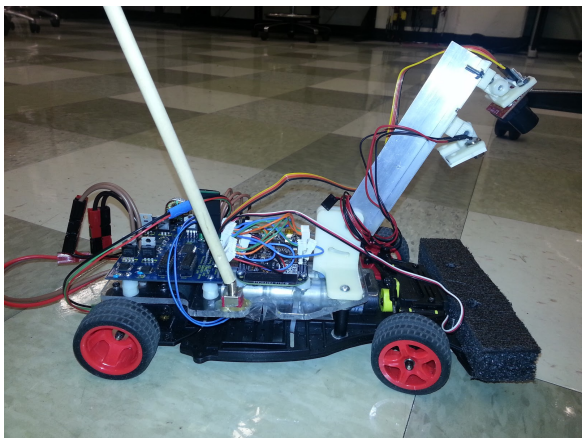
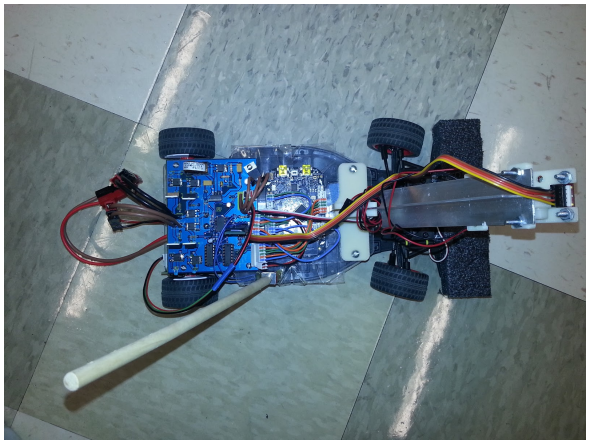
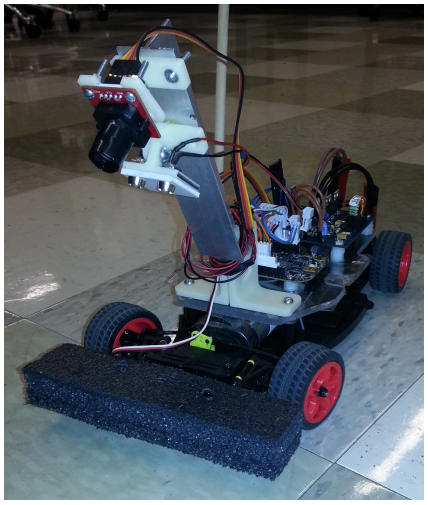
EECS 192 Progress Report

Team 8

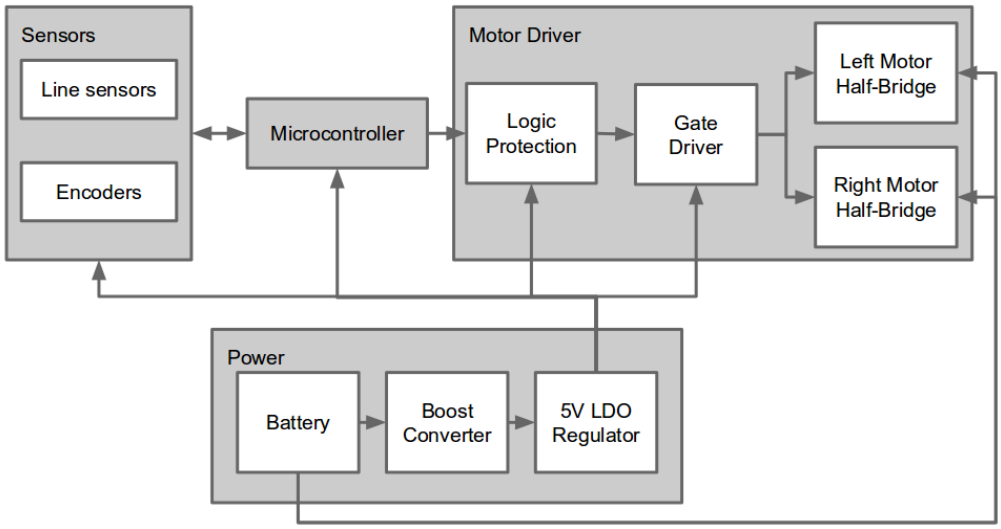
Songqi Gao
Greg Kahn
Kiet Lam

1. Current State of Project

We currently have most of the hardware and software functional. We are able to run the figure 8 at 2 m/s in varying light conditions. In preparation for the upcoming races, we will improve and debug the hardware and software so that our car can consistently complete the outdoor track.

