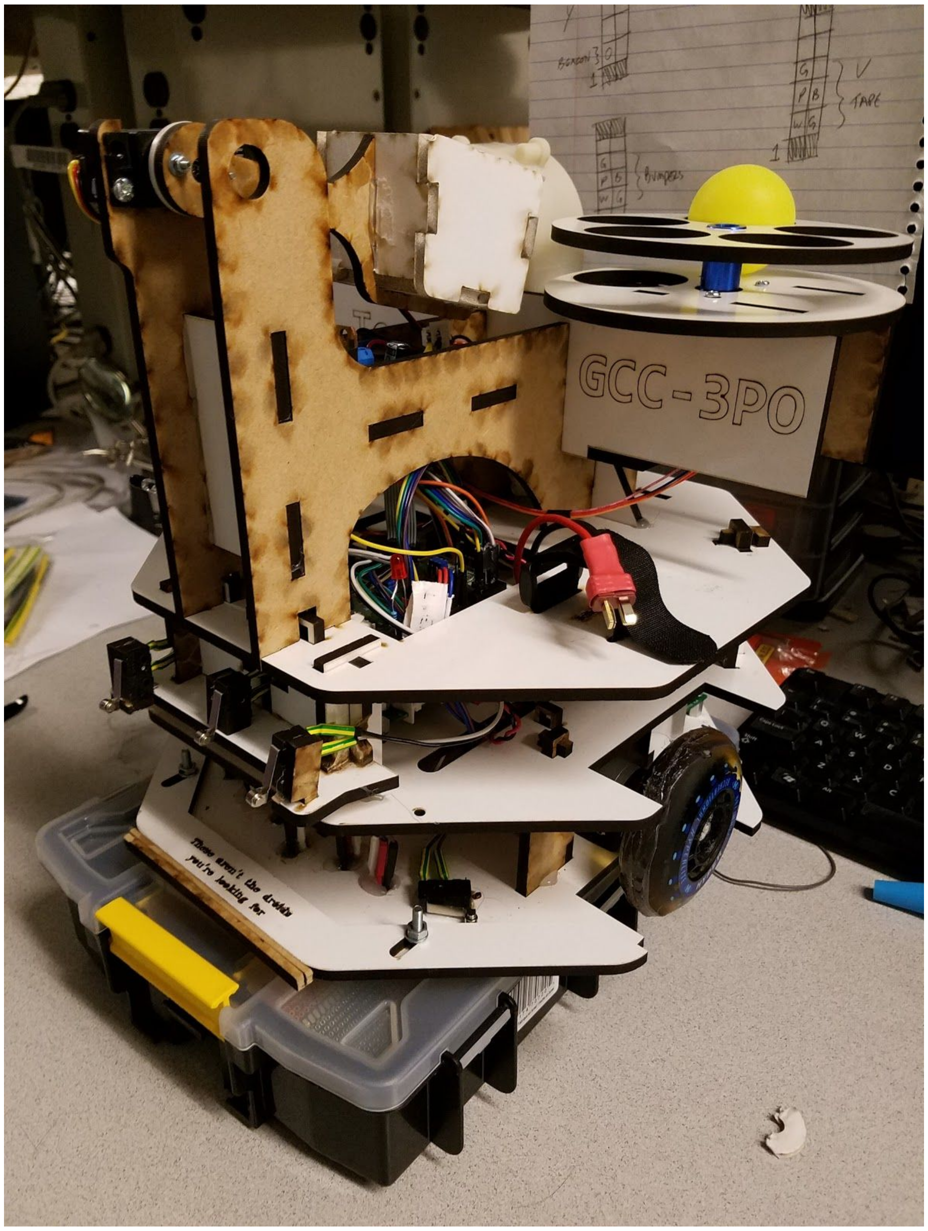
# CMPE118: Introduction to Mechatronics
# Final Project Report

Team GCC-3PO

Joseph Adamson
Trevor Muraro
Zili Wu



## Introduction

Slug Wars: The Last Slug-I. After several weeks of preparatory labs, we were finally ready to start the journey of creating our robot. The challenge: take down 3 AT-M6 targets, and then deliver the final blow to the evil Kylo Ren's ship. The field is bordered with tape, which our sensors must detect to stay within bounds.

The AT-M6 targets are scattered randomly throughout the field, demarcated by a wire generating an oscillating magnetic field. They must be shot with a ping pong ball, after which point the wire would go inactive, indicating a successful kill.

After shooting all three targets, the last step is to deliver a precise shot to the final target, during which the robot must deliver a 1.5" ping pong ball to the 2" target hole, 16 inches off the ground. A beacon emitting an IR signal at a specific frequency illuminates, indicating the position of the final target when it is ready to be shot.

The process of creating a machine to achieve this task brought us through all the steps of designing and fabricating a fully integrated system consisting of sensors, actuators, and the overarching state machine to control them all. During our journey we faced many problems, to be detailed below during the discussion of each phase of the design. A great many problems arise when bringing together elements which function perfectly well on their own. Each element of the system brings its own noise, which must be dealt with in a way which still allows complete functioning.

Ultimately, after a tiring 5 weeks, we have successfully created a robot to achieve the task laid before us. Below we detail our plans, methods, and challenges.

## Mechanical

### Robot Three-layout Platform

The design of our robot platform is based on the size requirement, the robot size need to fit into a 11-inch cube. We considered to build our platforms in circular or square shapes, however, we selected polygons with 8-sides because it has equilibrium length for all sides and sides are perpendicular to all 8 directions. Our robot could lineup to its center and rotate 45*X degrees (X as number of sides) back to the position facing forward when it hit on an obstacle. The width of the polygon is 10 inches, so we have extra inch for implementation before matching the size requirement.

Since the size of our shooting mechanism is about 5.5 inches height, our platform need to reach to 5 inches height to maximize the size of our robot. We used a three-level design for our robots that we could make more spacing for wiring and circuit board management. The levels were connected by supports using tab-and-slot construction. The supports between levels were glued to the platform below them, but held to the platform above them using MDF pegs put through holes in the supports. This allowed the robot to be disassembled level-by-level if necessary.

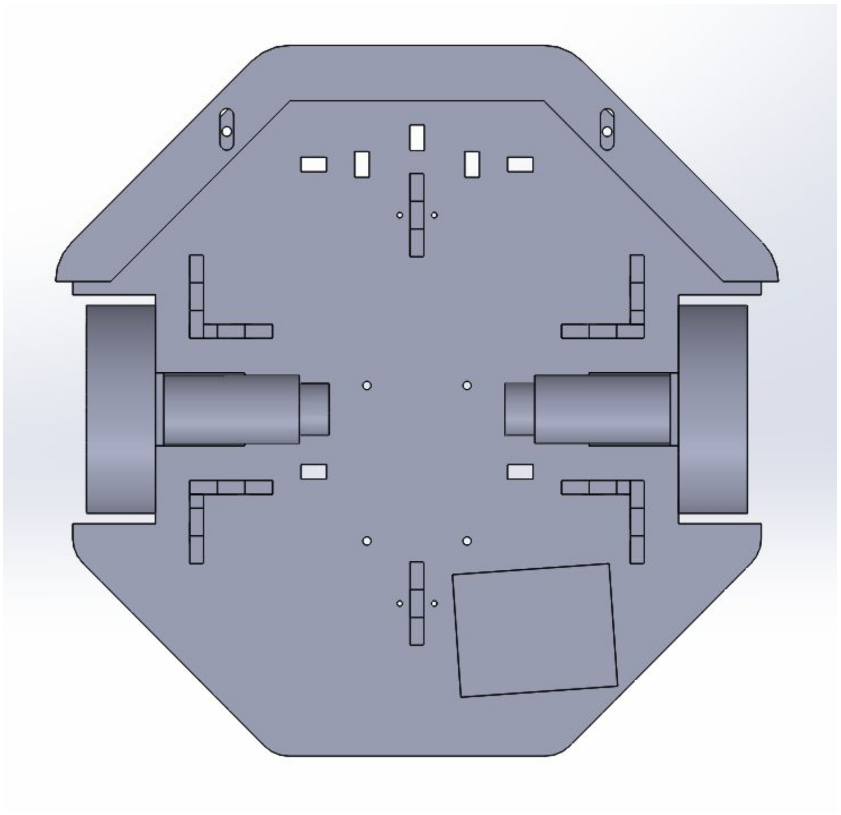


*Fig. 1: Bottom Platform Layout*

The bottom layout contained the movement and detection modules of our robot. We cut off a rectangle size groove from left and right side of the layout for placing the wheels. Since the wheels are surrounded by the bottom layout, they are protected and ensured their moving speed does not affect by hitting on the obstacles. Two motor mount brackets set on the central line of both end of grooves, the driving force ensured to power evenly for the platform. The H-bridge is placed in the back of the layout and the pin connector is facing outward of the layout, therefore, it is easier to manage the pin installation to the Uno32 and the power connection for the outside. For the detection modules, we have the tape sensors and bumper placed on the same layout. We designed a 7-tape senor detection, 5 sensors in the font and 2 in the back. The tape sensor circuit board is placed on the center of the layout to minimize the length of the sensor wires. This arrangement is also linked to the second layout for better wiring management with the Uno32 module. The bumper has similar shape as the front size of the platform, 3-side curve in 65 degrees which cover all hitting corner case of the robot. To avoid damage and scratches, we smooth both cover sides of the bumper, so the robot could slip through the obstacle. We added four L-shape supports toward the center of the robot, however, we found out that it did not provide enough support for the front and back of the robot since the final build is heavier than our estimation. We added two extra supports at the front and back on the central line to avoid sway while the robot is moving.

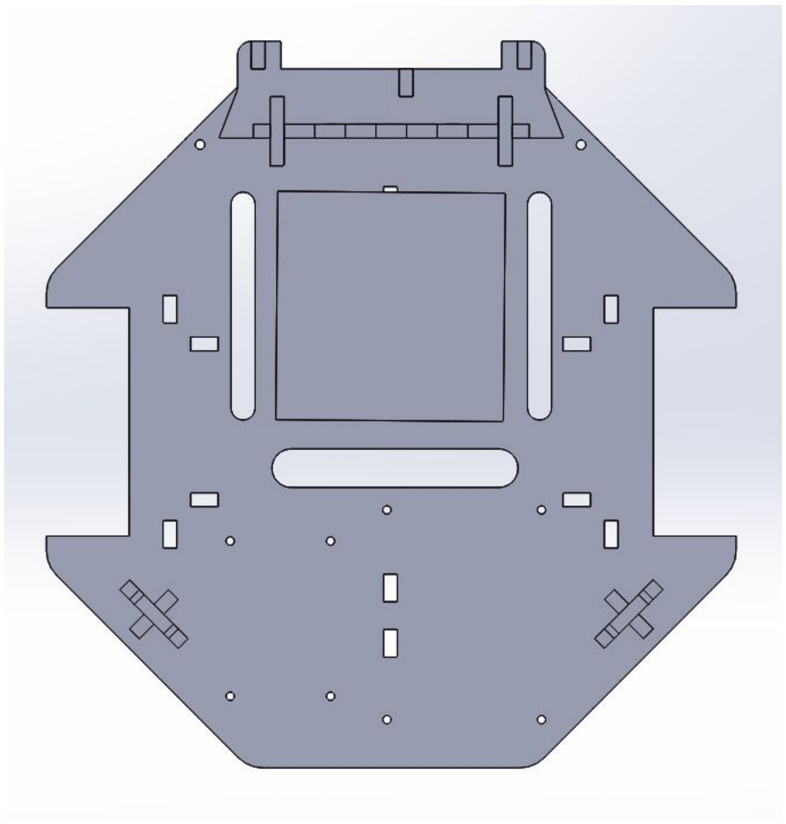


*Fig. 2: Center Platform Layout*

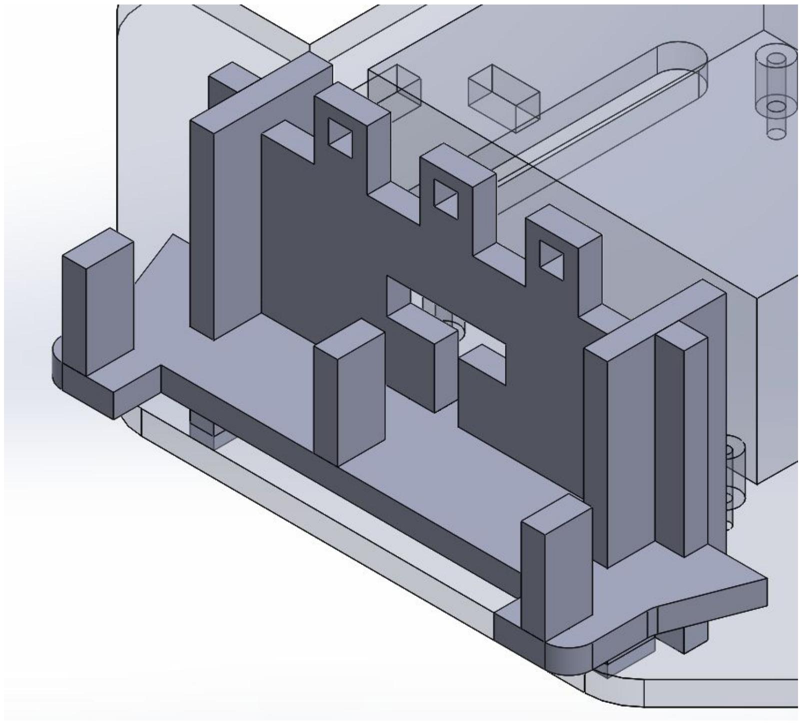


*Fig. 3: Front Support/Fiducial Sensor Mount*

Similar to the bottom layout, the center layout also has grooves for the wheels. The bottom layout contained all circuit board of the robot and holes for wiring management. The Uno32 is placed toward the front of the robot on the central line, with the connector banks on the left and right sides for wiring arrangement and power connection. The beacon board and the track wire board are placed on the back of the platform, located 2 inches away from the Uno32 to provide space for wiring. Rather than the support layout from the bottom layout, we designed to use two cross legs in the back and one stand for the fiducial hole sensors in the front, with the three supports positioned in a triangle shape for stability.  For wiring management, we added 3 slots around the Uno32, so the connector pins from the tape sensor and H-bridge board could be run from the bottom to the Uno32 and the power cables could be run through the side-holes of the layout.

The fiducial sensor holder is designed to hold three limit switches in position to detect the fiducial holes on the Ren Ship.  The side switches are held with their tips 1 inch in front of the robot, with the center one 0.4 inches back to account for the depth of the fiducial holes.  The holder is shaped to position the fiducial sensors near the inner edges of the fiducial holes, with the holders themselves being narrow enough to not get stuck in the holes.

