Microdelivery Robot, Based on Raspberry Pi and FRDM Board

{jdlopezm, arielmb, ztwang}@bu.edu

Introduction

One paramount issue to solve during a pandemic is the spread of viruses, and one solution to this is to reduce the contact risk among the population as much as possible by utilizing delivery robot technology. Among the systems currently in use, Amazon's Scout Vehicle and Starship Technologies' robots are able to realize the most efficient way to their destinations. We intend to conduct experiments to further explore building a microdelivery robot to help make our college and community safer.

In this project, our team aims to create a microdelivery robot by combining a Raspberry Pi and a FRDM board with sensors and algorithms. Collectively, we have extensive experience with building robots. Jonathan previously worked on vehicle projects using Arduino and hard programmed states and Ariel previously worked on vehicle projects, such as FRC robotics and VEX robotics in competitions, as well as OpenCV. Zhengtao has experience with working on communication systems and Arduino, but nothing related to robotics.

Proposed Considerations

Resources

The resources we were thinking of using for this project goes as follows:

- Aws Kinesis
- Wireless
- Ubuntu Linux Desktop 21.10
- Soldering iron
- Robot
- Lidar
- Cameras
- Cmake

The resources we used for the project were the following:

- Wireless
- Ubuntu Linux Desktop 21.10
- Soldering iron
- Robot
- Lidar
- Cameras
- OpenCV
- Cmake
- C++

- Java
- Python

Technical Risks & Risk Management

1. Overheating from the Raspberry Pi and FRDM board
2. Power draw and current flow
3. Connection issues with FRDM board and Raspberry Pi
4. Learning a new system and making sure we understand it
5. Can't use bash and python for building the system
6. Faulty hardware
7. People stealing the robot when doing tests
8. Interrupted by walking people
9. Learning the new software we have to use, for example, RoboMaker
10. Implementing the hardware in Raspberry Pi

Every engineering creation comes with technical risks of which we had to take into consideration. Some of the risks inherent in this project were obvious; for example, connection issues with the FDRM board & Raspberry Pi, learning the new hardware and software, and unforeseen failures that come with testing. Some of the obvious connection issues we foresaw were wire placements and connections, but a more complex issue we were concerned to may encounter was the power requirements between the two boards not matching or terminals malfunctioning.

To make this a successful school project, we had to learn new hardware and software packages to be applied to our robot, and learning something new always comes with some unforeseen challenges. The learning curve being applied to our robot may be grander than what we expected or the documentation may not be as comprehensive as we would like.

One of the final concerns or risks we anticipated would come from the unknown. One of the most common unknown risks comes to light when testing commences; for example, maybe someone will interfere in our robot's path to reach its destination or maybe the weather will destroy the robot. While some of these unknown risks may be unavoidable, we must be vigilant to those we can avoid at the time of testing. All of these risks mentioned above are things we expected to encounter and planned accordingly to avoid major setbacks on the project.

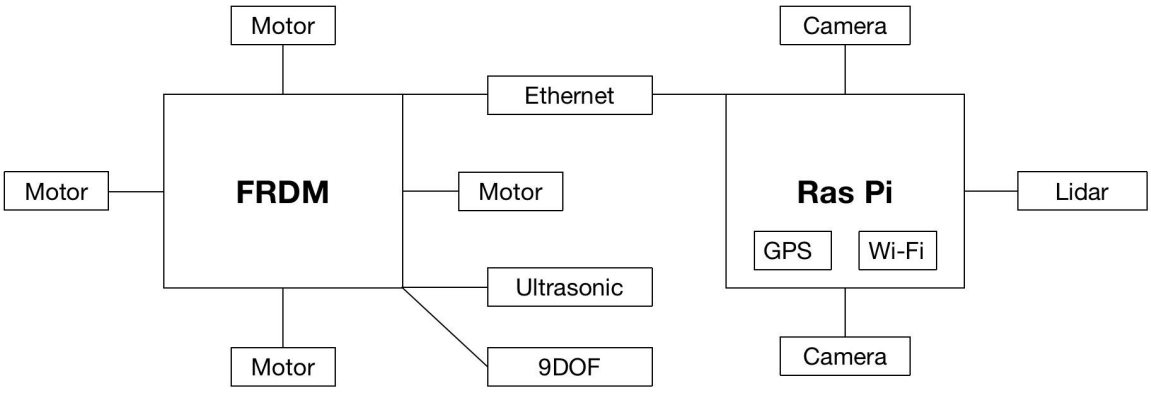
Technical Approach

Here is our scheduled timeline:

- Week 1
  - Designed the robot
  - Learned RoboMaker (or any other aws repos)

        - Split up the work
        - Worked on understanding OpenCV with lidar and cameras
        - Bought sensors
    - Week 2
        - Learned RoboMaker
        - Put the robot together
        - Worked on OpenCV with cameras
        - Started looking into lidar capabilities
        - Learned how to implement GPS
    - Week 3
        - Created model of what we need on any aws package
        - Programmed and connected the FRDM board to the Raspberry Pi over Ethernet and sent the code commands from the Raspberry Pi to it
    - Week 4
        - Created a bash script with opencv and connecting it to Kinesis
        - Worked on communication with kinesis to FRDM board
    - Week 5
        - Continued working on communication
    - Week 6
        - Constructed the robot
    - Week 7
        - Tested and debugged
    - Week 8
        - Finalized project

Here is our technical approach:



When creating a project that has several parts, we had to take into consideration how it will be modularized. The modular parts were as follows: designing the robot, programming camera vision, programming GPS data, connecting FRDM board, and lidar implementation. Each part was broken down into smaller parts based on each group member.

As part of the design of the robot, we added a simple box to store the packages that were being delivered. We also designed on SolidWorks a way for the Raspberry Pi and FRDM board to be situated such that it would be very intuitive for connecting all the sensors and motors together. The Raspberry Pi was connected to the FRDM board via an ethernet cable to send code, to act as a controller for the motors.

