ECE 118 Final Project

The Spy Who Slimed Me

Janelle Chen & Kohl Grunt & Grant Skidmore

(jnchen@ucsc.edu) (kgrunt@ucsc.edu) (gskidmor@ucsc.edu)

Professor: Michael Wehner

December 13, 2019

Contents

1 Design Goal & Overview

The goal of this project was to create a fully autonomous robot able to navigate around a light-colored field, locate an IR beacon tower, and dispense ping-pong balls into indicated drop-off holes. In the design process of this project, we incorporated topics such as signal manipulation, mechanical prototyping, sensor integration, and state machine logic. Our constraints for this project included a $150 budget and maximum size dimensions for our robot of an 11 inch cube.

Our final design included two ping sensors mounted at the front left and back left corners, facing the left, to allow the robot to recenter and align its left side with the tower. We went with a simplified launch mechanism using a single dispenser tower in the middle, facing the left as well, with a tape sensor to correctly identify where to dispense the balls. As for the launch mechanism itself, we used a DC motor to operate a spinning rotor which pushed out ping pong balls and relied on gravity to drop the ping pong balls down the angled dispensing tube, where its opening aligned with the mouth of the hole on the towers.

The final design of the robot is show in Figure 1. This design was able to complete the course with a 100% success rate of dispensing a ball in the correct holes. In UC Santa Cruz's semi-annual Mechatronics competition, our bot placed 2nd among the 18 competing teams. This report elaborates on design choices, state machine logic/code, and electrical hardware as we implemented and tested a system for the robot.

1.1 Mechanical Design Overview

We used SolidWorks to mechanically design and prototype the robot. From the preliminary design review, we incrementally worked from the bottom up as we tested our design's software and electrical components simultaneously. The tower's dimensions informed the basis of our prototype. A primary goal was to maximize the robot's dimensions so we could logically interact with the tower and simplify electrical and software design.

We started with the tower's dimensions, specifically how the IR beacon detector was mounted on the tower (See Section 2.3), as well as how holes were placed in the tower to inform the design choices. We were given tower specifications of 10 inches in height, and that the IR beacon was placed slightly below 11 inches, while the holes for the balls were centered at 8 inches with a diameter of 2 inches. We learned this information from the project overview as well as the assembly and CAD documentation for the towers. Figure 28 shows the design of the specified towers.

Following these guidelines, we designed our bot to have a mostly square base of 10 x 10 inches. This was followed by multiple, smaller layers to maximize space for our hardware components. The first layer sported the most parts: 3 reflective infrared optical sensors (tape sensors), 2 motors, an H-bridge dedicated chip, a front-facing bumper, a power distribution board, and the microcontroller itself. The heavier parts were placed here to provide a lower center of gravity and prevent toppling on fast tank turns and rapid directional changes. The next layer housed an infrared signal detector board, a boost converter, and the other 3 bumpers. The third layer stored a secondary power distribution board to provide high-current ground and extra power pins; our ball dispensing tower and actuating motor; and our final tape sensor used to identify where and when to dispense the balls. This design will be covered in explicit detail in Section 3.

Figure 1: Final Competition Picture. This is the 2nd place team at the end of the competition holding their prize robot. The picture is akin to a farmer holding their prize pig at the county fair.

1.2 Electrical Design Overview

The electrical design centered mainly on analog filters for the custom sensors, and power distribution for the numerous high-current components on the bot. The given specifications for the beacon were an infrared signal oscillating at 2kHz while the track wire was measured at 25kHz to 27kHz. The filters needed to be carefully calibrated so as to maximize the signal amplitude and reduce the effectiveness of external noise.

The remainder of the electrical systems design involved careful monitoring of voltage levels and incremental testing of the full system as more and more components were added. This incremental testing saved many hours of debugging as we were usually able to identify malfunctioning components or unexpected consequences as soon as they occurred. Using the ultrasonic ping sensors as an example, we identified problems that required an additional first-order low-pass filter on the echo pins in order to get a readable signal, even at nominal clock speeds. Similar issues would have set us back numerous days and likely increased stress levels across the team. For explicit elaboration on the electrical design of specific components, see Section 2.

1.3 Software Design Overview

The software for our robot was based off the ES_Framework developed by J. Edward Carryer of Stanford University [1] and revised by Gabriel Hugh Elkaim of UC Santa Cruz. This software uses C to program the Microchip Uno32 microcontroller, which was used as a way of interfacing and controlling the inputs and outputs of the robot. The software has three central parts to it: events, services, and hierarchical state machines.

The events work as continually running routines that are useful for checking things like the beacon detector and the tape sensors. However, we decided not to use any events or event checkers in the final iteration of the bot. Instead we opted to institute all sensor input checking with services. Services are similar to events, but they give the user the ability to control how fast the signal is checked as well as having the ability to turn the service on or off. The final part were the hierarchical state machines, which is just a fancy way of implementing nested state machines. Having these nested state machines made the organization of control logic much more intuitive, without which we may not have been able to complete the project on time. For information on the software implementation of the robot, please refer down to Section 4.

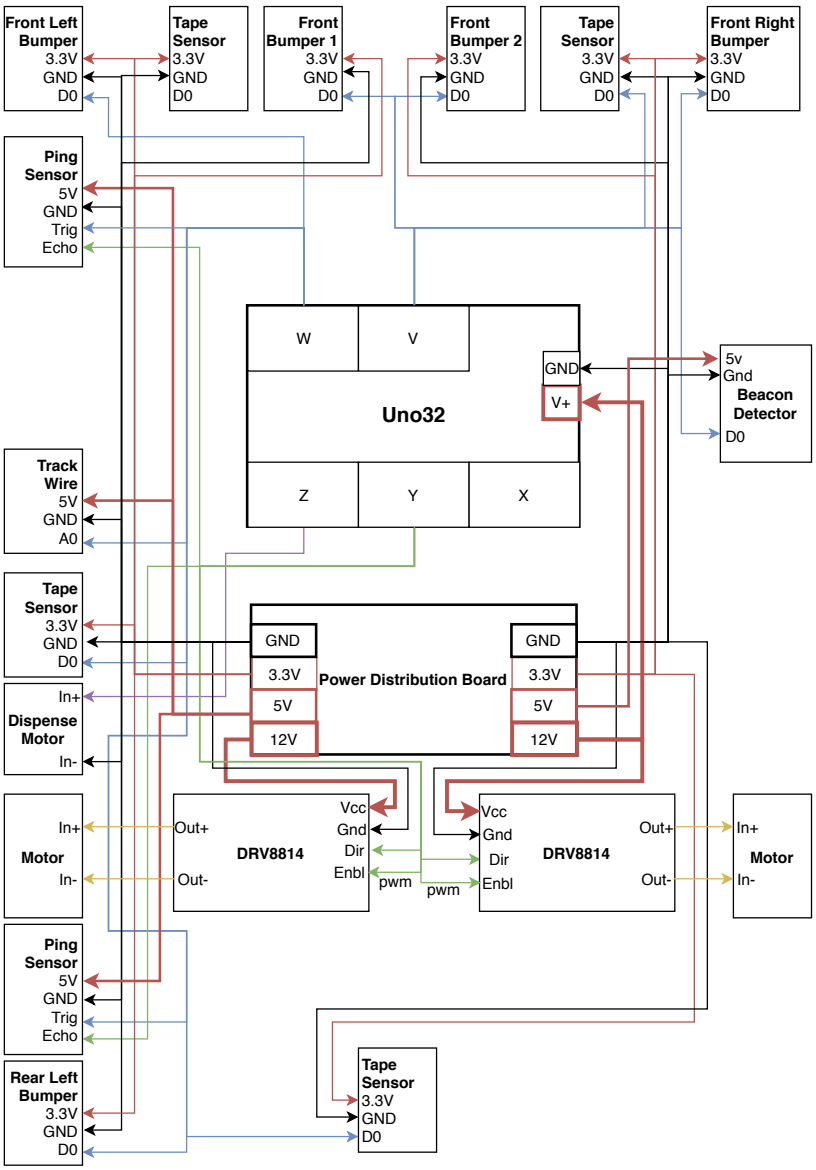


Figure 2: Block Diagram. This image shows the high-level block diagram of all connected sensors and motors. A0 designates an analog output while D0 designates a digital output.

2 Components

There are a number of components that found their way onto our robot's chassis. These included a wide range of useful sensors, power distribution components, and a central microcontroller acting as a hub to run everything synchronously. This section will look at each of the individual parts found on the robot including schematics, embedded code, and any other noteworthy information.

2.1 Uno32 Microcontroller

The Uno32 microcontroller from Microchip is a discontinued PIC32 microcontroller. Featuring a PIC32MX320F128H microcontroller with a clock speed at 80MHz, it was way more than enough firepower for the features we were running. Our fastest service (see Section 4.1 for all services), the ping sensors, only operated at 2.5MHz so we were able to drop the prescalar way down to extend our battery life. In order to program this microcontroller, we used the MPLAB IDE and a custom bootloader allowing us to use C on the bare metal. In total, we used 22 different pins on the Uno32 that directly interfaced with our sensors and motor control, not including power distribution pins. Some of the important pins we used were 2 pwm pins to control the enable inputs on the DRV8814 chips, see Section 2.10, and input captures 2 and 3 to control the echo pins of the ping sensors, see Section 2.6.

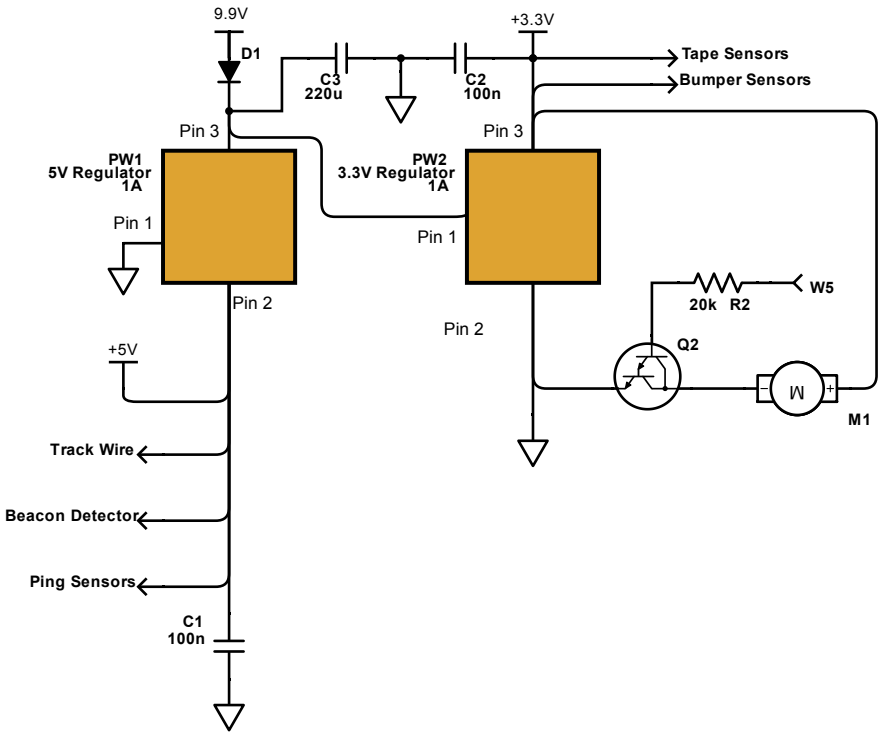


Figure 3: Power Distribution Board Schematic. This shows the design of the power distribution board and the integration of the ball-actuating motor. This board used a 9.9V input from the battery and created rails of 5V and 3.3V using linear regulators.

2.2 Power Distribution Board

Our custom power distribution board used the output of a LiFe battery, from 9.9V to 7.5V nominally, to drive two rails at 5V and 3.3V. We used 1 ampere linear regulators because they were cheap and we were not concerned about efficiency. To deal with the excess heat dissipation, we attached beefy heat sinks to each regulator, we never had a problem with them overheating or not working. The majority of our sensors used 3.3V because of the Uno32's limitations of 3.3V maximum ratings on the input and output pins. The only components that were powered with 5V were the ping sensors, track wire detector, and beacon detector. The dispensing motor, tape sensors, and bumper sensors ran off the 3.3V rail. Even though the block diagram specifies it, our power distribution board did not have a 12V rail on it; the 12V rail was supplied by a boost converter. This rail then supplied the Uno32 power and the power for the H-bridge chips used to drive the navigation motors.

2.3 Infrared Signal Detector

The infrared signal detector, otherwise known as the beacon detector, was an integral part of the design. Without it, we would not be able to locate the beacon towers and would not know which way to go. This was a challenge because the beacons were not particularly strong signals and we needed to read them from at least 8 feet away. Thankfully, at a low 2kHz, the beacons maintained signal integrity even at relatively long distances.

This was the first of our custom sensors and certainly the most complex. The sensor utilized a phototransitor as the initial sensor, followed by many stages of signal manipulation. The first of which is a transimpedance amplifier converting the current from the phototransistor to a usable voltage signal. This signal was also centered at 1.65V to preserve full signal integrity since we unanimously used a single-ended supply. After this, we ran the signal through a 6th order Chebyshev band-pass filter. With a 100Hz pass-band featuring a -1dB pass-band ripple, -35dB attenuation at 1.5kHz, and -36dB attenuation at 2.5kHz, this filter successfully isolated 2kHz signals. All stages of this filter were also referenced at 1.65V from the Vref op-amp. This filter took up the majority of the time spent on this circuit because we needed to get the center frequency as exact as possible due to our small pass-band range. After all the component tinkering, we managed to get a center frequency of 2.01kHz, which was close enough to the ideal 2kHz.

Following the band-pass filter, the signal was run through a non-inverting amplifier gain stage with an ideal gain of 300, centered at the 1.65 reference voltage. While this is a lot of gain for a single stage, we used the MCP6004 op-amp chip which has a gain bandwidth product of 1MHz [3]. This means that as long as we kept our gain below 500 (taking into account our 2kHz signal), we should not run into the problem of gain limiting at the op-amp output. This bumped our signal up to readable levels.

After the gain stage, we used a high-pass filter to remove the DC offset from the previous stages. Setting the corner frequency at 0.1Hz, we were able to remove nearly all of the DC offset without touching the 2kHz signal. The output of this stage was then run through a peak detector rectifier. We gave our peak-detector a large time-constant to ensure that we can quickly follow the beacon signal should we lose it for whatever reason.

The last stage was a single-threshold comparator that used an op-amp with no feedback. This

gave us an output of 0V or 3.3V depending on whether our signal was lower than the 230mV threshold or above the threshold respectively. Without any hysteresis, our beacon detector could potentially be a problem if we get a signal right around the 230mV range going into the comparator. However, we decided to mitigate this by implementing some debouncing in the code. The penultimate component we included was an indicator LED at the output of the comparator so we could quickly check if our beacon detector was working or not. Once we finished soldering, we the beacon detector was not actually working. The output of the comparator was always high with a minimum input of around 400mV into the comparator. After debugging, we noticed 350mV of noise on the power rail at 70kHz. We ended up adding the largest capacitor we had, a 470uF cap, and the noise was removed with the beacon detector working again.