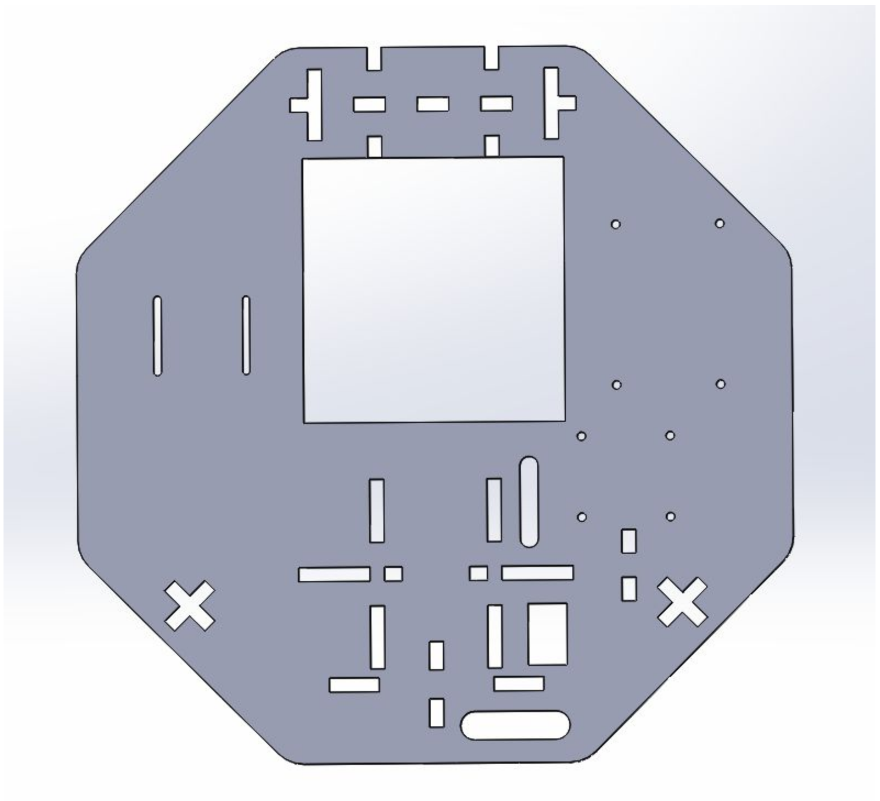


*Fig. 4: Top-layout Platform*

The top layout contained the shooting mechanism and the battery. To maximize our space, we cut off the section above the Uno32 on the top layout, therefore, we could reach the Uno32 for rewiring pins and reset button. Also, the wire jumpers could stretch out in between platforms and avoided bending and damaging the jumper head. To easily remove the battery for charging, the battery is placed on the left-hand side of the layout and held by a piece of velcro through the two parallel slots. The bumper and stepper boards are placed 2 inches away on the right-hand side of the layout, and 2 1.5-inch slots are located 2 inches away from the central line for wiring management. Wires and jumpers from the bumper board could reach through the holes down to the Uno32.

During implementation, we adjusted the position of the circuit board on the platform and resized holes for connection. We replaced the H-bridge to the bottom layout instead of the center layout to shorter the length of wires to reach the motor. We also repositioned all the broad that all power connectors are facing toward the center of the platform, and power jumpers could go through wiring holes on the platform to the Uno32. We had the cutoff section on the third layout for the Uno32, so the middle section has enough spacing for wiring to go through. The bottom section needed to be extended upward to rearrange the jumper connection. We resized both support arms for the platform, since the jumper's heads need to have half inches spacing to connected on the pin path on the board, we adjusted the bottom support arms to 1.8 inches long and 1 inch for the center support arms. We added names under the section for each circuit board, and laser cut them for visual recognition.

One of the challenge is that our robot did not turn precisely on the corner on the field because our wheels did not have enough friction while driving on a smooth surface for the DMF. We fixed this problem by covered a thin layout of hot glue on the wheels, so the wheel surface has enough friction to mount on the field. We also found out that our bumper was too short that it would not hit on the obstacle since we added the fiducial arm after the bumper design. We had an extra inch to rearrange the size of the bumper and we measured the distance from the bumper to the obstacle and Ren-ship, we glued two extra layouts of MDF on the front of the bumper to extended it by half inch farer. During installing our circuit board on the robot, we accidentally shorted the circuit on the Uno32 and burned the fuse for 5 times. We measured the input power to the tapes or regulators on each broad by the multimeter, however, we directly probing the power and ground pin and shorted the circuit with 9.9V flow back to the Uno32 broad. It also happened when we tried to test the continuity on the tape sensor where we probed both power and ground pin. It is important to make sure that we test pin connections under continuity mode on the multimeter and never probe the power and ground all the same time while powering the circuit.

## Tape Sensors

The tape sensor layout was designed with precision maneuvering in mind. There were seven sensors, positioned as such:

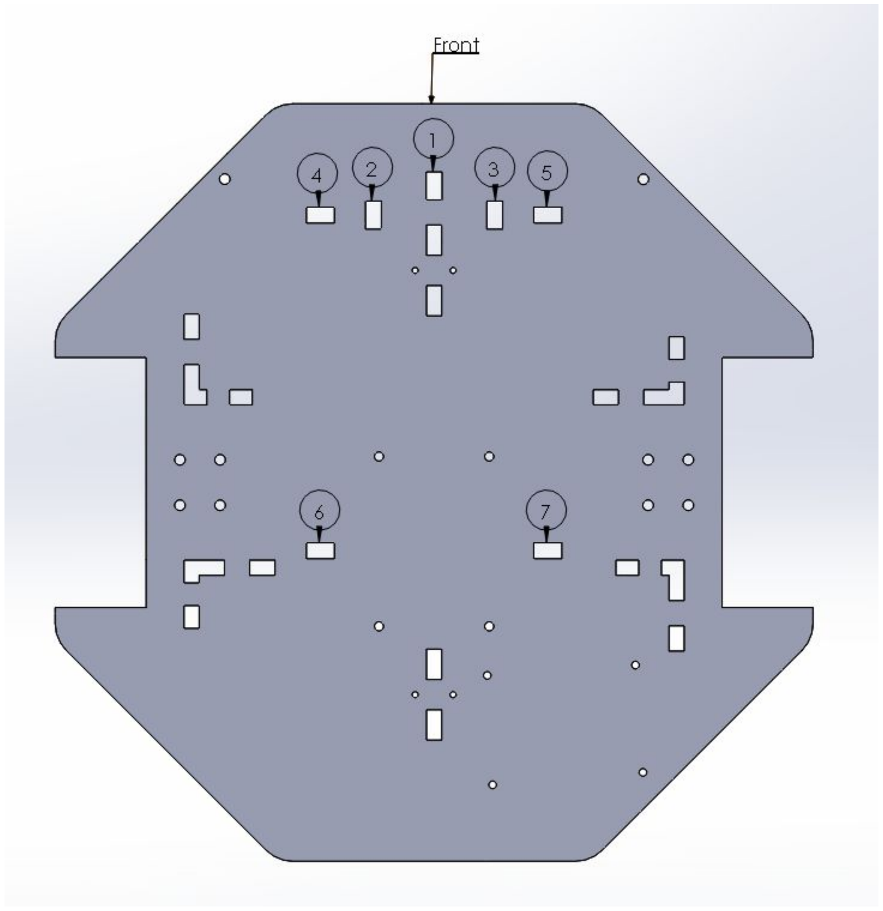


*Fig. 5: Tape sensor layout*

This would allow for very precise line following using the front-center trio of sensors:

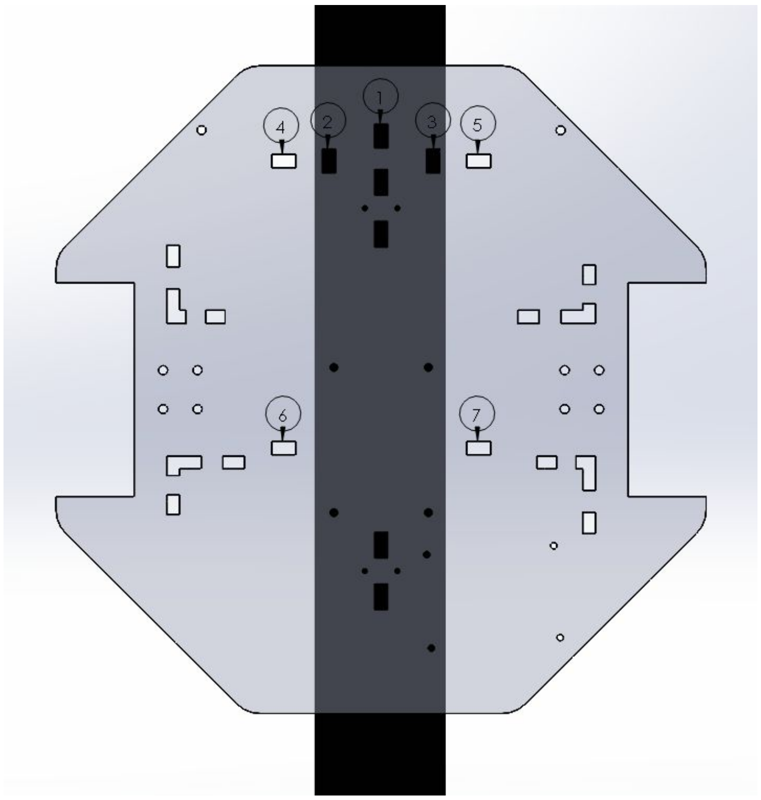


*Fig. 6: Line following*

Early detection of corners:

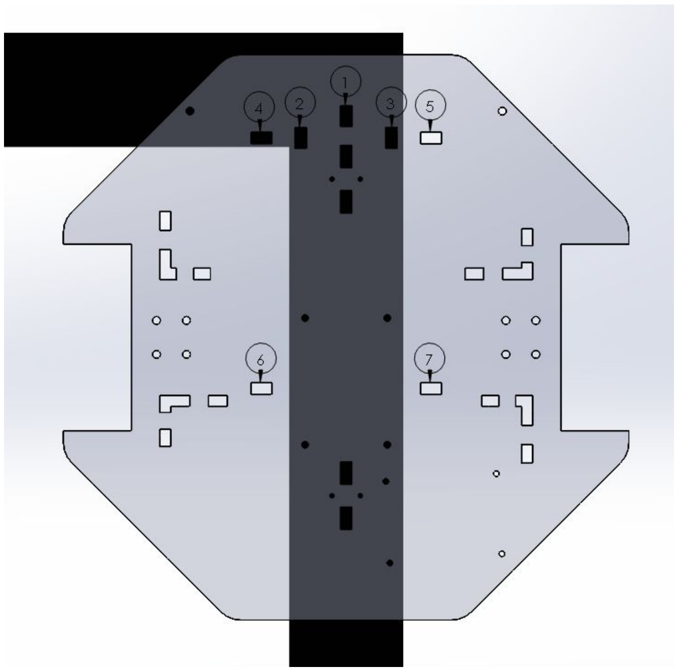


*Fig. 7: Corner detection*

Aligning the robot exactly on the corners, so as to get the robot as close as possible to corner targets:

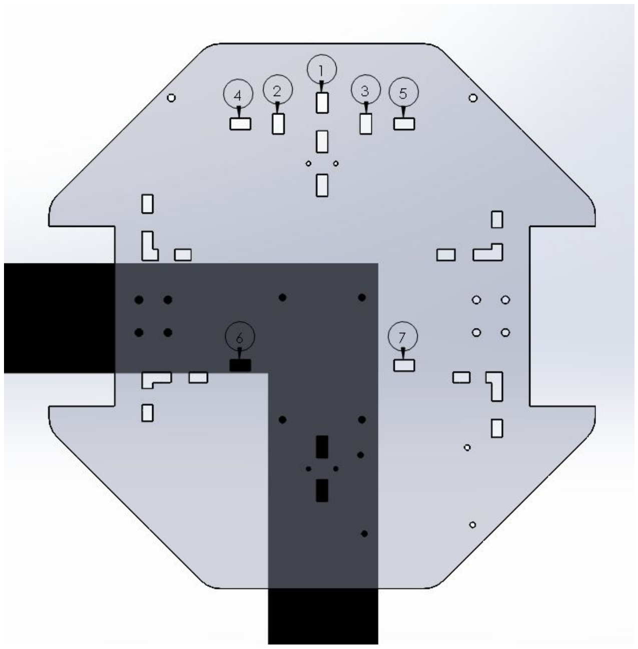


*Fig. 8: Corner alignment*

And precise alignment on the T of tape facing the Ren Ship:

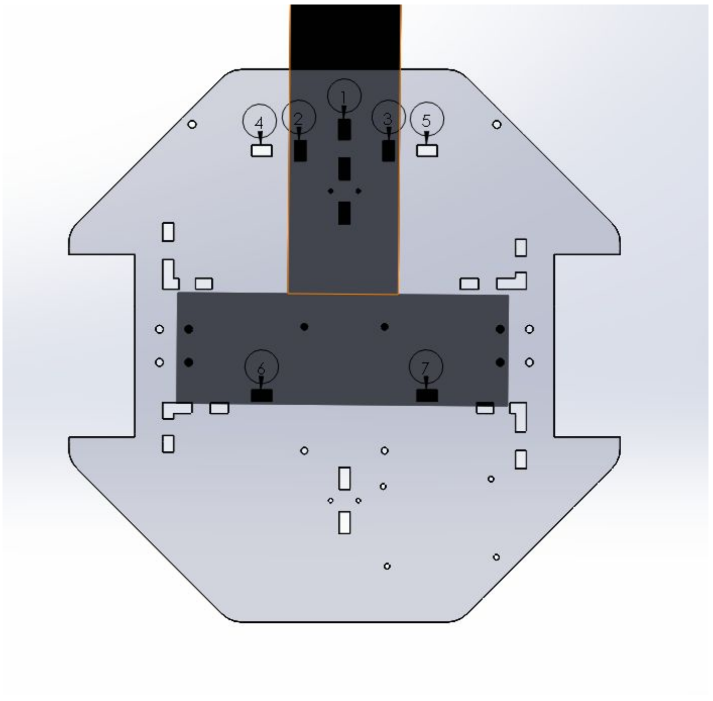


*Fig. 9: T alignment*

This is the only position on the field in which this particular combination of sensors on and off tape is possible.

## Ball Delivery

The design of the ball delivery mechanism started with the arm. It was to be mounted on a servo 10.5” off the ground, carry one ball, and pivot to deliver that ball exactly to the Ren Ship’s 16” high hole. The end of the arm was a square “bucket” carrying the ball. It was shaped so that when the arm was in the raised position, the “floor” of the bucket would be at a 10-degree downward angle, letting the ball roll out easily. When not in use, the arm could tilt backwards until all of it was below the 11” height limit.

