CMPE 118 - Final Project

Team: Yub Nub

Robot: CMPEwok

December 10, 2015

Partner 1: Mariette Souppe msouppe@ucsc.edu
Partner 2: Danny Eliahu deliahu@ucsc.edu
Partner 3: Gilberto Barrios gibarrio@ucsc.edu

Website: https://cmpe118teamyubnub.wordpress.com/

Section: Preliminary Design

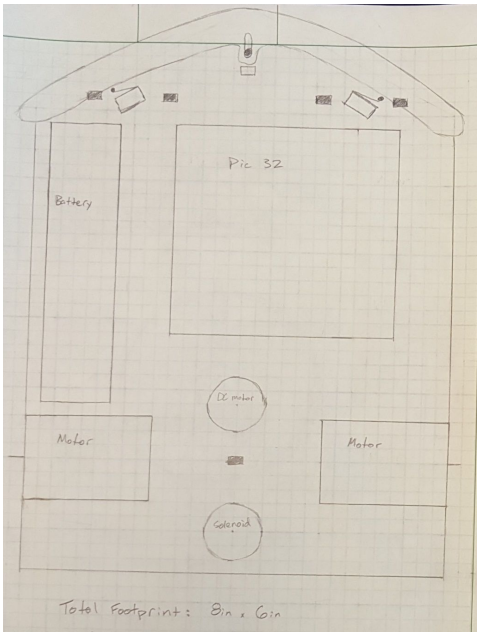
Below is our preliminary design for our robot. The design concept that we came up was to create a robot with two levels. The bottom level consisted of our hardware such as the battery, motors, UNO stack, H-bridge, tape sensors, and track wire sensors. The top level consisted of the ball launcher, the beacon detector, and the beacon filter.

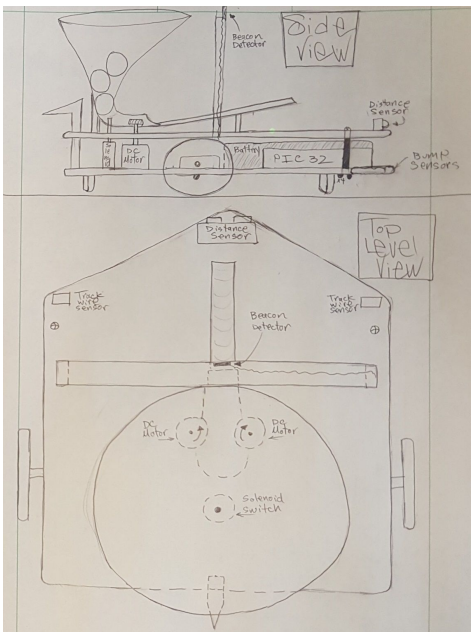
We took measurements of the components we had access to so we could have a realistic idea of how things would fit onto our robot. As for the other components, such as the ball launcher, we estimated rough locations. At this point in the design we were unsure how big of a motor or wheel we would need to successfully launch the ping pong balls. Therefore, we left the top level of the robot as open as possible so that we would have enough room to accommodate various launch motors.

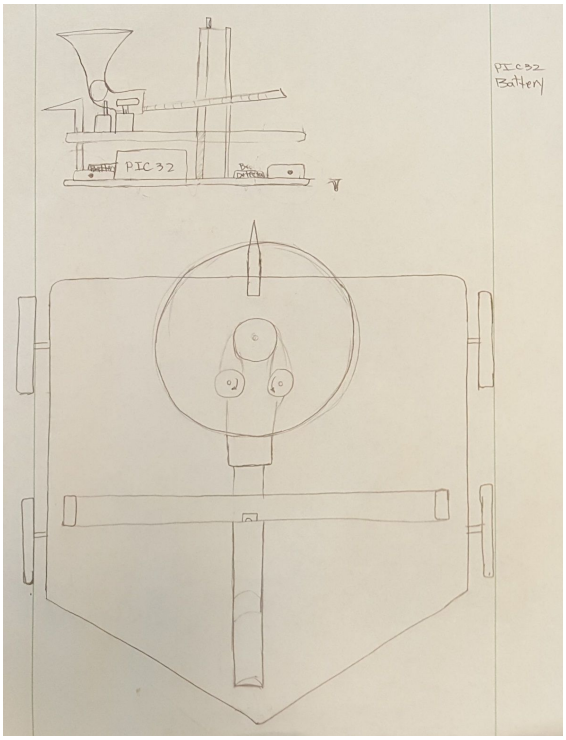
The general idea for the ball launcher was to have the ping pong balls fall into a funnel and through a tube where a solenoid would stop the ping pong ball. The solenoid would be released when an opponent was detected. The ping pong ball would shoot out by an constant running motor.

