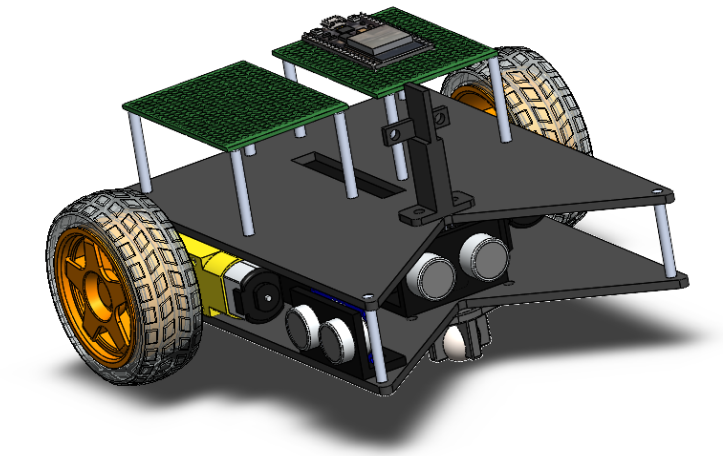
# Final Report: Grand Theft Autonomous



*Team 3: Jason Friedman, Niko Simpkins, Andres Voyer*

# Contents

## Functionality

Our overall approach to win the GTA game was to develop a simple, yet agile and quick robot. Robust mechanical, electrical and software components were all necessary to achieve the desired functionality. The mobile base was constructed using a differential drive architecture with two motors and a singular, central caster at the front of the bot. The small form factor of our bot (7'' x7") in conjunction with the differential drive system allowed the bot to be very agile when maneuvering. Certain keyboard keys were mapped to control the movement of the bot through the graphical user interface on a constructed webpage. Since agility and speed was our top priority, we accordingly named our bot *Lightning McQueen.*

Our game plan to achieve the maximum possible points in the matches was to firstly focus on moving visible cans on our side of the field to the bonus square area. The beacon would follow. Next, in order to grain time points for autonomous control of our bot on the opposing side of the field, the plan was to have our bot autonomously follow walls, ultimately making its way back to our visible side of the field. Lastly, general defensive tactics would be employed to deter other bots from stealing cans in scoring position from our side of the field. In order to achieve this gameplan, the following functionality was necessary:

### *Manual and Autonomous Can Grabbing*

Holding true to our design criteria of simplicity, the bot uses a V-shaped can collector in the front. By driving the bot towards a can, the collector guides the can to the center of the robot and pushes the can to the desired location. In order to autonomously move to a can, we employed the use of a single Vive location sensor to give the bot's position on the field. Given the cans would be transmitting their XY location via UDP, our bot receives the respective can's XY location and moves towards it using an autonomous movement algorithm that will be described in the following sections.

### *Autonomous Wall Following*

Our bot's wall following technique employed the use of two ultrasonic distance sensors; one on the front of the base and one on the right side of the base. This would allow the robot to autonomously follow the wall's of the field only in a counter clockwise direction. Once the bot is placed on the field, it uses the ultrasonic sensors to detect the closest wall and position itself parallel to it. Next,the bot moves down the wall with the side-mounted sensor ensuring an appropriate distance. Once the front-mounted sensor detects an obstacle or a wall in close proximity, the bot rotates 90 degrees left or counter clockwise. This repeats until the bot completes a full lap. The control algorithm will be discussed in the following sections.

### *IR Beacon Detection*

The IR beacon detection for the 23 Hz and 700 Hz beacons would be accomplished using two phototransistors separated by an optical barrier. This would be mounted to the highest point on

the car to have a clear field of view to the beacons. Functionally, the bot spins in a circle until one of the phototransistors detected a beacon. At this point, the rotation would be slowed to decrease error and continue until both phototransistors detect the same beacon. This step would orient the front of the bot in line with the beacon. Once pointing at the beacon, the bot would move forwards until the front-mounted ultrasonic sensor detects a proximal object, meaning the bot arrived at the beacon.

### *Our Accomplishments*

We were able to successfully implement and demonstrate the individual functionality for user control of the bot through a webpage, autonomous wall following, and autonomously moving to a given XY location of a can. Given time constraints, we were not able to implement the IR beacon detection and UDP receiving of the can's location. Rather, we decided to pass a target XY location for the bot to move autonomously to via our webpage interface. We were not able to compete in the GTA competition due to COVID concerns, and therefore did not find it necessary to receive UDP packets. One major issue that we ran into was discrepancies in the XY coordinate system for differing fields and tests. This would cause significant errors in our autonomous movement since the algorithm required a right-hand satisfiable coordinate system (meaning the Z axis would need to be pointing in the upwards direction). We were able to correct this by running control tests before each run to validate the XY coordinate system and adjust our algorithm accordingly. Lastly, we initially ran into issues breaking out of autonomous modes via our webpage, but were able to fix this by incorporating a *botmode* variable that the handlers would edit. Therefore, by pressing any of the keys (Q,A and P,L) mapped to our user controls, the bot would break out of the autonomous mode it was previously running.

## Mechanical Design

### *Frame/Base*

The frame was constructed using two pieces of lasercut, eighth-inch acrylic. Both the base and upper level shared the same shape and structure; only differing in the mounting holes necessary for the corresponding components on each. The rectangular shape of each level also included a V-shaped cut out in the front to allow for effective pushing of the cans.The base and upper level were separated by standoff fasteners to allow sufficient space and clearance for the electronics on the base layer. This allowed for easy assembly and disassembly. Moreover, the top layer included a rectangular hole to allow for passage of wires from the electronics on the lower level to the upper level circuit boards. We decided to stray away from a holonomic drive system due to the added complexity and space requirements of 4 motors required. Moreover, we wanted to minimize the weight of our bot to increase its speed and limit the need for higher torque motors.

### *Electronic Mounting*

The main components that inhabit the base layer are the 5V portable battery bank, the 2 motors, the 20mm caster wheel, and lastly the two ultrasonic sensors. The upper level was reserved for the majority of the circuit board layout including 2 perfboards raised by standoffs and the 3D printed mount and divider for the phototransistors. All components, but the battery bank, were mounted to the base and upper level through the use of mechanical fasteners. This ensured a strong and robust connection while allowing for disassembly when debugging was needed. The battery bank was fastened to the base layer using double sided tape, as this was a temporary use and would need to be removed after the competition. The two ultrasonic sensors were attached using a 3D printed mount with the connection pins oriented upwards. Both motors at the rear were attached using store-bought motor mounts and fasteners. Lastly, the 20mm caster ball was also mounted on the base layer using fasteners.

***Performance and Revisions***

A major aspect that had to be redesigned multiple times was the method and location of the motor mountings. First off, the motors were mounted too close to the center axis of the base, which caused the car to be unbalanced with not enough weight on the front caster allowing it to tip over backwards. Also, we originally explored using a slotted and glued motor mount, but we found the method to be fragile. For these reasons, we used store-bought motor mounts from Adafruit and mounted the motors as far back as possible to improve the stability of the base. We explored using a 16mm caster at first; however, the smaller diameter made it more prone to get stuck on obstructions on the floor (like cracks in the field). Therefore, we switched to the 20mm caster which performed much better and would not get stuck. We were worried that raising the front of the bot would be a drawback due to the possibility of tipping the cans over when pushing them. However, after some testing with the final weighted cans, the change in height did not affect the pusher's efficacy. Another aspect that gave us trouble was the fact that the protruding inner portions of the motor axel’s did not allow enough space for the 5V battery pack. In an effort to tightly pack components together, we snipped off the respective inner portion of the motor axles. This did not have an impact on motor performance and allowed the battery pack to fit between both motors. Lastly, after attaching the wires to the ultrasonic sensors, we noticed that there was not enough clearance to re-attach the upper level. This problem was easily fixed by adding an additional quarter-inch standoff to raise the height of the upper level. The wires were then easily able to be threaded through the rectangular hole in the top layer to reach the respective circuits.