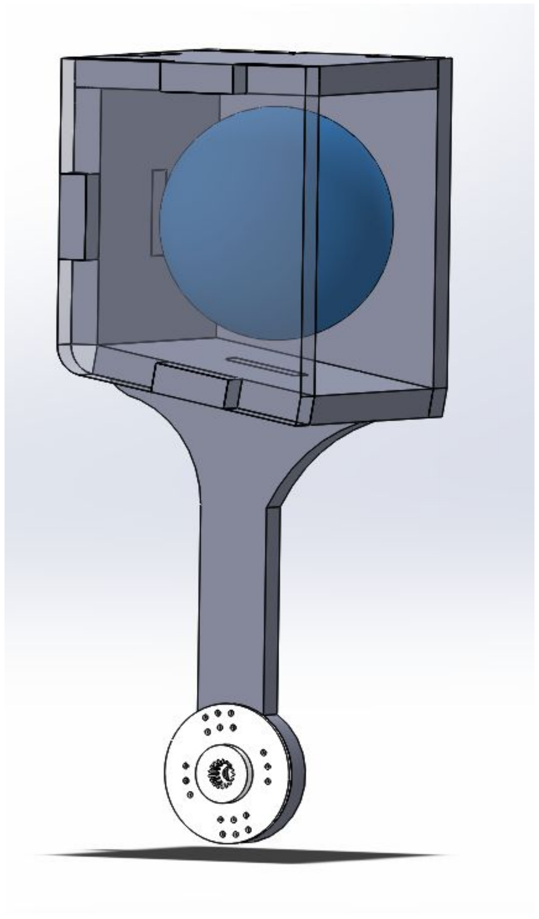


*Fig. 10: Arm design*

For the ATM6 targets, we used a sideways-facing wheel-driven ball launcher. A 5-ball revolving carousel, turned by a stepper motor, dropped balls into the “barrel”. A ramp rolled the falling ball into a motor-and-wheel assembly from a ping-pong ball launcher. The wheel used by this assembly was made of soft rubber, allowing for better gripping of the ball. Finally, another, shallower ramp was placed at the end of the barrel to give the balls extra lift as they left.

Ball in carousel:

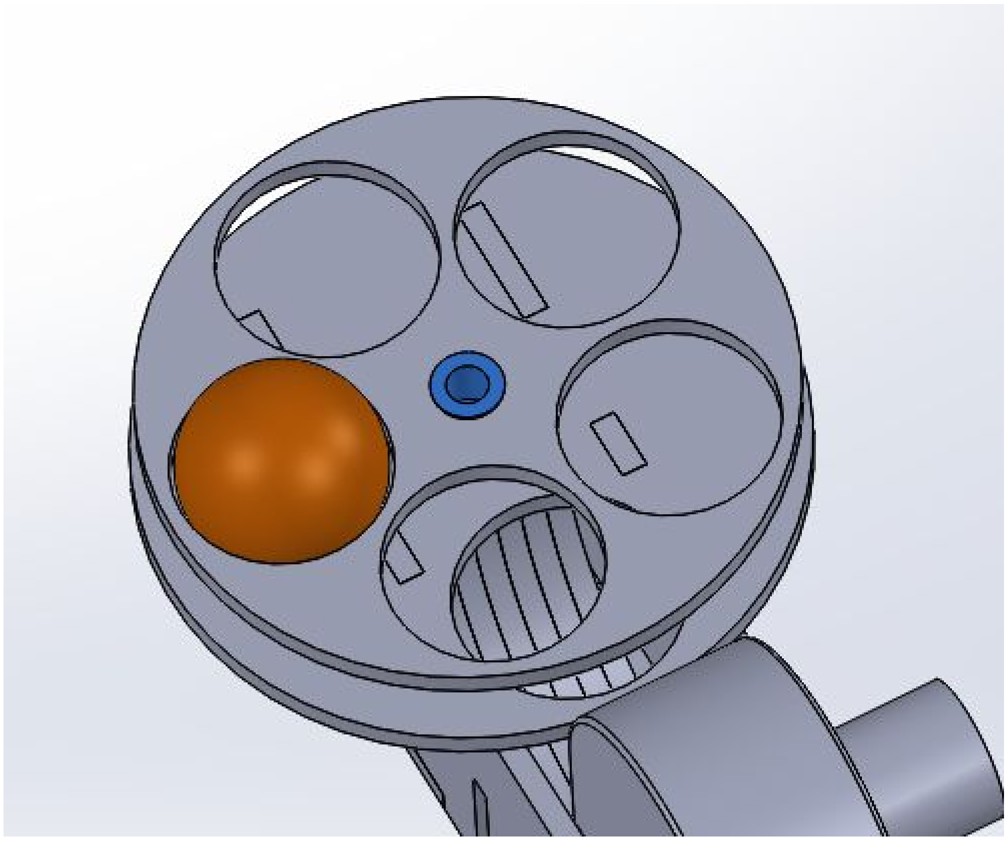


*Fig. 11: Launcher carousel with ball*

Ball in launcher:

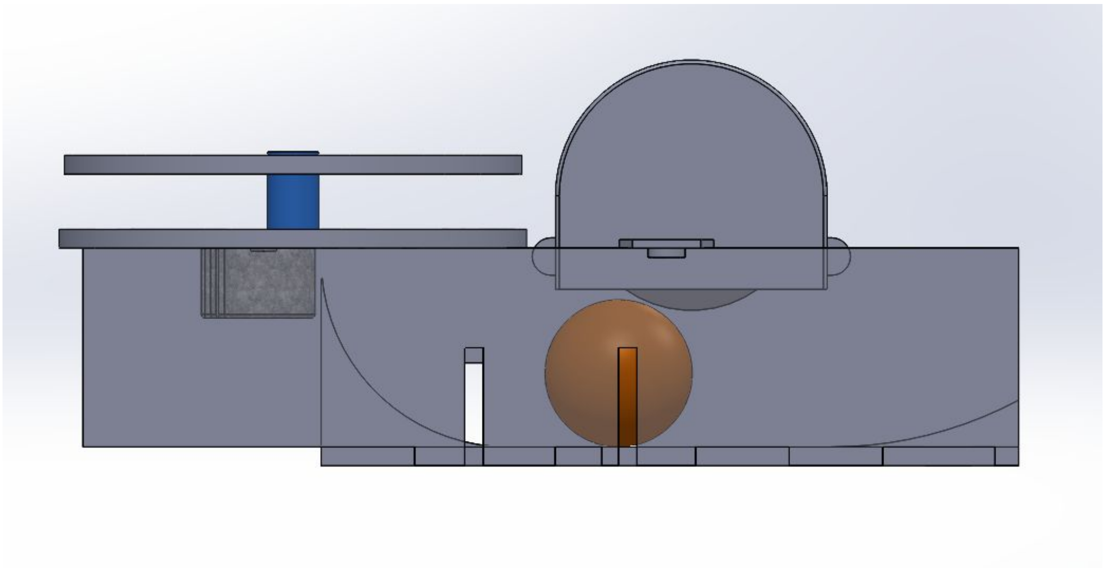


*Fig. 12: Ball in launcher.  Ramps and stepper motor (left) are visible.*

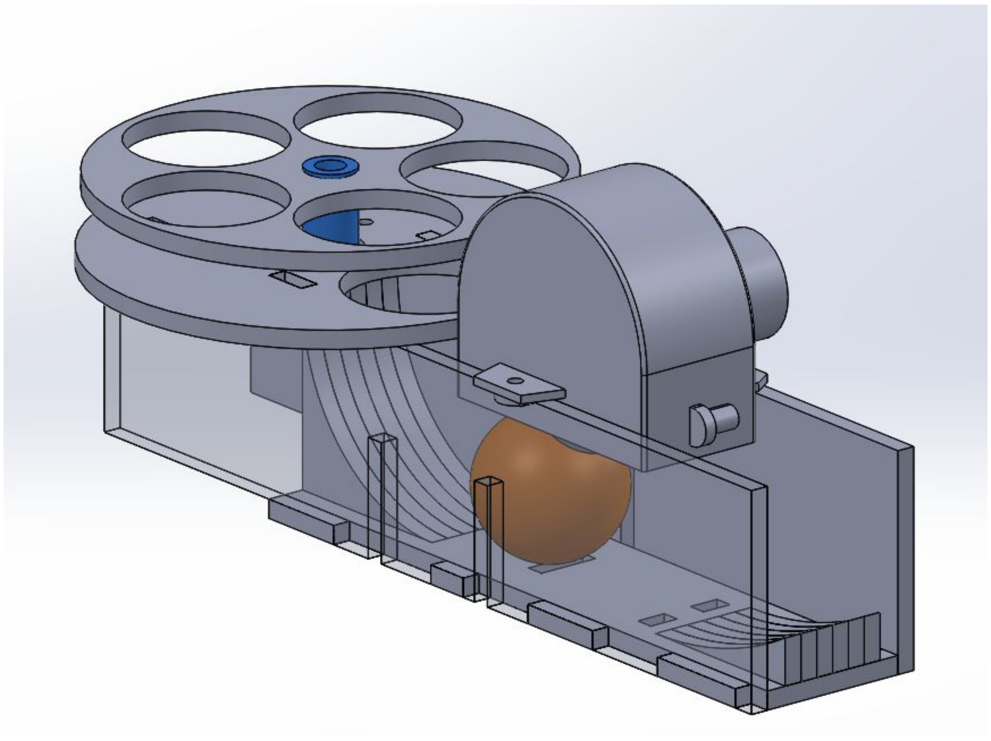


*Fig. 13: Dimetric view of ball in launcher.  Ramps are clearly shown.*

With the arm’s position on the robot decided by the need to deliver balls directly to the Ren Ship, we used a test assembly to position the launcher where the arm would not interfere with it. The position we found had the launcher mounted towards the back of the robot, with the “floor” of the barrel precisely 6 inches off the ground.

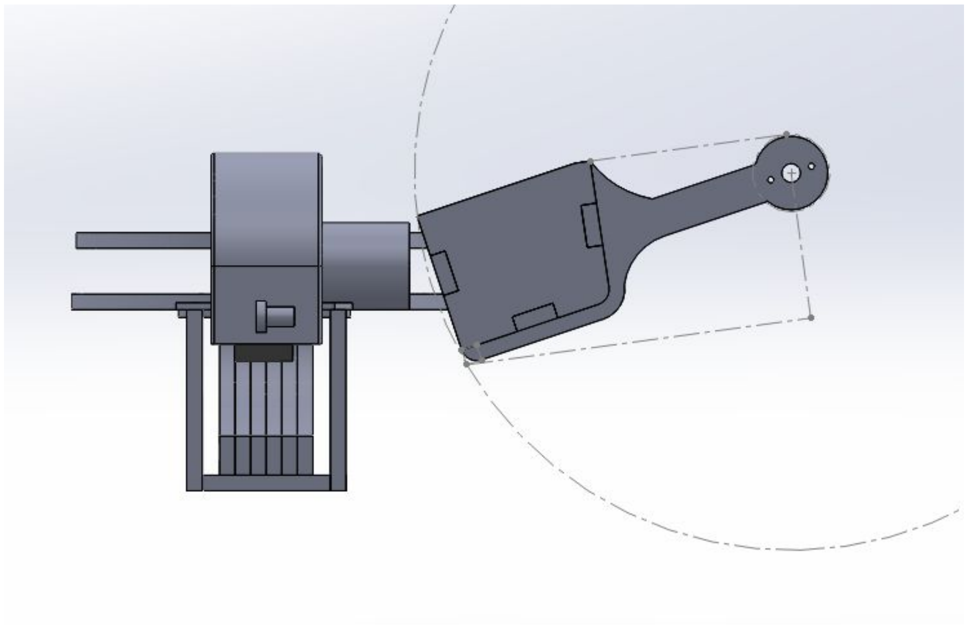


*Fig. 14: Arm and launcher sideways view.  Note sketch circle denoting sweep of arm*

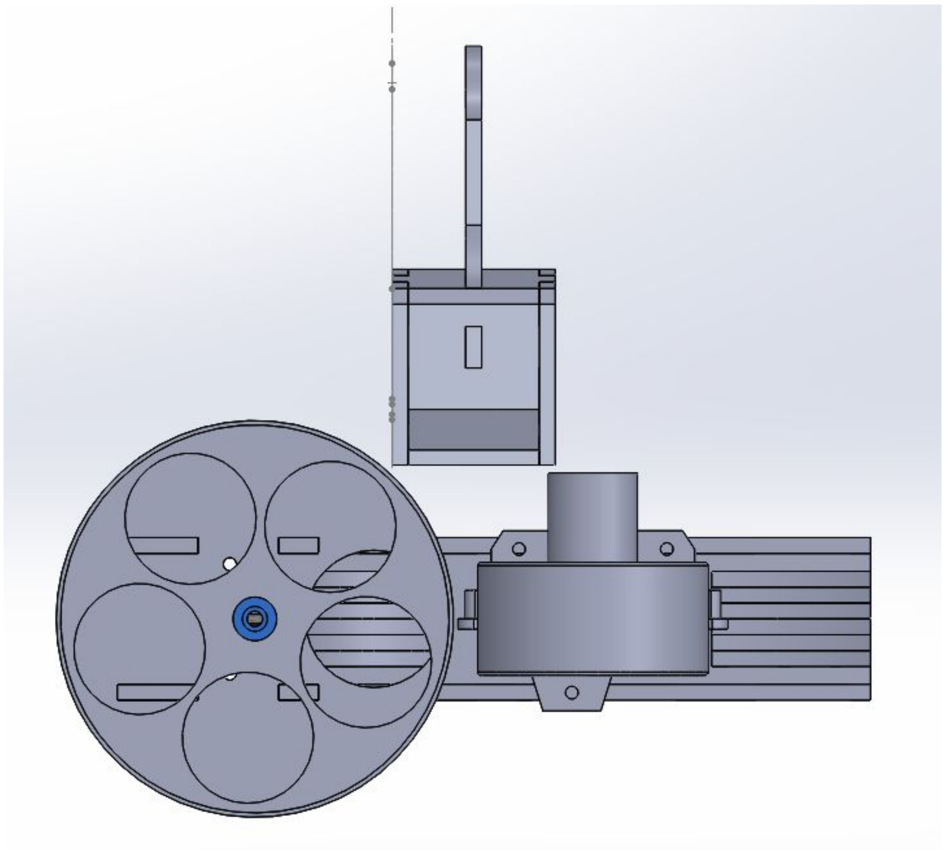


*Fig. 15: Arm and launcher top view.  Note gaps between arm and launcher*

After the base and ball delivery mechanisms were designed, a support structure was designed to connect the two.