We wanted the ability to shoot one ping pong ball at a time, but we also did not want waste money or space with two solenoids (the method implemented by most teams). Therefore, our preliminary design was to have the solenoid toggle a lever that would allow only one ball to roll through at a time. This method did not end up working because the solenoid we purchased did not have a large enough throw.

Our preliminary design for the bumpers was to mimic the concept used by the roaches, but to make the height of the bumper taller, thus allowing us to detect bumps from high or low elevations.


Pic 32
Battery
DC motor
Motor
Motor
Solenoid
Total Footprint: 8in x 6in


Beacon Detector
Side view
Distance Sensor
DC Motor
Battery
PIC 32
Bump Sensors
Distance Sensor
Top Level View
Track wire sensor
Track wire sensor
Beacon detector
DC Motor
DC Motor
Solenoid Switch


PIC32
Battery
PIC 32

Section: 1st Design

Our first design is our preliminary design put into Solidworks with several modifications. The modifications made to each component are explained below.

Beacon Detector

Originally the beacon detector was positioned at the front of the robot. After discussing the design with our mentor we decided to move the beacon detector to the tower which was already holding the beacon hat. Because the opponent is required to mount their beacon at 11in, this change allowed us to better detect the enemy robot.

Ping Pong Ball Release Mechanism

As mentioned above, we were not able to use a solenoid to release the ping pong balls because the purchased solenoid did not have a long enough throw. We also experienced trouble attaching the end of the solenoid to the lever. Therefore we changed the solenoid to a servo motor but kept the same general mechanical design of using a lever to only feed one ball at a time. We found what we believed to be a suitable servo motor on Amazon and ordered it immediately so that we would not be set off schedule.

Ball Launcher

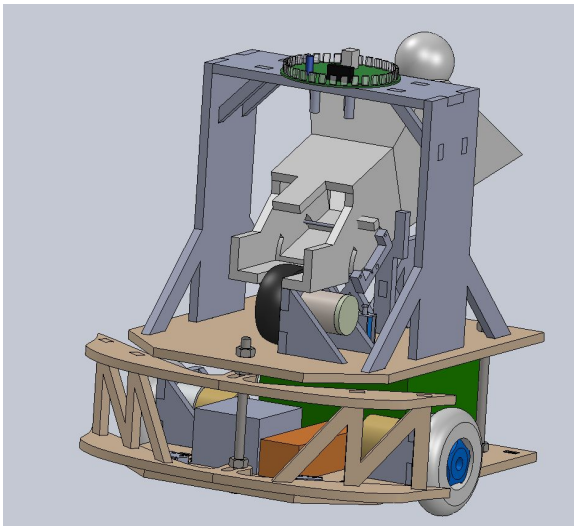
Next, we changed how our ball launcher looked in general. The preliminary ball launcher sat on a stand in the middle of the robot and we did not use a pipe for the ball to go through. In this design, we realized we would need much more space for the funnel, so we moved the launch wheel forward. We also had planned to use two launch motors, mounted side by side. After conducting tests, we concluded that we could get by with only one motor mounted under the balls.