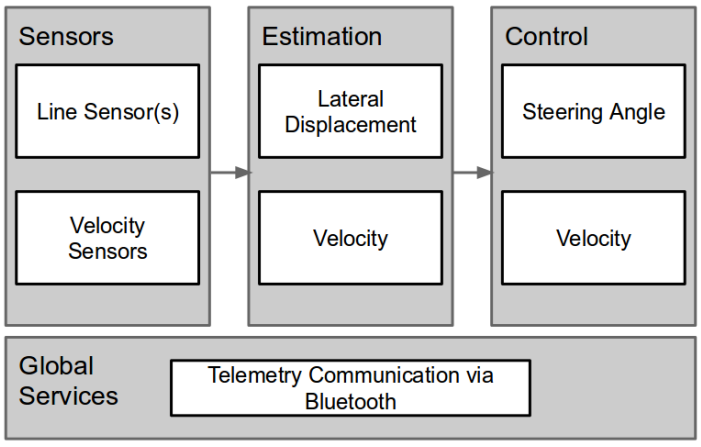
Hardware



Below is the progress report for each module (not done, partially done, done):

- Sensors
    - Main line sensor - still need to determine the optimal height and angle
    - Secondary line sensor - we are currently planning on mounting active lighting in its place
    - Wheel encoders - only one wheel encoder is operational
- Microcontroller
- Motor Driver
- Power

Software



Below is the progress report for each module (not done, partially done, mostly done):

- Sensors (see hardware above)
- Estimation
    - Lateral displacement - need to test on outdoor track
    - Velocity - need to test responsiveness
- Control

    - Steering angle - add derivative term and tune on outdoor track
    - Velocity - test controller from HW2
- Global Services
    - Telemetry communication via bluetooth - works from one of our laptops, need to get others up and running

2. Hardware Documentation

Figs. 1 and 2 show different angles of the CAD model of the RC car. Fig. 3 shows a labeled top view of the car with pictorial representations of the position of all the major hardware components and connections of the motor drive circuitry.

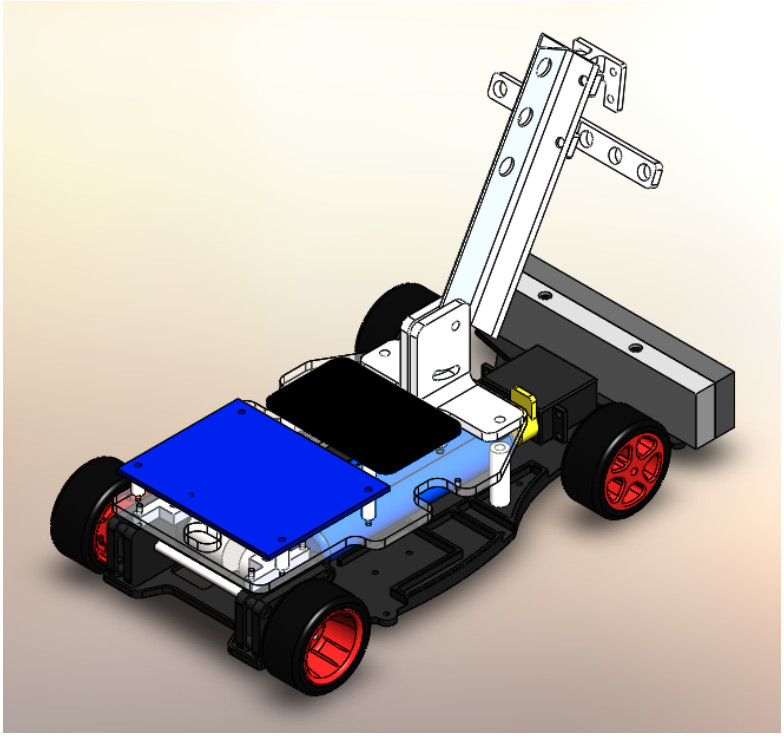


Figure 1: Isometric view of RC car. The blue panel represents a mockup of the PCB, and the black panel represents a mockup of the FRDM-KL25Z MCU.