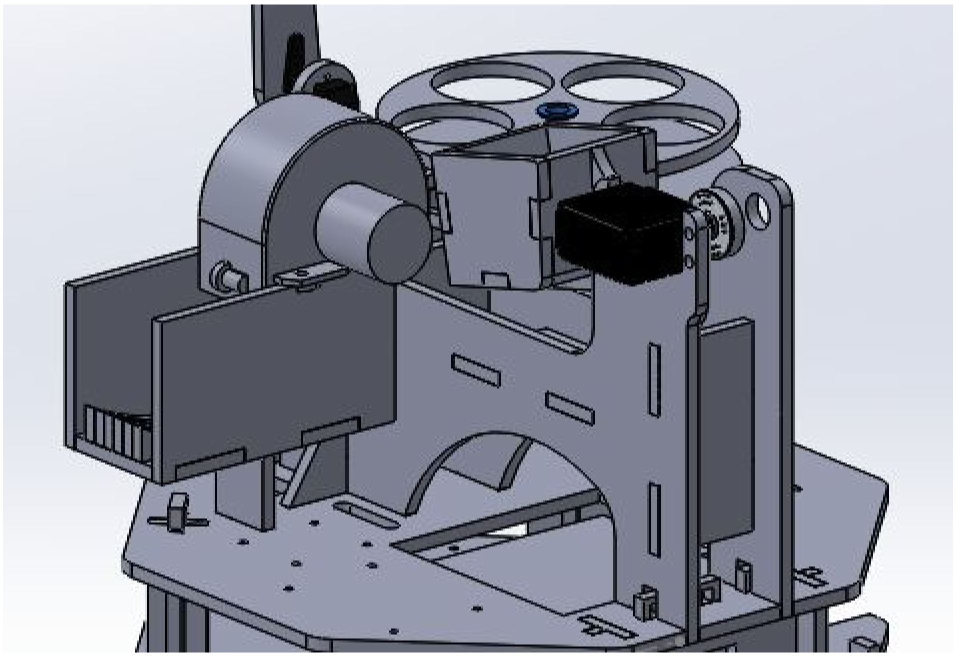


*Fig. 16: Ball delivery systems on completed robot*

### Phototransistor Holder

In addition to the ball delivery systems, a rotating “tail” arm on a servo was added to the robot to hold the phototransistor for the beacon detector circuit.  This arm would fit in the 11” cube when pivoted down and to the side, but would hold the phototransistor at the beacon’s 18” height when rotated upright.  Since we ended up not using the beacon detector, the “tail” and its mount were left off the robot.

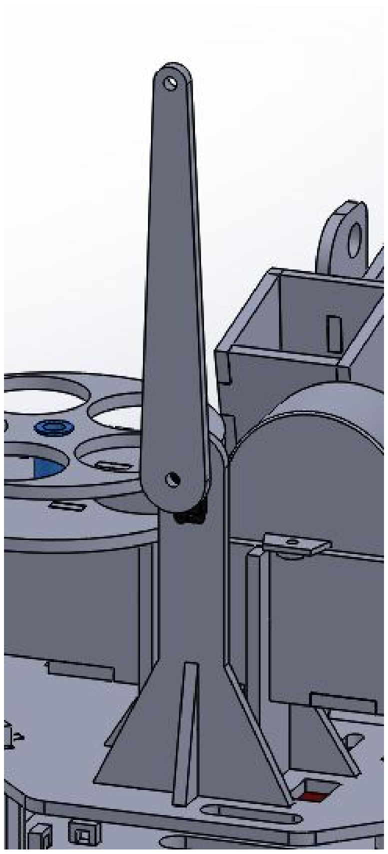


*Fig. 17: Phototransistor “tail” in upright position*

## Electronics

### Track Wire Detector

In order to detect the AT-M6 targets, we need to be able to sense an oscillating magnetic field. As with Lab 1, a tank circuit with an inductor is used to detect and amplify this signal. The circuit consists of three gain stages followed by a peak detector, in order to feed a smooth DC value into the ADC. The sinusoidal signal is amplified around a bias voltage of 1.65 volts, since we found the op-amps to behave in interesting ways when trying to rail them too far below 0 (operating them in single-ended mode). The first two stages provide a gain of 15 each, followed by a variable stage utilizing a potentiometer for fine-tuning.

The output at the end of the gain stages is a strong sine wave, which was tuned with the potentiometer to provide a 3.3V maximum (1.65V amplitude centered at 1.65) when the inductor was about two inches from the trackwire. This provided a good starting range for where the sensitivity of the circuit should be, as with normal operation of the robot we expect the solenoid to be about this distance from the targets.

Lastly, the peak detector components were chosen based upon what we found to be successful during previous labs.

To provide power, a 3.3v regulator was used.

The circuit:

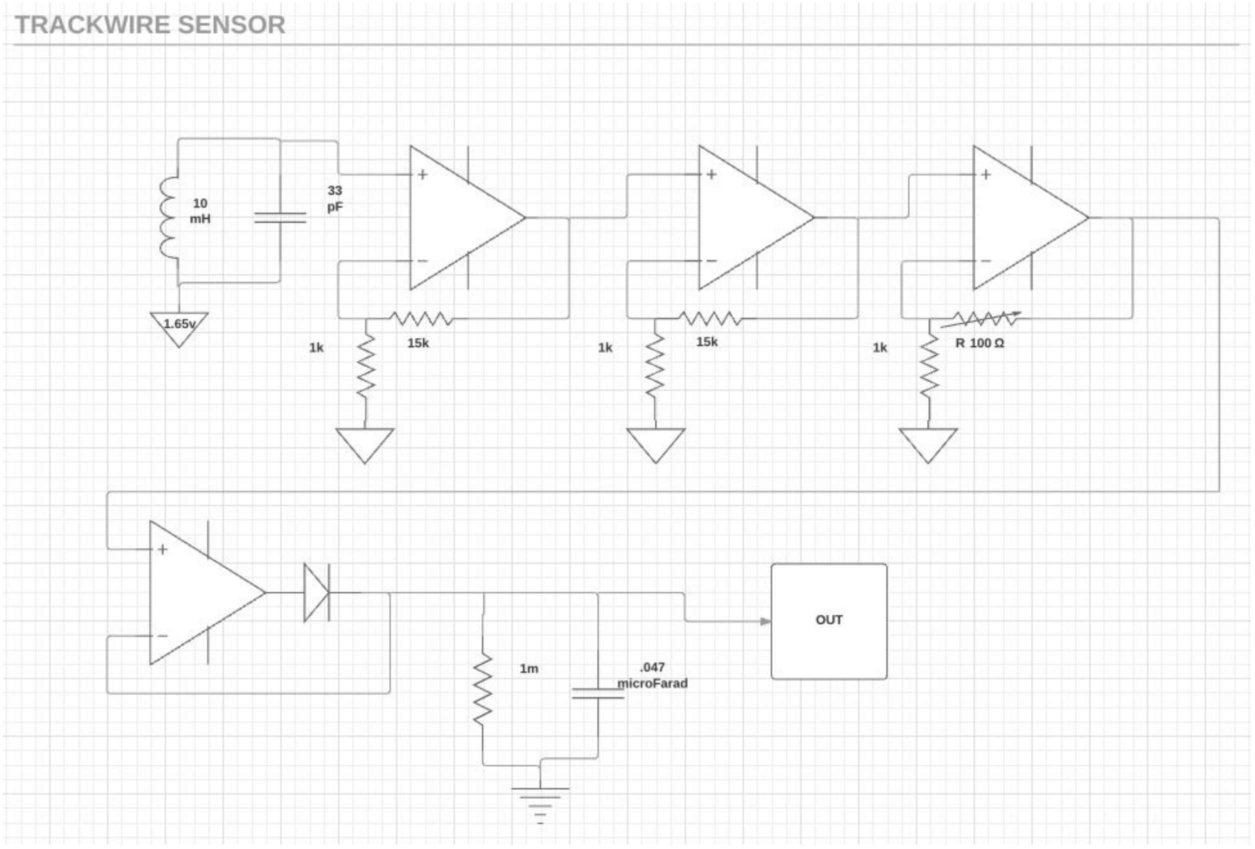


*Fig. 18: Trackwire Sensor Design*

The board:

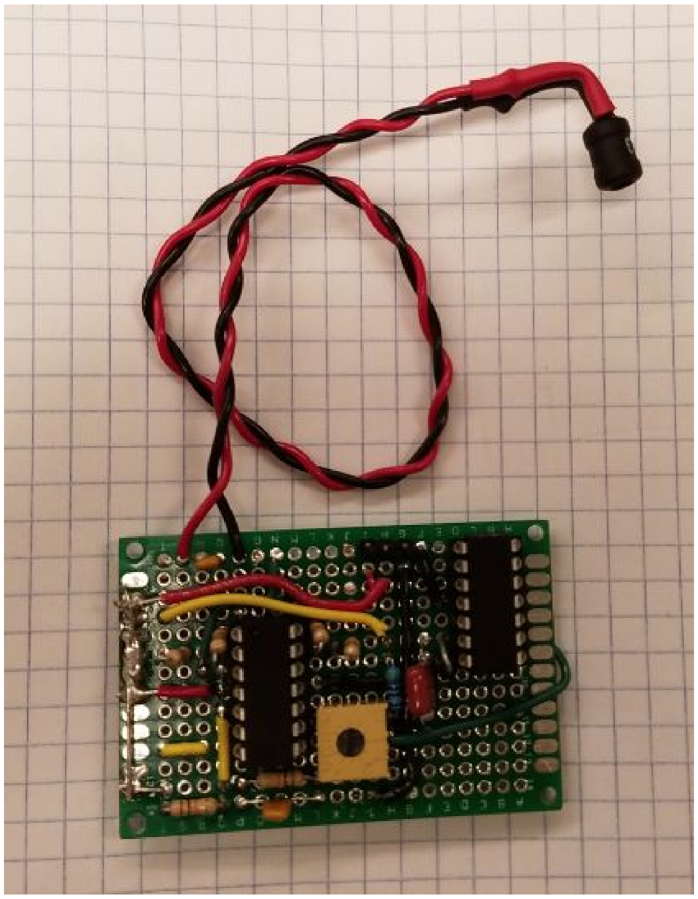


*Fig. 19: Trackwire Broad*

## Tape Sensors

After it was decided to use 7 tape sensors, the task at hand was to drive them all with a single pin, and to be able to read each individual phototransistor. It was to our advantage to run them all from a single voltage, so as to minimize the amount of wiring necessary to run every optical sensor.

The goal was to run 25 mA through each of the Vishay TCRT5000's IR LEDs, with each having a common ground at the collector of a TIP122. The LED's are expected to have a voltage drop of about 1.25 volts, so using 3.3 as the supply voltage was no problem. After running experiments to determine the collector-emitter voltage of the TIP, we were able to wire a tidy circuit which achieved our purposes. The chosen resistors allow a proper amount of current to flow through, and the same rail is used to read the phototransistor side in a sourcing configuration.

A major problem with the circuit manifested when we realized there was very strange noise in our tape sensor readings. The culprit: small capacitors on the regulator! This was a helpful learning experience about noise and how it propagates through the system; without a large enough capacitor to smooth the regulator output, the circuit was receiving very high frequency noise and generating faulty readings. This was fixed by swapping the capacitors. After this, we beefed up the regulator capacitors on all the rest of the boards.

The Circuit:

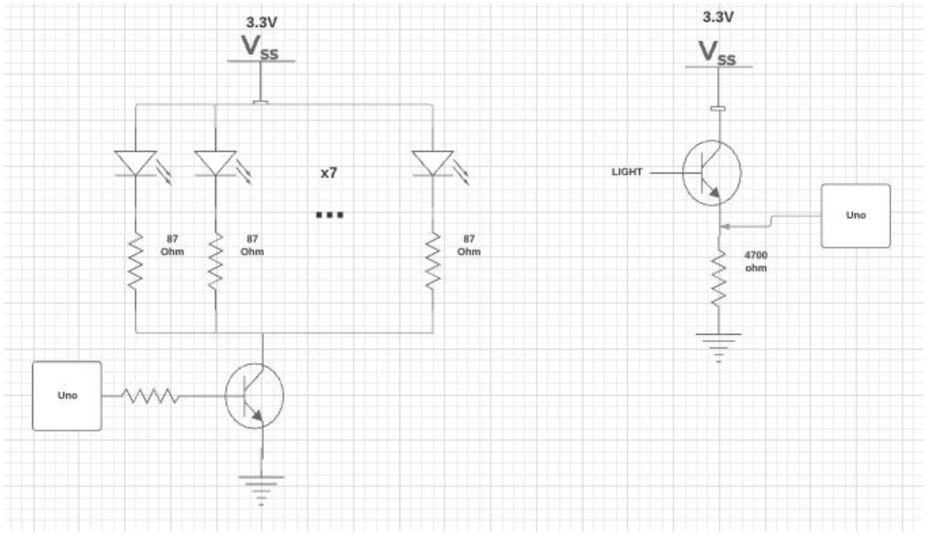


*Fig. 20: Tape Sensor Design*

The Board:

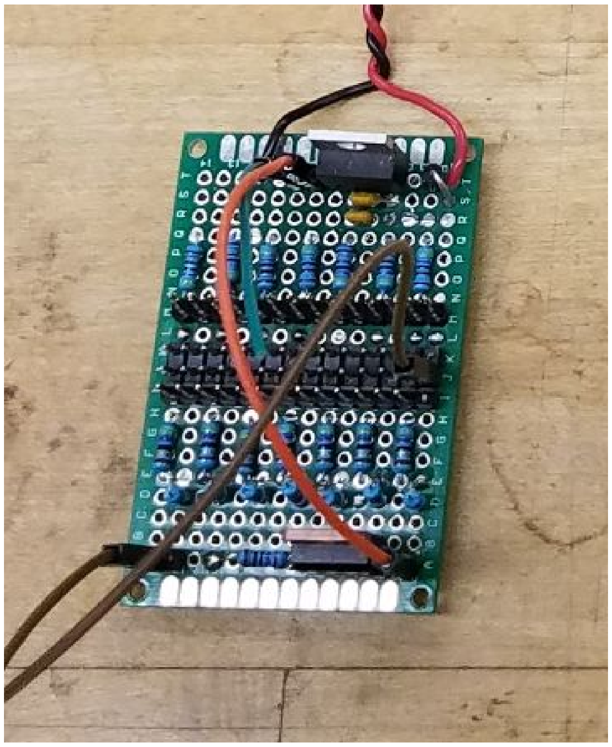
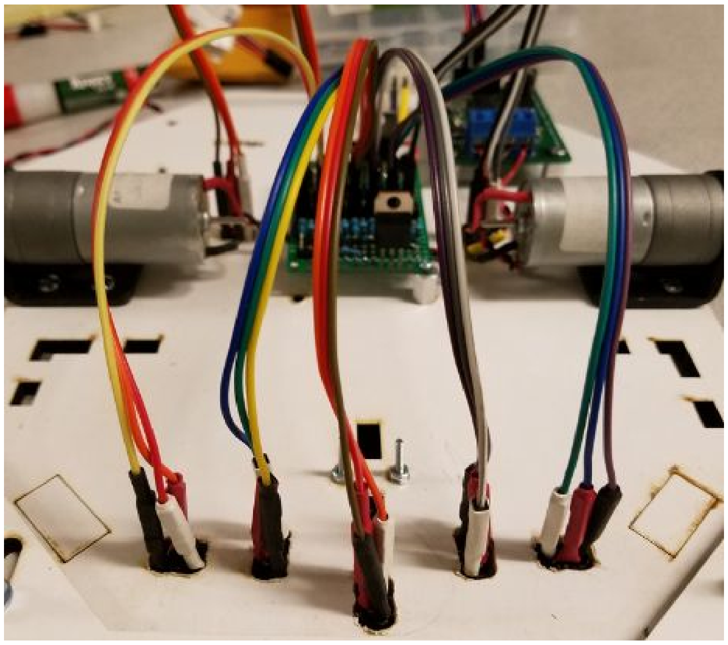