In the end, this actually turned out to potentially be one of the strongest assets on the bot. With the low comparator threshold, we were able to maximize the range at which the signal could be detected without worrying about clipping. Also with the 6th order filter, any noise, even at 1.5kHz or 2.5kHz became irrelevant. In testing, we measured the maximum range of our beacon detector at roughly 30 feet (9.14 meters) and had no minimum range, meaning you could use the beacon detector at point-blank range without problems as well. Unfortunately, this was total overkill for the competition and the project as a whole, but was still neat to have anyway simply for the satisfaction that we had the best beacon detector of all the teams.

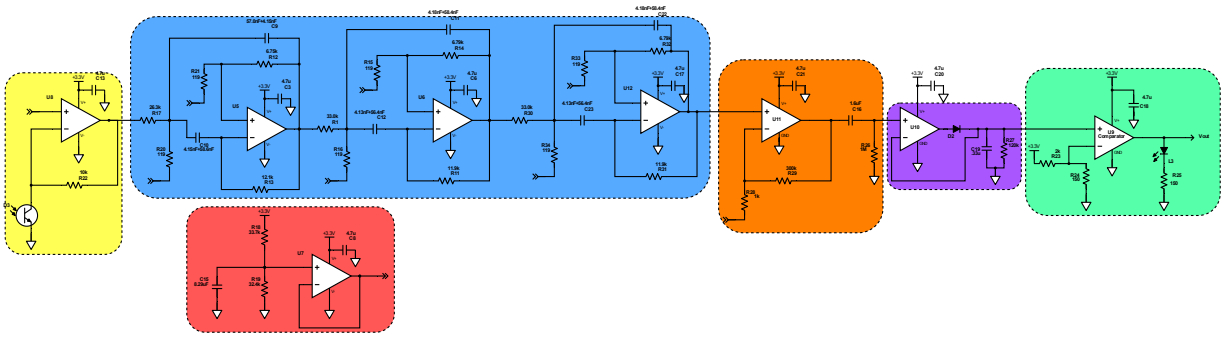


Figure 4: Infrared Signal Detector Schematic. Also known as the Beacon Detector, this board uses a transimpedance amplifier followed by a 6th order Chebyshev band-pass filter to identify a signal from an infrared led oscillating at 2kHz. It then uses a single gain stage into a peak detector rectifier and a single-bound comparator to get a working digital signal of 0-3.3V for the Uno32 microcontroller.

2.4 Electromagnetic Field Detector

The electromagnetic field detector, commonly referred to as the track wire detector, was the second of our custom sensors. This sensor was similar to the beacon detector, but varied due to the focus on gain as opposed to the previous focus on filtering. This switched focus made the track wire many times easier to make than the beacon detector.

The first stage of the track wire detector was a passive parallel tank circuit. This had an LC resonant frequency of 25.3kHz, which was perfect considering the measured frequency of the track wires were 25kHz to 27kHz. This tank circuit also had a 510kΩ resistor to lower the Q-factor on the circuit. This flattened the 25.3kHz resonant frequency to easily detect 27kHz signals as well.

If we needed to be more selective, we could have chosen to use a much smaller resistor to sharpen the resonant frequency response of the tank circuit.

Following the tank circuit, we had our first gain stage. This gain stage featured a non-inverting amplifier with a gain of 10.1. Throughout this circuit, we were much more careful with our gain stages because of the lack of quality filters in any stage, so we wanted to limit the noise introduced by the power supply.

After the first gain stage, we ran the signal through a unity-gain buffer and then into a high-pass filter. This high-pass filter had a corner frequency of 3kHz so as to not disrupt the track-wire signal at 25kHz but still provide lots of attenuation for low frequency noise. We ran the output of this filter into two consecutive non-inverting gain stages. The first of these gain stages had another gain of 10.1, but the second one used a potentiometer to bump up the gain theoretically to 500, while providing a nominal gain of 1. This made our circuit rather robust because of the fact that we could change the gain to whatever we needed it to be depending on the track wire signal.

After the gain stages, we used a peak detector as our final stage, so that we could rectify the signal enough to be used by the Uno32. With a high-enough time-constant, we were able to effectively linearize the signal because of its relatively low signal frequency. While it definitely was not completely linear, the signal decayed slow enough to the point where the Uno32 was not able to tell that the signal was not linear, this meant that we were able to use the signal as a digital output.

This sensor was a minimally used component, albeit integral to the proper functioning of our robot. This circuit definitely did not see the same amount of time and consideration as the beacon detector, but it still worked without a hitch. While the beacon detector may have been over-engineered for this project, the track wire detector definitely met the same standards as the rest of the project. We were able to detect the track wires through a layer of MDF from about 3 inches away (7.62 cm) at the strongest point at 1.5 to 2 inches away (3.85 to 5.08 cm) at the weakest point.

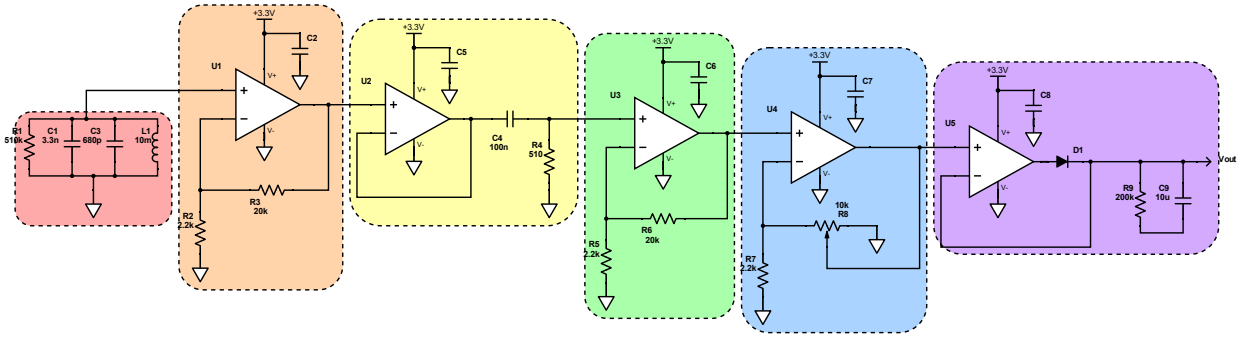


Figure 5: Electromagnetic Filed Detector Schematic. Also known as the Track Wire Detector, this circuit uses a passive tank circuit with a frequency range between 25khz and 27kHz. The signal passes through several non-inverting amplifier stages for gain, as well as a passive first-order high-pass filter before running through a peak detector rectifier. We decided to use this peak detector with a relatively large time constant so that we can get a real-time read out of the analog signal from the the electromagnetic source.

2.5 Reflective Infrared Optical Sensors

We used reflective infrared optical sensors, otherwise called tape sensors, to detect whether the bot was on black tape or white MDF. The difficulty of the tape sensors is separating between the wavelengths received. This would have taken a lot of time on our part, a lot of time that could easily be avoided by opting for the cheap, pre-made option. Therefore, we decided to buy a 10-pack of TCRT 5000 sensors that we could use on our bot instead. These came with both an analog and digital output as well as a potentiometer to control the threshold for the digital output. This turned out to be a great decision as these chips worked well out of the box.

For our bot, we put three tape sensors on the bottom to ensure we stayed on the field outlined by black tape (2 at the front, 1 at the back) and 1 tape sensor on the side of the bot to find the tape on the tower wall. We ended up using the digital output of all the tape sensors and we had no problems with them throughout the entire project. This was definitely a success.

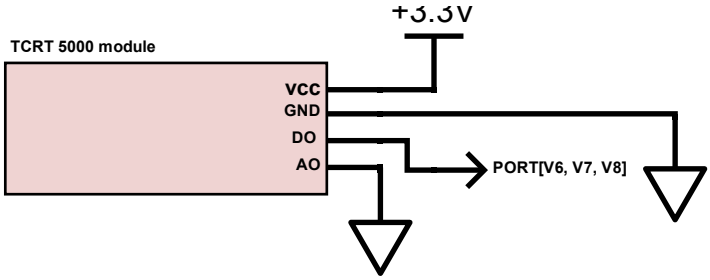


Figure 6: TCRT 5000 Reflective Infrared Optical Sensor. We decided to buy a pack of TCRT 5000 chips to read whether the bot saw a white background or black tape. We used the digital output port so we could get a simple output signal for the Uno32. The digital signal was controlled by a potentiometer to change the final comparator thresholds, effectively changing when the signal turned on low or high.

2.6 Ultrasonic Ping Sensors

The ultrasonic ping sensors were both a huge challenge and a huge success for the entirety of the project. They undoubtedly gave us the most trouble to set up, both from messing around with the timers as well as debugging a bunch of random errors. We also had to re-map the ping sensors since the distances they returned were not equal, plus we wanted to map the echo times to actual distances in inches. Eventually, once we got everything sorted out, the ping sensors were able to quickly and accurately tell us when we were parallel to the wall and when to turn to go around the walls. They were a large reason why our bot was so efficient in going around the tower and why we ended up getting 2nd in the competition as well.

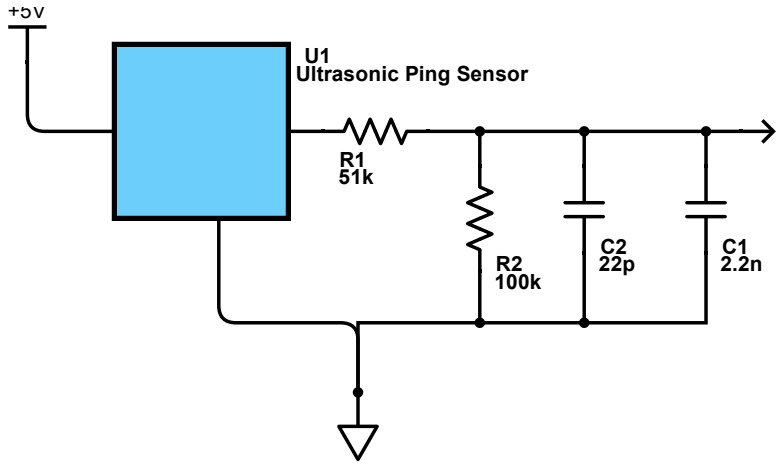


Figure 7: HC-SR04 Ultrasonic Ping Sensor Schematic. These are the ping sensors that we used to navigate around the towers. The schematic shows a voltage divider and low-pass filter on the output because we got a bunch of noise on the echo pin of one of the ping sensors. This filter removed all of that noise and allowed the ping sensors to work off 5V.

The ping sensors work by sending a digital high signal to the trigger pin of the ping sensors for at least 10us. Once 10us has elapsed, the ping sensors sent out a burst of 40kHz ultrasonic signals from one of the speakers. Then, as soon as the 40kHz burst has finished being sent out, the echo pin goes high. This pin stays high until the other speaker receives the 40kHz signal, at which point the echo pin goes low. This whole process does not take more than a few microseconds, especially at close distances. If the receiver does not pick the signal at all, it will drive itself back low after 38ms.

The provided specs for the ping sensor state that the formula for mapping the time in microseconds to the distance in inches is to use the following formula [4]:

$$D_{in} = \frac{pulse\ width(\mu s)}{148}. \tag{1}$$

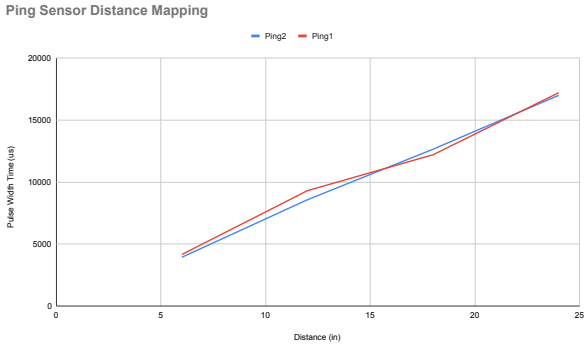
This was not accurate to what we saw with the data. Using this formula gave us wildly different values that we were expecting. Therefore, we decided to do our own mapping, by taking ping sensor measurements at different inch values and transforming the returned pulse width measurements to values in inches, which we later multiplied by 10 to get a values in tenths of inches. The actual distance mapping formulas for each ping sensor are shown below:

$$D_{in,PingF} = \frac{pulse\ width\ (\mu s) - 2125}{767}, \tag{2}$$

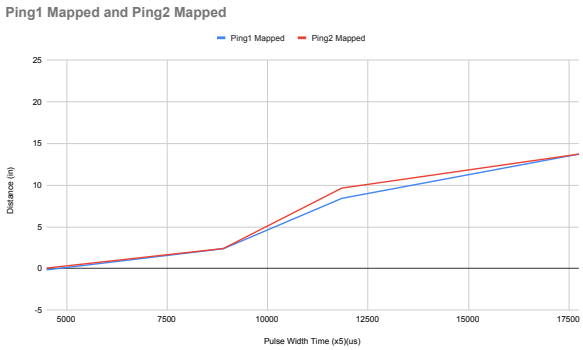
$$D_{in,PingB} = \frac{pulse\ width\ (\mu s) - 2084}{710}. \tag{3}$$

These were made by taking the average pulse width measurements of each ping sensor and fitting them to a specific formula. This means we took each ping sensor's values at distances of 3, 6, 12, 18, and 24 inches and used fitting software [2] to transform the ping values to the specified distances and return a formula in the form of an inverted line with given constants. We then took the constants across all distances and found the ideal values by weighting the shorter distances

higher and averaging all the values. This gave us the formulas above. When testing, the results we found were accurate at short range, and slightly less consistent at longer ranges. The graphical results of the before and after are shown in Figure 8.



(a) Unmapped Ping Sensor Data. This plot shows the data we collected from the ping sensors at different distances. The results show that one of the ping sensors is not terribly consistent and that at shorter distances, the ping sensors do not match up well. What we did from here is take the ping sensor pulse width values and transform them to the correct value in inches by fitting the data to a formula at each of the given distances.



(b) Mapped Ping Sensor Data. This takes the data gathered in the figure to the left and converts the pulse width of the ping sensors into distances in inches. The y-values in inches are what the code will read at the given ping sensor pulse widths. Especially at lower distances, the ping sensors are much more consistent and also accurately match up to the distance we expect in inches.

Figure 8: Ping Sensor Data Mapping.

2.7 Boost Converter

The boost converter is a very simple component, mostly because we did not really need to do anything. We purchased the XL6009 DC-DC boost converter which had a potentiometer that allowed us to change the voltage on the output of the boost converter from anywhere between 35V and 3.3V. We set it at 12V and connected the positive input terminal to the battery output and the negative input terminal to ground. This gave us a 12V rail for our Uno32 (Section 2.1) and the H-bridge module (Section 2.10).