Overall, the performance of the bot was great. Its small form-factor and simplicity allowed it to be quick and agile both when manually controlled and when performing the autonomous functions. Moreover, the low center of gravity, afforded by the heavy 5V battery pack on the base layer, made the bot incredibly stable to outside interferences.

## Electrical Design

The electrical design can be broken down into three main circuit components interfacing with an ESP32 and Power Distribution, Motor Driving, and Vive Detection. As mentioned in the functionality segment, we had plans to add a beacon detection system in which case we would add a Teensy circuit with an IR Photosensor. It should be noted that all our circuits were designed and implemented using the breadboards. This was in part to optimize time constraints but largely to accommodate for more design adaptability. Additionally, using the breadboard facilitated the prototyping phases as it made a night and day difference throughout the debugging process.

### *ESP32 and Ultrasonic Sensors*

The single ESP32 was the central processing unit for all the functions of the bot. It is the brains behind the logic and control of the robot. The ESP32 receives input commands from the HTML webpage over WiFi and the logic loaded on the microcontroller therefore drives the auxiliary circuits. The 5V and 3.3V outputs of the ESP were used to power many of these circuits as well like the PWM for the motor drivers and the Vive detection circuit. More specifically, the two ultrasonic sensors interfaced directly with the ESP via a trig pin and echo pin. In total, 4 ports were needed for the ultrasonic sensing circuit. The sensors were powered using 3.3V. When testing the ultrasonic sensing circuit, we noticed that it did not effectively read distances greater than 50cm. This was a shock due to the advertised maximum range of around 400cm. At first we thought it was a hardware issue, but later resorted to implementing fixes in the logic. We discarded any readings above 50cm in our code and also implemented a software filter to average readings to eliminate noise. These edits greatly improved the accuracy and efficiency of our ultrasonic sensor readings.

### *Power Distribution*

In terms of power distribution, an Anker Powercore 1300 was used. It was able to supply 5V at 2 amps to two seperate USB ports. One USB port supplied 5V power to the ESP. We then decided to implement a micro usb circuit board adapter to power the motors with 5V directly to the H-bridge. This decision to use an external 5V power source was made to maximize the motor performance given the 5V power from the ESP32 port would not supply enough power to run the motors at their maximum potential.

### *Motor Driving*

We used dual yellow DC motors to drive our robot. The motors themselves were controlled through a dual H-Bridge. The dual H-bridge enabled us to control both motors, their directions, and frequency all from the ESP32. More specifically, the ESP32 would receive input controls from through the WiFi access as they were recorded by the HTML graphic user interface (GUI). The ESP32 would output either 5V or 0V which would signal to the H-Bridge whether to drive a

servo clockwise or counterclockwise. The speed of the motor, however, was set using PWM and we decided to use the same speed for both motors for design efficiency. That said, most of our ESP32 pins were allocated for direction control inputs to the motor controller.

### *Vive Detection*

Our Vive detection circuit used one photodiode provided to us. The circuit was built in accordance with the provided schematic in class and would send the data to the ESP32 via a singular pin. The Vive diode was soldered to a piece of perfboard with header pins attached in order to prototype the circuit on a breadboard. We did encounter some noise and erroneous readings in the Vive circuit due to components moving around in the breadboard and the diode being dirty. We often found ourselves wiping the diode clean when the Vive would not be locked on and reading accurate values. After some more practice, we were also able to eliminate some of the errors in the Vive readings with software corrections.

### *Beacon Detection*

Building off the aforementioned setup, the beacon detection system we envisioned for our robot would consist of a traditional Op-Amp, a duel color Red/Blue LED, two basic 220Ω resistors, a 1kΩ resistor, a 10kΩ resistor, and a 4MΩ resistor. Using these different pieces our beacon would search through atmospheric noise for the desired signal by filtering out signals beyond a set threshold. The beacon worked well when tested on its own using the Teensy. That said, we planned to integrate the Teensy and the ESP32, so that the ESP32 could read outputs of the Teensy to detect beacons, but were not able to, given the time frame.

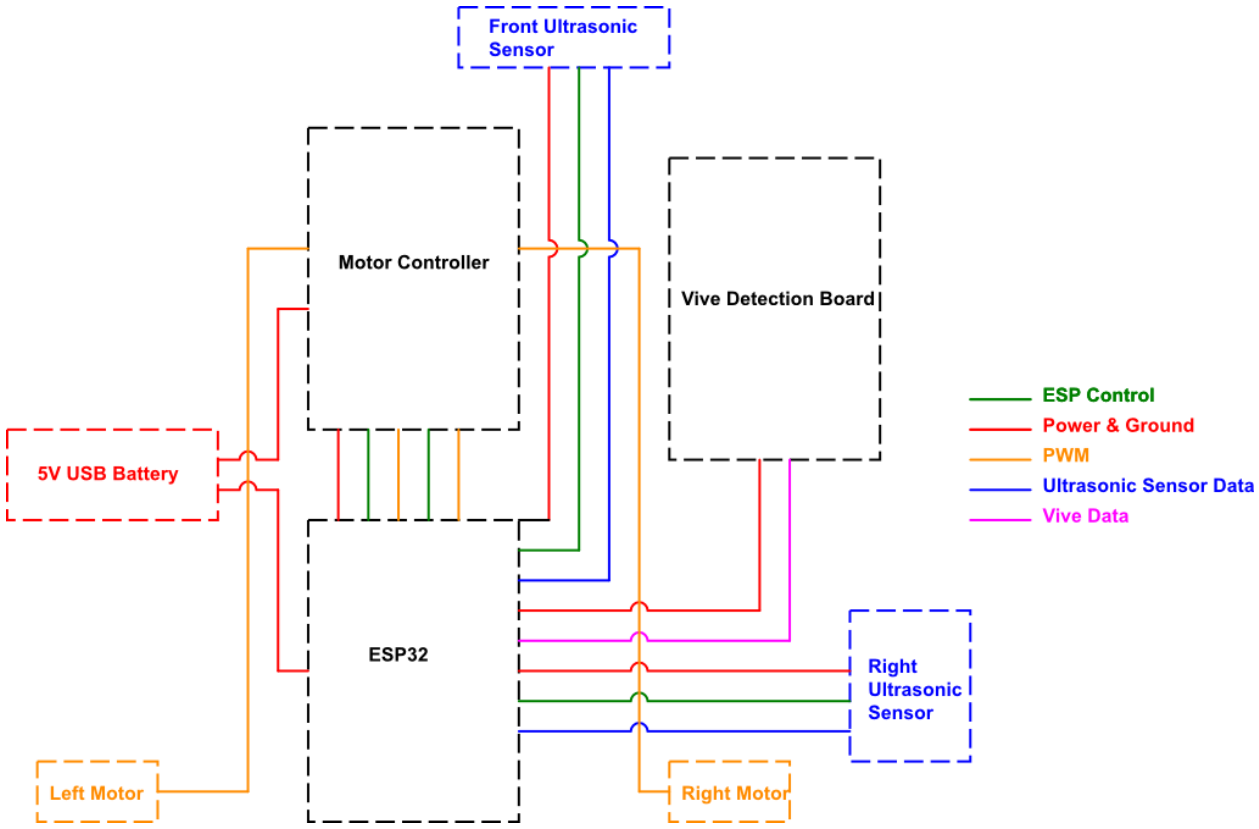
### *Integration Issues*

The integration of the hardware went smoothly for the most part. A majority of the bugs we encountered were software issues. However, a major issue that we encountered when testing individual behaviors or functions was the lack of a switch to cut power to the motors. We ended up using a switch alternative by unplugging the 5V power wire to the H-bridge when we wanted the motors to be idle. This worked for most of the debugging and testing process; however, it did not function as an effective and robust method to switch power to the motors off and on as the connection would often be very fragile when not plugged in properly. Once we were able to integrate all functions into the webpage and call certain behaviors via the GUI, this was not an issue since the default state of the bot would be idle until a command was passed to it.