When programming the whole system it was broken down into parts, with each part having its own separate task to complete all in C++. The whole system was mainly run off of the Raspberry Pi, due to the amount of processing power it can support. The Raspberry Pi will be running Ubuntu Linux Desktop 21.10, just for the ease of use, since we all knew how to use that distro. Each part of the system was run through a thread, because we are running multiple systems simultaneously. The GPS data was directly received and connected from the raspberry due to the amount of data being drawn from where the robot is in the world. Camera vision was through OpenCV. We planned on using 2 cameras to have Binocular Stereo Vision, similar to humans. The cameras received data at about 120px120p, to ensure that we didn't kill the Raspberry Pi with everything else that was running on it. This would help us receive data such as lights changing, people walking in the way and other information. We also looked into lidar capabilities to help make sure that the robot knows its 3D space and surroundings so that it would not fall into holes and hit walls. While the robot was in motion, data was sent back to AWS Kinesis to enable us to check that the information from the sensors were being received correctly, visually. We created a bash script to allow the system to constantly run on the Raspberry Pi. This would make sure that it never died during use. All of the systems that were built would be connected together through a CMake file and then sent through the bash script. We also used the 9DOF, 9 degrees of freedom breakout board from adafruit that would help take in more data such as acceleration, magnetic orientation, and angular velocity. This would let us know if the vehicle had been moved by someone, bumped into, going too fast or anything else.

Milestones

Here were our estimated milestone goals.

- Milestones
    - Monday Week 1
        - Getting starting with project
    - Monday Week 2
        - Getting the programming starting
    - Monday Week 3
        - Piecing Together the Robot
    - Monday Week 4
        - Getting robot to communicate
    - Monday Week 5
        - Working code
    - Monday Week 6
        - Putting the code together with the robot
    - Monday Week 7
        - Working Robot
    - Monday Week 8

- Completing robot

Discussion of Failures

The Raspberry Pi was a problem to begin with. Jonathan brought his Raspberry Pi to school and we could not connect it to the school's internet for the first 45 minutes. We also ran into another problem which was that the screen was touchscreen, but without a touchscreen keyboard enabled. So, running to the desktop computers to grab a keyboard was our only option. Some quick thinking happened and we were lucky that we were already in Ingalls Engineering Library. In Ingalls, there are a number of desktop computers that are hardwired via ethernet to the school's internet service provider, ISP. We took the ethernet cable, usb keyboard and mouse; plugged them all into the Raspberry Pi to get everything setup. When it finally worked we encountered how slow wireless connection for downloading a stable working Raspian would be.

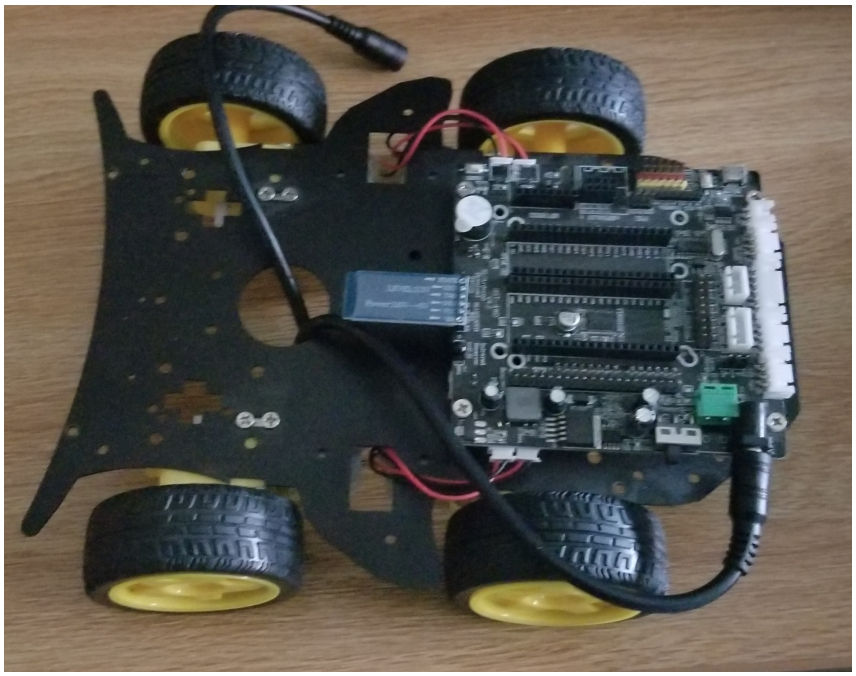
The design of the robot was based on an Amazon simple roverbot. The motors are simple brushed DC motors that are connected to a motor controller unit. The motors are connected in series on each side of the robot. The motors behaved like tank style driving; we tested this by sending a PWM into one of the 2 PWM input pins and two motors spun. The controller is a PID controller that is controlled by the FRDM board which is connected to a Raspberry Pi via ethernet. The Raspberry Pi will be running constantly to keep receiving data from the GPS and ultrasonic sensors, as well as sending that processed data to the FRDM board. The oscilloscope was used to test PWM output of the FRDM board for the PWM channels in an attempt to make a HC-SR04 template and PWM signal for the motors. We figured out what the duty cycle range should be from the motors using the oscilloscope, and realized how hard it was to generate a 10 μs pulse for the HC-SRO4 ultrasonic sensors. We were able to control the motors with the FRDM board now, once we got the duty cycle for the PWM situated. The code was essentially thrown together by using the sample codes from the IDE. We started reverse engineering and understanding how all of the code worked for the FRDM board because we wanted a way to connect the data from the sensors to the FRDM board programming.

The lidar sensor was another problem we tried working on. We sadly lost the custom UART to the serial converter that would have connected the lidar to the Raspberry Pi. We then started looking for an online store that might have it, but it had a 15-week waiting period. We purchased an off-the-shelf UART to serial converter, but the lidar wouldn't transfer the data properly, since the lidar had a custom UART to serial, instead of using the standard off-the-shelf ones. We figured this out since the manufacturer provided some code with which to test the lidar and we couldn't get the motors to spin. The lidar has 6 pins instead of the usual 5 pins on the standard UART to serial converter. The 6 pins consist of: 2 grounds, Rx, Tx, 5V VCC, Motor Controller (MOTORCTRL), and voltage for motor (VMOTO). The lidar is separated into 2 modules, as stated in the datasheet, the Motor interface module and the Core interface. The

motor interface has 3 pins: ground, voltage for the motor, and PWM input for the motors. The Core interface has 4 pins: Voltage for scanner core (5V), Tx, Rx, and ground. The motor and scanner both accept 5V, but the specialized UART to serial converter has two different 5Vs for each interface. The off- the-shelf UART to serial converter has 5 pins: VCC, ground, Tx, Rx, DTS, and CTS. The difference we encountered was the DTS/CTS with DTR. Upon reading the datasheet, DTR sends PWM information to the MOTORCTRL. The off-the-shelf doesn't have such capabilities, to our knowledge. We then decided to buy the parts of the cable that would allow us to solder the cable pieces together, but ended up realizing that the headers on the cable would interfere with GPS connection.