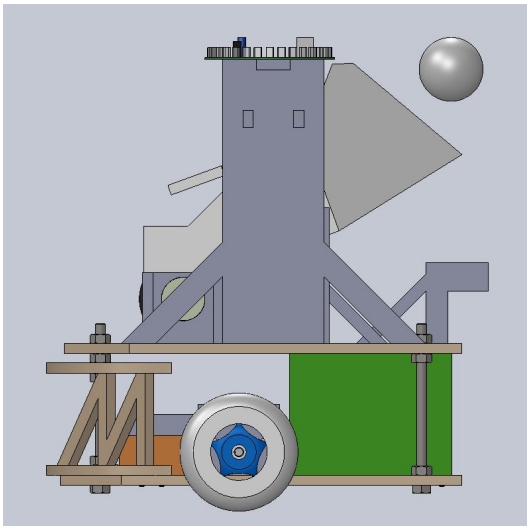
Bumpers

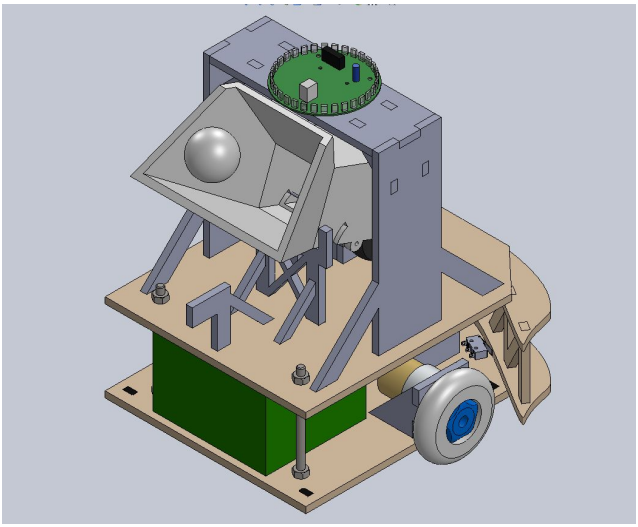
We changed the bumpers in order to make them longer as shown in the image below. This was done to accommodate the width of the wheels.

Wheels and Motors

During the preliminary design we had not yet decided what motors to use. At this point, we had received our motors and were able to design accurate motor mounts. The motor mounts were cut from MDF. We were faced with a challenge when attempting to interface the motor shaft with the roller blade wheels we had purchased. The motor shaft did not have the standard “D” shape cut in it. Instead, it had a hole drilled straight through its center. We tried to cut MDF pieces to interface with this motor shaft, but ended up deciding it would be a better use of resources to buy a shaft collar from Amazon. The shaft collar interfaced perfectly between the motor shaft and the roller blade wheels.







Section: Final Design

After having testing the previous design, many more changes were made in order to produce the final design. Each of these changes are described below.

Ping Pong Ball Release Mechanism

Unfortunately, the servo motor we purchased for the release mechanism was not ideal. It drew far more current than we expected and we blew a fuse on the power distribution board the first time it was plugged in. After speaking to other teams in the lab we were able to buy a much smaller servo motor that a different group was not using. After implementing this smaller servo motor the release mechanism worked as expected.

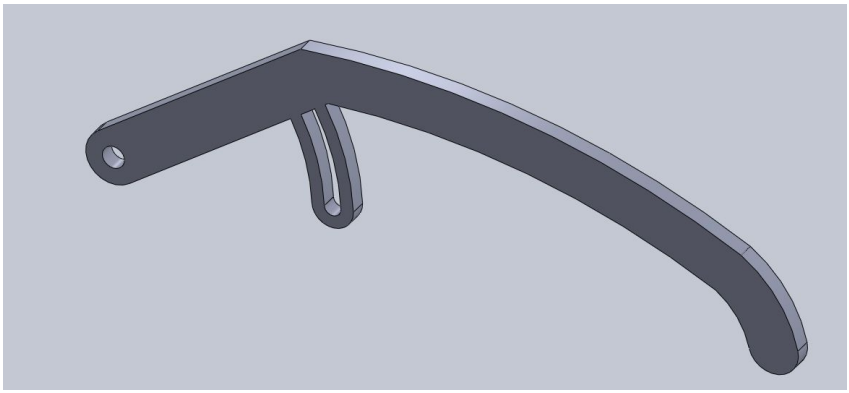
Ball Launcher

Up until this point we had only tested with 38mm ping pong balls. When we realized that the competition would feature 40mm ping pong balls we were forced to file out the inner diameter of the launch barrel.