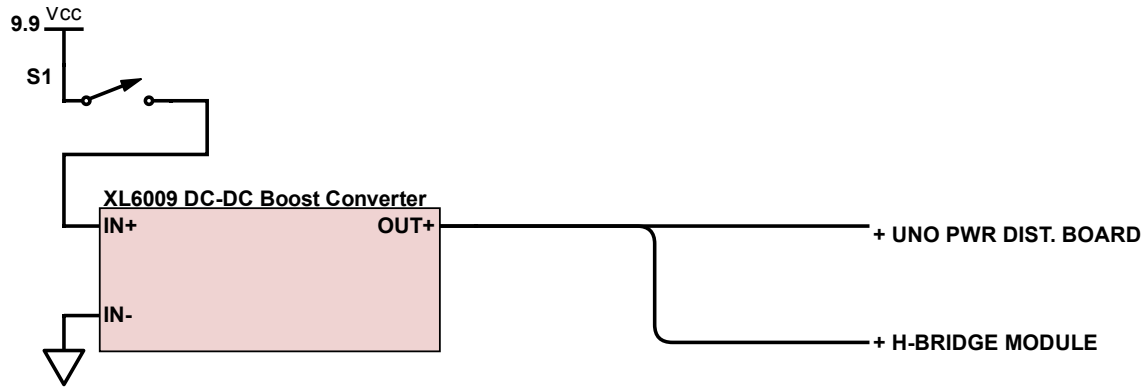


Figure 9: Boost Converter Schematic. We used the XL6009 DC-DC adjustable boost converter as inputs for our H-bridge and our Uno32. This allowed us to take the decaying battery voltage and step it up to a constant 12V. This drastically improved our timing consistency when debugging the state machines.

2.8 Bumper Sensor

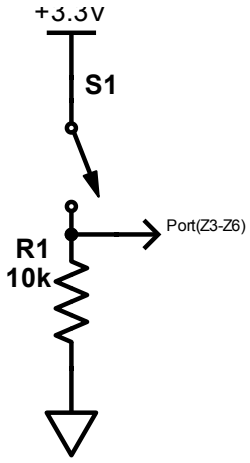


Figure 10: Bumper Sensor Schematic. We used cheap switches for our bumpers that were pulled down when not pressed and set high when pressed. The bumpers were also configured to spring back automatically to a default of not pressed.

The bumper sensors were an integral part of the project, so we made the critical choice to keep them as simple as possible. The bumpers were our way of detecting collisions with towers. The idea was to have a small piece connected of cut MDF that, when hit, would trigger a bumper switch. This bumper switch, as shown in Figure 10, would switch between high and low outputs when hit and released respectively.

Our specific mechanism that we used for the bumper was to have it essentially pivot around a specific spot. This allowed us to screw down that one spot on the bumper, which was as far as

possible from the switch itself, to allow the bumper to pivot around like a door. When hit, the bumper would swing in and hit the switch and the spring mechanism in the switch would push the bumper back out after the collision had been resolved.

We had 4 bumpers on our robot: front right, front center, front left, and back left. The front right, front left, and back left bumpers covered their respective corners while the front bumper covered the entire front face of the robot. This caught every part of the tower that the robot could hit and allowed us to respond anytime the robot was hit. The most important of the bumpers was the front left, as that was the main bumper that we used to follow the wall and stay as close as possible. We know THE more frequent we hit that bumper, the closer we are to the wall and the better we can be at determining when we have gone past the walls.

In practice, our bumpers worked phenomenally well. Especially considering we used cheap, re-used switches for our bumpers, our system was effective at detecting bumper collisions quickly and accurately. We did have some problems with the front bumper because it was so large that the screws holding it down sometimes impeded the movement of the bumper. We changed the design a bit here by moving it from underneath the bot to placing it on the first level and loosened the screws so that the bumper responded precisely while also not moving too much.

2.9 Ball Actuators

This section partly falls under the physical design section, so only the electronic design of the ball actuators will be discussed here. These were actually pretty simple; since we simply wanted to turn a motor when we wanted to deposit a ball, all we needed to do was connect a motor to one of the Uno32 pins. The problem is that the Uno32 only supplies around 20mA per output pin, which is definitely not enough for our purposes. The motor requires 700mA to 1A to turn on. Therefore, we decided to use a TIP122 Darlington pair to provide a high-current ground for the motor. We connected the Uno32 output pin to the base of the Darlington pair so that when we set that pin high, the motor would turn on. This part of the schematic is included in the Power Distribution Board schematic shown in Figure 3.

2.10 Motors

While we talked about motors in the previous section, the motors referenced here specifically discuss the motors used for moving the robot. These were controlled with a dedicated navigation H-bridge chip, the DRV8814. We simply needed to provide a direction signal and an enable signal controlled via pwm for each motor, and the DRV8814 would handle the power delivery and direction controls. We also removed as much inductive kickback from the motors as possible by connecting diodes from the input of the motors to Vcc. This meant that when the inductive kickback would go back through the chip, instead of messing up the IC, it would get rerouted to power as the diode would get turned on there, effectively removing almost all of the voltage spikes.

3 Physical Design

In our proposed design, we initially planned to use three separate ball launchers, so we needed to maximize surface area early-on. We used SolidWorks to prototype the mechanical design of the robot, exported these designs to RDWorks, and laser-cut with Medium-Density Fibreboard (MDF). We relied on lots of duct tape to construct the robot because we wanted to access our electrical components throughout testing. We made design choices that were informed by the functionality of our state machine logic as we made incremental progress in all aspects of hardware, software, and physical design.

We designed the robot to fit within 10 x 10 inch dimensions at the base giving us a 1 inch clearance of space for bumpers outward from the platform. From here, we created two smaller platforms to create compartments to hold our electrical components, as well as to extend the height of the bot. As we designed upward, we built a launch mechanism that aligned the tower hole with the center of the tube at a height of 8 inches. Nearly maximizing our height constraint, we mounted a beacon detector on top of the launch mechanism to stabilize it and guarantee that the beacon detector would not be blocked from receiving a signal at any given state. The platform design was beneficial to us as we also needed to mount our ping sensors to the front left and back left corners of the robot without disrupting the bumper sensors placed on the lower levels. The subsections that follow provide detailed explanation into the creation and design process in our mechanical design's functionality.

3.1 Chassis Base

3.1.1 Bottom Layer

We created a bottom base layer in rectangular form with angled edges to maximize surface area under the size constraint. The edges were designed for the purpose of mounting bumpers outward from the base's dimensions. On this layer, we mounted a front bumper with an extension for left-biased turns as we strategized movement about the tower. On this bottom layer, we also included details for tab and slot construction at the front, back, front-left, and front-right side as barriers to keep the electrical hardware inside of the bot. On this bottom layer, we cut out slots for tape-sensor mounts at the front-left, front-right, and back center based around our state machine logic (see Section 4.2.2). We noted the standard size for wheels was 3 inches in diameter, and approximately 1.5 inches across; there is dedicated clearance for the wheels. The CAD documentation for the base plate with dimensions is shown below as Figure 11. The front bumper is shown below as Figure 12.

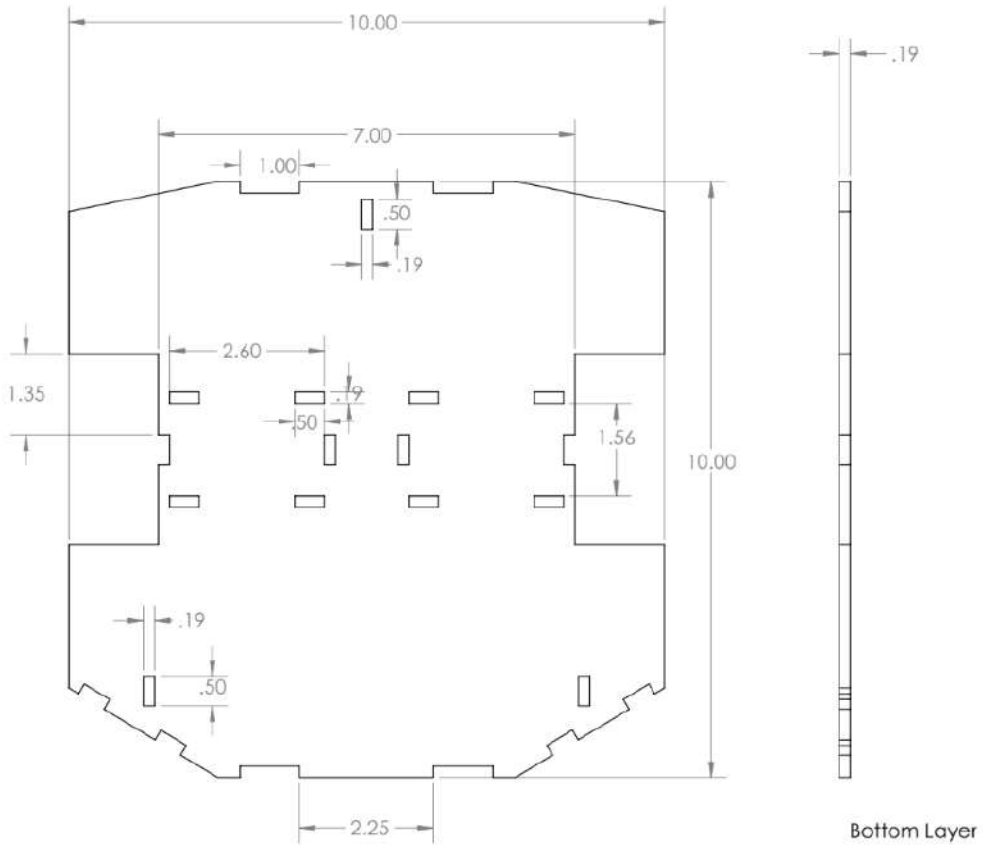


Figure 11: Bottom Layer: This bottom layer was informed by motor size dimensions for tab-and-slot construction, as well as wheel dimensions. We mounted tape-sensors into the slots at the front-right, front-left, and back center.

Figure 12: Front Bumper

3.1.2 Motor Mounts

We used tab and slot construction to secure the motors in place; the dimensions on the base platform were informed by the motors that we purchased. We mounted the motors at the center to stabilize the bot as we built upwards as well as to allow for tank turns where the bot spins without moving forward or backward. To secure these motors in place, we included a semi-circular mount, as well as side-walls, front-walls, and back-walls to secure the physical motor itself, as well as to isolate motor noise. The front-walls included 6 holes to screw the motors; this definitely secured the motors well when we added in the semi-circular mount. The back-walls included a hole to thread

wires from the H-Bridge's output for A and B side, respectively. CAD documentation for the motor mounts are shown below in respective figures Figure 13, Figure 14, Figure 15, and Figure 16. The constructed mount is shown below in Figure 17.

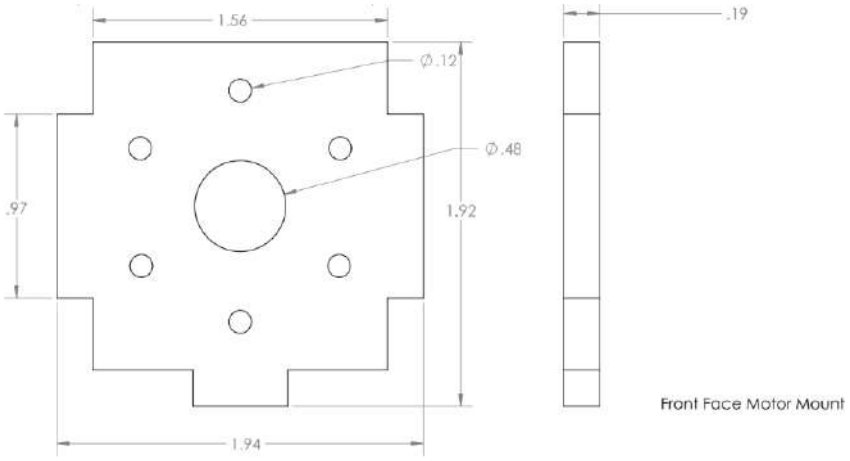


Figure 13: Front Face Motor Mount: There were 2 front face motor mounts for the 2 motors.

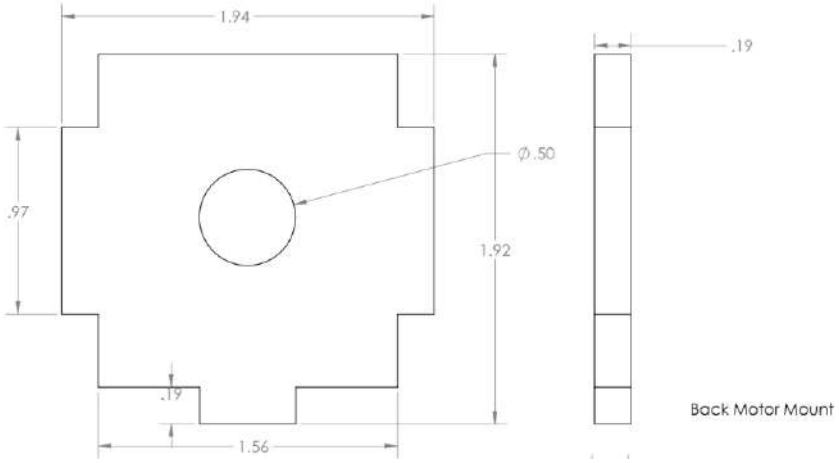


Figure 14: Back Face Motor Mount: There were 2 back face motor mounts for the 2 motors.

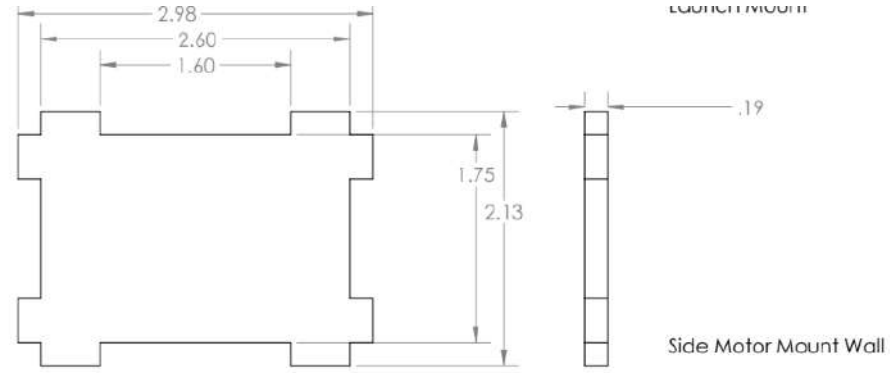


Figure 15: Side Motor Mount: There were 4 motor mounts with 2 parts dedicated to each motor. These side walls include tab-and-slot at its top to connect to the second-layer platform.