Motor Controller:

As mentioned earlier in the paper, we used many SDK examples provided from the FRDM-K64F SDK. First, we learned how to control the GPIO pins on the FRDM board as controlling them wasn't intuitive at first. The GPIO pins are going to act as digital outputs to the robot as 4 pins on the robot control whether it's going to go forward or backwards on each pair of motors. Once we got that going we learned to generate a PWM to the motors as that is how we can control the speeds. There are specialized PWM pins on the FRDM board and we had to cross reference with the wiring diagram to see which pin on the pin configuration corresponds to the pin on the physical board. Then, we did the same with the FTM pins on the FRDM board, to figure out where the PWM pins correspond to on the board. To get the robot to follow the GPS coordinates we needed some PID controller to control the robot to go towards that location. We needed a reference frame so we decided to use the accelerometer and magnetometer combo that is built in the FRDM K64F board. To get the GPS coordinates information, we used Ethernet for the FRDM Board. Unfortunately, the FRDM board had a fatal error towards our last stages of testing, so we had to switch to the Raspberry Pi within the last few days and programmed it in Python for its rapidness. The robot we used is the Yahboom 4WD Car as shown below:



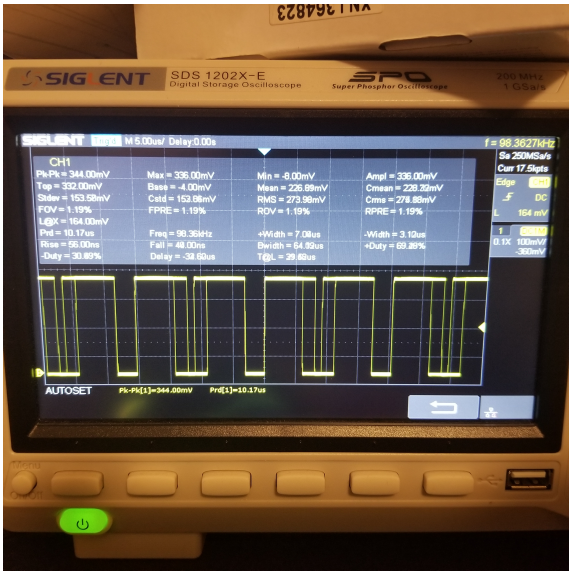
Our Robot

The GPIO LED output was the first example we looked at to learn how to write to the GPIO pins on the FRDM K64F board. This was a difficult first process, since everything was brand new and our prior experiences were with arduino. Fortunately, over time, this became more intuitive. This example taught us how to use the MCUXpresso config tool to activate certain pins that can be controlled either through GPIOA, GPIOB, GPIOC, or GPIOD. Each different GPIO port is separated and can be controlled in separate threads. The pins in the same GPIO port can be controlled separately as well, but are locked if the port is currently in use. We used this example to write to the robot since the robot has 4 different digital ports that control the direction the robot is moving. The next step we took was to write a PWM code to the motors to control their speeds.

The Freertos generic, FTM PWM Two Channel, and FTM Simple PWM examples were used to generate a pulse width for the motors. Originally we though we needed to hardcore the pulse duration which is why we first referenced Freertos generic. Freertos genetic shows us how to generate a timer with the built in clock. This method was too difficult and luckily there was a specific SDK package that is used to generate a PWM signal and that package is FTM. The two examples that help us learn how to use FTM are FTM PWM Two Channel and FTM Simple PWM. We first looked at FTM Simple PWM and that showed us how to control 1 PWM channel and that helped us figure out how to configure FTM pins using the MCUXpresso config tools. There are 4 FTM specific ports to use that can control 2 different pins per port. Later we found out that the robot has two sets of motors, so we need to control those motors with 2 different FTM ports. FTM PWM Two Channel helped us produce 2 different PWM channels to control

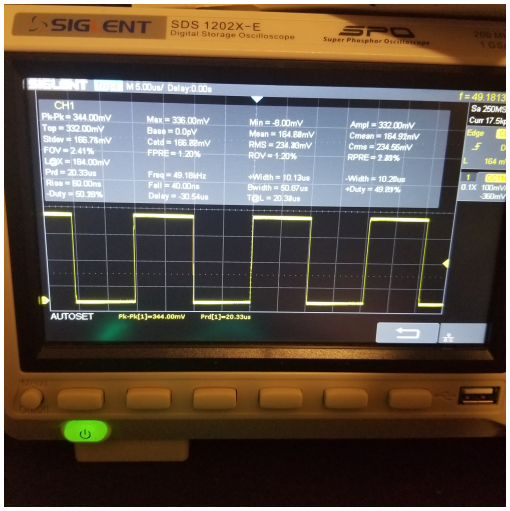
the motor speeds separately. We assigned 2 different FTM ports for each motor then controlled it that way. Next we tried to create an ultrasonic sensor code to get distance.

The SDK example FTM Timer provided code to set up a timer circuit using the FTM ports as the clock. We used this example to try and get a very precise pulse, 10 µs, to activate the ultrasonic sensors. The FTM timer works by setting a cycle time with the FTM ports as the clock and runs through the code that way. The example counts the seconds by adding 1 each time the code is cycled. This method is not incredibly precise as the professor provided and gives us messy pulses, as shown in the figure below.



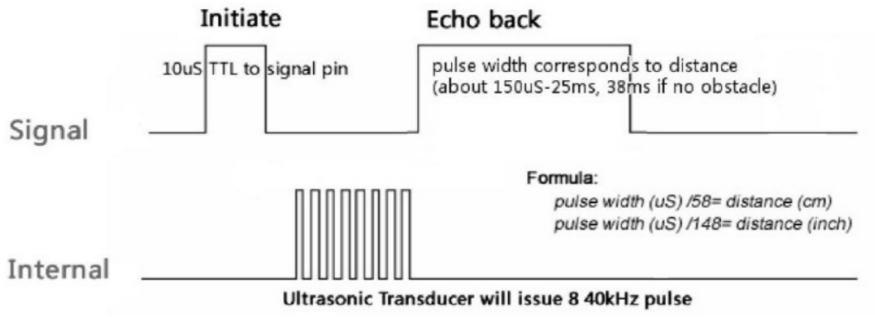
FTM Timer PWM

Another method we tried deals with a more sloppy approach but has a cleaner pulse wave and produces the desired 10 µs shown below; although, we could not figure out how to time a given pulse within the allotted time.