## Processor Architecture and Code Architecture

### *Overall Architecture*

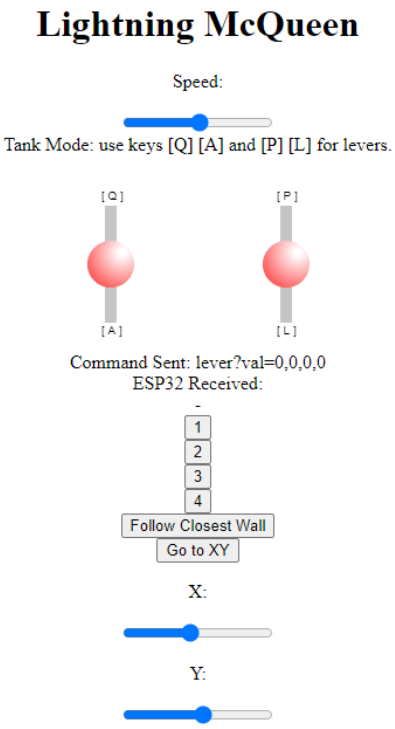
Our bot implemented a simple WiFi architecture to receive UDP packets and commands from the laptop. With 192.168.X.XXX being the local subnet, the laptop browser connects to the WiFi router (192.168.1.XXX) via the internet. Using the ESP32 station mode, the webpage is able to connect to the robots local IP (192.168.1.194). A block diagram of the overall electronic and processor architecture can be seen below:




### *Software Overview*

One major challenge of this project was writing the C++ and HTML code to control our robot, interface it with user input via HTML, and perform all five required tasks as detailed below. While addressing these problems efficiently and effectively, we tried to stay organized, and thorough yet concise. To that end, we spread out our functionality across several helper functions as described below, as opposed to cluttering hundreds of lines into Arduino's default *setup()* and

*loop()*. These functions all existed one after another in a single file *510_final_project.ino* to stay centralized. This structure promoted modularity in code, helping to minimize repetition, facilitate debugging and modifying, and promoting efficient contribution as a team. The following subsections describe our approach in dividing up this project then solving the many individual functions through logic, iteration, testing, debugging, and fine tuning. It's worth noting that the entire software suite was submitted on Canvas with thorough line-by-line comments to enhance readability and understandability of code, so the below specifications on the other hand primarily provide a general overview about our problem solving approach in developing this software.




### *Setting which Functional Task*

To differentiate between the many different functional tasks getting solved within a single big *.ino* file, we implemented a single global variable *botMode* integer variable, with integers from 0 to 3 (inclusive) corresponding respective to the functions of manual user robot control, wall following, go to XY, and go to Beacon. The 5th and final function of UDP broadcast is simply executing in the background no matter what, since it doesn't impact motor commands or car movement. In order to set which mode, the user can click on a corresponding HTML webpage button for each, triggering a specific C++ handler which updates the *botMode* value. Manual user control i.e. *botMode = 0* was the default, and would be set automatically whenever a manual wheel control button was pressed by the user; since nothing autonomous is happening here and all user button presses are handled by special handler functions, *loop()* need not include any additional functionality. We then cased over the remaining three *botMode* values in the central *loop(),* respectively triggering one of the three helper functions: *loopWallFollow()*,

*loopGoToXY()*, and *loopGoToBeacon()*. These helper functions all execute repeatedly to implement the logic achieving required functionality, as detailed below.

### *Manual Webpage Driving*

The webpage to control the motor driving of our robot was largely based on the tank mode concept we discussed in lecture. The baseline features for this portion of our webpage controller included a speed slider along with QAPL keypad control. We chose this mode over the joystick as we found the keypad commands to be much more user friendly in practice whereas the joystick was more visually appealing. Implementing keypad control of the motors meant programming our webpage to continuously read for keypad presses, then recording the char of the key pressed once the keypad was used. Next, the webpage would signal to the ESP32 which set of direction pins to power or invert said power. In the Esp32 code we used handlers and respective functions to use the char inputs to determine ESP32 pin usage, accordingly. Fortunately, the latency was negligible and the key tracking proved to be a key feature once properly configured to handle multiple signals. We were especially concerned about connectivity given we lost our antennae prior to developing these WiFi interfaces. Nevertheless, after troubleshooting the outdated tank mode sample we were able to implement these features almost seamlessly.

Regarding C++ code and logic specifically, controlling the car benefited greatly from certain helper functions which move certain wheels in certain directions. First we wrote 6 helper functions, using *digitalWrite()* to set the proper pins to HIGH or LOW in order to control either wheel to move forward, backward, or rest. We then congregated these 6 functions into a single *moveWheels()* function by accepting arguments about which wheel and which direction (if any) to move it. Finally we wrote helper functions to achieve more macroscopic functionality that directly assists with moving the car from one place to another; two functions rotated the car in either CW or CCW direction, two functions translated the car forward or backward, and one function sets the car to rest. We generally followed up any function call with a *delay()* call to give the car time to actually perform that motion before jumping to the next task. These helper functions became useful later on as well, in achieving the remaining 3 autonomous tasks of this project.

### *Wall Following*

Our approach to wall following software integrated closely with our selection of hardware, namely two ultrasonic distance sensors reading distance to wall in cm, assisted by our *getDistance()* function. Note that the robot must perform CCW revolutions about the rectangular game board's perimeter, meaning the right wall is generally pretty close (yet not colliding) to the robot's right end, while the front wall is further away. Thus we simplified the problem into continually checking for and upholding the following 3 explicit conditions: distance from right wall is more than 2cm, distance from right wall is less than 11cm, and distance to front wall is greater than 15cm. Note that these parameters were tuned through empirical testing, in order to troubleshoot and account for both the robot's non-zero volume and also the sensor's placement

relative to the robot. Within *loopWallFollow()* we first used a while loop to send the robot forward indefinitely until any of the above conditions break. At that point, the while loop ends and we handled the broken condition as follows

1. If the distance to front became too close (i.e. within 15cm), then the robot rotated CCW by 90* (* means degrees throughout this entire report) to translate that front wall into the right wall which we sought to follow. We performed a one-time experiment to measure the constant angular velocity of a rotating robot, and used this value to map from desired angle to required rotation time, requiring 0.750s of rotation.
2. If the distance to the right wall was too small i.e. below 3cm, then rotate the robot CCW in the opposite direction, to get further from the wall. We chose 10* after some trial and error to ensure that rotation sufficiently protects from wall collision, while also preventing an overcompensation away from the wall.
3. If the distance to the right wall was too large i.e. above 10cm, then we rotate CW by 10*, analogous to Case 2

Note that this *loop()* runs forever, with no termination condition. This is desirable since wall following is meant to continue indefinitely, or if the TA became satisfied, then we could just terminate manually.

***Go to XY***

*Logic Overview*

Our approach to nearing a provided Vive (X,Y) target "Can" position centered around performing repeated comparisons between the target position and the robot Vive positions, past and present. We defined several variables to update at each time step, repeatedly informing the status of robot position/orientation/angle (angle with respect to X axis), and displacement/distance/angle away from the can. See Canvas submission, around *510_final_project.in*o line 160 for details on which variables, and see *updateVals()* around line 500 for computations to update them (for example using *arctan2()* to find angles). Since our simplistic approach to hardware selection left us with only the options of translating forward and backward (i.e. not left or right) relative to the robot, we alternated between (1) rotating to tune forward robot orientation towards the target, and (2) moving straight to lessen distance away from target, until gradually nearing 0cm. These durations of time spent moving forward also informed knowledge of robot orientation (by comparing robot (X,Y) position before and after), which further informed knowledge of required rotation to approach can. This rotation occurred through the *reorient()* helper function, which rotates the car using *CW()* or *CCW()* based on computed offset angle (difference between angle to can and robot orientation angle), and also an angle (degrees) to rotation time (seconds) mapping similar to that described in *"Wall Following,"* based on constant angular velocity. Our *loopGoToXY()* function basically repeats these alternations between forward and rotation motion indefinitely, lessening the length of each

forward step as distance to target gets smaller, up until it's within 2cm thus we terminate by entering a while(true) loop.