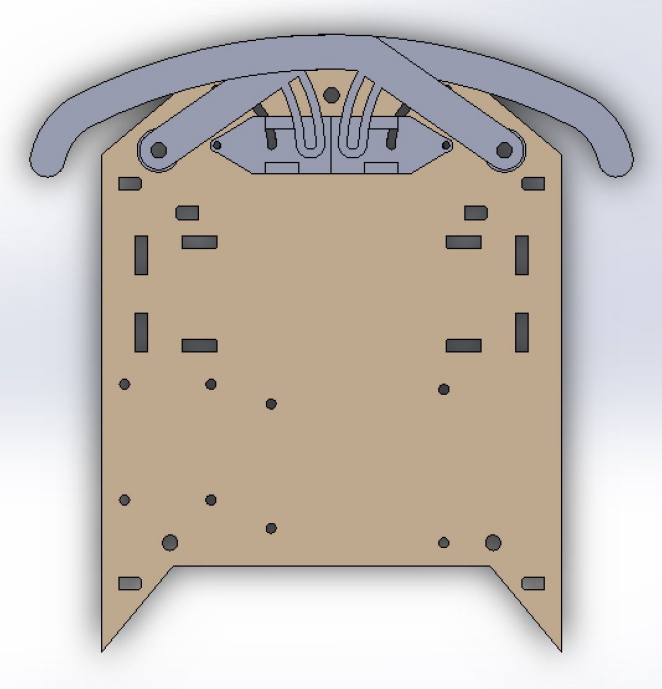
Bumpers

After testing the bumpers more rigorously, we realized that the design was not going to work well on our robot. The initial design was very similar to that of the roaches. The problem, however was that our robot has a much flatter front than that of the roaches. Because our robot did not have a pointy front, the bumpers would occasionally read a "double bump" if they were hit hard from the side. This was undesirable, so we designed a completely new bumper system (at this point it was easier to do this than to redesign our robot to have a more pointed front edge).

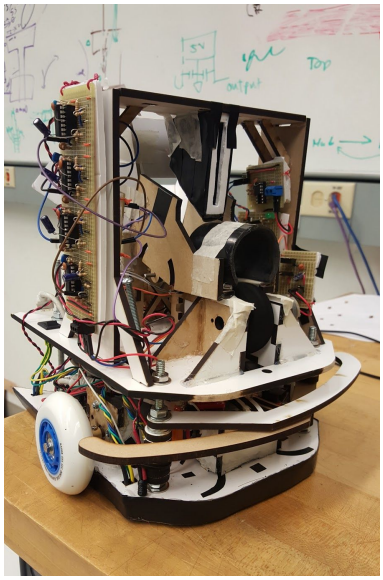
The new bumpers were designed to mechanically decouple the left bumper and the right bumper. Below is an image of one of the bumpers.