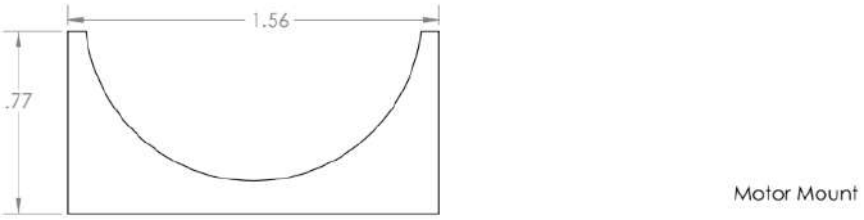


Figure 16: Circular Mount: We used 2 of these circular mounts per fully assembled motor mount to secure motors in place evenly.

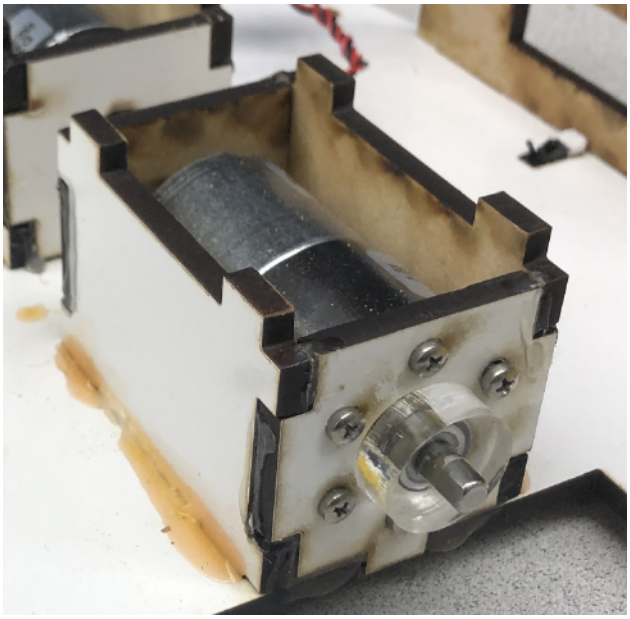


Figure 17: Constructed Motor Mount: The motor is held snugly and securely with the assembled motor mount and M3 screws.

3.2 Chassis Layers

3.2.1 Second Layer

Our second layer was the most simplistic in its design as it was a way for us to isolate electrical components and the motors, as well as for us to extend our height. The second layer is the same exact layout as the bottom layer with maximum dimensions at 10 x 10 inches with tab-and-slot construction details for the side walls at the front and back of the bot to house hardware within. This layer also includes slots for the extension to the third layer. The CAD documentation for the second layer and the connecting side walls from the bottom layer to the second layer are shown below in respective figures Figure 18, Figure 19, Figure 20.

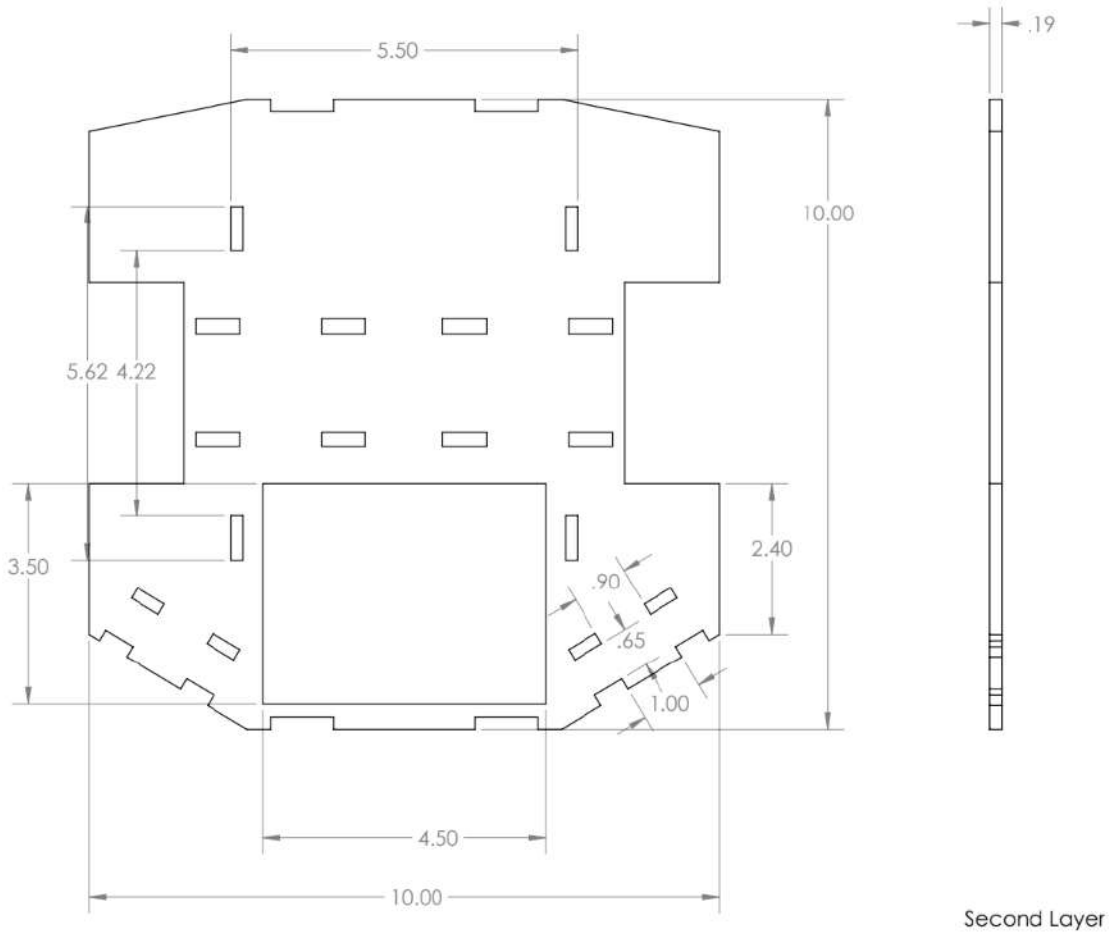


Figure 18: Second Layer: This layer has the same dimensions as the bottom base layer. Specific slots on this design are designed to connect from the base layer and to extend height to the third layer.

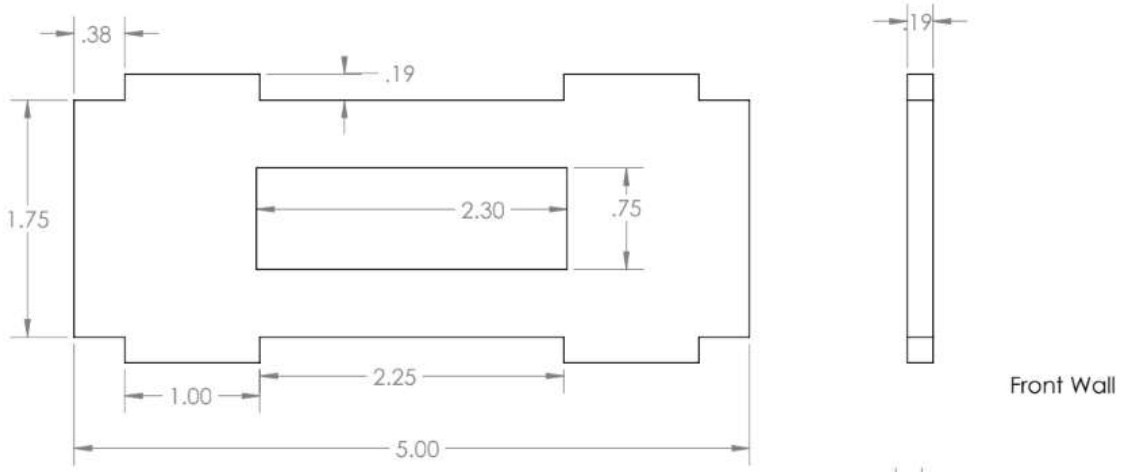


Figure 19: Front and Back Wall: We originally prototyped the design to have walls on the front and back of the robot. We found that we had some difficulty in mounting the front bumper, so we scrapped the front wall entirely when we accidentally sawed through it in an attempt to get our front bumper to fit.

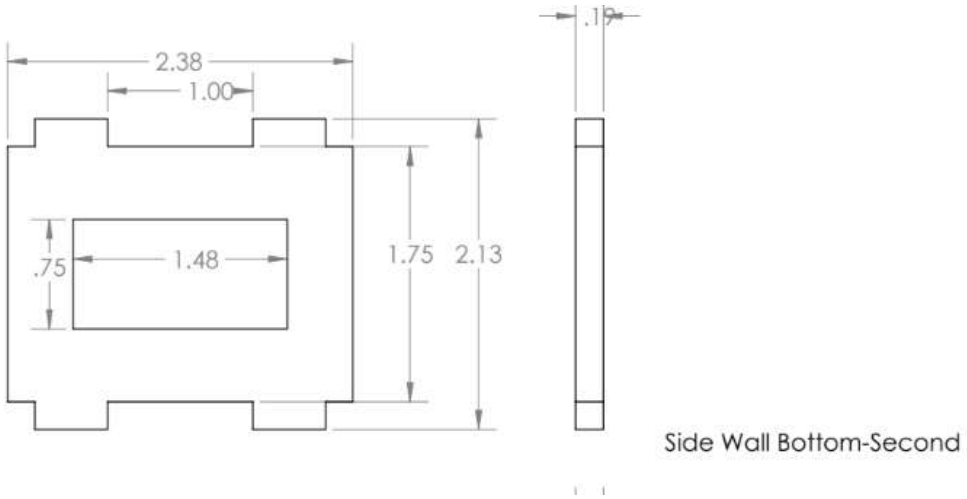


Figure 20: Side Wall (from Bottom to Second): This side wall is along the front-left and front-right edge that act as windows to the soul as our robot harnessed a personality. These walls acted as barriers for electrical components from base to second layer.

3.2.2 Side Bumpers

The second layer's main functionality was to contain the front-left, front-right, and back-left bumpers for interference detection. These front bumpers extended outward from the platform to 0.4 inches at maximum from the angled edge. From the side edge we wanted our bumpers to be parallel so we could catch the bumper_hit event, as a sensor reading, as soon as possible. The front-left bumper gives us the most information about our robot's movement in context of the tower, so this was a very important feature in both mechanical and software design as discussed previously in Section 2.8. The side bumper is shown below in CAD documentation as Figure 21 and construction as Figure 22.

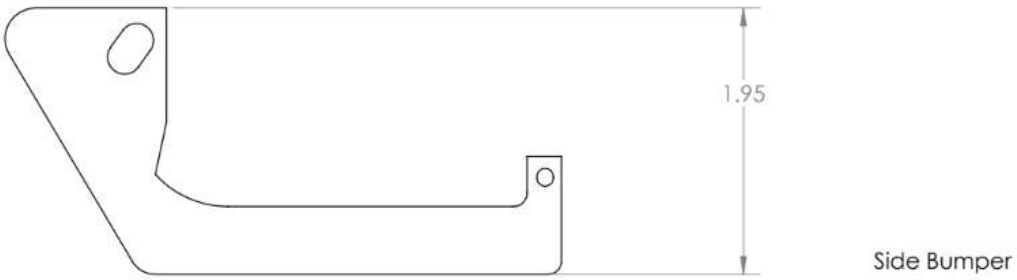


Figure 21: Side Bumper: We incorporated 3 side-bumpers for the front-left, front-right, and back-left. These bumpers give us information into the orientation of the robot as it moves about the tower.

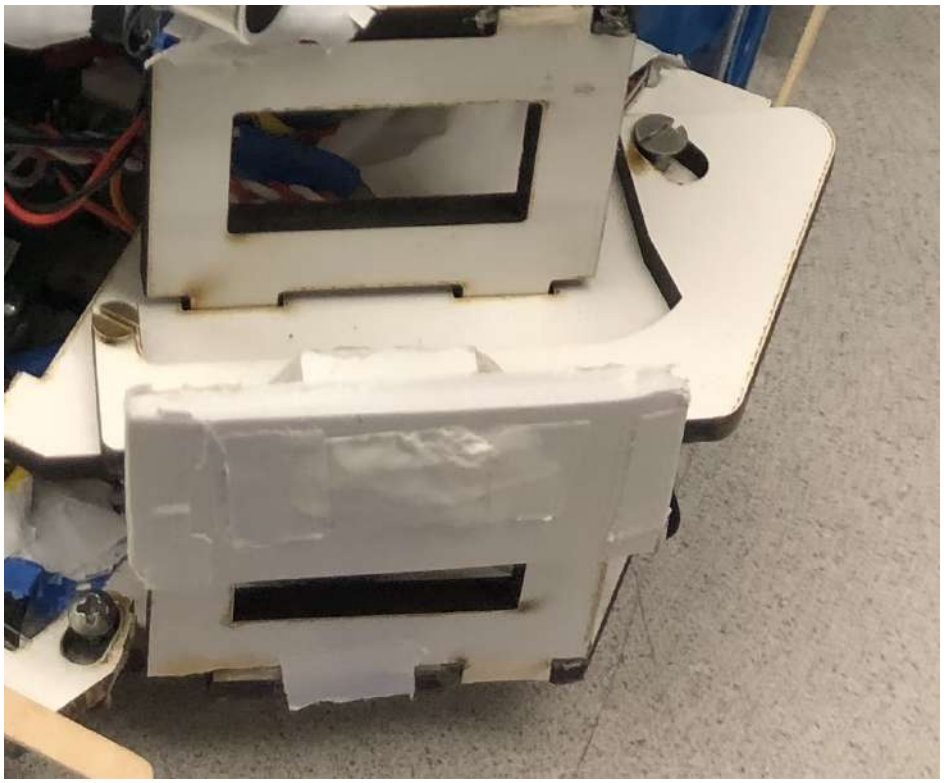


Figure 22: Constructed Front-Left Side Bumper: This front-left side bumper is the most critical bumper sensor, so we attached an extension to get closer to the tower. This would also help align us more parallel to the tower.

3.2.3 Third Layer

Early on, we thought about using three ball-dispensers, so this third layer maximized surface area as best as it could given this parameter. Over time, we learned that we could simplify and just use one ball dispenser as we tested a center ball dispenser with our code. The third layer connects from the second layer with tab and slot construction of 2 inch walls to extend height. CAD documentation for the third layer is shown below as Figure 23. The next section Section 3.3 discusses the ball-dispensing mechanism in detail.