FTM Delay PWM

We wanted these parameters since the device we are using as our sensor is the HC-SR04 ultrasonic sensor as our range finder since we couldn't get a UART to serial converter for the lidar in time. The HC-SR04 works, as stated by the user manual, by receiving a 10 μs then sending 8 ultrasonic pulses to the object then waits for the returning wave and outputs a 5V pulse until the microphone does not receive any more pulses. The initial pulse is called the trigger pulse and the returning pulse is called the echo pulse. The echo pulse can be converted to distance by taking the time of the pulse width and using the speed of sound to convert that to distance. The operation is shown below as given by the HC-SR04 user manual:



Trigger Echo Operation

We couldn't get this working on the FRDM board and we had the ok to write this in python, so we switched this over to the Raspberry Pi and the process is the same but more consistent. Our original range finder sensor was a lidar unfortunately we lost the custom UART to the serial converter, so we couldn't use it. The custom UART to serial converter is shown below:

Lidar UART to Serial converter

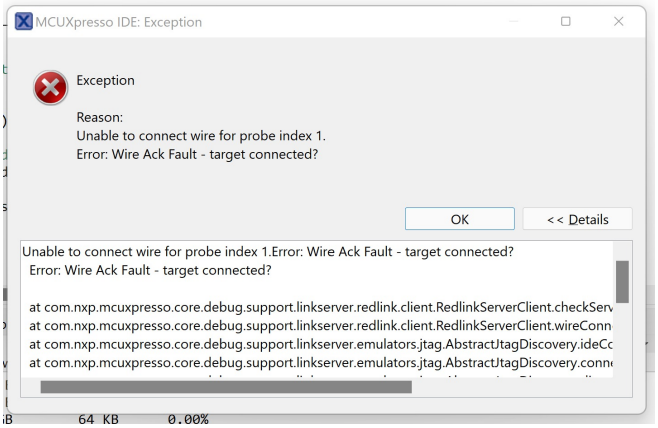
We purchased an off-the-shelf UART to serial converter, unfortunately it does not come with the capabilities this custom converter has. This converter has the typical outputs: GRND, Tx, Rx, and 5V VCC, but the difference is that it has two 5V ports for the sensor and motor in the lidar. The DTR does come with UART to Serial converters but the one we purchased didn't seem to have that, instead it had DTS and CTS output ports. As time was drawing nearer to the

deadline we decided to swap to ultrasonic sensors. The next step we took to figure out on the FRDM board is to get the robot's axis but using the built in magnetometer and accelerometer.

The SDKs examples we used are: cmsis I2C read accel value transfer, ENET Tx Rx transfer, Freertos generic, FTM PWM Two Channel, FTM Simple PWM, FTM Timer, and GPIO LED output. The accelerometer, while built-in, uses the I2C protocol to send information to the Cortex M4 processor. We used the I2C example in an attempt to get accelerometer and magnetometer readings, and we did but the readings were in raw values so we read the specific accelerometer (FXOS8700CQ) data sheet to convert the raw values into g's. There are 3 sensitivity modes that the data sheet provides: 2g, 4g, 8g modes. Each mode has a different conversion value, and the mode we chose was 2g since the robot was not moving that fast. The conversion value for 2g is 0.244 mg/LSB, and LSB stands for least significant bit. The magnetometer also sends data via I2C protocol and also sends that data with raw values. The datasheet provides the conversion for this as well, and the conversion is 0.1 μT/LSB. Now we have the axis we need the GPS coordinates to develop a PID controller to drive the robot. switched to the Raspberry Pi as the controller.

The next SDK used was the ENET Tx Rx transfer. This SDK shows us how to use the ethernet port built in the FRDM board. The purpose of this is so we can receive the GPS coordinate and ultrasonic data from the Raspberry Pi. The example shows us that the ethernet sends byte data over the ethernet so we have to figure out how many bytes the raspberry pi was sending. The GPS coordinates are floats and each float is 4 bytes in C, as well as 1 byte being sent over for the ultrasonic sensor to indicate if the robot is close to an obstacle or not. The total bytes being sent is then 13 bytes. Once we got all the prerequisites running, we combined this all to one file to run the robot.

To get the robot to follow the GPS coordinates, as stated earlier, we used a PID controller. The accelerometer/magnetometer sensor gives the robots its personal axis and we utilized that axis to generate the angle from the front of the robot to the GPS coordinates. With those relative vectors established, we calculated the angle between the coordinates to the GPS vector and set the desired angle to 0 so the axis aligns with the GPS and follows it. Fortunately for us the FRDM board has a built-in PID controller function. We set the Kp, Ki, and Kd values for the PID controller as well as the measured angle and desired angle. The PID controller then calculates the error in between and seeks to reduce that error to 0. Unfortunately, as we reached testing this PID controller we happened upon an error with the FRDM boards that persisted to both our teammates FRDM boards and could not solve the issue within the allotted time. The error is described below:

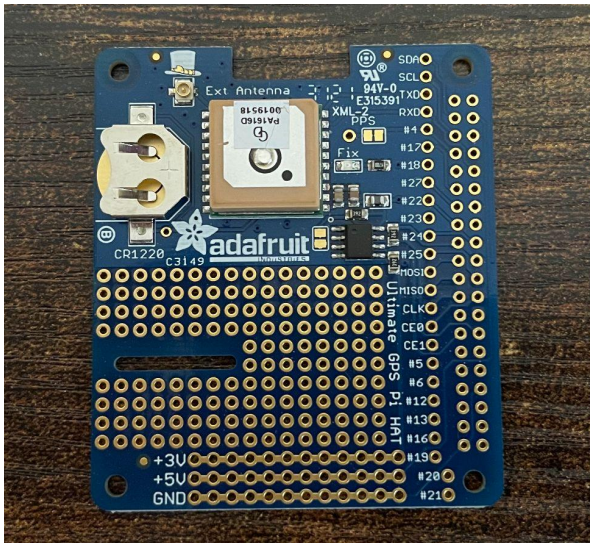


Fatal Error for the FRDM Board

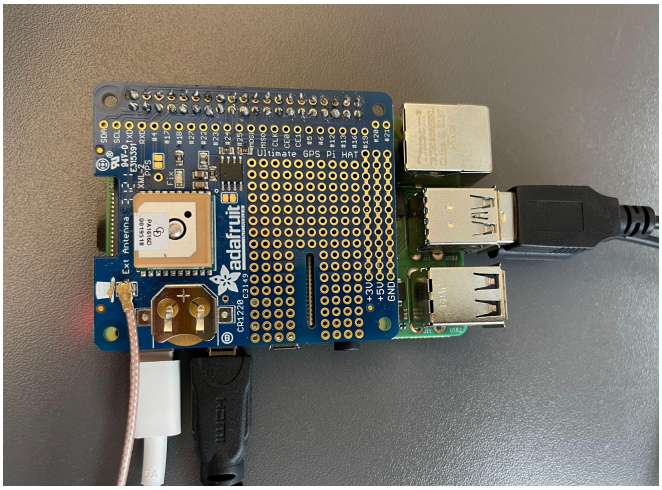
We did test the motor controller during the PWM phase to see if the PWM outputs the desired results which it did, but unfortunately some error happened during the last stages of the motor controller. The next topic is detailing our endeavors with the GPS sensor.

GPS

The GPS had many problems starting out. The original code was designed for an arduino and then we found code for the Raspberry Pi GPS, but sadly it was written in Python. We then realized that if we wanted to test the GPS, we would need to solder the pins to the GPS shield, but since we didn't have a soldering iron, we had to wait until we had time to get to the EPIC lab when it was open to get in.



Original GPS module

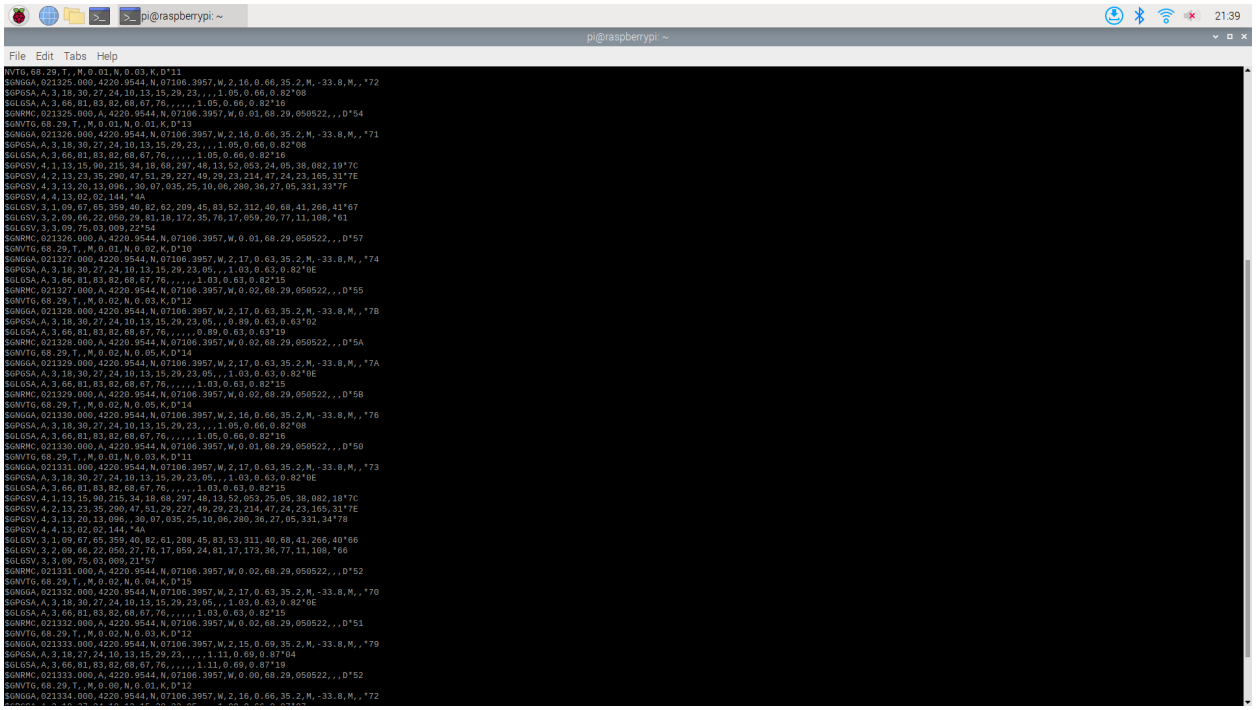


Soldered GPS module connected to the Raspberry Pi

When the GPS shield was finally soldered, we then started testing and running different examples from online sources to help us understand how the GPS worked better. First, we needed to set up the GPS and make it receive data from satellites. Because the GPS module needs a very ideal environment to obtain data from the sky, we also had to buy an external antenna to make it work more effectively.

A good position for antenna

The position of the antenna is also a problem with which we had to deal. It needed to be placed close to the window or outdoors and the upper side had to be directed straight to the sky. Through trial and error, we searched many places to find the right location, because we couldn't connect to the satellites indoors, even using the external antenna. Eventually, we found a window that led to the outdoor patio. and the GPS finally worked. When it did not get signals from satellites, the signal light would blink a red light on and off once every second which meant "NO FIX." After successfully connecting, the light would blink once per 10 seconds, which meant "FIX."