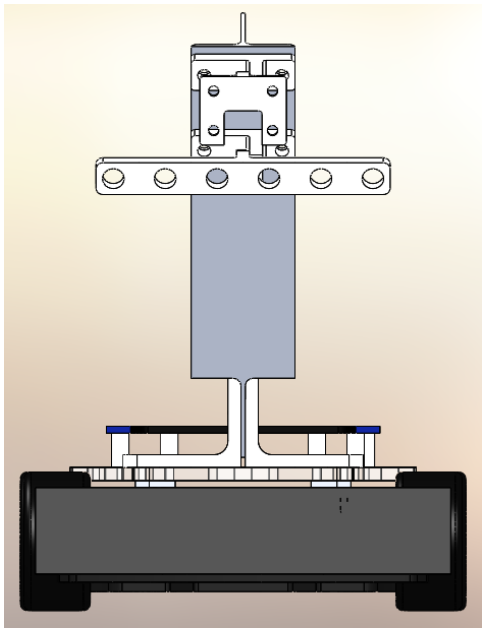


Figure 2: Front view of the car. An aluminum T-channel provides an elevated position for mounting the line sensor. The sensor mounts (shown in white near the top of the image are angle-adjustable).

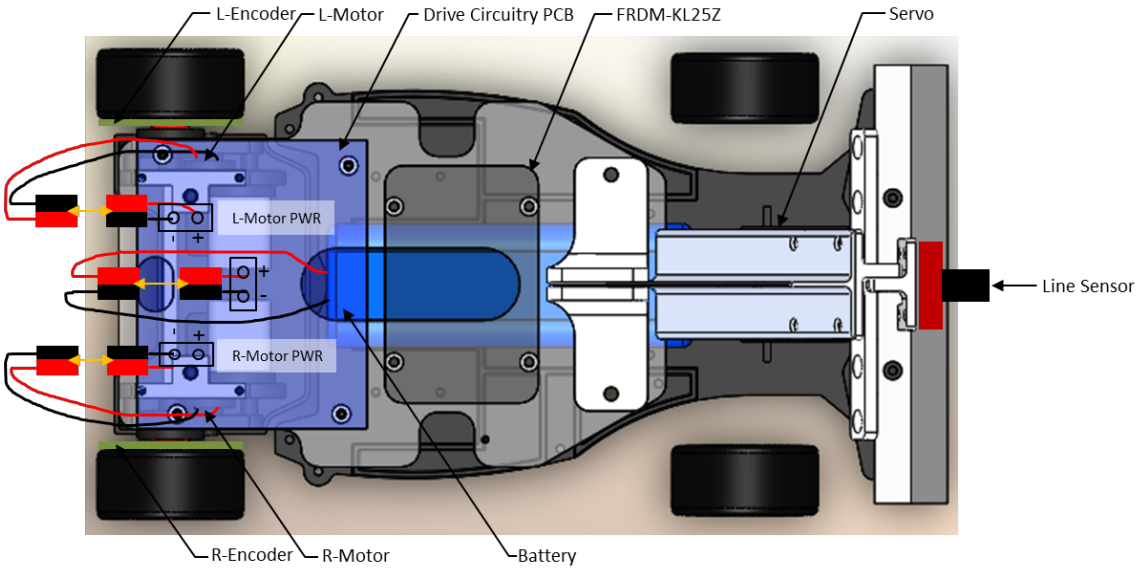


Figure 3: Top view of RC car showing positions of each major component. Note that the connection of the left motor to the power out from the PCB is red-to-black to reverse the rotational direction of the wheel such that both wheels spin "forward".