*Debugging & Refining Vive (X,Y)*

Due to usage of arctan2() to convert from orientation vector to orientation angle above the positive horizontal X axis, and this angle's central role in reorienting the robot towards the can, it was critical to use a right handed coordinate system. In other words, since we're assuming rotation to occur about a vertical upward-oriented Z axis, we must ensure that the perceived Vive X and Y axes lend themselves to this upward-oriented Z. This however became problematic when we noticed the Vive X and Y coordinates had swapped (via Piazza @1012) yielding a right handed Z axis that points downward. After some debugging confusion, we noticed that the logical consequence of this forced rotation to occur in the opposite direction. To remedy this, we created functionality for the can and robot (X,Y) readings (via *setCan()* and *updateRobotXY()*, respectively) to check a boolean which manually records whether the vive coordinates are swapped at that particular time, and if so, to swap them back before our logic executes. While we were refining the vive readings anyway, we also converted them to meters (using a measured ratio of 2000 vive units : 1 meter) for convenience in distance reading logic. It's worth noting that this realization was not immediate, that is, it took lots of trial and error to diagnose the issue as (X,Y) flipping. Part of our debugging efforts included implementing *printVals()* which basically just printed all the global state tracker variables so we could identify the point at which issues started to arise.

***Beacon Tracking***

For beacon tracking, our goal was to design our software based on interfacing with the hardware. Namely we'd include two IR photodiodes near the front of the car, separated by a long planar divider. This divider ensures that if the car is oriented to the right of the Beacon then only the left photodiode picks up the signal since the right photodiode is blocked; and vice versa. Only when the robot is facing directly at the beacon will both photodiodes trigger "true." More precisely, we use two boolean helper functions rightBeaconOn() and leftBeaconTrig(), both the ESP32/C++ analogue in circuitry and code to our solution to the previous Lab 2 problem 2.4.3. Translating this approach to software, we'll occupy our loopBeacon() function by casing on the values of rightBeaconTrig() and leftBeaconTrig(), which are constantly re-evaluated based on constantly-changing car position and orientation. The casing goes as follows:

1. *rightBeaconTrig()* true AND *leftBeaconTrig()* true: execute *forward()* since the car must be pointing toward the beacon
2. *rightBeaconTrig()* false AND *leftBeaconTrig()* true: execute *CCW()* since the car must be oriented to beacon's right
3. *rightBeaconTrig()* true AND *leftBeaconTrig()* false: execute *CW()* since the car must be oriented to beacon's left

4. *rightBeaconTrig()* false AND *leftBeaconTrig()* false: execute *CCW()* arbitrarily assigning a rotation direction until a signal is picked up by either photodiode, informing further action

Note that the conditions are continually rechecked at frequent timing intervals, to ensure that no one command is executed for too long, even after its initiation conditions have become false. By rechecking and redirecting the car, we effectively implement feedback control to robustly ensure the car will reach the beacon.

***Broadcast UDP***

The final task of transmitting robot vive XY reading via broadcast UDP was a standalone function so to speak, in that it neither interferes with nor is affected by the other 4 functions. As such, there was no need to create a 5th *botMode* setting for this function. Rather, we simply added the broadcast logic within Arduino *loop()* to execute always regardless of mode. That said, we were instructed to send readings at a frequency of once per second, and empirically noticed that sending quicker than that actually gave rise to bandwidth overflow errors. To ensure 1 Hz, we first created a global static variable *timeLast* to record when the most recent UDP broadcast occurred. We then continually recheck with each *loop()* iteration whether the current time (as evidenced by *millis()*) is over 1000ms greater than the previous *timeLast*. If so, then it sends out the broadcast UDP (following the code template from Lecture 22) and updates *timeLast* to the current time, preventing premature followup broadcasts. If less than 1000ms have passed, then it simply skips to the other *botMode* dependent functionality if necessary (ex. *loopWallFollow()*) until 1000ms have passed. Finally, resetting the robot ID was straightforward, as we simply stored this data into a global static *RobotID* integer variable (values ranging from 1 to 4, inclusive), and updated it directly through handlers which got triggered whenever any HTML robot ID buttons were pressed.

## Reflections

***Jason Friedman***

Looking back, MEAM-510 was certainly a journey. We learned how to seamlessly integrate mechanical and electrical components into complex interdisciplinary and autonomous systems, and how to write software to the microcontroller and prompt these systems in real time through sensory input. We certainly learned plenty of highly technical features like reading data sheets, motor selection, using Op-Amps, and event-driven design of software just to name a few. However the more memorable lessons from 510 were more abstract ones. Mark clearly cares a great deal about how we approach the task of learning in general. For one, it's best to learn and work in teams, and ask for help whenever you need it. In the real world we'll never succeed alone, so it's best to learn the skill of teamwork early on. Additionally, many resources were available to us in 510 (ex. Lecture content, Canvas, MEAM labs design page, TA office hours, Piazza, asking friends, the internet, data sheets, reddit threads, etc.), so it became our responsibility to leverage these to our benefit rather than getting overwhelmed by the abundance of options to choose from. Furthermore, we learn more from making mistakes than from correctly following all the instructions covered in lecture; even if it means staying in GM lab debugging until 4am, this makes for a more memorable learning experience, and helps develop more robust problem solving skills when the solution is truly unknown. Other problem solving skills learned include breaking down all problems into smaller chunks. For example, this final project would have been extremely overwhelming if we didn't take it one step at a time, and simply change plans / figure it out as we go, after a certain feasible amount of pre-emptive planning. Furthermore, we learned that it's expected for things to inevitably go wrong, and it's best to keep a clear head while debugging, and systematically perform functional unit tests and iterate over many logical possible fixes, guided by feedback in real time (ex. Print statements). Last but not least, we learned that all problems (big and small) have several possible solutions, and you gotta test it out to see if you've found it.

***Niko Simpkins***

In retrospect, I learned a lot from my time in MEAM 510; that said, my main takeaways from the course related to (1) the process of learning and troubleshooting complex systems, (2) the challenges of managing timetables for experimental technology and prototyping, and (3) the importance of asking for help. Coming into the course I had a decent base of experience with troubleshooting code from my previous endeavors in computer science. (1) To my dismay, the troubleshooting process still can become exponentially more complicated when working with two complex systems (the code and the electrical systems), then even more so with the three complex systems (the code, electrical systems, and hardware), and so on. Sometimes the error

would be in the code, more often the error would be in the electrical circuitry, but sometimes it would be faulty hardware, and if that weren't enough sometimes it would be the auxiliary systems (i.e. the oscilloscope) that you might have wasted hours by presuming they worked properly because you just watched them work properly for someone else. Thankfully, this process became significantly daunting as I developed my own technique for troubleshooting. This entailed separating the functionality of the design from the theory for each system, checking the theoretical correctness of each, then working top down to test the respective subcomponents in conjunction then isolation as necessary. (2) The hours it could take to find out that a breadboard was faulty were demoralizing at best; however, these trials brought me to my second insight regarding the challenges of managing timetables for experimental technology and prototyping. I quickly learned there is no quick fix or surefire method of managing these timetables especially when they conflicted with obligations from other classes or spheres of life. I would consistently run into an impasse where I could give another hour or two toward a potential yet uncertain breakthrough in my mechatronics assignments at the sacrifice of time necessary to complete work or study for another course. I quickly learned there was no quick fix for this dilemma as enormously time consuming conflicts could stem from the most unpredictable of events like sudden covid infections or having GM keycard access revoked randomly throughout the semester. This gave me a lot of respect for companies and institutions that deal in innovative technology and the adaptability it must take to execute even grander lesser known projects. (3) Despite the inevitability of these obstacles I found hope in my third and final insight, asking for help. This course would have been unimaginably more difficult had I not leveraged my resources in instructors and students alike. I quickly began to note what TAs and students were most helpful and developed a support network for myself that enabled me to work and learn way more effectively. This lesson was the most impactful as it was integral to resolving several critical troubleshooting fiascos and seemingly impossible time conflicts. All in all, it goes to show that no matter how deep into tech you find yourself, people will always be your most valuable asset.