Raw data from GPS

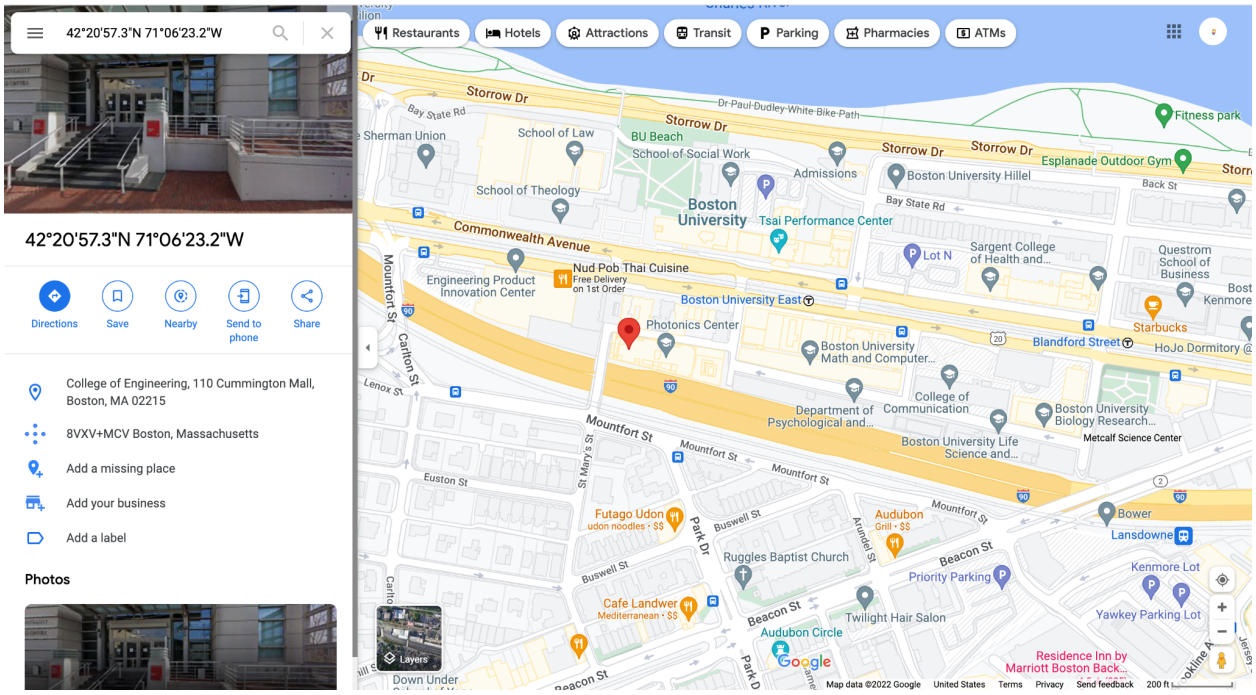
Below is the information we needed to use to get our location data from the GPS.

*$GNRMC, 205829.000, A, 4220.9548, N, 07106.3860, W, 0.04, 73.56, 040522,,,D*5D*

The first part 205829.000 is the current time GMT (Greenwich Mean Time). The first two numbers 20 represent the hour and the next two are the minutes, the next two are the seconds and finally the milliseconds. So, the time when the code shows is 8:58 pm and 29 seconds. The GPS does not know the time zone of its location, so we have to calculate it into our timezone (which is 4:58pm and 29seconds EST).

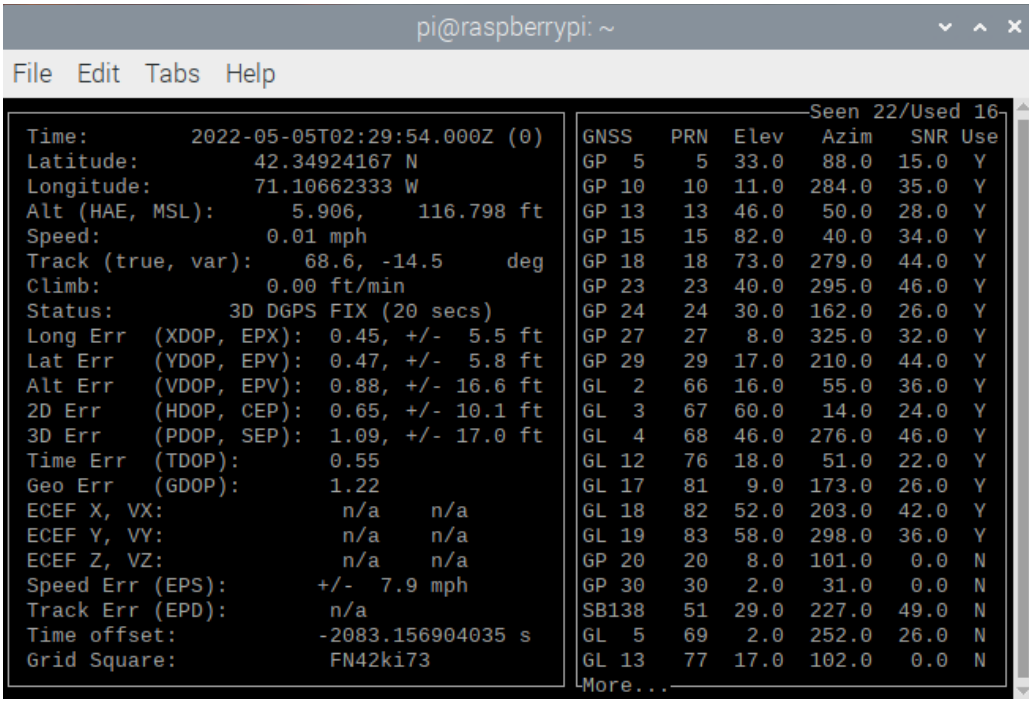
The second part is the “status code.” V means that the data is Void (invalid) and A means that it's Active (the GPS could get a lock/fix). As shown in the code, the status is Active.

The next 4 pieces of data are the geolocation data. According to the GPS, the location is 4220.9548, N (Latitude 42 degrees, 20.9548 decimal minutes North) and 07106.3860,W. (Longitude 71 degrees, 6.3860 decimal minutes West). We used 42°20.9548'N 71°06.3860'W (=42°20'57.3"N 71°06'23.2"W) to find our location in Google maps, or using sign +/- to represent the NSWE, where N and E are positive, S and W are negative. So, finally we typed the location to figure out where we were by Google map.