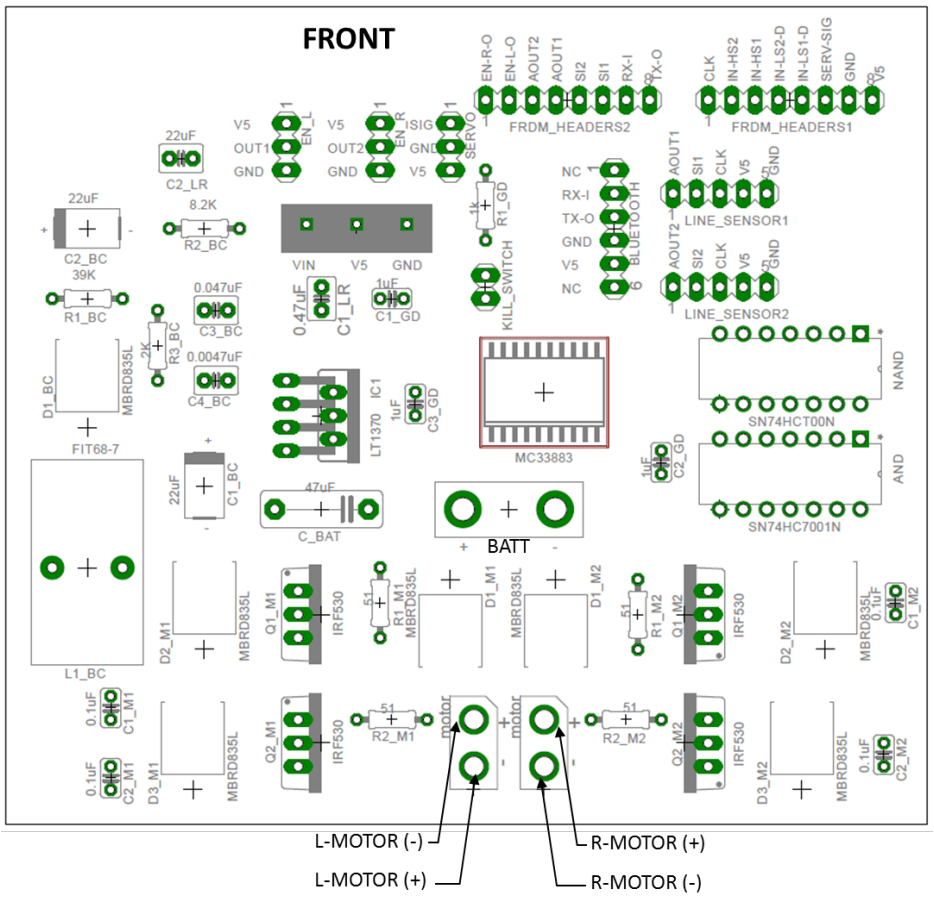


Figure 4: Labeled car driver PCB. FRONT designates mounting orientation on car.

The car drive circuitry PCB is mounted on the rear portion of the car. In front of it sits the FRDM-KL25Z MCU, which controls all of the movements and responses of the system. Each pin on the PCB that interfaces with either the MCU or other hardware is labeled with what it connects and which specific pins perform each task. EN_L and EN_R refer respectively to the left and right optical encoders. The pins on the MCU that are used to interface with the hardware are shown in Fig. 5. A legend for each of these pins is as follows (from left to right):

SI1: Line sensor 1 signal
SI2: Line sensor 2 signal
IN-LS1: Left motor low control signal (PCB routs this signal to the gate driver chip)
IN-LS2: Right motor low control signal (PCB routs this signal to the gate driver chip)
IN-HS1: Left motor high control signal (PCB routs this signal to the gate driver chip)
IN-HS2: Right motor high control signal (PCB routs this signal to the gate driver chip)
RX: Bluetooth receive pin
TX: Bluetooth transmit pin
EN-L-O: Left encoder signal
EN-R-O: Right encoder signal
CLK: Line sensor clock

SERV-SIG: Servo PWM signal
AOUT1: Line sensor 1 analog output
AOUT2: Line sensor 2 analog output
V5: 5V power line from linear regulator on the car drive PCB
GND: Common ground