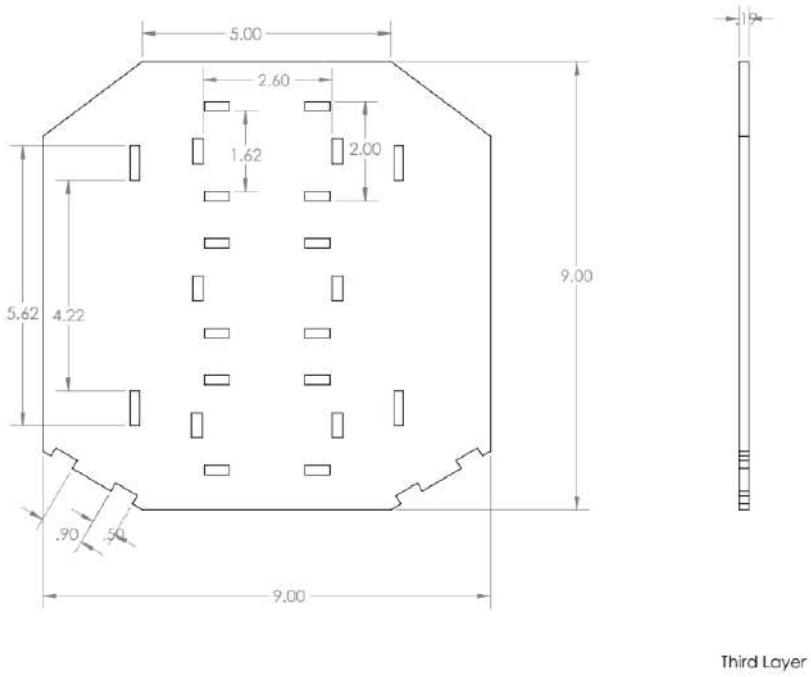


Figure 23: Third Layer: This third layer's main functionality was to act as the base platform to the ball launchers. Although we scrapped the three dispensers, the platform could fit the ping sensors on the front and back.

3.3 Ball-Dispensing Mechanism

Our ball dispensing mechanism was unique in its usage of a DC motor and technique for range-extension. As our third layer was created, we would always be guaranteed alignment to the holes with the ball-dispensing mechanism that we had prototyped in the early stages of the project. Throughout testing, we made minor changes to the design to increase its consistency. This included re-angling the tube and adding an extension to the rotor.

The rotor operated off a DC motor and as we sent a high-signal to that pin, the rotor would drop down towards the hole. To make the rotor move back into its initial place, we simply used a counter-weight and a Popsicle stick to force it back. We relied on gravity at this point. We figured that we could add an extension onto the rotor to guarantee that we could make it into the holes; we added a thin film that was flexible and lightweight.

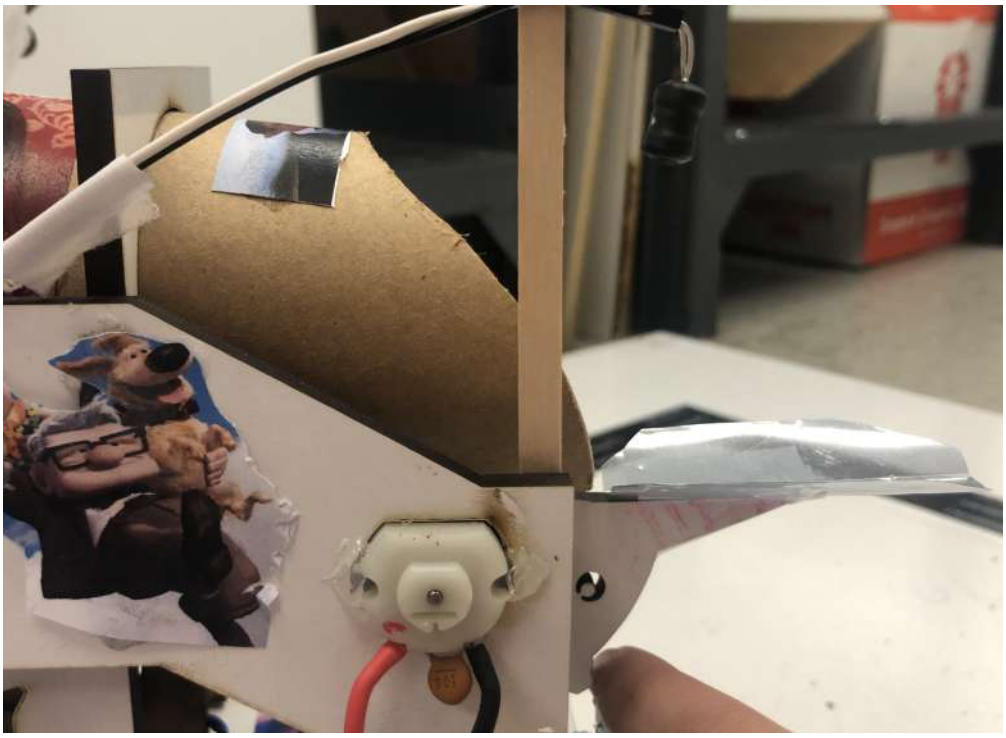


Figure 24: Ball Dispenser: This is a side view of our ball dispenser. We added 1.5 inches of a thin film and taped this onto the rotor to extend our range.

3.4 Final Product

We created an assembly consisting of all SolidWorks parts to see how our design lined up to the tower. As a verification process, we simply lined up the ball dispenser to the hole to see if it would be a mechanical success, given that our implementation of state machine logic and sensor integration was correct. The real success came in from aligning ourselves parallel to guarantee that this was a feasible option so we wouldn't have to make more adjustments in this design process. Needless to say, we only had to make adjustments to appeal to our flame aesthetic. The assembly below in Figure 25 is a SolidWorks prototype of our robot. The constructed version is shown below in Figure 26 and Figure 27.

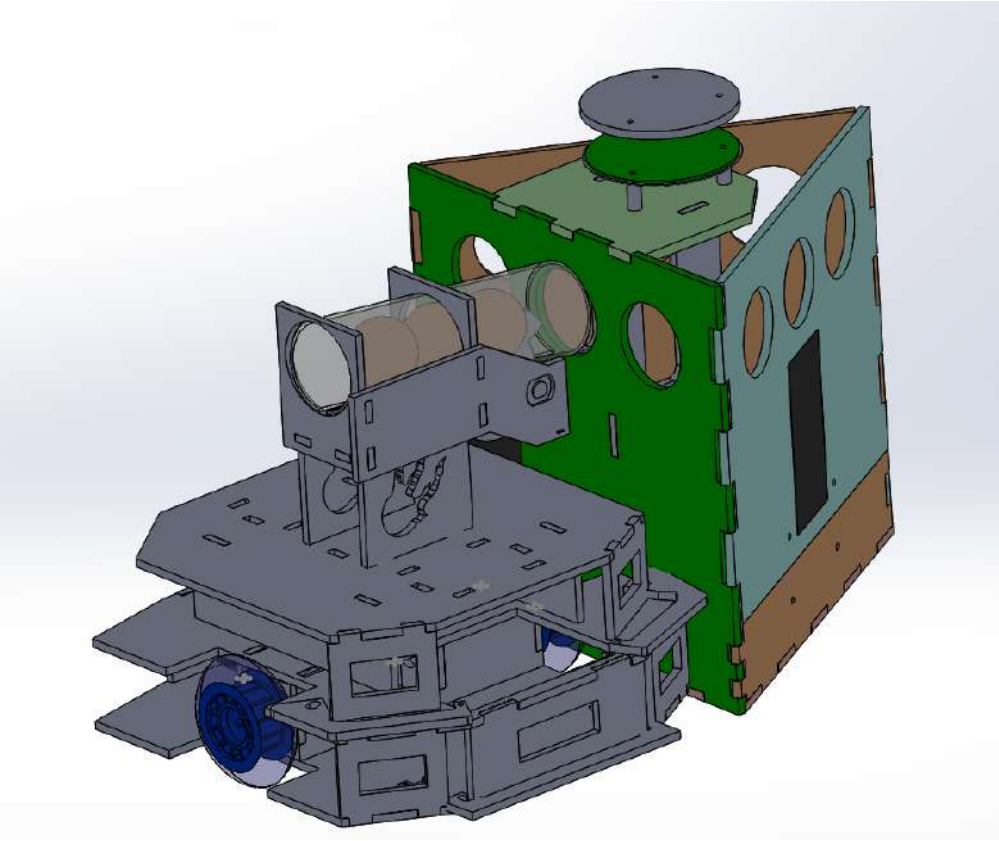


Figure 25: SolidWorks Assembly: This is our final prototype before we started to get creative for fully functional reasons.

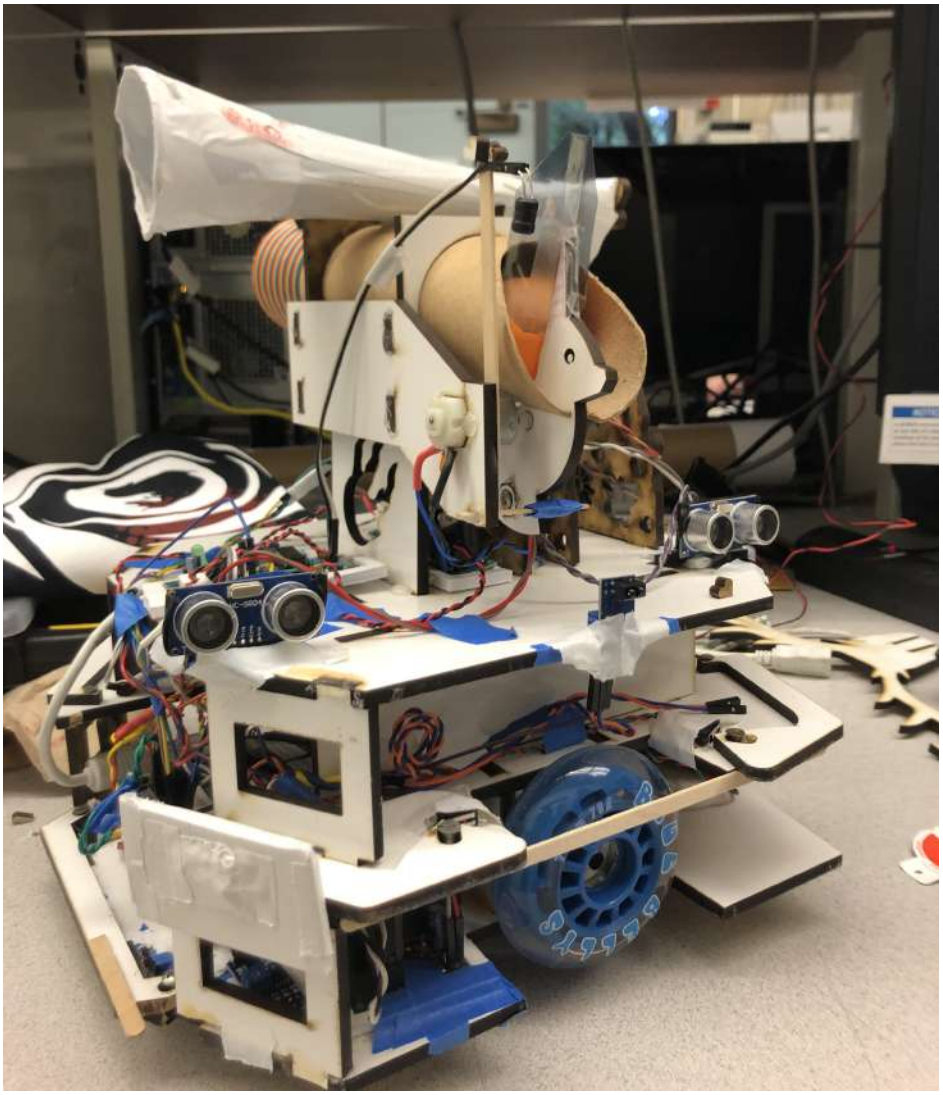


Figure 26: Final Robot (1 of 2): As you can see, the beacon detector is mounted to the top of the ball dispenser in its final constructed form. Some of the electrical components to be seen here are the ping sensors mounted to the front and back.

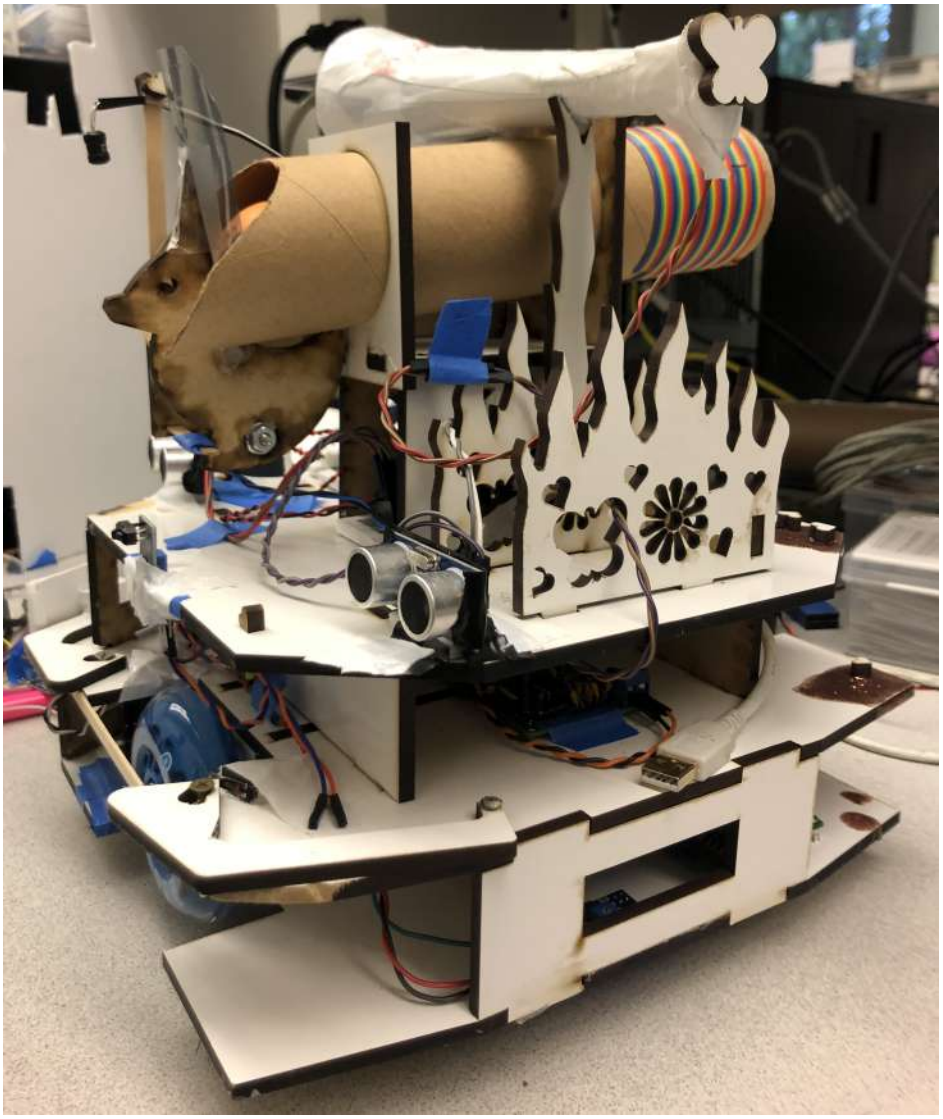


Figure 27: Final Robot (2 of 2): Functionality and beauty.