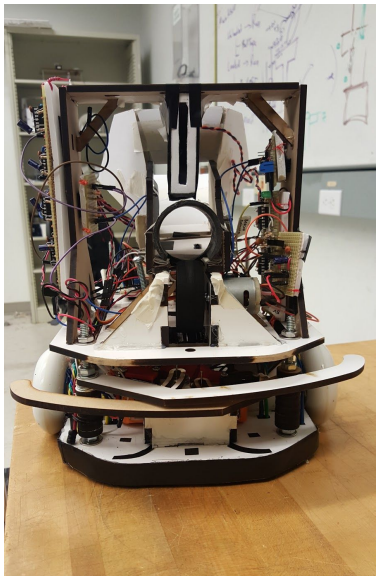


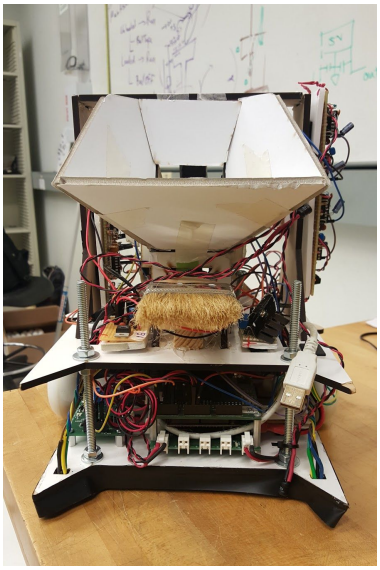
Two of these bumpers were mounted on the robot such that the middle 1.5in of each bumper overlapped in the center of the robot. This can be seen in the image below.



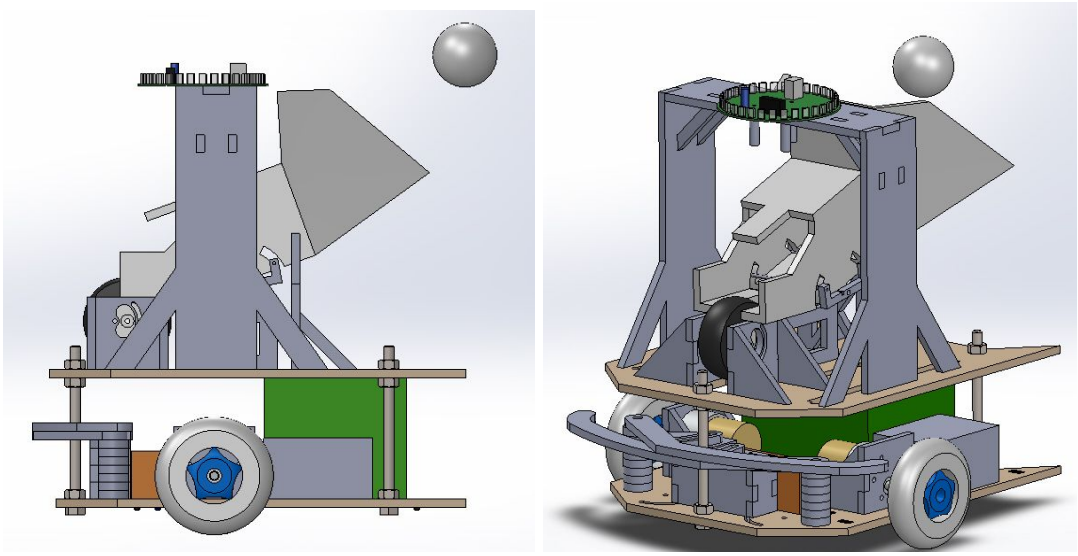
Final Pictures:

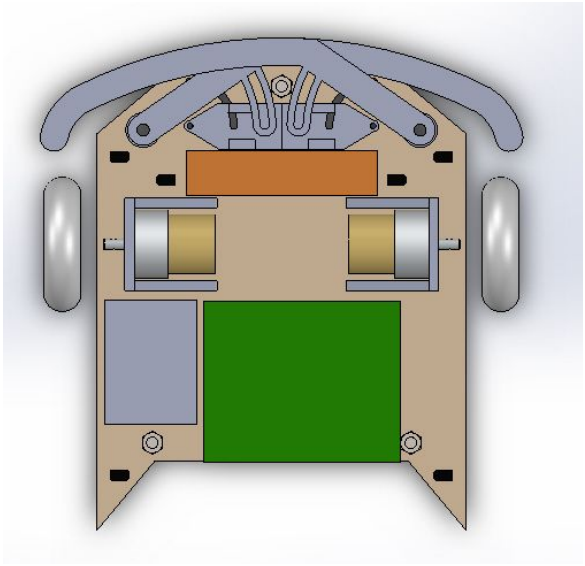






Solidworks images are on the following page …





Section: Bumper Sensors

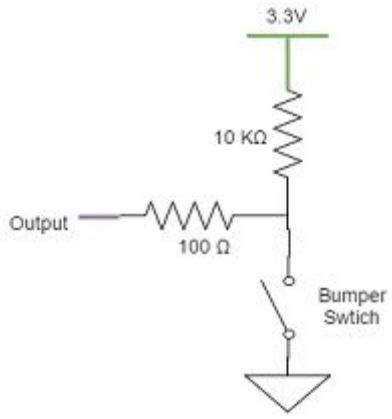
Subsection: Design

We ended up changing the design of our bumpers three times throughout the time of the project. These design changes are all described in the sections above.

Subsection: Circuit

The bumper circuit for this part of the robot was simple. The circuit consisted of two resistors, the power source of 3.3V and the bumper switch. The mechanical switch that was used for our bumper mechanism had the diagram of the circuit on the switch. After figuring out how the mechanical switches worked, we made a simple circuit on the breadboard and connected it to the oscilloscope to check the signal when the switch got triggered. We specifically chose to power the bumper switch at 3.3V because the output of the switch that would go into the UNO stack can not exceed 3.3V or we would blow out a fuse. These

switches worked fairly well, however sometimes there would be a lose wire or the switch would get stuck. These were early problems we came across and fixed them quickly and we had spare switches incase one really failed on us. Below is the schematic that was used for our bumpers.



Subsection: Code

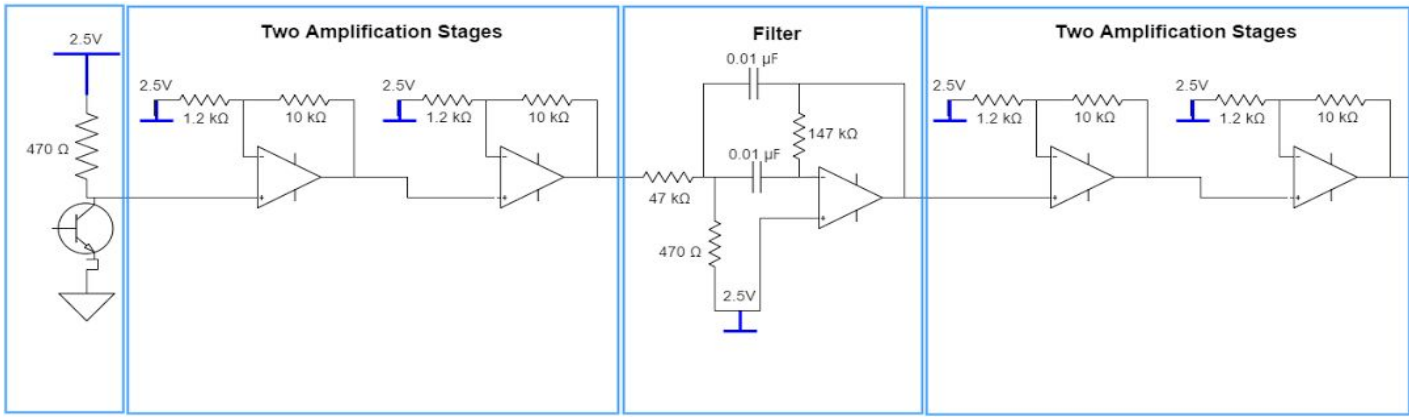
The implementation for the bumper sensors was the same idea for Lab 0, except that our code was modified for two bumper sensors versus four bumper sensors. We used the function Ewok_ReadBumpers() which would read each bumper value, where one of the bumper inputs was bit shifted so we could read each bumper value side by side. The return value of the Ewok_ReadBumpers() were the states of the bumpers, where 0x01 means that the right bumper got triggered, 0x02 means that the left bumper got triggered and 0x03 means that both bumpers got triggered. Before checking which bumper got triggered we first checked if there was state change. If the previous state and the current state are not the same we know that was a change in the bumper state. Then we check which bumper got triggered which was previously mentioned. Lastly, we post the event into our state machine and the state machine will handle the behavior of the robot when a bump has been detected. Below is the code that was used for the bumper sensors.