*Fig. 21: Tape Sensor Board*

Attached to this board are the seven sensors; we soldered each Vishay unit to a set of three jumpers, sharing a power rail and providing TIP ground and signal wires. This wiring scheme allowed the layout to be quite tidy:



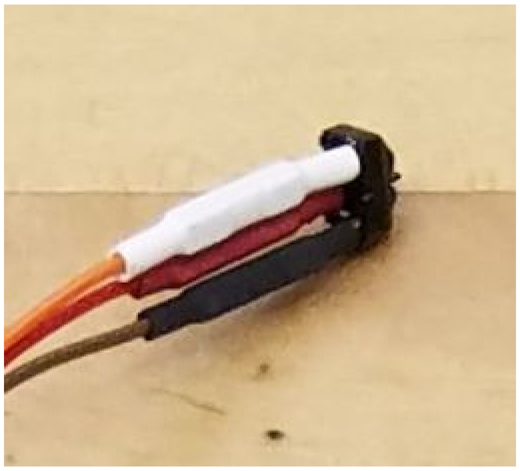


*Fig. 22: Tape Sensor Wire*

## Bumpers/Fiducials and Servo Board

In order to properly read all of our limit switches, we designed a board which would consolidate the switches, provide a pull-down resistor to each, and send the output to the uno. A pull-down resistor was necessary in order to keep the pins from floating when the switches were disconnected, ensuring that the uno would see true ground when the switches were open. When the switches are closed, 3.3 volts appear across the digital pins.

This board was also a convenient place to install a 5 volt regulator in order to power the servos, as well as consolidate their signal pins. We made sure to use large enough capacitors

The circuit:

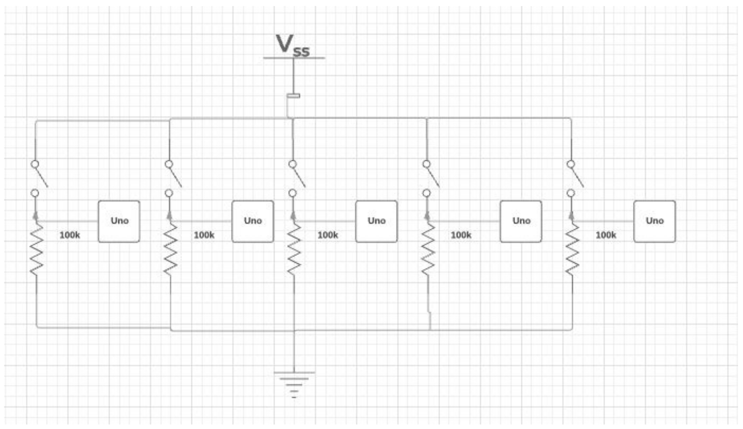


*Fig. 23: Bumper Board Design*

The board:

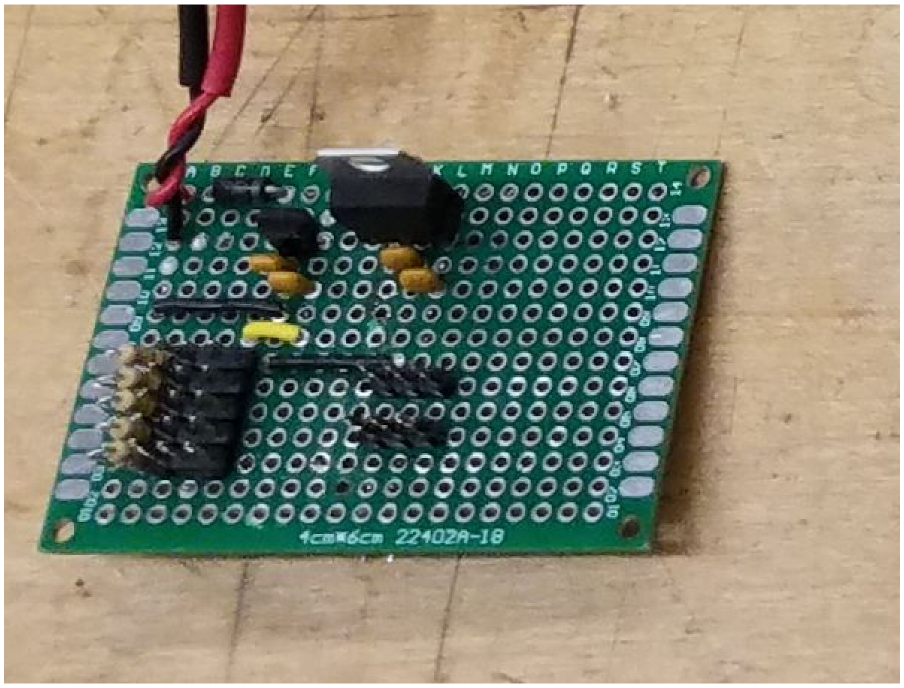


*Fig. 24: Bumper Board*

## Beacon Detector

Initially, we planned to use the beacon detector in order to orient ourselves during the first few seconds of the match. As detailed above, the phototransistor would have been placed on a servo-controlled arm at the rear of the robot. We were armed with a well-functioning detector from the previous labs, and the board is installed onto our robot. However, it came to light that there are ways to orient ourselves without the beacon, and thus have opted not to use it.

## Ren Sensor

During the course of our design, it became apparent that it would be to our advantage to be able to differentiate between hitting the normal obstacle and hitting the ren ship. This allows different maneuvers for different targets. As it turns out, the a tape sensor works wonderfully for distinguishing the two; an off-the-shelf detector was added for this purpose, with the sensitivity knob adjusted for operating within range. The sensor was installed as follows in the top right corner of the top layer:

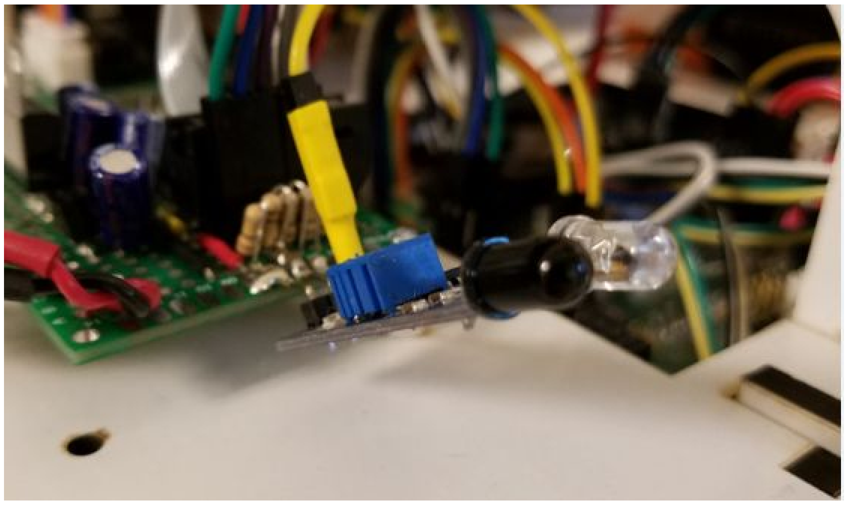


*Fig. 25: Photosensor Board*

The usages for this sensor are detailed within the state machines described further below.

## Launcher Driver Board

In order to drive the launcher motor, we created a driver board using a TIP122. This allows us to run the motor at full battery voltage and PWM it down to a reasonable speed. The simple circuit is as follows:

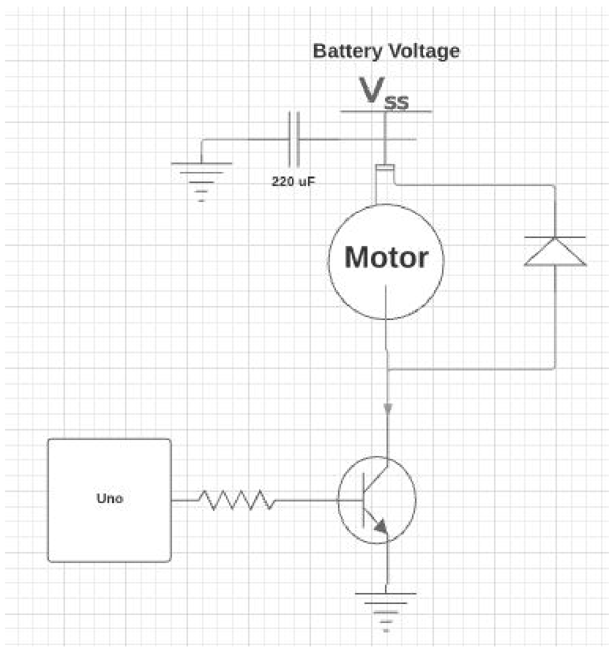


*Fig. 26: Motor Board Design*

This setup was inspired by the motors lab, with a snubbing diode and smoothing capacitor to deal with inductive spikes and noise.

The Board:

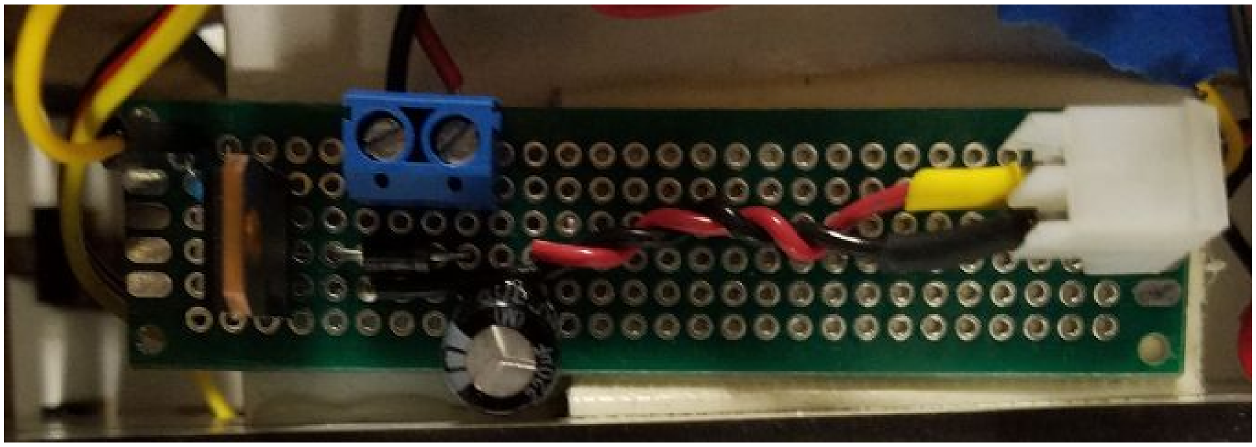


*Fig. 27: Motor Board*

# Software

## Sensor and Actuator Drivers

### Tape Sensor Driver

The tape sensor checker began life as a service. This service implemented synchronous sampling using a simple four-state state machine: it would first turn the tape sensor LEDs off, then take a set of samples, then turn the tape sensor LEDs on, then take another set of samples, then repeat. Every time the LEDs-off samples were taken, they were stored in a 7x1 array - one sample for each tape sensor. When the LEDs-on samples were taken, the LEDs-off samples for the corresponding sensors were subtracted to get the differences, which were stored in a 7x8 ring buffer. This stored the last 8 values of each sensor for use in an 8-point running average filter. The averages were then converted into sensor on/sensor off values using hysteresis.

Rather than storing the tape state as an enum, the states of all seven sensors were stored as a 7-bit bitfield, with 1 indicating a sensor being on the tape and 0 indicating a sensor being off the tape. Whenever one or more tape sensor changed status, the old bitfield was shifted left by seven bits and combined with the new bitfield. The combined 14-bit bitfield was then sent to the state machine as the event parameter of a TAPE_SENSOR_EVENT.

The state machine contained helper functions that would take a 14-bit tape event parameter and return a boolean value indicating whether the parameter indicated a certain event had occurred. For example, this helper function was one of the first ones written:

```
int driftingLeft(int tapeParam)
{
    int returnVal = 0;

    int newParam = tapeParam & (TAPE1_NEW_BIT | TAPE2_NEW_BIT | TAPE3_NEW_BIT);
    int oldParam = tapeParam & (TAPE1_OLD_BIT | TAPE2_OLD_BIT | TAPE3_OLD_BIT);

    if ((newParam == (TAPE1_NEW_BIT | TAPE3_NEW_BIT))
            && (oldParam == (TAPE1_OLD_BIT | TAPE2_OLD_BIT | TAPE3_OLD_BIT))) {
        returnVal = 1;
    }
#ifdef HELPERFUNCTION_TEST
    printf("driftingLeft: %i\r\n", returnVal);
#endif
```

```
    return returnVal;
}
```

When passed a tape parameter, this function will return 1 if and only if sensors 1 and 3 are currently on tape and sensor 2 is off tape, when sensors 1, 2 and 3 were all on before. This function ignores the status of sensors 4-7.

Unfortunately, this service turned out to have a very nasty bug. In certain positions, a sensor's state would "flicker", rapidly transitioning between reading as on tape and reading as off tape. This was the cause of multiple problems in our early state machine. We tried to solve the issue by adding debug output, multiple state machine redesigns, adding more debug output, reworking the helper functions, disassembling the robot to test the sensors, giving each tape sensor individual hystereis thresholds to compensate for differences between sensors, turning the tape sensor service into an event checker without synchronous sampling, and adding even more debug output.

After several days of this, including multiple times when we thought we had fixed the bug only for it to finally crop up again, we finally found the culprit: a single erroneous line of hysteresis code meant that the sensor values were not actually being evaluated with hysteresis. Instead, the sensors read as "on tape" when the ADC value was below the low hysteresis threshold, and "off tape" when the ADC value was above the *low* - rather than high - hysteresis threshold. While we had checked the outputs of the running average before, we had not thought to do so only when a state change occurred, and thus missed the running average output "bouncing" around the low hysteresis threshold when the flicker occurred. Once we found the problem, the fix took 30 seconds.

### Bump Sensor Driver

The limit switches for the bumpers and fiducial sensors were debounced using the exact same ring-buffer method as the tape sensors. The only differences were that there were 5 sensors instead of 7, and the service kept a running sum of the last 8 samples instead of a running average. If a limit switch was pressed less than 4 times during the 8 samples, it was considered pressed. If pressed more than 4 times, it was considered unpressed. If exactly 4 times, it was considered to have not changed state.