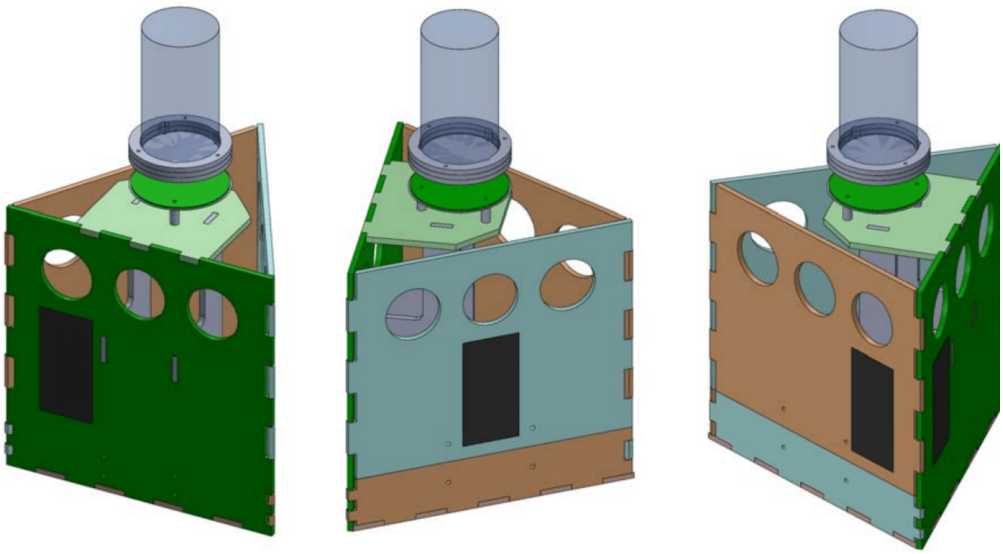


Figure 28: Beacon Towers. These are the beacon towers that the bot was required to dispense the balls in. Each tower has a single face lined with an inductor radiating an electromagnetic signal at 25kHz to 27kHz as well as tape under one hole per on each face. The only 'correct' hole on each tower is identified by the side with the electromagnetic signal and tape underneath the hole.

4 Software Implementation

4.1 Services

4.1.1 Beacon Detector

The beacon detector service was set up to periodically check the output of the beacon detector circuit at 100 Hz. The output of the circuit was analog 0-3.3V, so the service read in the analog value and compared it with a threshold. If it read a value above the threshold for the beacon, it would then debounce the input for 150 ms to make sure that it was an accurate reading. If the beacon detector reading remained high for the duration of the 150ms, it would post a beacon detected event. The somewhat long debouncing period was chosen so that the robot would have a bit of a buffer time to get lined up with the beacon before moving forward. Once the beacon detector service found the beacon, it would then continue to poll until the beacon was lost. Losing the beacon had no debouncing period because our bot made no action if the beacon was lost.

4.1.2 Bumpers

We wanted the bumpers to be as responsive as possible, so we had the bumper service running at 500Hz. This meant that the service responsible for the bumpers had to be lightweight so that it would not bog down the rest of the system. To do this, all of the debouncing and comparisons were done with bitwise operations.

Each of the four bumpers is allotted one byte of a 32 bit integer. At the beginning each call to the service, this 32-bit buffer is shifted over one bit, and the oldest bit of each byte is cleared. Then, each of the bumpers is read as a bit and added to the youngest position of that corresponding bumper's byte. Then, each of these four bytes is checked for a desired bit combination. If the byte is equal to 0x7F, then it indicates that the corresponding bumper has been pressed. Likewise, if the byte is equal to 0x80, then it indicates that the corresponding bumper has been released. If either of these conditions is true, it then checks the last event invoked by that bumper. If there has been a change from the last event, it then

4.1.3 Tape Sensor Service

The tape sensor service utilized the array of reflective infrared sensors to tell whenever the robot was driving over the tape on the field. Running at 200 Hz, the service would record the digital output of the each tape sensors. Every other reading of the tape sensors, it would check to see if the last two readings were the same. If the last two readings were the same, it would check to see if they had changed from he last two readings. This short debouncing period of 10 ms was ideal because the tape sensors had almost no noise on the digital output, and the bot needed to be able to respond to the tape very quickly.

If the bot found that the output for a tape sensor had been high for two cycles, and the last event for that tape sensor had been low, the service will then post a TAPE_SENSOR_TRIPPED event. The parameter of this event is set to be 0x1, 0x2, or 0x4 depending on which tape sensor was activated. If the opposite occurs, the service will insted post a TAPE_SENSOR_CLR event with the correnssponding event parameter.

4.1.4 Ping Sensor Service

The ping sensor service was operated using a state machine two separate timers. The first timer, located in the ES_framework was used for controlling the ping sensors and moving the service through its state machine. This timer was used for time-insensitive jobs such as sending an activation pulse to the ping sensors and periodically checking to see if the sensors had responded. The second timer was used by the input captures to measure the exact length of the return pulse. This timer, timer3, had to be configured manually using its control registers. we set it to have an 8-cycle prescaler, but this could be changed based on the ratio of resolution to measurable distance we desired. We used one input capture for each of the two ping sensors. These were configured to record the first rising edge and all consecutive edges after that. The state machine then waits for two edges to be captured then finds the difference between them.

For our design, we required two ping sensors. The structure of the state machine was designed to have the two ping sensors operating in an alternating pattern. This was done because the ping sensors required a short cool down period. As one sensor was cooling down, the other sensor was being started. This meant that we could always have one sensor running at a time, and slightly parallelized the process. We waited for each sensor to take 3 measurements, then averaged them, mapped them, and compared the results of both of the sensors. The averaging was done to reduce noise on the sensors, but a low number of samples was chosen to improve measurement time.

In order to make the data as manageable as possible, we made sure to get both ping sensors mapped to output the nearest 10^{th} of an inch. We did this by taking hundreds of samples of each sensor at set distances, averaging them and finding the trendlines. For an in-depth explanation of this process, see section 2.6. We then set up a number of event conditions to be tested after each completed cycle of the service's state machine. These compared the values of the two ping sensors to see if they were parallel, and within ball launching range. If this was the case, it would return an event stating such. Otherwise, if either ping sensor was reading very large distances, it would return a ping far event. These events were posted to the HSM to allow the robot to navigate about the tower and line up to drop the ball.

4.2 The Hierarchical State Machine

4.2.1 TopLevel

The top level HSM is essentially how we controlled which of the sub states was being called at any given time. This allowed us to isolate certain sub states, like the dancing sub HSM, from other sub states. This also allowed us to ignore some of the sensor inputs by only checking for events posted by the services when we actually wanted to respond to them. For example, when we initially look for the beacon in Init_Locate_Beacon, we only respond to the beacon_found event and not tape sensor events or bumper events. Now we will walk through the high-level overview of the TopLevel HSM.

The first state, Init, was used to initialize all the subHSMs and the reset all the module level variables. The only other use for this state would be if we needed to reset the entire program for whatever reason. This was not actually used, but having the functionality was a reliable back-up.

The next state, which also was only used directly after the Init state, was Init_Locate_Beacon. This state basically just had the bot tank turn in a circle until it found the beacon. The idea here is that we do not want to go off the field - that would be a worst-case scenario - so we opted to just spin in place if the beacon detector fails for whatever reason. Now if you are someone dedicated enough to read down through this section, you may be thinking, "That seems like a really stupid and arbitrary decision; it is better to just start driving after a certain amount of time." Well, we would like to take the chance to point out how incorrect you are.

The following is the recount of actual events in the competition. After our first round, one of our friend's team needed ping pong balls, so we moved our bot to grab them some balls and in the process accidentally pulled off the second layer of our bot. Now this is not a huge issue because we just needed to replace it back into the tab-and-slot construction joints, which we did. But what we forgot to do was to put the beacon detector pins back on the perf board. We essentially had no beacon and did not even know it. In our next round, we realized something was wrong when the bot was just spinning in circles; we thought we were done for simply because we accidentally unplugged the beacon detector. However, while we stayed on the field because we just were spinning in place, the other team's bot actually drove off the field without depositing any balls causing them to automatically disqualify. Therefore, we won because of a minor design choice to only respond to the beacon signal and not risk driving off the course. And this is how you spin to win in Mechatronics.

Returning back to the state machine outline, after finding the beacon signal in Init_Locate_Beacon, we move into Drive_To_Beacon where we call DriveSubHSM and respond to bumper events to move us into the Dancing state. All tape sensor events here are handled by the DriveSubHSM, which will be discussed in Section 4.2.2. Something we were conflicted with until the end was transitioning to ReLocating when we lose the beacon signal. This was a difficult decision because we had problems where we would lose the beacon signal and then turn the wrong direction to find it again, so instead of re-orienting to continue driving to the same beacon, we would find a different beacon and go somewhere else. In the end, we decided to just get rid of this transition because once we found the beacon, we felt confident that the bot would drive in the correct direction and eventually collide with the beacon tower, starting the DancingSubHSM.

Once we did collide with the tower, we would move from Drive_To_Beacon to Dancing, in which

we call the DancingSubHSM to navigate around the tower. This state machine is covered in detail in Section 4.2.3. This state only responded to a Ball Dispensed event posted by the LocateHoleSubHSM. This would indicate to the bot to stop dancing around the tower and begin the sequence of finding the next tower.

After dispensing a ball, the bot would go into Drive_Buffer1 where it reverses until it moves out of collision range of the tower. This we decided to just control with a timeout event once the ping sensors indicated that the bot was past a certain point. This was mainly just done to simplify some aspects of the HSM. We needed to do a little testing but eventually our robot was very consistent. Obviously we could have instituted more complex maneuvers with the track wire detector or bumpers but this just felt like the most simple and consistent method.

Once we determined we were far enough away from the tower, the bot would move into ReLocating where it just tank turns looking for the next beacon. The idea here is to spin away from the tower we were just at so that the bot is nearly guaranteed to find a different tower other than the one we just went to. Also, this state differs from the Init_Locate_Beacon state because it responds to other events besides the Beacon. First off, it uses a smart search routine to find the beacon: initially bot turns in the clockwise direction but it will switch directions it if hits tape. We also have built in functionality in the case of losing the beacon signal; the bot turns in the opposite direction of the previous instance in which it lost the beacon signal.

The ReLocating state also reacts to a timeout event; therefore, once the bot has been in this state for a certain amount of time, it will automatically stop spinning and transition to wandering. Wandering is nearly identical to Drive_To_Beacon, except that it can drive in a non-straight direction. It still responds to tape sensor and bumper events, and even beacon found events, but this state has the added capability of driving left or right justified depending on the most recent tape sensor to be triggered. This allows the bot to maneuver around the tape square in the middle of the field. This state exists for the sole purpose of catching cases where the beacon detector stops working in the middle of the challenge. Planning for this case will allow the bot to continue without the beacon and hopefully complete the challenge.

The other transitions of the Wandering state are identical to the Drive_To_Beacon state, but this state can also transition to Drive_To_Beacon if it ever finds the beacon while wandering. The reasoning here is that if we pass through the beacon's signal while driving in a left or right justified manner, we can straighten out the robot in the hopes of running into the tower. That is about all there is to the TopLevelHSM, it really just acts as a way to organize the rest of the state machines and to respond to sensor inputs and direct the code to the correct subHSM.

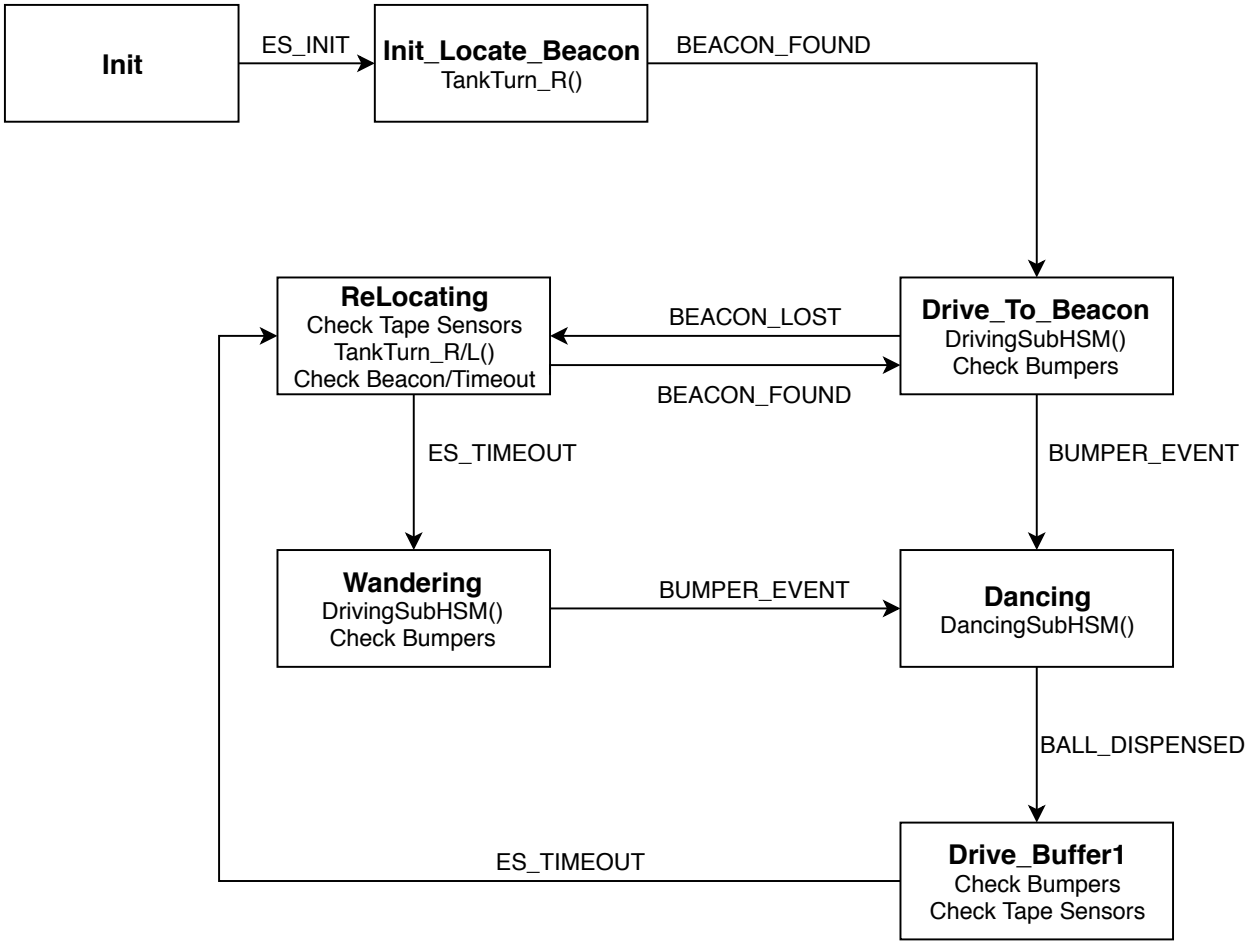


Figure 29: Top Level HSM Block Diagram. This shows the progression of the top level states and how they transition and reset upon dispensing a ball.