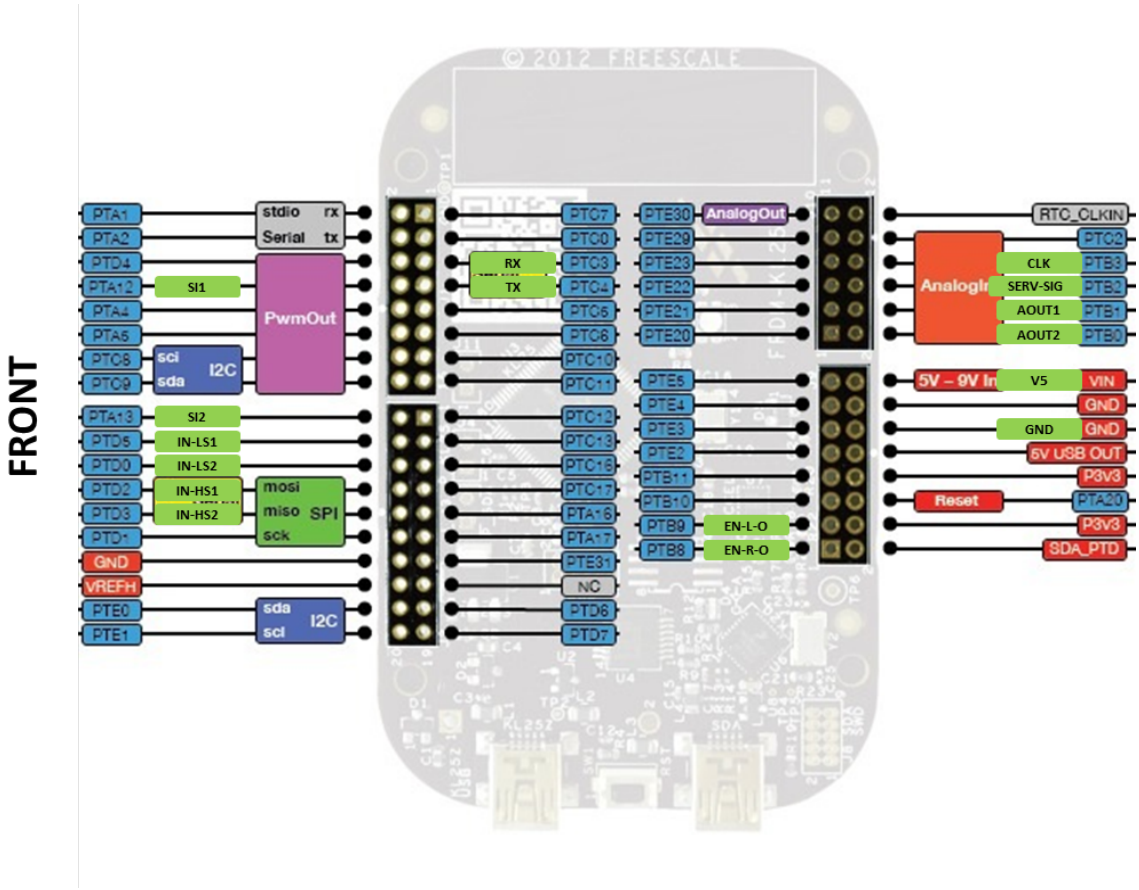


Figure 5: Labeled pinout of FRDM-KL25Z MCU. FRONT designates mounting orientation on car.

3. Proposed Control Methods

a. Velocity Control: We previously had a PI control for our speed control, but it was not functional on the figure-8. So currently we are not using a velocity control and are just using an open-loop PWM reference. We plan to
   i. Debug our encoder / velocity control code to find which was the problem.
   ii. Tune PID constants so that we have an optimal response time and as low of an overshoot as possible.
   iii. Incorporate higher level logic to have better velocity reference to our PID controller instead of a constant velocity reference.
      1. We can do simple logic like m*v^2/r to determine the reference velocity to the PID controller.

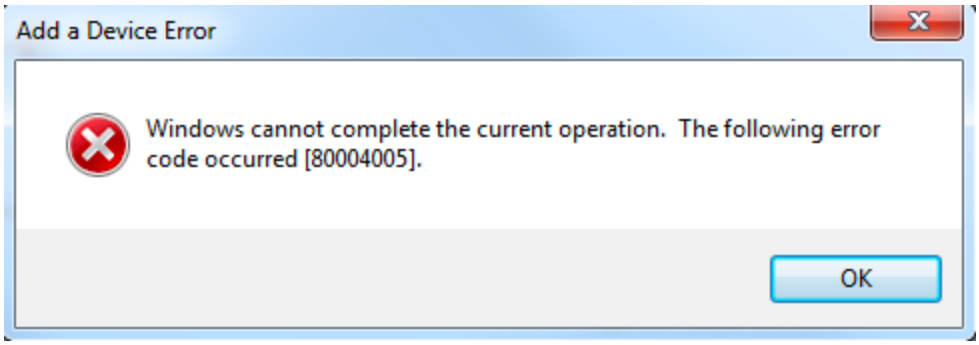
b. Steering Control: Currently we are using a simple P control for our steering control. We calculate our estimate of the line center to [-1, 1], corresponding to the line center being all the way in the left or all the way in the right of the line sensor. We plan to
    i. Test the robustness of our steering control for different parts of the track like pretzels, loops, crossings and etc...
c. Stabilization: Currently we have no verifiable way of making sure our system is stable besides running the car on the figure-8. We plan to
    i. Test on the physical figure-8 / track and see the response through telemetry on Bluetooth.
    ii. Test the step response of the system to verify that it is stable.
    iii. Determine sampling rates and use simple system identification to understand the stability of the system in discrete time.

