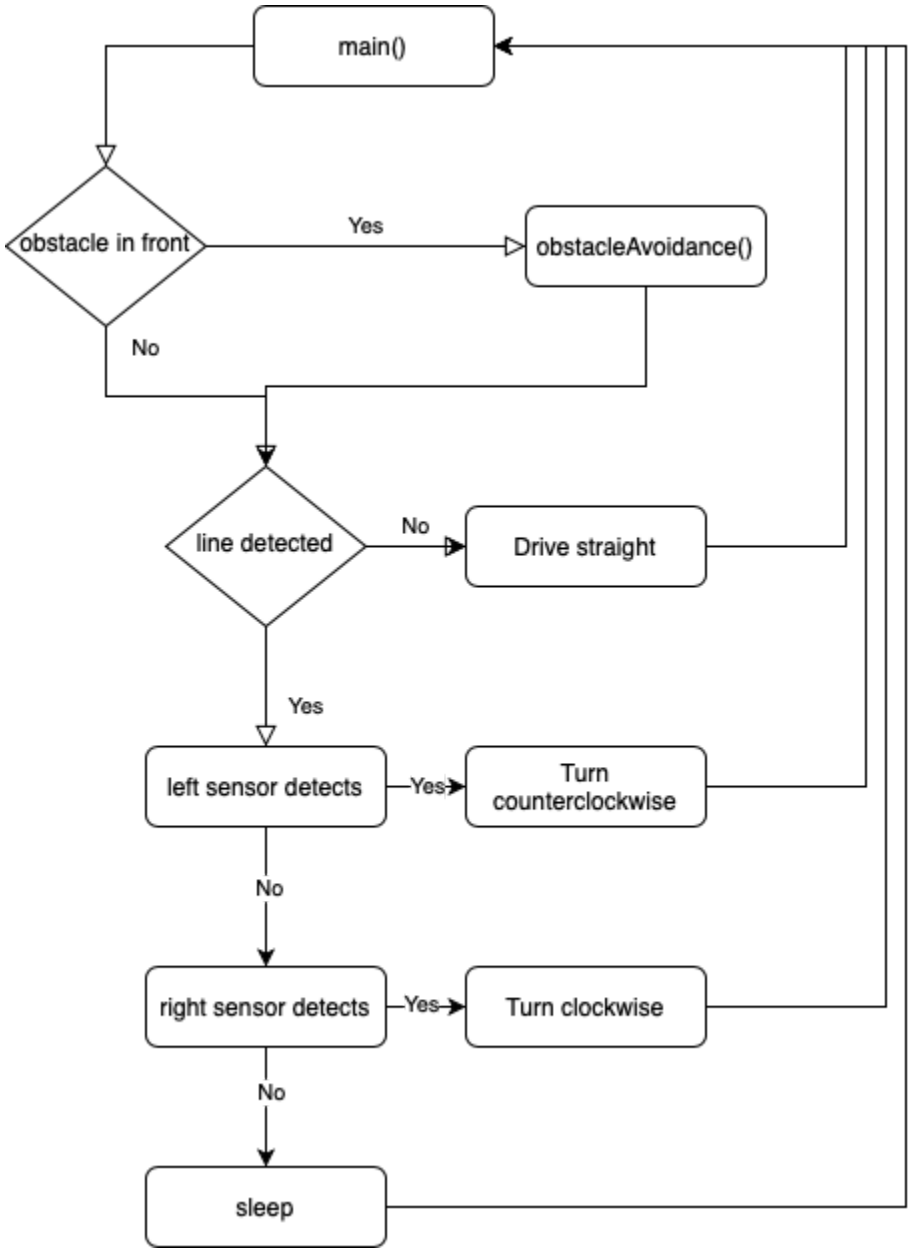
   i. lidarInit() → initialize file descriptor for reading lidar values. Returns a copy of the file descriptor for usage in functions.
   ii. lidarTerminate() → close lidar file descriptor
   iii. readLidarWriteGlobal() → read the file descriptor opened by lidarInit(). Tokenize output and check to see if an object is in front of the car and/or behind the car.
3. Test engine functionality
   a. Obstacle avoidance
      i. obstacleAvoidance() → while an object is in front, strafe left. Then, drive straight forward until the object is not detected behind the car. Then strafe right until the right line sensor detects the line and the left one does not. Then stop the car.
4. Write line sensor code:
   a. Global variables lineLeft and lineRight hold the value output by the line sensors. They are updated by lineSensorLeftRoutine() and lineSensorRightRoutine() respectively.
   b. In the main execution loop inside main.c, the values of lineLeft and lineRight determine how the car moves
      i. While no line is detected go straight
      ii. While left line sensor detects line, turn counterclockwise
      iii. While right line sensor detects line, turn clockwise
      iv. If both sensors detect line, keep current motion until line is centered