The bump sensor driver was literally copy-pasted from the tape sensor driver, and thus also had the hysteresis bug, although since it operated on binary inputs the effect was negligible and did not result in "flicker". However, it was still fixed when the tape sensor driver was.

### Motor Driver

The motor drivers provided us the interface required to interact with our primary actuators, the DC motors. The core functionality for this driver was to provide a way to set independent speeds for both the left and right motors, around which all other motor functionality (driving forward/backward, tank turns) was based. The speed control utilized PWM, via the provided

libraries. In order to keep the motor speeds constant even when the battery begins sagging, the duty cycles were scaled to the battery voltage, in order that the motors would always see the same average DC voltage through the square waves. To achieve this, a MAXIMUM_TARGET_VOLTAGE must be defined, and it must be below the battery's typical operating range. To provide sufficient headroom, the driver will never place an effective DC voltage greater than 8.5 across the motors. The battery voltage is available via the AD.c module.
The scaling algorithm is as follows:
(Desired Speed / 100) * MAX_TARGET_VOLTAGE / BATTERY VOLTAGE = newDutyCycle
Where desired speed is between 0 and 100. This creates a duty cycle which will always evaluate to the same DC value, since when the battery voltage drops, the duty cycle raises proportionately.
We found it valuable to have the ability to pause the motors without losing the speed value previously written to them; that is, a pause() command can be issued to the motors which stops them. When resume() is subsequently called, the motors return to their previous states. This was achieved by keeping the left and right speeds as a module-level variable, which the setting functions can interact with. The motors also have a PAUSED flag, and calls to any of the functions will only set the motors if this flag is not set.
The motors have proven to be the largest cause of noise in the system. Even with the provided H-Bridge performing inductive snubbing, the motor functionality still interfered with other components (namely, the trackwire sensor). This is to be expected, given the widely fluctuating magnetic fields generated by motors in constant change. The mitigation if this noise is discussed below.

### Trackwire Driver

The trackwire driver required some finesse, as the motors wreaked havoc with their irregular magnetic fields. Since we opted to skip hardware filtering on the sensor, software is where we were able to knock out our noise. It became clear that the highest magnitude noise came from the motors starting or stopping (when changing direction, not necessarily the spikes during each PWM cycle) as DC motors generate large spikes due to their very nature. The understanding of this we gained during the motors lab was quite useful here.
To start, the motors were probed for the frequency and general waveform of their noise. Many frequencies were present; unfortunately, one of those frequencies was the 25 kHz we were looking for in the trackwire.
Ultimately, our solution was to use the driver to stop the motors and allow them to settle before taking a trackwire reading. Every 200 ms, the motors are paused and a timer is set for 10ms. When the timer expires, a reading is taken and the motors are resumed using the pause functionality described above. We had to play around with this number in order to find the right settling time for the motors; anything less than 10 and we would see false positive events. This method sacrifices some torque, but we ultimately saw no negative effects. It also sacrifices sampling rate; however, for our purposes this method has worked flawlessly.

With the ability to take clean readings, the trackwire detection becomes a simple matter of software peak detection. The data is first run through a moving average filter to smooth out any ADC noise. Then, each filtered sample is compared with a static variable, highestSeen, which keeps track of the highest sample seen during data collection. If the incoming sample is higher, it becomes the new highestSeen.
When this variable is found to be past a certain threshold, the driver moves into the NEAR_TRACKWIRE state, and posts an event indicating such. At this time, we know that we are approaching the center of a target. In this state, the incoming data is then compared with the highest value; if the data starts dropping and is LESS than the highest seen, we know we have reached a peak. A TRACKWIRE_PEAK event is then posted to the main state machine. Finally, after seeing a peak, each incoming sample is compared to the NEAR threshold. If found to be less, we know that the target has shut off, indicating a kill; the LOST_TRACKWIRE event is posted and the state machine resumes hunting its targets.

## Other Actuators: Servo and Feeder

The servo and feeder mechanisms (the feeder being run by a stepper motor) were controlled in software by essentially creating wrappers for the provided libraries to provide functionality specific to our robot. For example, the servos.h module provided functions to raise and lower our servos to specific angles, with functions such as servos_extendFront() to place the actuators exactly where we needed them to be. The values necessary for our specific movement were determined experimentally and left as #defines within the modules. We are left with a servo module that abstracts us from the required pulse time of a servo and simply actuates the system with a single function call.
The only wrinkle here was that the feeder stepper we chose is a unipolar motor, and thus requires slightly different driving patterns as compared to the bipolar motors seen in previous labs. Whereas the bipolar motors need enable and direction commands, the unipolar motor simply needs to have each of its coils connected to ground in sequence. An example modification is in the state machine for a fullStepDrive, indicating that we simply sink current through two of the coils at a time:

```
void FullStepDrive(void)
{
    switch (coilState) {
    case step_one:
        //sink the first two phases
        STEPPER_PIN1 = 1;
        STEPPER_PIN2 = 1;
        STEPPER_PIN3 = 0;
        STEPPER_PIN4 = 0;
        if (stepDir == FORWARD) {
            coilState = step_two;
        } else {
            coilState = step_four;
        }
        break;
```

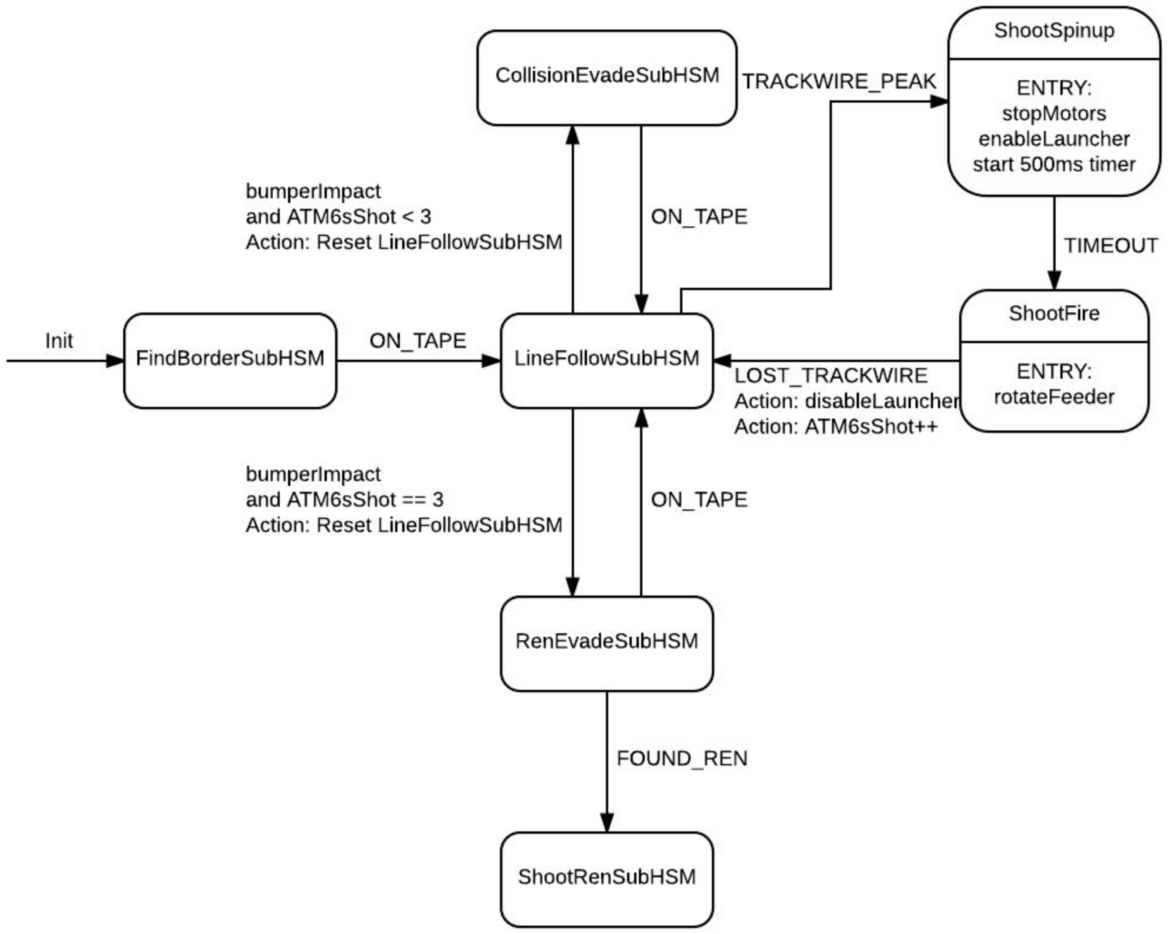
Each of the four pins was connected as a digital output, and connected to the handy driver board included with our stepper.
After making these modifications, abstracting this movement into a motors_rotateFeeder() command was simply a matter of determining how many steps per revolution, and dividing it by 5 in order to move our feeder enough to load one ball into the cannon.

During testing, it came to light that due to an oversight, the feeder mechanism would only rotate during every other function call, due to the motor stopping in a state which wouldn't respond to further commands. Resetting the IO pins upon the completion of the rotation fixed this problem.

## State Machine

### The Original State Machine

The initial state machine was the one the tape sensor layout was designed for. It was a complex setup using all seven tape sensors, designed to handle almost every edge case we could think of. The high-level HSM was as follows:



ShootSpinup
ENTRY: stopMotors enableLauncher start 500ms timer
CollisionEvadeSubHSM
TRACKWIRE_PEAK
bumperImpact and ATM6sShot < 3 Action: Reset LineFollowSubHSM
ON_TAPE
TIMEOUT
ShootFire
ENTRY: rotateFeeder
Init
FindBorderSubHSM
ON_TAPE
LineFollowSubHSM
LOST_TRACKWIRE Action: disableLauncher Action: ATM6sShot++
bumperImpact and ATM6sShot == 3 Action: Reset LineFollowSubHSM
ON_TAPE
RenEvadeSubHSM
FOUND_REN
ShootRenSubHSM

*Fig. 28: Complex HSM top level*

The only simple element here is the process for shooting ATM6s, which simply involves stopping the robot, spinning up the launcher, firing one ball and waiting for the trackwire to deactivate. If the robot could not reliably line itself up for a shot based on trackwire peak sensing alone, we could turn the ATM6 targeting into another sub-HSM.

You can also see a helper function here. bumperImpact() took a bumper parameter and returned 1 if and only if at least one limit switch was now pressed when none were before - in other words, if the robot has just impacted an object with either the bumper or the fiducial sensors.

The Find Border sub-HSM is as follows:

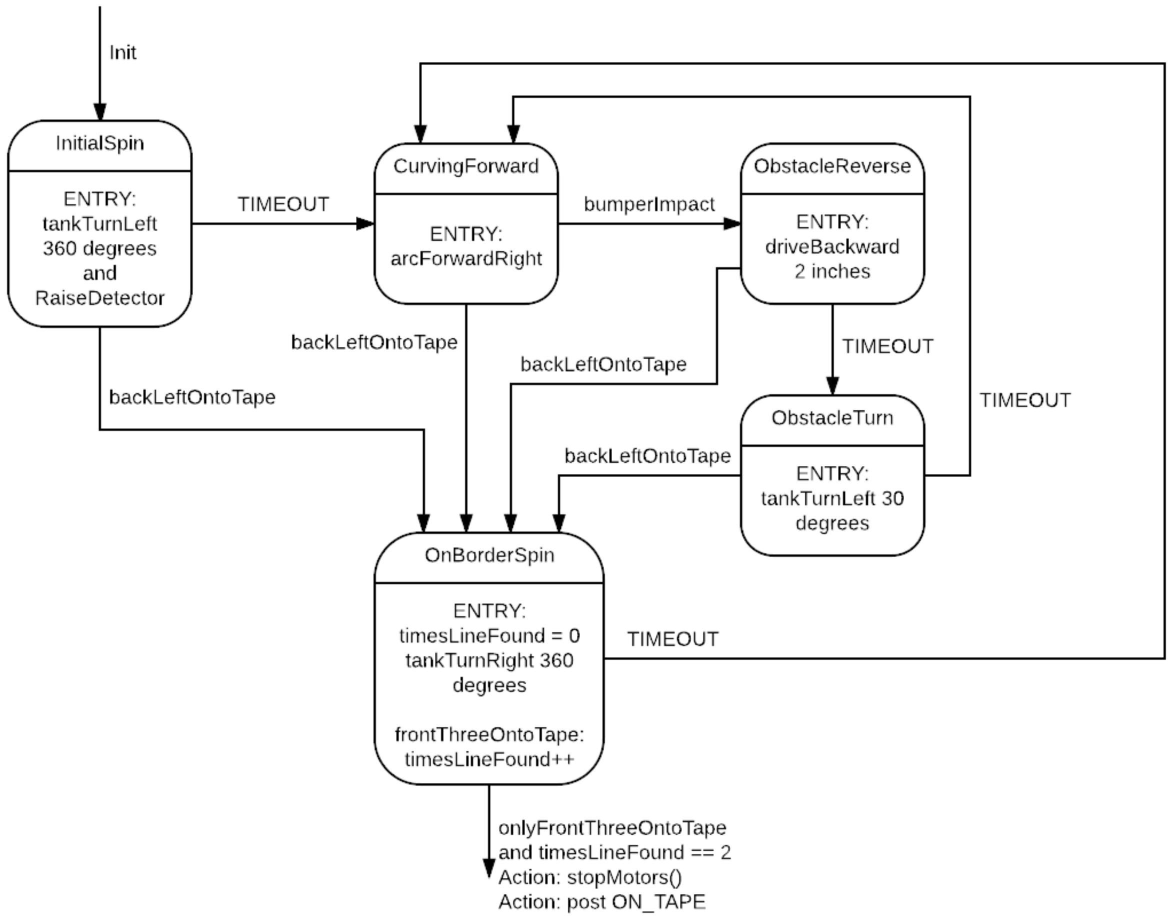

Init
InitialSpin
ENTRY: tankTurnLeft 360 degrees and RaiseDetector
TIMEOUT
CurvingForward
ENTRY: arcForwardRight
bumperImpact
ObstacleReverse
ENTRY: driveBackward 2 inches
TIMEOUT
ObstacleTurn
ENTRY: tankTurnLeft 30 degrees
TIMEOUT
backLeftOntoTape
backLeftOntoTape
backLeftOntoTape
backLeftOntoTape
OnBorderSpin
ENTRY: timesLineFound = 0 tankTurnRight 360 degrees
frontThreeOntoTape: timesLineFound++
TIMEOUT
onlyFrontThreeOntoTape and timesLineFound == 2
Action: stopMotors()
Action: post ON_TAPE

*Fig. 29: Complex Find Border sub-HSM*

This state machine was designed to get the robot onto the border in any and all cases. First, the robot spins 360 degrees to detect if it is already on tape. Then, it drives in a curving forward arc to the right until it hits tape or impacts an obstacle. In the event of an impact, it backs up slightly, turns left slightly and then drives in an arc to the right again, eventually working its way around the obstacle.