Our testing location shown in the Google map

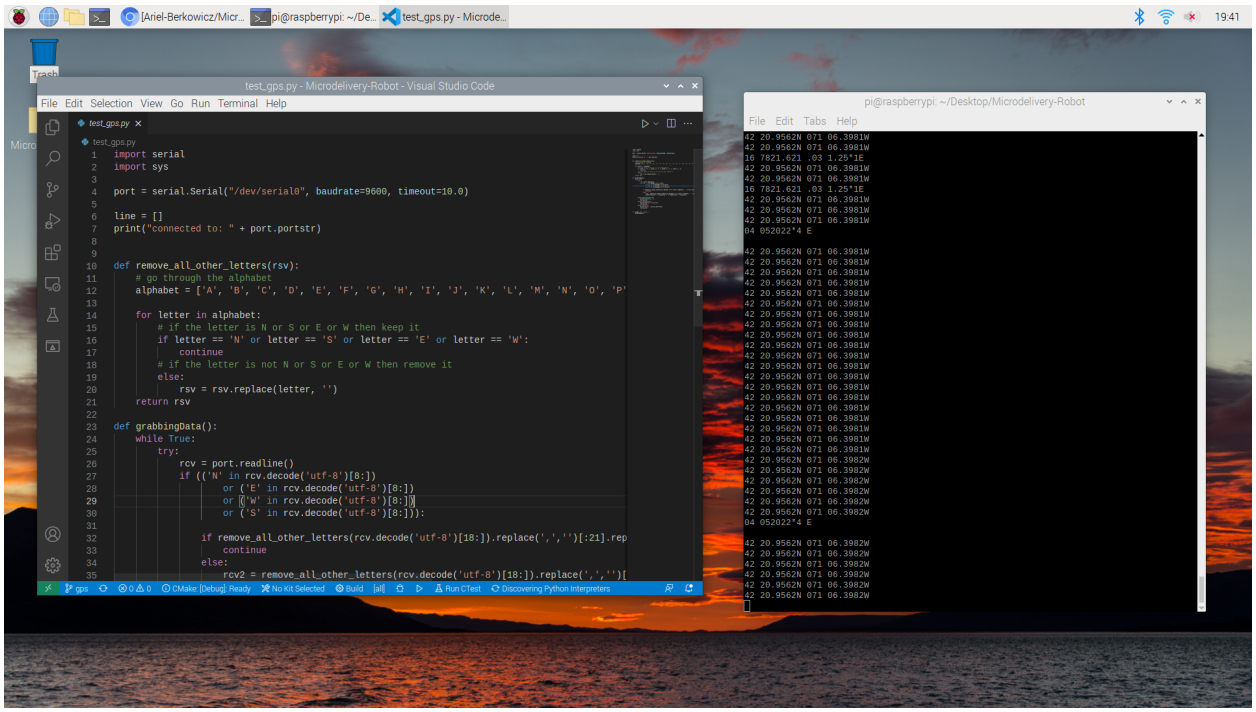
Bingo! The location(Red Pin in the Google map) is very accurate. The testing of our GPS is exactly above the west entrance of the Phonetics center. This GPS is often accurate to 5-10 meters and it perfectly made it.



GPS data in GPSD

After the process, we planned to use the GPSD daemon to make the GPS raw data to a standard form that is efficient to read and use. We tried many ways to use the GPSD data but it didn't work. We then started researching and coming up with new ways of getting around this problem. We came up with one way which was by taking the data that was being processed every second from the GPS module and then using a C based script to figure out where we were in space. It didn't work as you can tell. We then looked into writing a bash script to do the same thing but for some reason our device wasn't being read at all but was being read just in general based on the initial data. We then started looking into programming in Java instead but the device wasn't being read either and we then decided to use the library that was working and then see if there was a different way to find the GPS device but sadly it didn't work.

We also tried many different ways to use C++ or Python for data reading but failed. The most problems we encountered were the libraries we couldn't use. As a result, we decided to use the raw data from GPS. Because the GPS module receives the data every second, we could update our robot location simultaneously. At the same time, the GPS would generate some useless data that we don't need such as the information of satellites. First, we removed these unnecessary things and left the only data we needed. When processing the remaining data, we try to make it into an effective form for Google map. We discarded all the other information but the latitude and longitude: the location numbers and the direction marks.

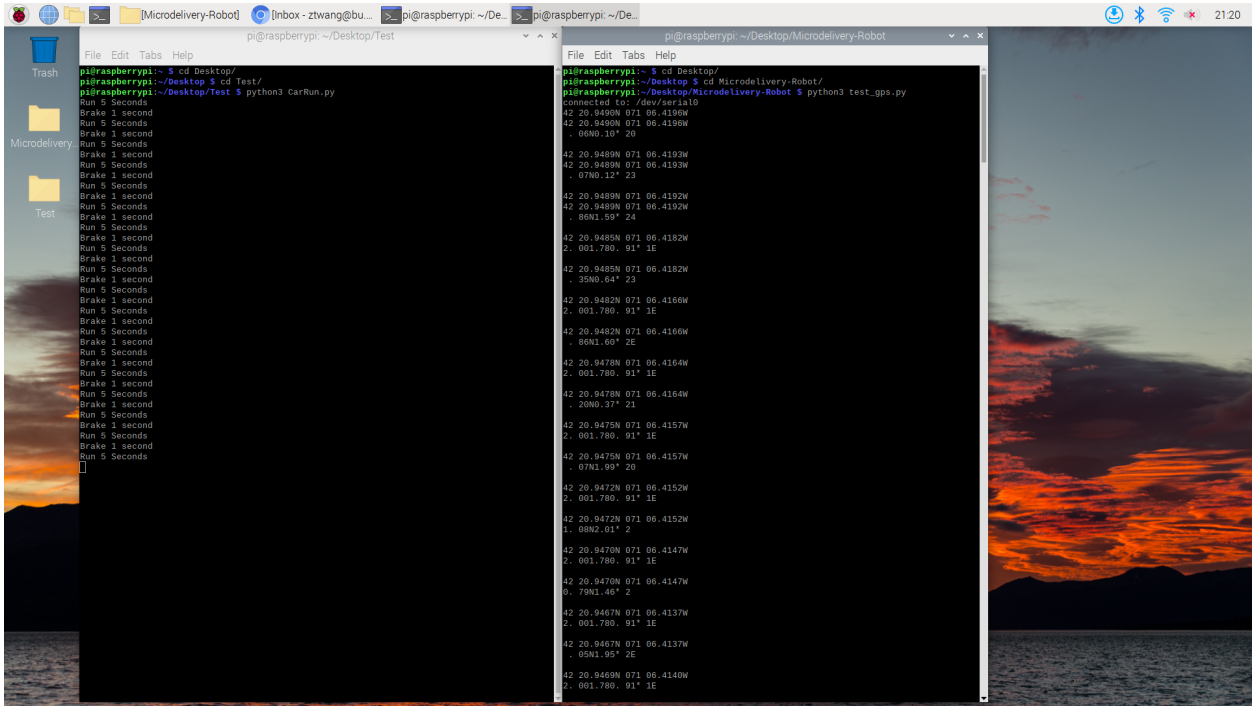


Data Reading

Since the data is ready, it's time to use it for robot localization.
We also had a plan for visualizing the robot's location and movements. We tried to use Pubnub to stream data and visualize it by Google map. First we need to register an account and create a publish key, a subscribe key, a secret key for Pubnub and an API key for Google map which we need to pay money for using it. Next we create a html file where we can directly see the visualized data, just like on Google Maps. Unfortunately there were so many errors when we dealt with it, so we deleted it and canceled that plan.