4.2.2 Driving

The DrivingSubHSM is likely the most simple state machine n the entire project library. This state machine has 5 main states that control the robot's movement when it is not at a tower. This includes driving forward, reversing, reversing left, reversing right, and u-turning. The transitions between states are directed by the tape sensors on the bottom of the bot, which are passed in as events from the TopLevel.

The first state, Forward, has two different functionalities depending on which state the bot is in at the TopLevel. If the bot is in Driving_To_Beacon, then the forward state will tell the bot to drive straight. However, if the bot is in the Wandering state at the TopLevel, then this state can change the way that the bot moves forward. Primarily, if no tape sensors have been tripped since the bot entered the wandering state, then forward still directs the bot to drive straight. However, if the bot hits one of the front two tape sensors, then the forward state will direct the bot to move forward with a bias in one direction. Tripping the front left tape sensor tells the bot to have a left justified movement where it still goes forward, but with a slight turn to the left. And this pattern is mimicked if the front right tape sensor is tripped, where the bot will move forward but turn slightly to the right. See Figure 30 for the corresponding code segment this movement pattern refers to.

```
case Forward: // in the first state, replace this with correct names
    if (Lost_Flag == 1) {
        Roach_LeftMtrSpeed(50);
        Roach_RightMtrSpeed(80);
    } else if (Lost_Flag == 2) {
        Roach_LeftMtrSpeed(80);
        Roach_RightMtrSpeed(50);
    } else {
        Roach_LeftMtrSpeed(70);
        Roach_RightMtrSpeed(70);
    }
```

Figure 30: Justified Direction Driving. This part of the DrivingSubHSM directed the bot in a manner corresponding to the most recent tape sensor event. The code works by using the Lost_Flag variable set by the TopLevel when calling this subHSM. A Lost_Flag of 1 will have the bot move left justified, a Lost_Flag of 2 will have the bot move right justified, and anything else will have the bot going straight.

The reasoning behind integrating this unusual behavior comes from the idea of tape following. This is the act of driving along the tape, but without actually crossing the border. The benefit is that the bot can guarantee find its way around the tape without getting stuck. It will also help the bot to find the same tower again after losing the signal from the tape event.
Speaking of tape events, the way we respond to tripping those sensors is by transitioning from Forward to either L_Turning or R_Turning. These two states handle turning after the bot runs over tape on the field. The procedure here is to force the bot to respond to the tape event before returning to forward movement. This is done by both checking for a timeout event to tell the bot that it has turned for long enough and by checking that the triggered tape sensor is clear. For obvious reasons, we do not want the bot to continue moving forward before it has cleared the tape sensor, but we also want to ensure the bot has turned for a reasonable amount of time as well. Once both conditions have been satisfied, it will return to Forward where it will either drive straight again or it will drive in a specific, justified direction.

The last two states are connected in the sense that they both contribute to the bot performing a full U-turn. The idea is that if we trip both front tape sensors within a short time period, we want to have the bot back up, and turn around to continue in the opposite direction. These are both performed on timeout events instead of sensor inputs. Thankfully, we have never had a use for this scenario because usually only one tape sensor gets tripped at a time. Still it is nice to have the functionality.

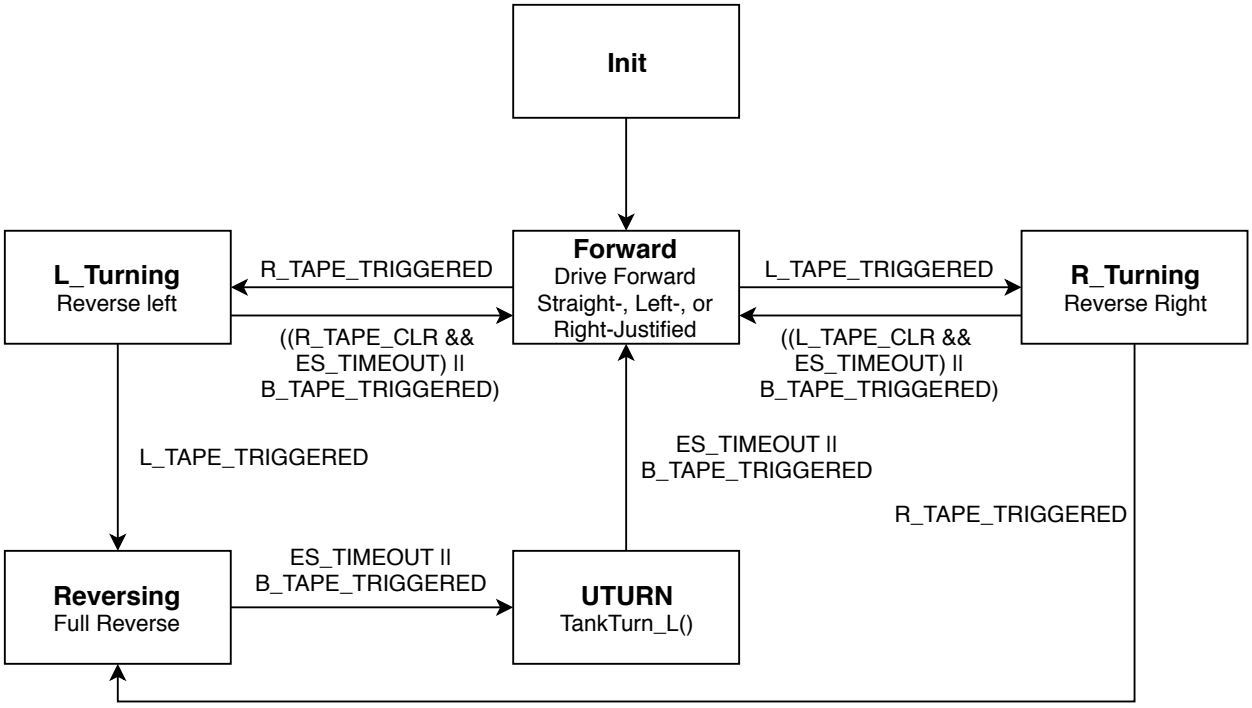


Figure 31: Driving subHSM Block Diagram. This shows how the motors were controlled based on inputs from the tape on the ground.

4.2.3 Dancing

The DancingSubHSM is easily the most complex state machine in the project library. This state machine gets called from the TopLevel from Dancing, which is transitioned to whenever we have a bumper event. Its purpose is to efficiently navigate the bot around a tower and to indicate when the bot has successfully navigated to the center of the correct side and is parallel to the face of that side. Once the bot has determined it is parallel and close enough to the correct side, it will call the last subHSM, LocateHoleSubHSM, to dispense the ping pong ball.

The analysis here will be lengthy, so bear with us as we try to elaborate on each part of the DancingSubHSM. To simplify the analytical discussion, we have broken it down into 7 distinct parts: Entry, Left Bumper Response, Right Bumper Response, Back Bumper Response, She Cruzin, Parallel Close Response, and Parallel Far Response. Also please keep in mind that this part of the project was the most time consuming and mind numbing experience of the quarter, that is why many of the state machine names are rather ridiculous and in no way are intended to be taken seriously or offer offense. It was just another way to keep us sane throughout this project. If you find any of the state machine names offensive, we encourage you to read up on their specific sections which explain why each of the states are aptly named.

Entry
Starting with the Entry section, this is where the state machine remains idle until a bumper event gets passed down from the high level. Since this state machine only gets called after an initial bumper event; therefore, that initial event directs how the state machine first reacts. Since obviously, if the right bumper is tripped, then the state machine moves into RB_Reverse to handle

that response. Consequently, we move into LB_Reverse if the front left or front center bumper are tripped. The only time we move back into Entry is after a BALL_DISPENSED event from the Cha_Cha_Real_Smooth.

Left Bumper Response
The next stage is the Left Bumper Response. This is what the state machine does to handle any left or center bumper events, which can be transitioned to from any state in the state machine. We want to have this type of transitional freedom to ensure that our bot does not get stuck against the left or front bumpers. The routine here is to immediately back up to the left, away from the wall until the front left bumper or front center bumper gets released. At this point the state machine will transition from LB_Reverse to LB_Buffer and start a timer. In LB_Buffer, the bot waits for the previously set timer to elapse. Until it does, it will continue to reverse in the same manner as LB_Reverse. Once the timer elapses, the state machine transitions to Criss_Cross where the bot tank turns clockwise for a number of milliseconds. This is just to get the bot re-oriented parallel with the wall without using the ping sensors. After a short timeout, the bot will transition from Criss_Cross into She_Cruzin, which is the next stage of the DancingSubHSM.

She Cruzin
The She Cruzin response is a nod to the state called She_Cruzin, where the bot moves forward again in a left justified manner. This behavior is similar to the tape following behavior of the previous section (Section 4.2.2), except now we are wall hugging. The idea is to move along the wall as quickly as possible while still maintaining a low average distance from the wall. This procedure is carried out by continually switching between the Left Bumper Response states and She Cruzin until the ping sensors indicate that the bot is parallel to the wall.

She Cruzin also has a small added behavior that is performed on a timeout event in She_Cruzin. After a certain amount of time where no bumper events have been triggered, the state machine will briefly transition to Quickly, which is the mislabeled state below and to the left of She_Cruzin in Figure 33. This state simply has the bot do a hard turn to the left for a short period of time before transitioning back to She_Cruzin if no bumpers were tripped or to LB_Reverse if the left or front bumper are tripped.

Back Bumper Response
This response is more error catching than an actual response. The idea is that if we misjudge one of our turns and hit the back left bumper for whatever reason, we can immediately respond to that event and provide some kind of error correction. Once the back bumper is tripped, the bot will immediately drive forward in a left justified manner with the intention of hitting the front left bumper. This will hopefully force the bot back into its normal pattern of wall hugging.

```
#define GoTo_Entry nextState = Entry;\
                makeTransition = TRUE
#define GoTo_LB_Reverse nextState = LB_Reverse;\
                makeTransition = TRUE
#define GoTo_RB_Reverse nextState = RB_Reverse;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, RB_REVERSE_TICKS)
#define GoTo_LB_Buffer nextState = LB_Buffer;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, LB_REVERSE_TICKS)
#define GoTo_Switch_Stance nextState = Switch_Stance;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, SWITCHING_TICKS)
#define GoTo_She_Cruzin nextState = She_Cruzin;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, CRUZIN_TICKS)
#define GoTo_Woah_Nelly nextState = Woah_Nelly;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, NELLY_TICKS)
#define GoTo_Vibe_Check nextState = Vibe_Check;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, 1);\
                Ping_Flag = ThisEvent.EventParam
#define GoTo_Backup_Dat_Trunk nextState = Backup_Dat_Trunk;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, DAT_TRUNK_TICKS)
#define GoTo_Ramming_Speed nextState = Ramming_Speed;\
                makeTransition = TRUE
#define GoTo_Cha_Cha_Real_Smooth nextState = Cha_Cha_Real_Smooth;\
                makeTransition = TRUE
#define GoTo_Quickie nextState = Quickie;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, QUICKIE_TICKS)
#define GoTo_Quickie2 nextState = Quickie;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, QUICKIE_TICKS2)
#define GoTo_Straight_Pride nextState = Straight_Pride;\
                makeTransition = TRUE
#define GoTo_Heterophobe nextState = Heterophobe;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, HETEROPHOBE_TICKS)
#define GoTo_D_Flip_Flop nextState = D_Flip_Flop;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, FLIPPING_TICKS)
#define GoTo_Criss_Cross nextState = Criss_Cross;\
                makeTransition = TRUE;\
                ES_Timer_InitTimer(DANCING_TIMER, CRISS_CROSS_TICKS)
```

Figure 32: GoTo #Defines. These pound defines made the DancingSubHSM many times more efficient and look a lot cleaner. Instead of typing out all the lines to make transitions each time, we decided to make the GoTo statements that handled all that for us. Not only is the DancingSubHSM legible, but these #Defines also remove the possibility of error occurring from timer mistakes.

Parallel Close
Parallel Close is the response that the bot goes through once the ping sensors identify that the bot is parallel, in the center, and close enough to a face of the tower. This would indicate that the bot is ready to begin the sequence for dispensing a ping pong ball. But before it can do that and call the LocateHoleSubHSM, the bot needs to identify that this is the correct face of the tower.

This section handles the side-checking sequence in Vibe_Check, where the bot checks for the electromagnetic track wire signal. The transition to Vibe_Check occurs under the conditions stated in the above paragraph from any state. The rationale is that as long as we can satisfy those criteria, regardless of what state we are in, we should immediately transition to Vibe_Check to see if we can begin dispensing a ball. If Vibe_Check determines that this is indeed the correct face of the tower, then the state machine moves into Cha_Cha_Real_Smooth where LocateHoleSubHSM is called. If this is the wrong face, then the bot will respond in the same way as a Parallel Far event.

Parallel Far
The Parallel Far response is the sequence that defines how the bot quickly maneuvers from the center of the wrong face to the next side of the tower. The transition to this stage comes from a Parallel Far event in any state or a low track wire signal from Vibe_Check. To begin the Parallel Far response, Straight_Pride, will drive the bot straight until the front ping sensor detects that the front of the bot is beyond the face of the tower. This state is name Straight_Pride because nobody wants to have to undergo this response, similar to how nobody legitimately believes in straight pride. We would much prefer to be close enough the right face so we can Cha_Cha_Real_Smooth.

Regardless, once the front of the bot is beyond the face of the wall, we begin a timer that will allow the bot to continue a little farther and give it some distance from the edge, to give the bot some room to make the following turn. The bot then sharply turns to the left in Heterophobe until the front left bumper or front center bumpers are tripped. The reasoning behind this name is that after going straight for a long time, we no longer want to go straight and instead need a sharp turn. If we are in this state for too long, we will forego trying to find the tower and admit we are lost. We then transition back to Entry and post the Ball Dispensed event to the TopLevel to reset the system. Otherwise, once we hit the left bumper again, we can resume normal behavior of switching between Left Bumper Response and She Cruzin.

That wraps up the analysis of all of Dancing stages. Again, the idea is to simply wall hug the tower until we find the correct face and can identify we are parallel, in the center, and close enough to that face. Once we can do that, we call the next subHSM, LocateHoleSubHSM to find the tape on the wall and dispense the ball.