***Andres Voyer***

The most important thing I learned was the power of compounding knowledge and to not be discouraged by taks no matter how niche and inapplicable they may seem in the moment. I also learned that integration always takes 5x longer than you think and you must plan ahead for it. I found the final project to be very interesting and rewarding. Being able to apply all the small skills we learned through the course of the semester to see tangible results was awesome. The best part of the class was the culture of collaboration and help. Although the GM lab was often a hectic mess, the TA's, fellow classmates, and the rest of the teaching staff were always incredibly helpful via Piazza or in person. I had the most trouble with the transition from the Teensy microcontroller to the ESP32. Changing IDEs and transferring the Teensy-specific skill to the ESP32 was challenging. The one thing I think could be improved for the class is the format of the recitations. GM was always very busy and I often found it hard not only to find a workstation but also focus once there. I think it would be a more rewarding and effective way to reinforce concepts in recitation if the class was split up into smaller sections with the recitations taking place in the GM lab with physical demonstrations. Not only would this more tangibly demonstrate concepts in a different method than lecture, but it would also control the amount of

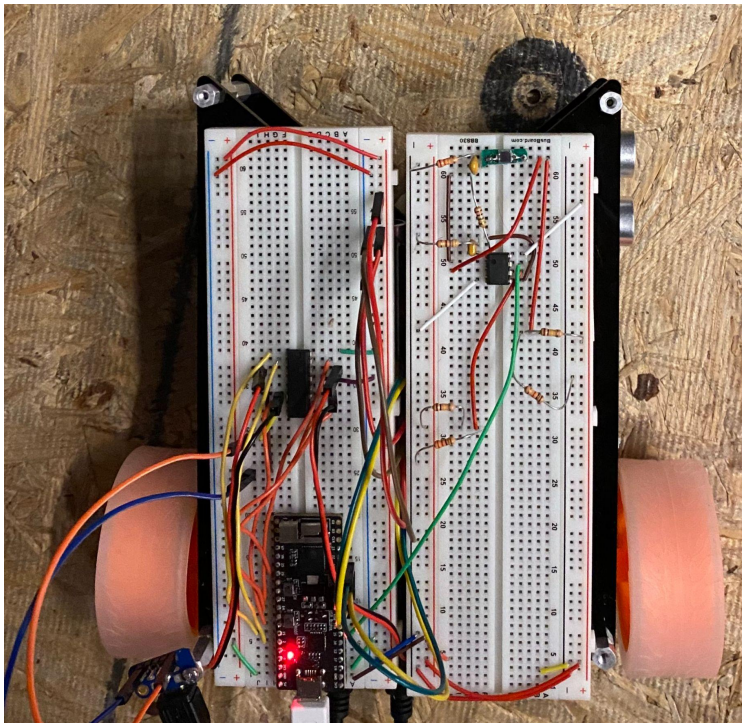
people in GM at certain times to make sure everyone has sufficient access and time with the right tools and teaching staff. Overall, I really enjoyed the class and am very happy that I will have such an interesting piece to add to my portfolio. Although the class was tedious and time consuming, it is worth it in the end!

# Appendix

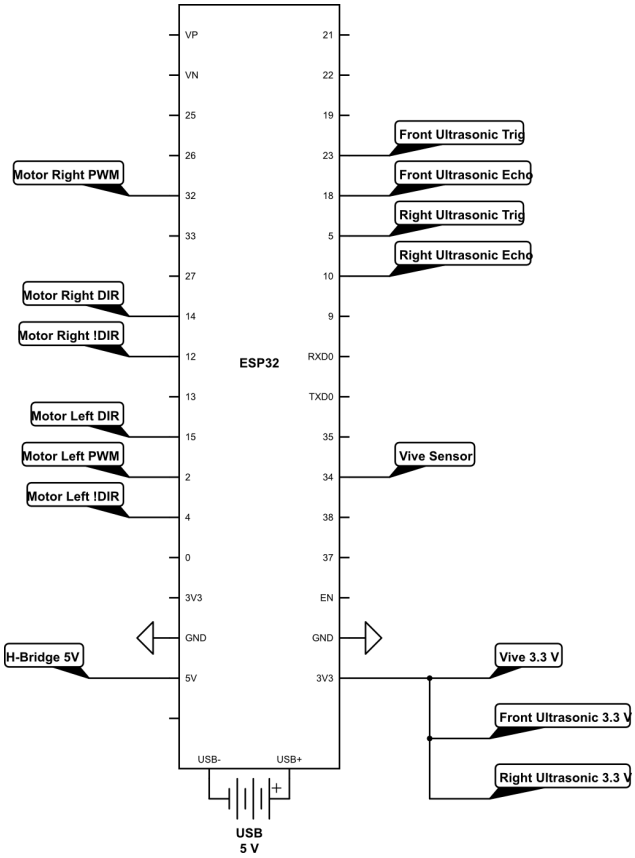
## *Bill of Materials*

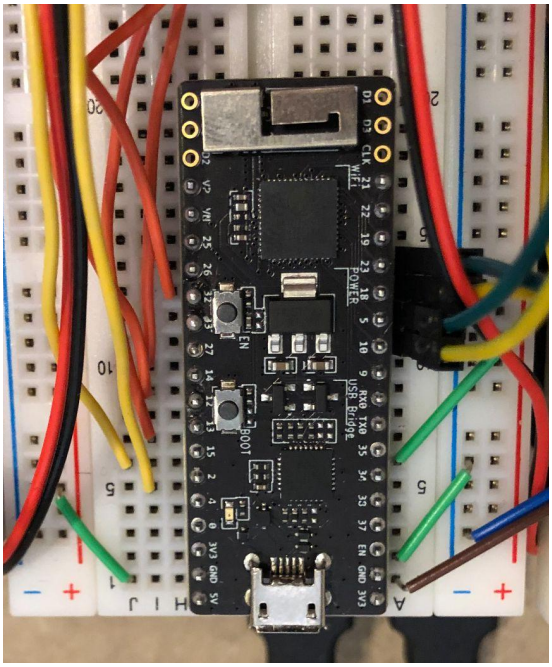
| Brief description | Full product description | Existing count | Number to purchase | Cost/unit | Subtotal | Link to product (shortest link possible) | Product identifier (ASIN number if Amazon, part number otherwise) | Status |
|---|---|---|---|---|---|---|---|---|
| Wheels | Orange and Clear TT Motor | 0 | 2 | $1.50 | $3 | https://www.adaf | 3766 | Received |
| DC Motor | DC Gearbox Motor - "TT M | 2 | 0 | $2.95 | $0.00 | https://www.adaf | 3777 | Received |
| Motor Mounts | Motor Mounts for TT Geabc | 0 | 2 | $1.50 | $3.00 | https://www.adaf | 3768 | Received |
| 16mm Caster | 16mm Height Metal Caster | 1 | 1 | $1.95 | $1.95 | https://www.adaf | 3949 | Received |
| 20mm Caster | 20mm Height Metal Caster | 0 | 1 | $1.95 | $1.95 | https://www.adaf | 3948 | Received |
| 5V Power Source | PowerCore 13000 | 1 | 0 | $39.99 | $0.00 | https://us.anker.c | a1215 | Received |
| Short Micro USB Cable | USB cable - 6" A/MicroB | 0 | 2 | $2.95 | $5.90 | https://www.adaf | 898 | Received |
| Ultrasonic Sensor | Ultrasonic Distance Sensor | 0 | 2 | $3.95 | $7.90 | https://www.adaf | 4007 | Received |
| ESP32 | ESP32-PICO-KIT V4.1 | 3 | 0 | $23.99 | $0.00 | https://www.ama | B00RSPTHE0 | Received |
| USB Breakout Board | USB Micro-B Breakout Boa | 1 | 0 | $1.50 | $0.00 | https://www.adaf | 1833 | Received |
| Ultrasonic Mount | Ultrasonic Sensor Mount | 2 | 0 | N/A | 0 | 3D printed | N/A | Received |
| IR Mount | Phototransistor Mount | 1 | 0 | N/A | 0 | 3D printed | N/A | Received |
| H-Bridge | SN754410NE | 1 | 0 | $3.09 | $0.00 | https://www.digik | 380180 | Received |
| Op Amp | TLV272 | 2 | 0 | $1.30 | $0.00 | https://www.digik | 454247 | Received |
| IR Sensor | LTR-4206 | 2 | 0 | $0.40 | $0.00 | https://www.digik | 214965 | Received |
| Perf Boards | Perf Boards | 2 | 0 | N/A | 0 | Ministore | N/A | Received |
| | | | | | | | | |
| | | | | Total | **$24** | | | |
| | | | | Budget | $150 | | | |
| | | | | Remaining Budget | $126 | | | |