The ultrasonic sensors are being used to check the distance of an object that is presently in front of the robot. When the robot sees an object about 50 centimeters from the sensor, it will stop and then turn left to right and right to left to make sure nothing is in their way to pass around the object. If there is another object on the left it will turn to the right and pass it by that way. If the object was on the right it would pass it by the left.

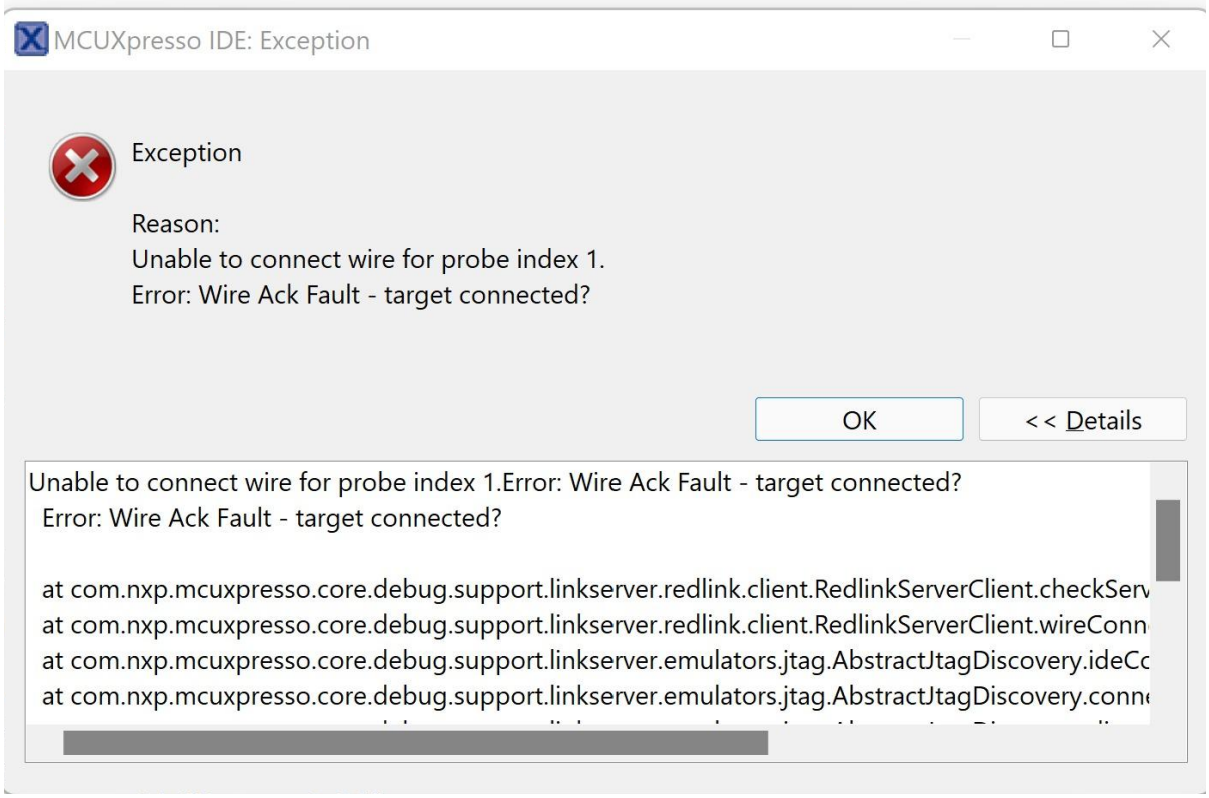
Since we spoke to Professor Babak Kia, we were allowed to start using python and other programming languages to fix a lot of the bugs. As we tried doing so, time didn't allow us to continue working on our other courses.



We were able to find a couple of examples using the GPSD which is designed for the antenna on a raspberry pi in python but were unsuccessful. We were able to find a solution by grabbing the pure data from the antenna. The pure data was based on compass coordinates: North, South, East, and West. The antenna had a lot of unnecessary numbers, alphabetical letters, and symbols

that messed up receiving and understanding the data currently. We took care of that by creating simple preprocessing that was able to remove 98% of the unnecessary information. The only negative part was having to deal with the sporadic times of random single cases such as other compass directions that didn't make any sense. We wanted to make sure that the data we were receiving was acceptable for using an API such as Google Maps or MapBox for testing later down the road. We chose to use a GPS shield for the raspberry pi as a form of connecting to an antenna that was placed outside because the buildings we are in are so thick that are similar to a faraday cage. Which makes it harder to read the GPS signal from the satellite correctly. From there we were going to take that data and use the magnetometer and accelerometer from the FRDM board to find the location of where the robot is going.

Ariel was setting up the Raspberry Pi



After the FRDM board stopped working, we then pivoted from the FRDM to the Raspberry Pi. We then started looking into the easiest way to get the robot to move. We were able to find online a prebuilt sample code for the exact robot we were using. We grabbed that code and then realized that the GPIO pins that they had were not going to work because they were based on the GPIO header on the raspberry pi. The GPIO pins were all being used by the GPS Shield so we found that the shield had GPIO pins header locations that were free that we could use so we then switched the code to take into account the headers that were free which worked. While testing the code for the run command we noticed that half the motors were going backwards which made a circle for the robot. We tried changing half of the motors that were going backwards but for some reason the motors wouldn't run when we flipped from forwards to backwards on them at all. We were lucky that the other set of motors would switch without any issue. The robot then was driving only backwards from now on but we thought about it as though it was the new way of driving forwards. The motors are using a motor controller that uses PWM connections to let the motors know how to turn at a certain speed and action whether forwards or backwards. We chose to run the simulation for 5 seconds of running and a 1 second time for braking just to be able to test the system without causing the robot to run off the table. We weren't able to finish everything in time for connecting all the parts but we were going to create a queue system that would split the sockets in programming to allow both the antenna and the driving part to run at the same time.

Video
https://youtu.be/bcVRozAmJ3M

References

https://learn.adafruit.com/adafruit-ultimate-gps-hat-for-raspberry-pi/basic-test
https://github.com/YahboomTechnology/RaspberryPi-4WD-Car/blob/master/4.Code/python/CarRun.py
FXOS8700CQ - https://www.nxp.com/docs/en/data-sheet/FXOS8700CQ.pdf
FRDM K64F Wiring Diagram - https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F
FRDM-K64F Freedom Module User's Guide - https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F
HC-SR04 User Manual - https://web.eece.maine.edu/~zhu/book/lab/HC-SR04%20User%20Manual.pdf