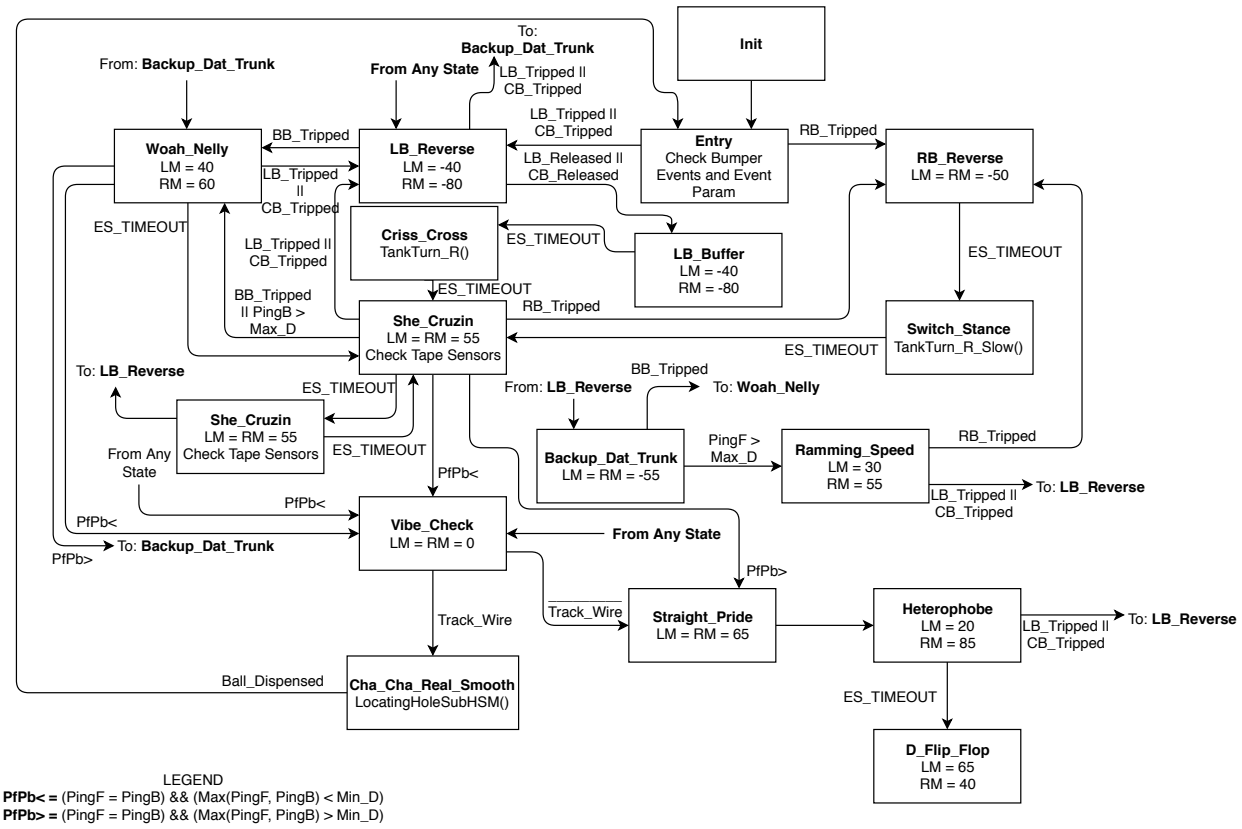


Figure 33: Dancing subHSM Block Diagram. While convoluted, this outlines the process for going around the tower. Many of these states were solely implemented to catch edge cases while the majority of the time the bot would transition between the LB_Reverse state progression and She_Cruzin to effectively navigate the tower's sides.

4.2.4 LocateHole

LocateHoleSubHSM is the final subHSM of our project library. This subHSM deals with finding the tape on the wall indicates the correct hole and provides the control sequence to dispense the ping pong balls. This state is called from within DancingSubHSM once the bot has identified that it is parallel to, in the center of, and close enough to the correct face of the beacon tower.

In this subHSM, the bot will initially tank turn clockwise or counter-clockwise depending on which ping sensor distance was higher when the bot identified that it was parallel with a face. This tank turn is very brief, only for 10 milliseconds. Then the bot will begin to move forward and backward along the face of the wall until the tape sensor under the ball dispensing pipe locates black tape. It performs this movement incrementally, starting with small movements and gradually increasing the distance that it moves from the center of the wall until it finds the tape.

Once the tape sensor finds the black tape, the bot will transition to Ride_At_Dawn and creep forward until the tape sensor goes low again, indicating the end of the tape. Here the bot transitions from Ride_At_Dawn to Retreat. This is done so we know that the bot starts at the same relative location to the hole. Once the tape sensor goes low, the bot backs up a specific amount specified by a timeout event. Determining the length of this timeout event took lots of testing and

playing with numbers to get it right. Once we landed on a specific number, the bot was able to almost always able to perfectly line up the dispensing pipe with the tower's hole.

After the timeout event, the bot pauses quickly as it transitions to Give_Me_A_Sec. This is just to give the bot time to turn off the motors and let it get situated before dispensing the ball. After a short timeout, the bot turns on the ball-actuating motor on the transition to Squirting_Dat_Sweet_Slime and the ping pong ball comes out of the pipe and into the hole on the tower. After two seconds, the bot turns off the actuating motor and posts a Ball Dispensed event to the TopLevelHSM and resets itself back to Dancing. At this point the ball should be successfully deposited in the correct hole on the tower and the TopLevel should know to begin looking for the next beacon tower.

In practice, our code worked really well. In the competition when all the code was perfected, we did not miss a single instance of getting to a tower and dispensing a ball in the wrong hole or missing the holes all together. In the check-off, we were able to get the first 5 tower over 2 runs, after which we turned off the bot and celebrated our success. I firmly believe that we had the most efficient and consistent method of dispensing balls combined with the physical fabrication of the bumpers. The only reason we lost the competition was due to unlucky placement of the towers and the fact that the other team's bot pushed us out of bounds, which we did not even plan for as it was not part of the project requirements. Even so, we did really well and this project library is a large contributor to that success. This is further elaborated in Section 5.

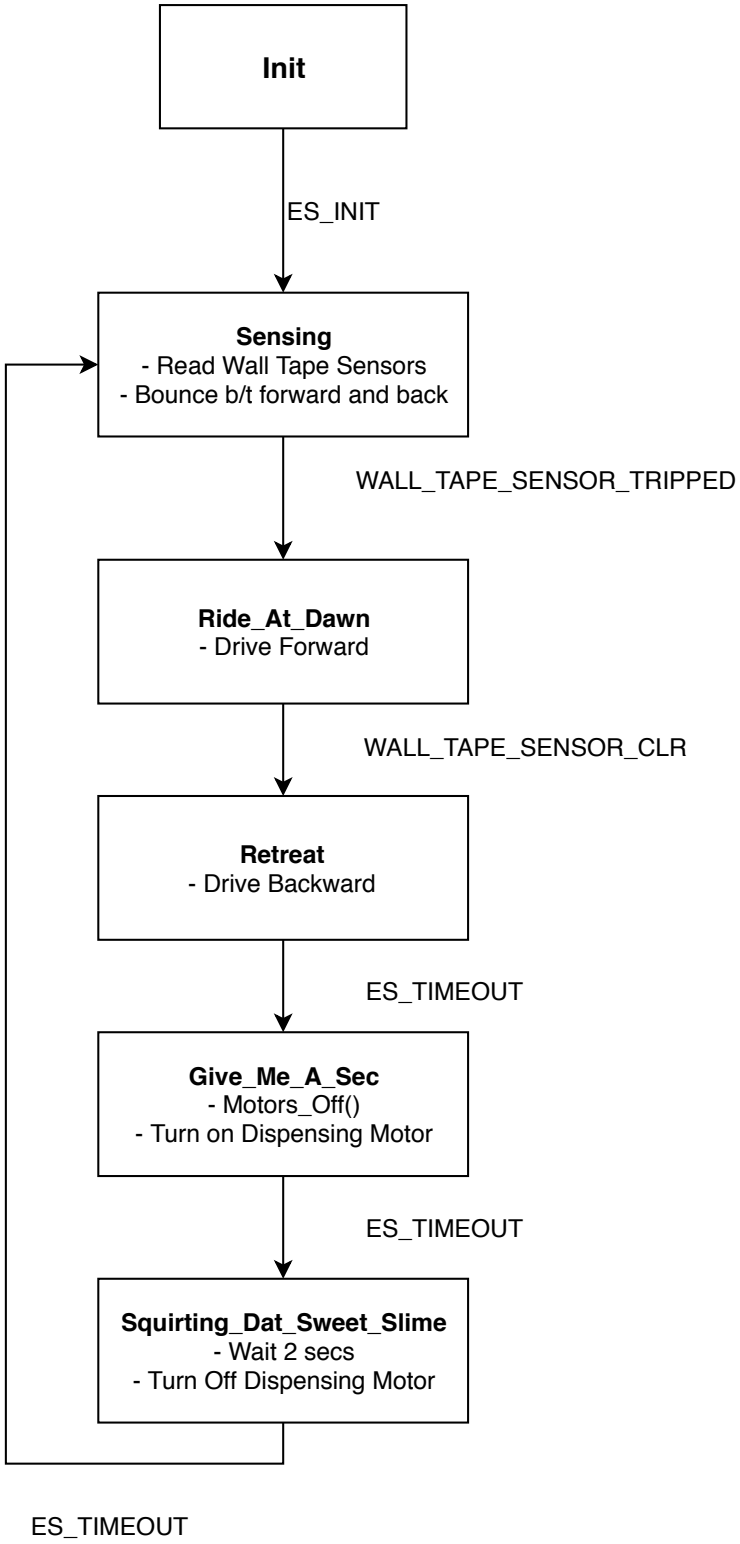


Figure 34: LocateHoleSubHSM Block Diagram. This state machine was used to navigate to the location of the tape on the wall to find the correct hole, dispense the ping pong ball payload, and signal the rest of the system to reset.

5 Conclusion

Holistically, our project was mostly a success. We obviously encountered numerous obstacles throughout the four-week project. But after the first two weeks, the majority of our challenges were behind us, allowing more time for fine tuning as opposed to debugging. Each major milestone was achieved ahead of schedule, proving to not only the TAs, but ourselves too, that designing and building an autonomous robot was actually doable in three weeks (with no all-nighters). And not only did we do it, but we did it well, surpassing our supervising TA's expectations. To be completely honest, our TA was so surprised that he bought the whole team a pitcher of beer during the post-competition celebration.

For this project, our team was given a budget of $150, but we still had to supply the money ourselves. Regardless, we came in way under budget at $79.39. This was among the lowest amount of money spent on any of the robots. The biggest attribute to this low cost was our re-usage of DC motors as ball actuators. Instead of buying a set of RC servos, we re-purposed old motors and designed a unique mechanism utilizing this uncommon component. Inevitably, all parts of the robot worked successfully throughout testing and competition, but we had to struggle before we hit our stride.

5.1 Challenges

If we want to talk about challenges, we undoubtedly need to talk about the ping sensors. Even just the name sends ironic laughter throughout the room as everyone understands the struggle of getting these pesky sensors to properly work. The difference between the ping sensors and the rest of the electro-mechanical system is the clock speed they need to run at. All other parts of the robot work at super-millisecond timing. The ping sensors worked at microsecond timing and this presented a challenge for the ES_Timers framework. While trying to get the ping sensors to run off separate timers, we still needed a way to call the service that handled each of those ping sensors. And this is what caused a spiraling chain reaction of mistakes that culminated in bugs across all sensor services.

The solution to this problem was to remove the recurring timers that we initially put in all the services. Thankfully, this solved the problems in the other services without breaking the ping sensors. But in total, we spent nearly a week debugging and rewriting the ping service. Outside of this obstacle, the front center bumper was an unexpected challenge. We initially mounted that bumper on the bottom of the robot, but the problem we encountered involved the the screw holing it in place. This screw would jam the bumper and prevent it from smoothly moving back and forth. We tried to fix this by moving the bumper to the top of the bottom layer, but we had to saw off a decent amount of the sides off that bumper. Still, this provided a temporary fix to the problem; if we were to redesign the robot, this would be a primary concern for the physical design process. Outside of these two issues, we did not have any other significant challenges that took unnecessarily long to figure out. So now we can move onto the successes of our project.

5.2 Successes

The ping sensors belong here as well. As we were one of the two groups in the end who successfully implemented ping sensors, this is a big source of pride for our team. Not only is it a moral victory, but it was the main contributor to the robot's efficiency in navigating around the beacon towers. Another of our big victories was the ball dispenser mechanism. Outside of just the motor actuator, our ball dispenser used only two total components. This was the fewest of any of

the teams as we subscribed to the idea of simplicity first. We figured out that the key to consistency is to simplify; reduce the number of moving parts and you limit the sources of potential error.

One of our other successes came from the physical design as a whole, especially the bumpers. Our chassis had more than enough space and was durable enough to withstand impacts with the towers. Each of the bumpers was extremely responsive yet securely in place, and on top of that, we covered all the edges that we wanted to without any panel gaps. From a design perspective, there would be minimal physical changes in the second iteration of our robot.

The final success with our robot was the dancing state machine. This state machine was easily the most complicated part of our project, yet we had no problems debugging or with any of the states not working properly. It was clean, efficient, and worked in implementation as well. Overall, there were many more challenges and successes we encountered throughout this project, but to name them all would be repetitive, bordering on boring. Instead, we believe it is safe to say that our diligent work on our robot yielded tangible results in the form of a 2nd place competition finish with a simple, yet efficient design. Thank you for taking the time to read through this report and if you have any questions, please feel free to email any of the persons on the title page of this report.

6 Acknowledgements

We received guidance and debugging help from assigned tutor Daniel Brenner as well as the rest of the CMPE 118/L teaching staff.

We also want to thank John Pei for his help in making acrylic molds for our wheel-to-motor mounts. Without him, it would have cost quite a bit more to go buy the acrylic and make the molds ourselves.

We also want to thank all the other participating teams for an exhausting and rewarding quarter; while the Mechatronics experience was definitely a lot of work, it was without a doubt worth it in the end.

References

[1] J. Edward Carryer. *Introduction to Mechatronic Design*. Pearson, 2010. URL: https://www.amazon.com/Introduction-Mechatronic-Design-Edward-Carryer/dp/0131433563 (visited on 12/07/2019).

[2] *GNUPlot Homepage*. URL: http://www.gnuplot.info (visited on 01/14/2019).

[3] Microchip. *MCP6001/1R/1U/2/4 1 MHz, Low-Power Op Amp*. Microchip. URL: https://ww1.microchip.com/downloads/en/DeviceDoc/21733j.pdf (visited on 12/07/2019).

[4] ITead Studio. *Ultrasonic ranging module : HC-SR04*. URL: https://www.electroschematics.com/wp-content/uploads/2013/07/HC-SR04-datasheet-version-2.pdf (visited on 12/07/2019).