## *Overall Board Layout*

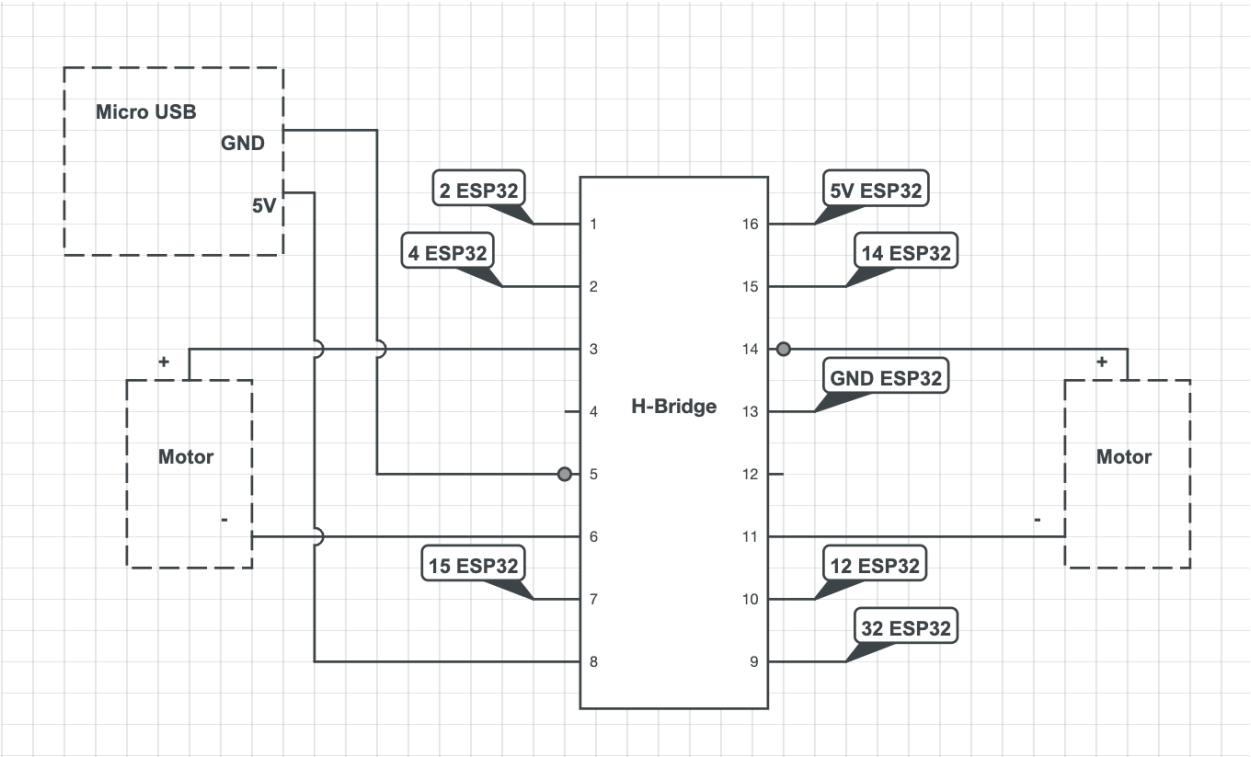


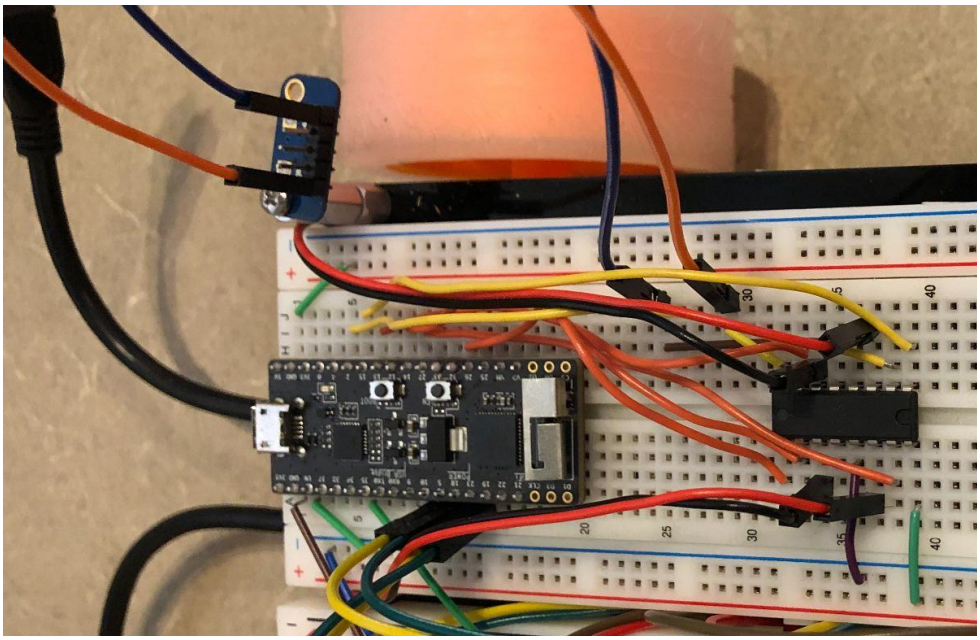
## *ESP32 Port Allocation - Schematic and Board*