If at any point in this process tape sensor 6 passes onto tape (as determined by the helper function backLeftOntoTape()), the robot spins 360 degrees and looks for the front trio of sensors to pass onto the tape while sensors 4 and 5 are off tape (with onlyFrontThreeOntoTape()). It looks for this to occur twice within the spin, because unless the robot is on the field border, the front sensor trio cannot get onto the tape twice in one spin without sensors 4 or 5 being on the tape at least one of those times. This was determined by moving a foamcore cutout of the robot baseplate over the field and seeing what tape sensor patterns were possible in what positions.

While the combination of the right curve and the detection on sensor 6 were intended to position the robot on the border for a counterclockwise loop, it was possible in some edge cases to position the robot on the border facing the wrong way. That was, however, accounted for.

The Line Follow sub-HSM is as follows:

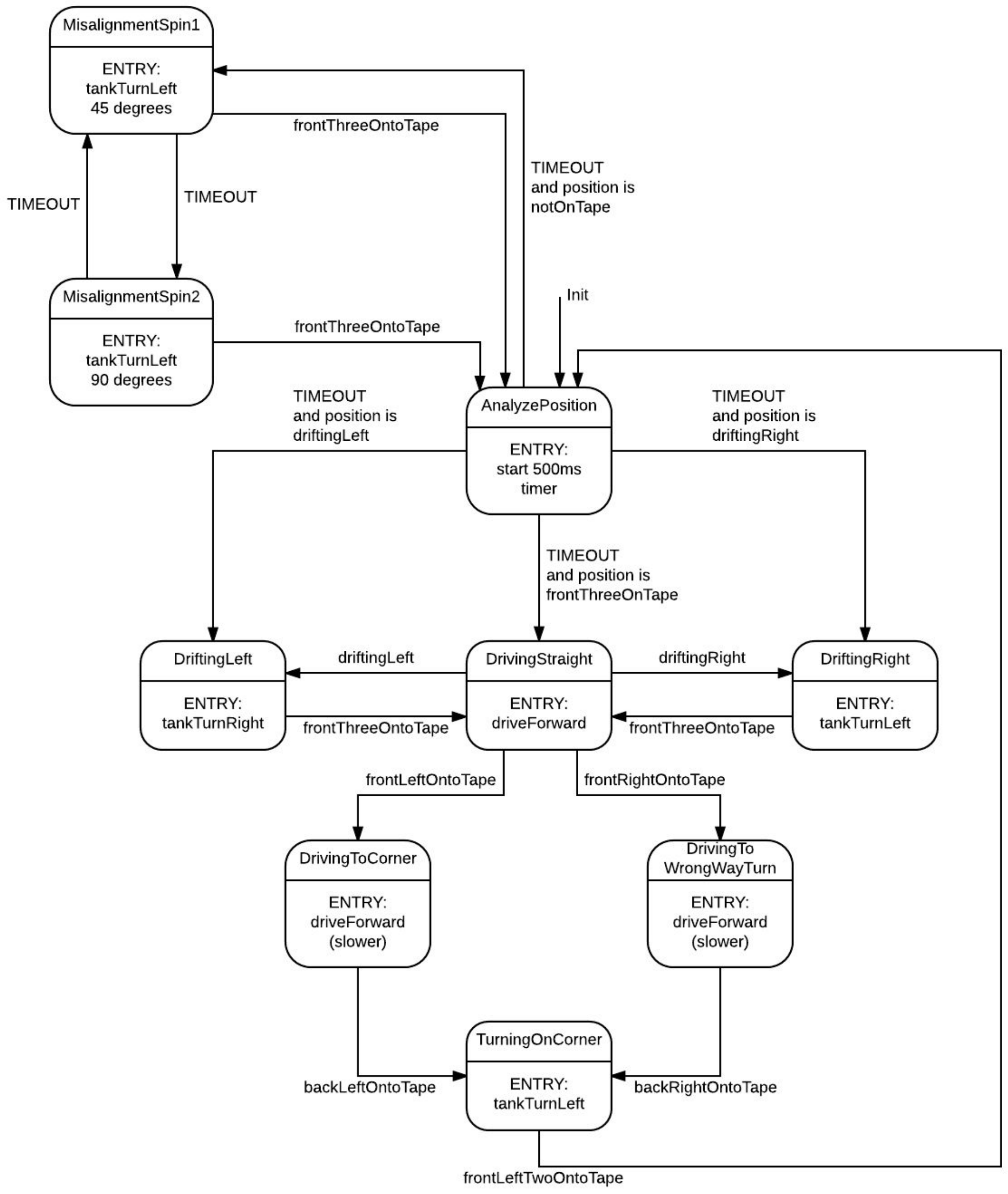


*Fig. 30: Complex Line Follow sub-HSM*

This sub-HSM starts by waiting 500ms for the robot to stop any previous motion, then calling a function, senseAlignment(), to read the status of the front trio of tape sensors.  If not all three sensors are on the tape, the robot corrects this.  If either one or none of the front trio are on the tape, the robot turns back and forth in place until it finds the tape.

The robot drives straight on the tape, immediately correcting any drift with a tank turn.  If sensor 4 or 5 hit tape, indicating an upcoming corner, the robot will slow down and wait for sensor 6 or 7 to hit tape, respectively, indicating that the robot is centered on the corner.  If sensor 4 hit tape, the robot is approaching a left turn, and will turn left at the corner.  If sensor 5 hit tape, the robot is approaching a right turn, meaning that it is driving the wrong way (with the launcher pointed towards the inside of the field.)  As such, the robot will turn around.  After a turn, the robot will realign itself and continue.

The Collision Evade sub-HSM is as follows:

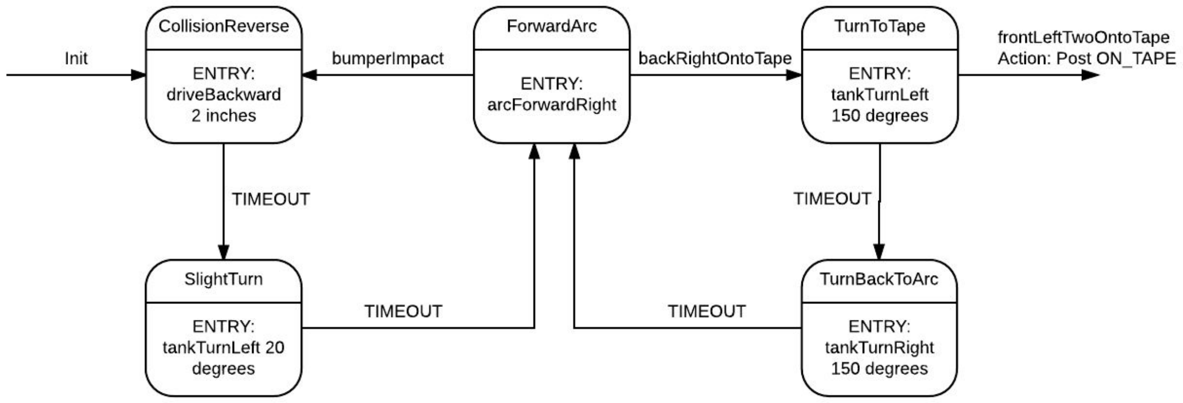


*Fig. 31:  Complex Collision Evade sub-HSM*

This is the simplest sub-HSM.  In response to a collision, the robot will back up 2 inches, tank turn left 30 degrees, then drive in an arc forward and to the right.  In this way, the robot will bump its way around an obstacle, following it closely.  When sensor 7 hits tape, the robot will turn left 150 degrees, looking for the line with the leftmost two sensors of its front trio.  While there is a position on an alignment mark where this may result in the robot lining up on the mark, the presence of an obstacle prevents the robot from reaching that position.  If the robot does not find the tape in its 150-degree turn, it turns the 150 degrees back and keeps driving.

The Ren Collision Evade sub-HSM is as follows:

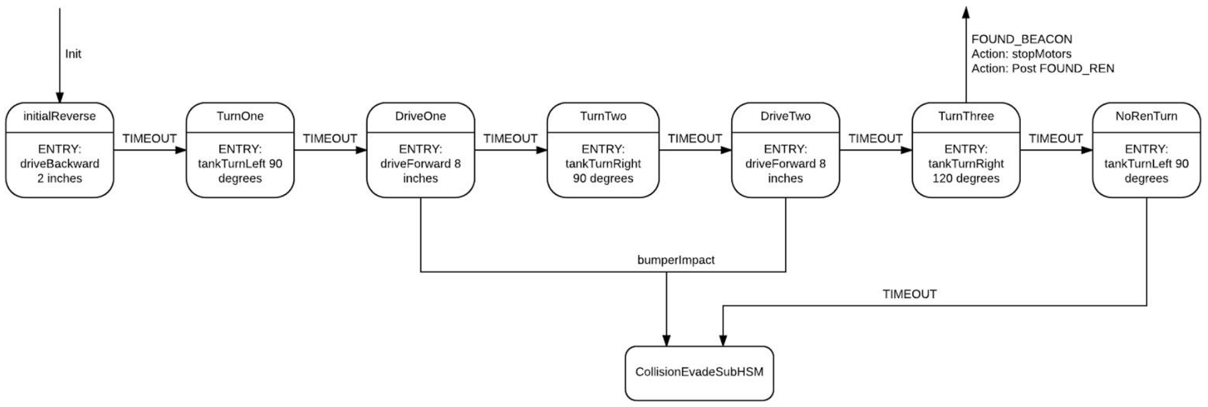

Init
initialReverse
ENTRY: driveBackward 2 inches
TIMEOUT
TurnOne
ENTRY: tankTurnLeft 90 degrees
TIMEOUT
DriveOne
ENTRY: driveForward 8 inches
TIMEOUT
TurnTwo
ENTRY: tankTurnRight 90 degrees
TIMEOUT
DriveTwo
ENTRY: driveForward 8 inches
TIMEOUT
TurnThree
ENTRY: tankTurnRight 120 degrees
TIMEOUT
NoRenTurn
ENTRY: tankTurnLeft 90 degrees
FOUND_BEACON
Action: stopMotors
Action: Post FOUND_REN
bumperImpact
TIMEOUT
CollisionEvadeSubHSM

*Fig. 32: Complex Ren Collision Evade sub-HSM*

This is simply a sequence of straight lines and 90-degree turns which, if the robot bumped into the Ren Ship, should result in the robot positioning itself in front of the Ren Ship, then turning and looking for the beacon. If the beacon is not found or the robot impacts an obstacle, it goes back into regular collision avoidance, as it hit the obstacle instead of the Ren Ship.

This sub-HSM was the least developed at the time we stopped working on the complex HSM.

The Shoot Ren sub-HSM was intended to start from the position facing Ren, align on the T, then advance to the Ren Ship, confirm positioning with the fiducial sensors, and deliver the final ball. It was never diagrammed.

In addition to the sensor helper functions, the HSM was also equipped with a suite of motor helper functions. The HSM could request, for example, a 45 degree left turn at 80% PWM, and the helper function would set the motor speeds and directions using the motor driver functions, calculate the time needed to complete the maneuver, and set a timer to return when the maneuver completed.

Unfortunately, reality did not live up to the expectations for this state machine. The inertia of the heavy robot, combined with the motors we used, meant that the robot was simply incapable of the precision maneuvering this state machine required. In fact, it was incapable of any kind of precision maneuvering, regularly overshooting lines and turns by inches or tens of degrees. Even switching to a braking-capable H-bridge barely helped. Attempts were made to fix this. For example, the alignment portion of the line following sub-HSM was a later addition intended to deal with misalignment coming out of a maneuver. Likewise, the robot often listened for only the two sensors of the front trio that would hit the line first when turning to the line, in an attempt to minimize overshoot. It also slowed down when approaching corners, in an attempt to minimize overshoot there Unfortunately, it didn't work. The robot still overshot corners and turns, and had trouble getting on the line. senseAlignment() itself turned out to have a bug that inverted its readings, which took roughly a day to track down. We were not making progress fast enough.

While the complex HSM may have been able to work on our hardware given time and effort, we simply did not have the time. When it became clear that completing the complex HSM would be more effort than starting over, we started over.

## The Simplified State Machine

After beginning on a fresh sheet of paper for the new state machine, we came up with the following state machine. While it leaves more edge cases open, it integrates better with the nature of our hardware, and allows edge cases to be plucked out as we find them. Each section is discussed in detail below:

TOP LEVEL:

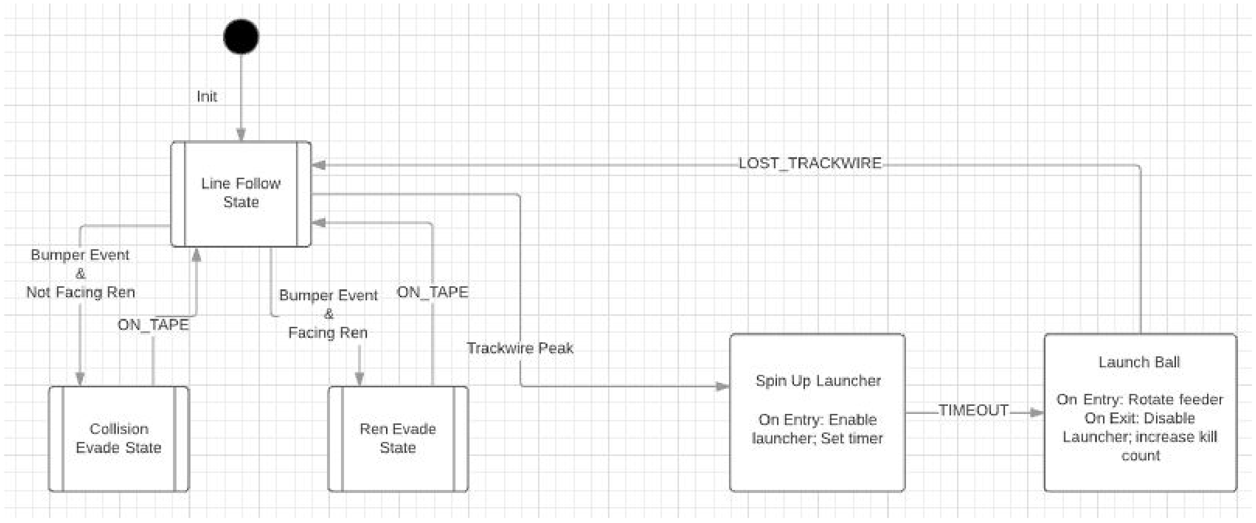


*Fig. 33: Toplevel-HSM*

At the top level, the simplified behaviour can be seen: follow the line until a trackwire is found, shoot it, then move on. The addition of the Ren Sensor was of great benefit to the new state machine. With the ability to differentiate between the Ren target and the normal obstacle, we can define two different behaviours for each, governed by the Collision and Ren evasion subHSMs. The launch procedure was placed at this level since it is only two states, consisting of spinning the launch motor up, and dropping a ball into the barrel.
After each AT-M6 is shot, a counter increases for how many targets have been taken down. As detailed below, the Ren evade state will execute the final shot if this counter indicates there are no more targets to shoot.