Libraries/Software

- rpLidar C++ library, modified by Faiyaz
- PCA9685, DEV_Config, dev_hardware_i2c, sysfs_gpio

Code Flowchart

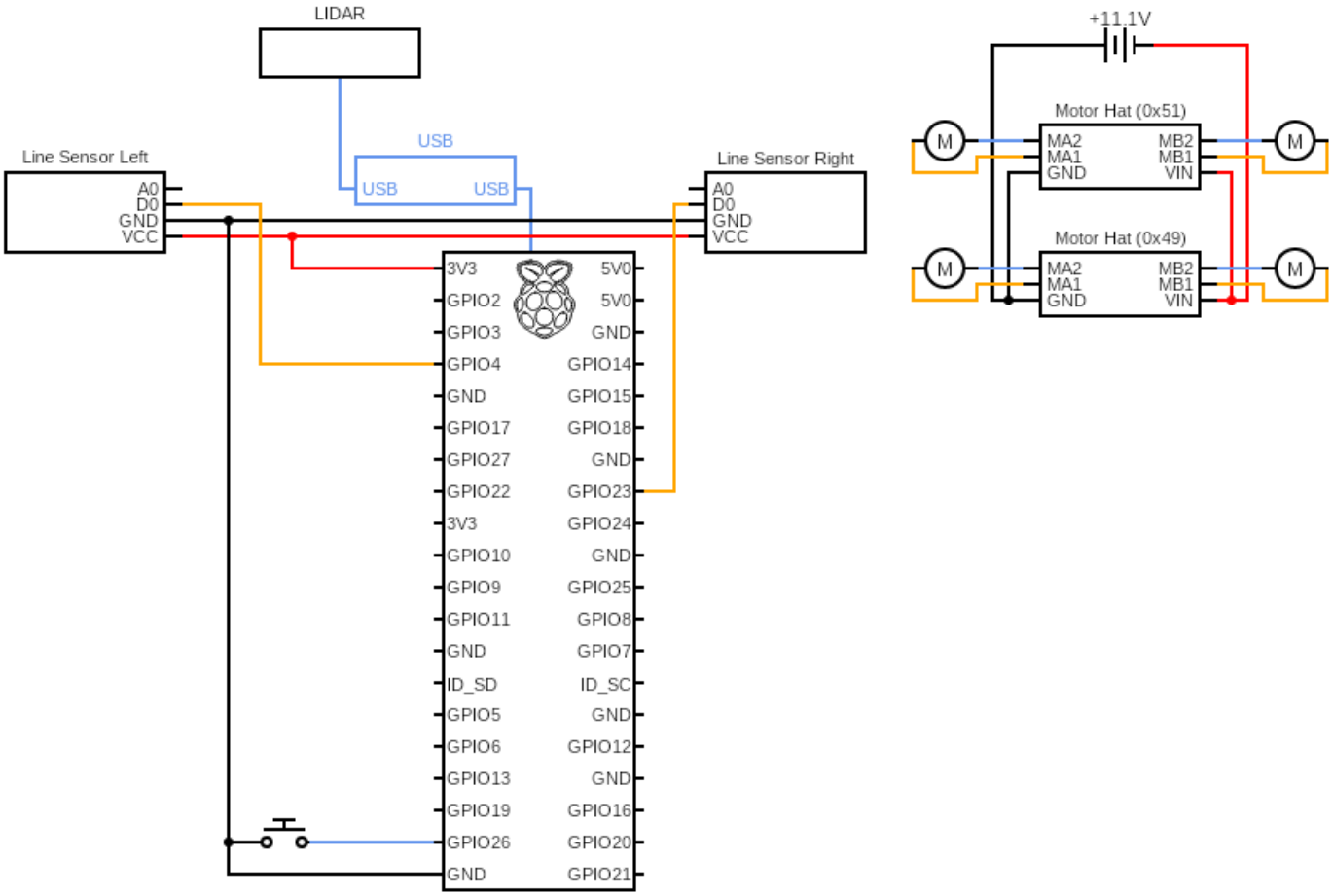




Pin Assignments

GPIO Pin	Use
26	Button
4	Front left line sensor
23	Front right line sensor

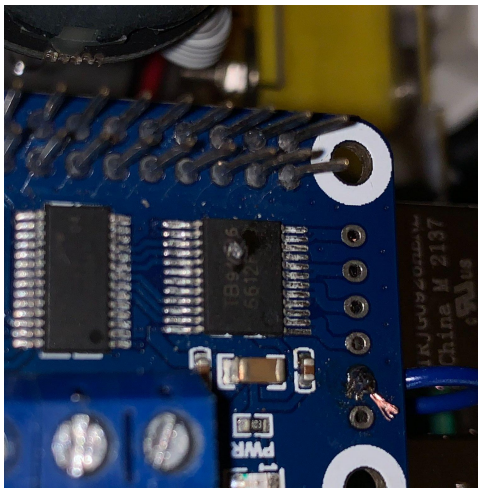
Hardware Diagram



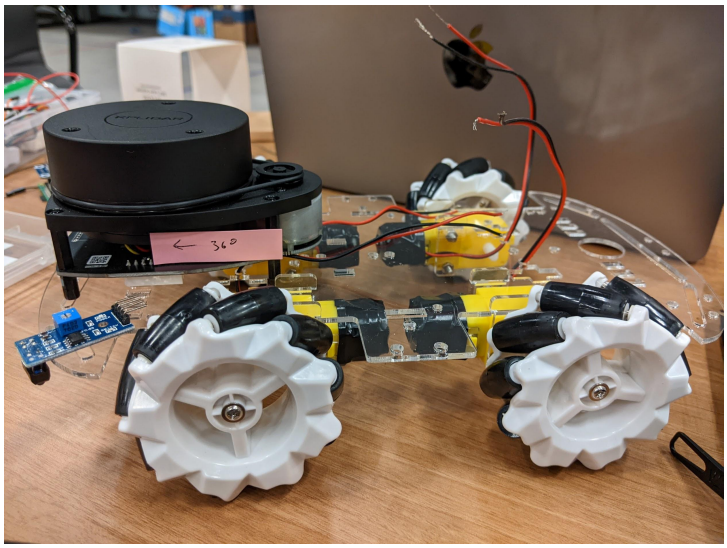
Reflections

Issues

A. Our batteries kept draining very quickly. We suspect that keeping the LiDAR on during development time is what caused this issue.
B. When first using the batteries to power all 4 motors, the top Motor HAT burned out likely because of a faulty ground connection to the other HAT.



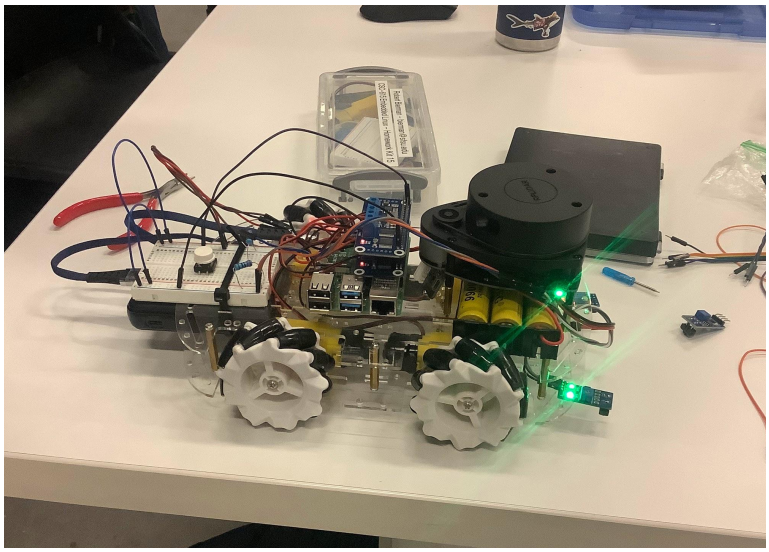
C. We originally tried to use a single acrylic board to house the sensors, electronic components, and batteries. But there was not enough room for all the electronics. The battery packs and the raspberry pi also blocked the Lidar sensor's field of view.



D. The LiDAR drivers presented an initial learning curve since they were written in C++, and the installation of the device, while documented, was not entirely straightforward.

Solutions / Successes

A. The battery draining issue became much more apparent once we started to run the final course. The best that we could do at the time was to swap out the low batteries with fresh ones and disconnect the battery pack when it was not in use.
B. On the old Motor HAT we used jumper wires to supply power to the top Motor HAT. On the new Motor HAT we chose to solder the wires directly to the Motor HAT board.
C. We decided to go with the two layer chassis design. Using metal standoffs, we added the second acrylic board just enough to fit the motors and the raspberry pi's battery pack. We also use the metal standoffs to raise the lidar above the other electronics.



D. After installation of both the driver for the USB and the A1 LiDAR software, we ran extensive tests on the built software. Using the LiDAR was possible by researching pipe(), fork(), exec(), and popen() to see how the best way to call the C++ built code in C. I talked to Professor Bierman about it, and he suggested using pipe() since that is how it was done in the past, but I found that using popen() was more straightforward since it handles the fork()ing and exec()ing of a specified command.