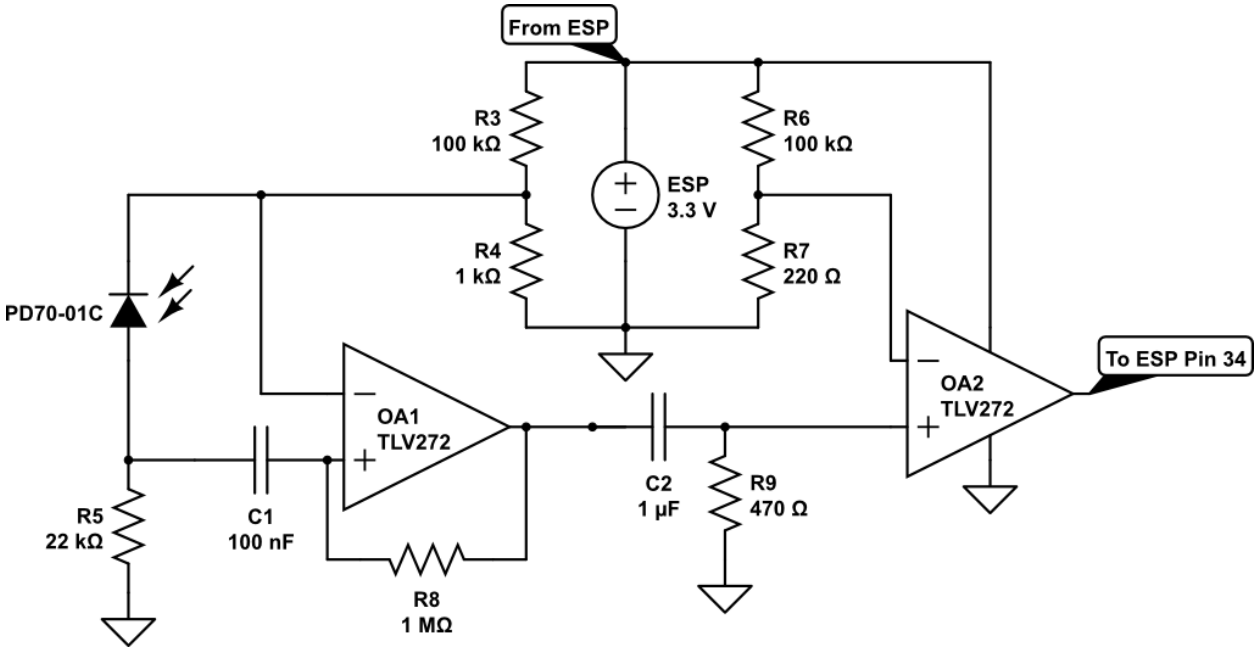



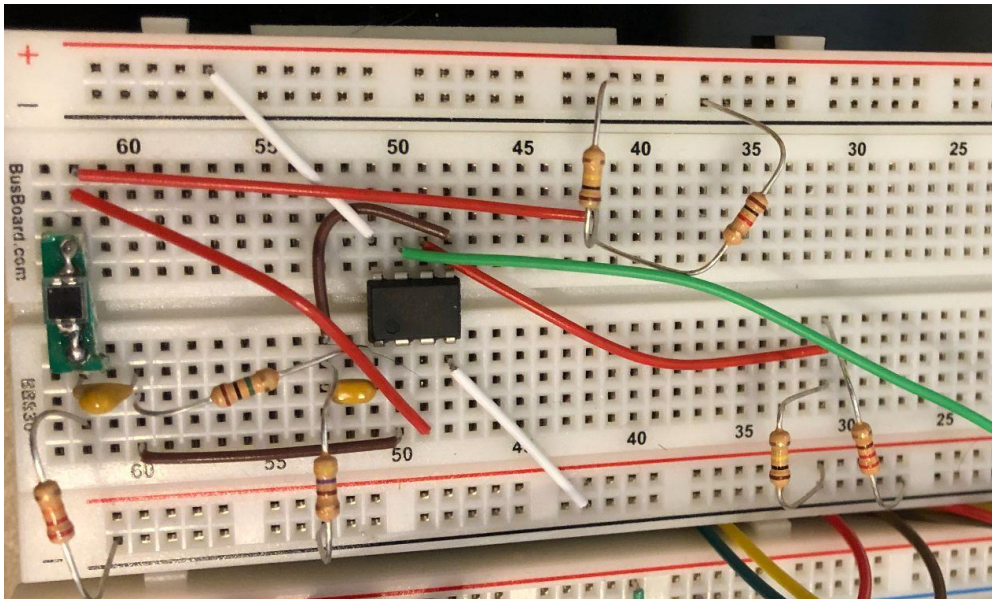
***Motor Controller - Schematic and Board***