4. Interim Budget

All are approximate

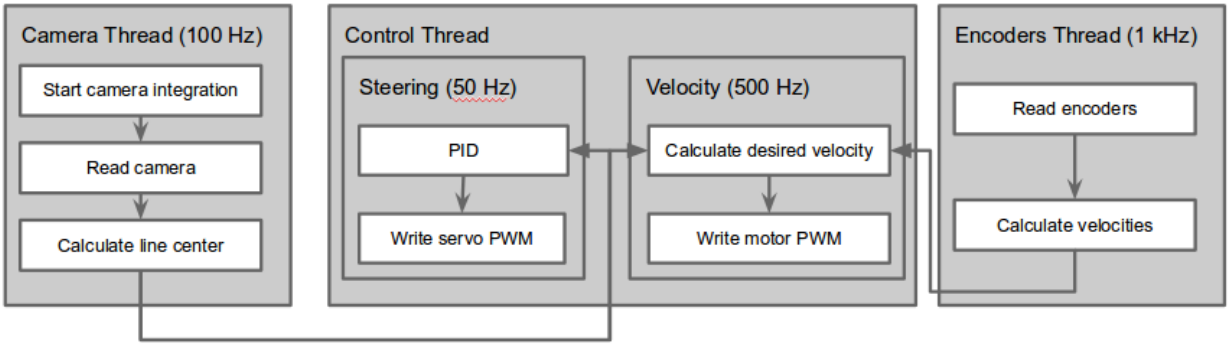
- Motor Drive: 10 hours
- PCB Design: 10 hours
- Boost converter: 20 hours
- CPU board understanding: 5 hours
- Line sensor: 20 hours
- Velocity sensor: 6 hours
- Software: 45 hours
- Bluetooth: 13 hours
- Hardware construction: 20 hours
- Total: 144 hours
- Out of pocket expenses: about $30.

We have ran into significant problem getting bluetooth to work, especially getting it to work with Richard's telemetry package. The lab's computer gives a generic error that did not give us clues on how to debug it.



We were finally able to get telemetry over bluetooth working using a Macbook Air. Figuring the correct PWM pins on the Freedom board with separate timers took a lot more time than expected because the PWM pins interfered with each other and the documentation on the pins was not very easy to find.

5. Refined Proposal for Software Architecture



We designed our software to be modular while also limiting the number of threads. The main thread is the control thread, which reads in values from the estimators (camera and encoders) and then calculates and sets the desired steering/wheel set points. The line camera is read on its own thread because we need to consistently read it so that the integration time is consistent. The encoders are on their own thread because they update at a very fast rate.

Each logical component (camera/control/motor/steering/encoder/IO) are in their own separate classes. All constants (e.g. FRDM pins, PID constants, etc.) are kept in a single header file.

Details of each subsystem:

- Camera
    - Completion: 80%
    - Estimated time to complete and debug: 3 hours
    - We are currently able to do the figure 8 in varying light conditions at 2 m/s. To deal with outdoor and varying conditions, we are going to write an auto-calibration function that sets the integration time so that the line reading is neither too low or saturated. We also plan to read the camera values using FastAnalogIn because it currently takes us 3ms to read the data. We could run into issues if calculating the line center cannot be done within the integration time, in which case we'll have to create a separate read camera thread and use a double buffer.
- Steering
    - Completion: 80%
    - Estimated time to complete and debug: 2 hours
    - Using a P controller we are able to do the figure 8 at 2 m/s. We need to put everything into SI units and then tune the P controller by analyzing data output by telemetry. We will judge the controller based on stability and course completion. Then, we will do PD to enable more aggressive control.
- Velocity
    - Completion: 80%
    - Estimated time to complete and debug: 1 hours

    - We will first determine the maximum constant velocity at which the car can complete the course. Then, we will play with linear/non-linear mappings that map the current lateral error and min/max velocities to a desired velocity. We will use a script to determine the best mapping based on the telemetry data.
- Encoders
    - Completion: 75%
    - Estimated time to complete and debug: 2 hour
    - We need to confirm the velocity estimate is stable and updates quickly enough. We can determine stability from the telemetry. We will test update speed by inputting a sinusoidal PWM and ensuring the velocity estimate has a similar shape. Finally, we will fix the scaling by having the car do the figure 8 at a constant velocity, timing the runs, and determining the error in the velocity estimation. We may also switch to using interrupts, but will hold off on this because we will have to switch pins and we also heard teams are having trouble.
- Global Services
    - Communication
        - Completion: 100%
        - Line waterfall
        - Line center
        - Estimated velocities
        - Steering angles

6. Additional Resources Required

Besides active lighting, we are done adding electrical/mechanical components. We do not foresee needing additional resources.