We observed that the launcher motor also trigger the trackwire. To safeguard against this, the motor is set to automatically shut off within one second of firing the ball.

LINE FOLLOWING:

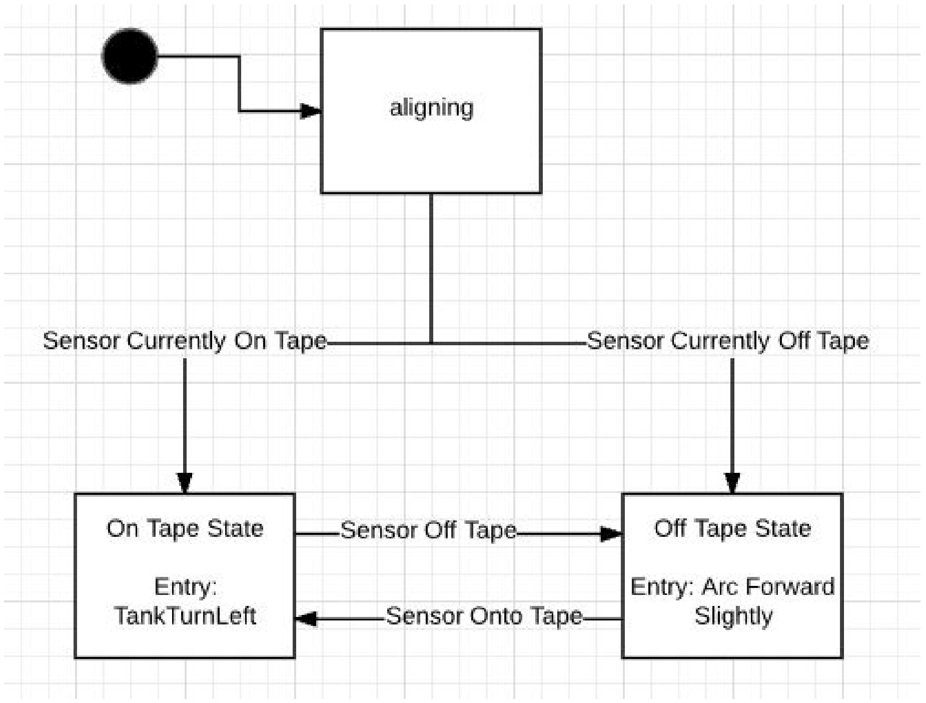


*Fig. 34: Line Following sub-HSM*

The line following behavior seen in this subHSM is the single sensor approach: If the sensor is on tape, get it off by tank turning. If the sensor is off tape, drive forward until we hit tape. This forward motion is done with a slight arc, ensuring that during long stretches we will still occasionally perform correction turns and hug the border properly. In this manner, we ensure that we are always driving counterclockwise, and always staying within the border. Corners are no longer done with neat 90 degree turns, but the radius is small enough that corners are easily handled with no more than 3 corrections.

The subHSM starts by polling the one tape sensor for its current position, and starts off in the corresponding position. With this method, there is no need for an initial border finding state; we just go.

COLLISION EVASION:

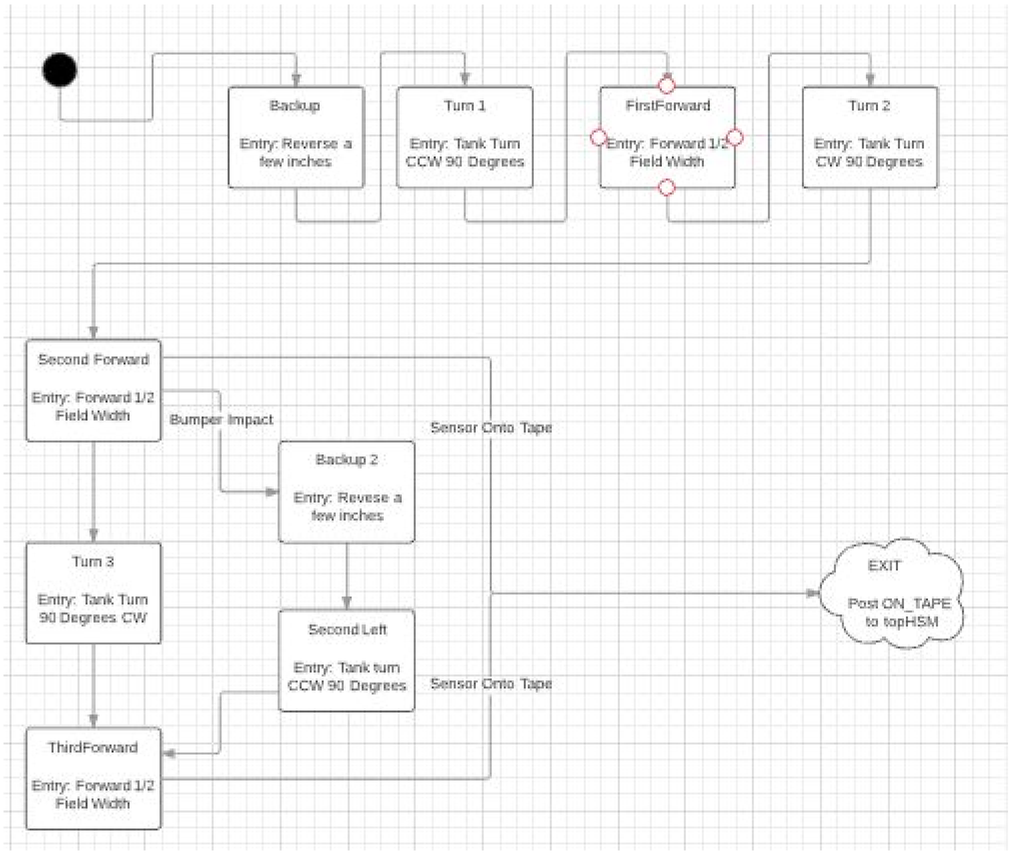

Backup
Entry: Reverse a few inches
Turn 1
Entry: Tank Turn CCW 90 Degrees
FirstForward
Entry: Forward 1/2 Field Width
Turn 2
Entry: Tank Turn CW 90 Degrees
Second Forward
Entry: Forward 1/2 Field Width
Bumper Impact
Sensor Onto Tape
Backup 2
Entry: Revese a few inches
Turn 3
Entry: Tank Turn 90 Degrees CW
EXIT
Post ON_TAPE to topHSM
Second Left
Entry: Tank turn CCW 90 Degrees
Sensor Onto Tape
ThirdForward
Entry: Forward 1/2 Field Width

*Fig. 35: Collision Evade sub-HSM*

The strategy for avoiding collisions with this subHSM is to do a box turn when an obstacle is seen. We will only expect to see obstacles on the border, since we will only ever be driving there. As such, when we see a collision and its not the Ren ship, we can avoid getting confused by alignment marks by driving away from the obstacle, all the way out to the middle of the field. Listening for tape events during the second forward traversal catches the case where the obstacle is in the corner, as shown in the following two diagrams:

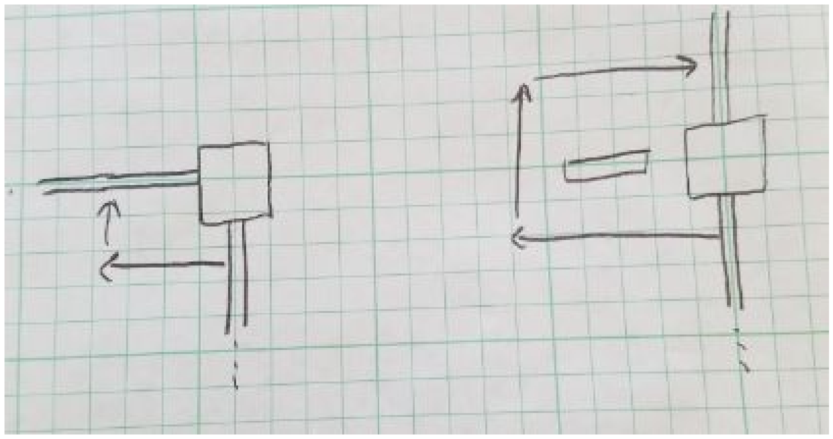


*Fig. 35: Collision Evade Figure*

Each arrow without an exit condition in the diagram above is assumed to be the ES_TIMEOUT event that signifies when the requested maneuver is complete. We spent some time carefully calibrating these times for accurate turns and traversals, and the result is a neat evasion which avoids the alignment marks each time; with this methodology, our one-sensor line following can stay as simple as it needs to be without complicated conditions for rejecting the mark. We just drive around them!

REN EVASION:

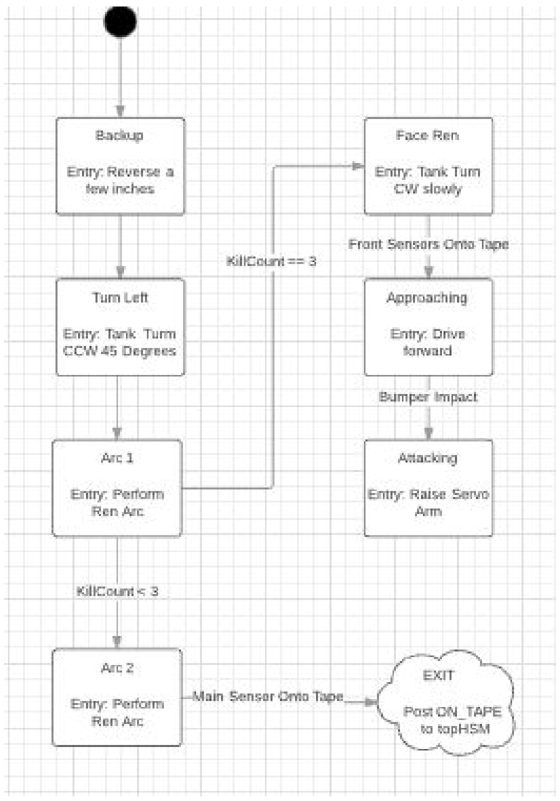


*Fig. 36: Ren Collision Evade sub-HSM*

Since we can tell when it's Ren we have run into, we can perform a very specific arc in order to bypass the ship and continue on our traversal around the field. The timers for this arc were fine tuned in order that we can reliably bypass the ship regardless of the initial impact angle.
If the killCount is 3 during this time, then we are ready to execute the final shot; after the completion of the first arc, the robot lands roughly on the T-shaped alignment. A tank turn then finalizes our placement using the tape sensors until we are ready to drive forward and deposit the ball.

## Conclusion

After an exhausting marathon, we have created a robot, and very nearly finalized its operation. After careful testing of the hardware, software, and their interaction, we are left with an autonomous bot of our own design, and it feels darn good. We set ourselves up with the proper tests, including verbose output and test harnesses, to ensure and confirm that everything is integrating together correctly; we have watched the robot flounder around on the field countless times, tweaking bit after bit; we have probably worked harder on this than on anything else. As we reach the conclusion, we can look back on this as a rewarding experience. It has been said that robotics is the art of getting a thousand small things right at once, and after an endeavor like this it is certainly apparent that this is a true statement. Regardless, the end is in sight, with countless hours of teamwork and effort leading us to this point; a complete robot.

## Bill of Materials

| ITEM | PRICE | QTY | DESCRIPTION | Source |
|---|---|---|---|---|
| Drive Motors | 31.98 | 2 | SainSmart 12V DC InstaBots Motor 201RPM x2 | https://www.amazon.com/SainSmart-InstaBots-Encoder-Arduino-Control/dp/B01GCM7QEM/ref=sr_1_1?ie=UTF8&qid=1510800615&sr=8-1&keywords=sainsmart+12vdc+201rpm |
| Stepper Motor | 2.598 | 1 | 28BYJ-48 5V stepper + Driver Board | https://www.amazon.com/LAFVIN-28BYJ-48-ULN2003-Stepper-Arduino/dp/B076KDFSGT/ref=sr_1_1_sspa?s=industrial&ie=UTF8&qid=1510800785&sr=1-1-spons&keywords=stepper+motor&psc=1 |
| Motor Mount Bracket | 8.99 | 1 | L-shaped Bracket 2-Pack | https://www.amazon.com/Black-Metal-Mounting-Bracket-Holder/dp/B073JQ43DS/ref=sr_1_6?s=industrial&ie=UTF8&qid=1510800749&sr=1-6&keywords=25mm+dc+motor+mount |
| Shaft Couplers | 2.5 | 1 | 4 5mm to 8mm shaft couplers. 1 used | https://www.amazon.com/Wangdd22-Coupling-Coupler-Alumlnum-Casing/dp/B01M8QXY8N/ref=sr_1_9?s=industrial&ie=UTF8&qid=1510800715&sr=1-9&keywords=5mm+to+8mm+coupler |
| Launcher Motors | 8.9 | 1 | 2x Mounted Launch Wheel/Motor Combo | https://www.ebay.com/itm/3VDC-Motor-Set-Wired-Together-with-On-Off-Hi-Lo-Switch/161475882616?hash=item2598b66e78:g:gUwAAOSw~1FUXBDx |

| | | | | |
|---|---|---|---|---|
| Caster | 2.99 each plus 8.99 shipping | 2 | Pololu 3/8 inch metal ball caster | https://www.amazon.com/Pololu-Ball-Caster-Inch-Metal/dp/B01CTUPE7M/ref=sr_1_1?ie=UTF8&qid=1510890018&sr=8-1&keywords=pololu+ball+caster |
| MDF | 10.5 | 3 | BELS $3.50 Each | BELS |
| Foamcore | 12 | 3 | BELS $4 Each | BELS |
| Limit Switches | 8 | 10 | 10-pc Switch Set | Fry's Electronics |
| 5V/3.3V Regulators | 3 | 5 | BELS | BELS |
| | | | | |
| Digital Proximity Sensor | 5 | 1 | | Provided by Team Hitchhikers |
| Standard Servo | 8 | 1 | | Already owned by Joseph |
| Mounting Hardware (Bolt Set) | 7 | 1 | Assorted Nuts + Bolts | Already owned by Joseph |
| Rollerblade Wheels | 3 | 2 | | Already owned by Joseph |
| Electronic Components | 12 | 1 | (Resistor, Capacitor sets, Op-Amps, Regulators) | Already owned by Zili |
| | | | | |
| **TOTAL** | 123.468 | | | |