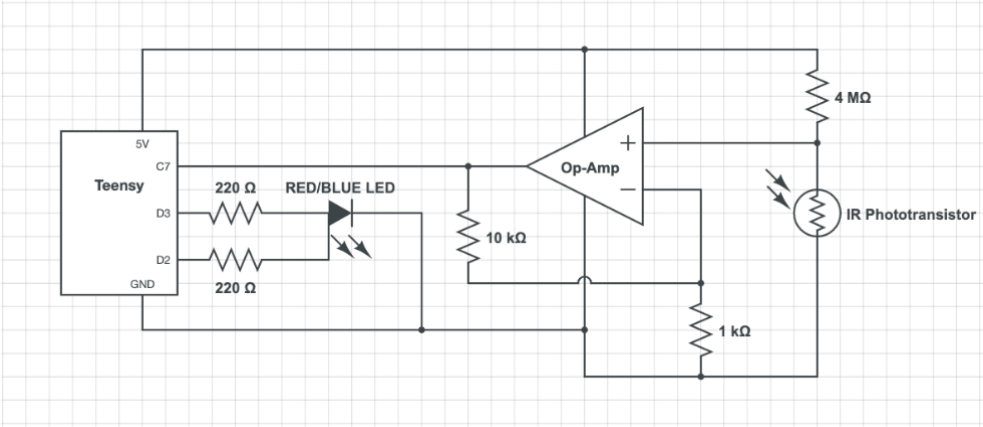



***Vive Detection - Schematic and Board***

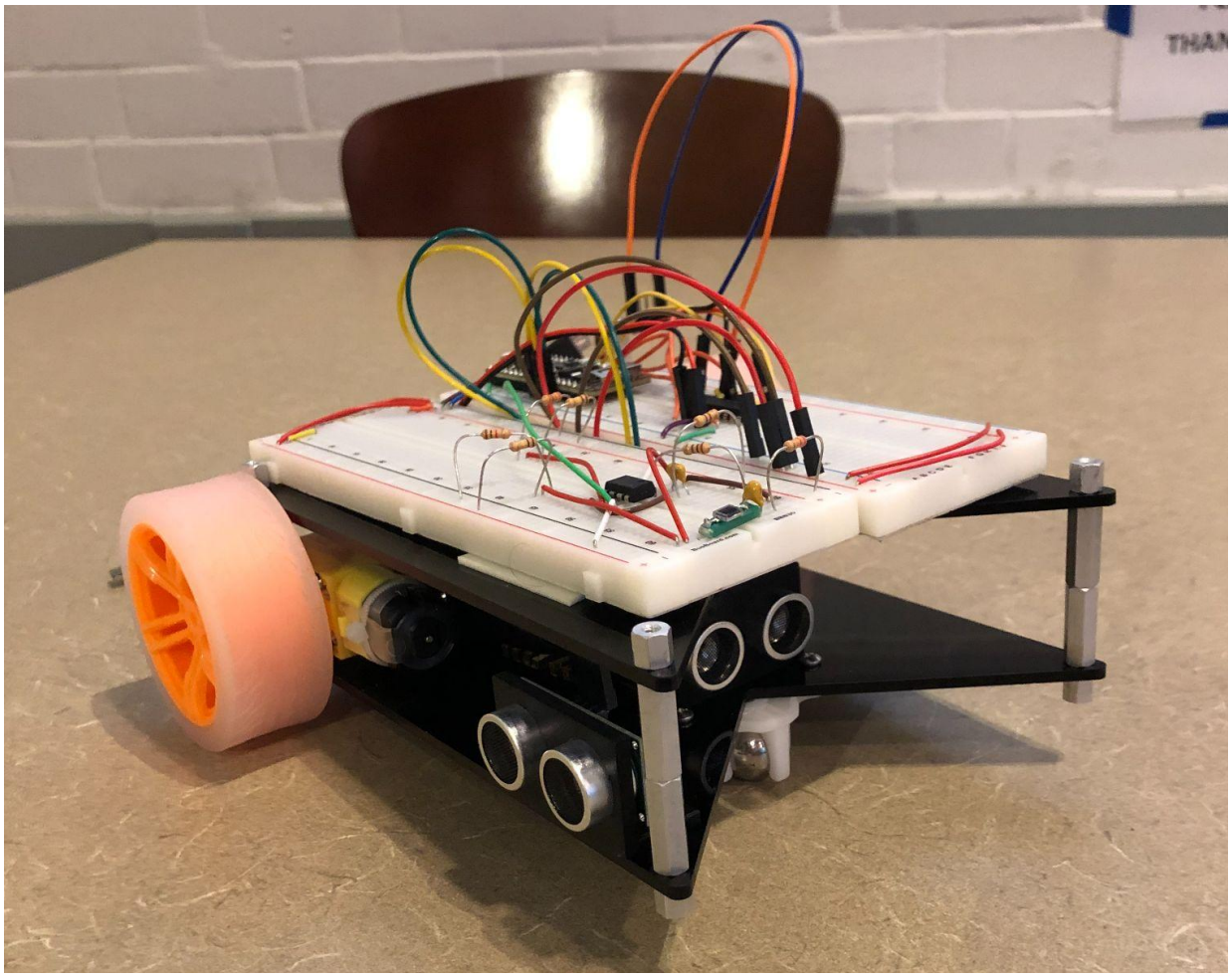


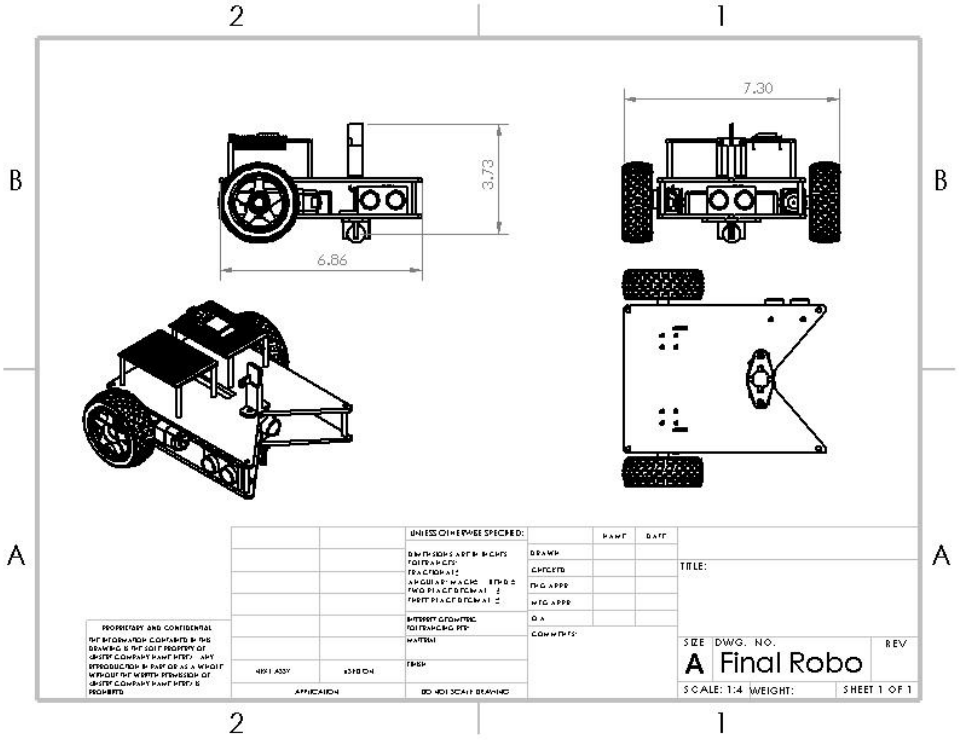
***Beacon Detection - Proposed Schematic***



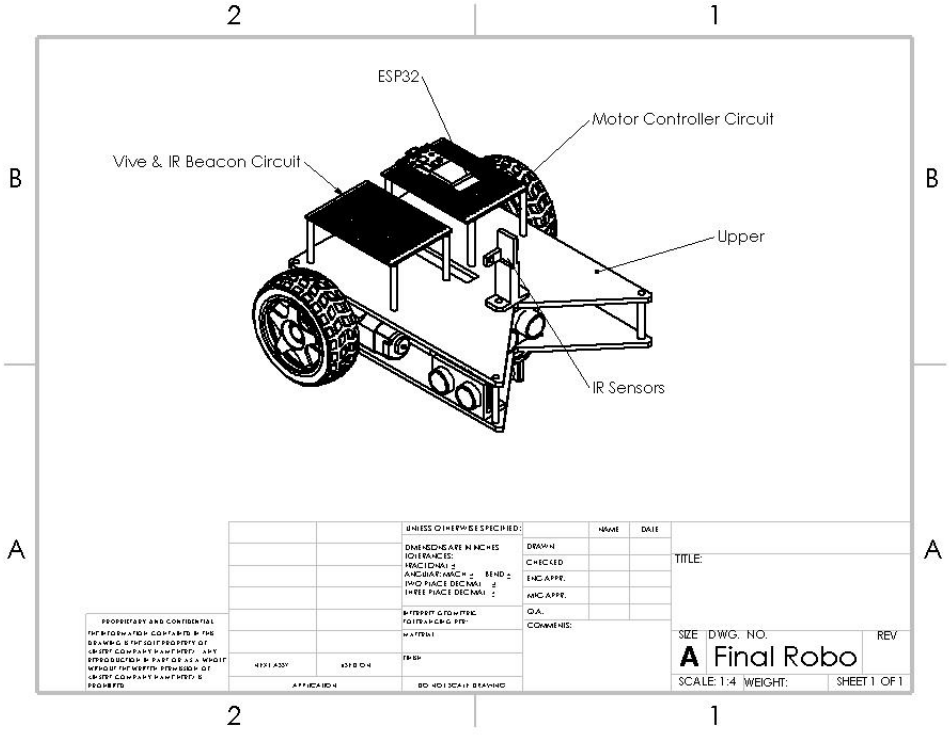

***Final Robot Picture***



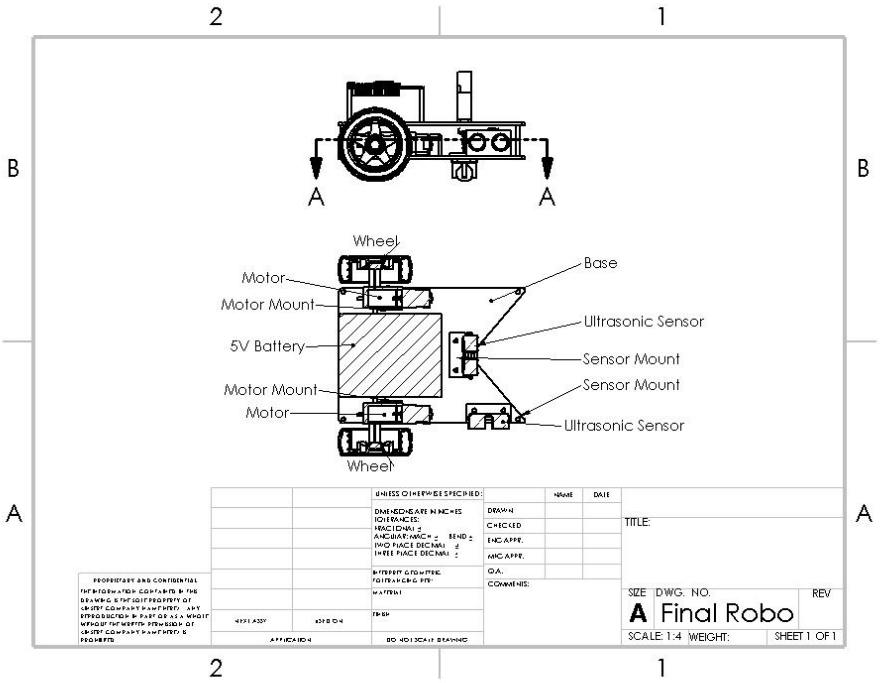
***Overall Dimensions Drawing***




***Upper Layer: Component Drawing***



***Base Layer: Component Drawing***




***Datasheets***

[Ultrasonic Distance Sensor: RCWL-1601]

***Videos***

[Autonomous Wall Following]

[User Control of Robot]