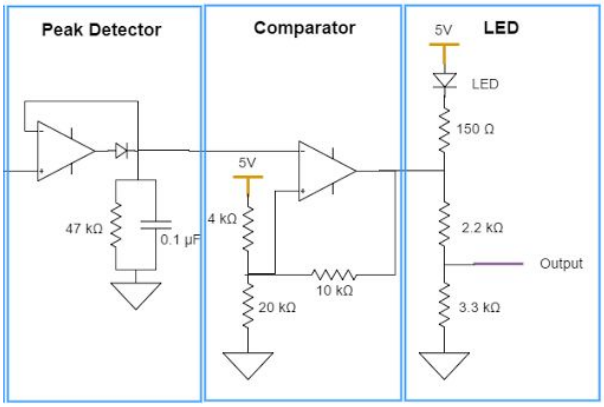
Bumper code: https://www.dropbox.com/s/cwqpb3fl9sn66n8/BumperService.c?dl=0

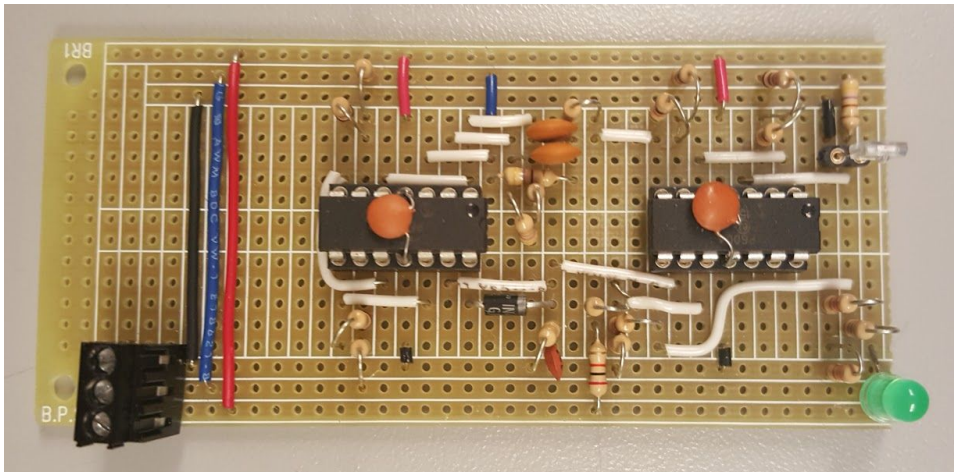
Section: Beacon Detector

Subsection: Circuit

The only person that had a somewhat working beacon detector was Danny, but he was not confident in his beacon detector so the team agreed to solder a new one. The design that the team choose was Mariette's design because of past performance of the design. The design consisted of a total of 4 gain stages where each gain stage was 10, one bandpass filter, a peak detector, a comparator, and LED voltage divider. Below is the schematic that was used for the our robot and the circuit itself.







One of the difficulties that came up while soldering the circuit was one gunning the first four stages in the schematic and not having anything work. The circuit should have been soldered module by module. Even though we already knew this to begin with, we thought it would be quicker to solder all of the modules at once. We were wrong. After spending some time troubleshooting the circuit we just scratched it and started fresh. Fail

early and often. The second time around the circuit was checked after every module and after every module we got the results that was expected each time expect after one on the gain stages. We had trouble seeing an amplification after the fourth gain stage and the reason being was that the wire was connected to the positive side of the op amp and not to the output pin. After that minor error, everything worked fine. We tested our beacon detector with all of the beacons in the lab where we were station and in the fabrication lab, where the playing field is placed, and our beacon detector was fine. The beacon was able to detected from diagonally and horizontally, so from anywhere on the field.

Subsection: Code

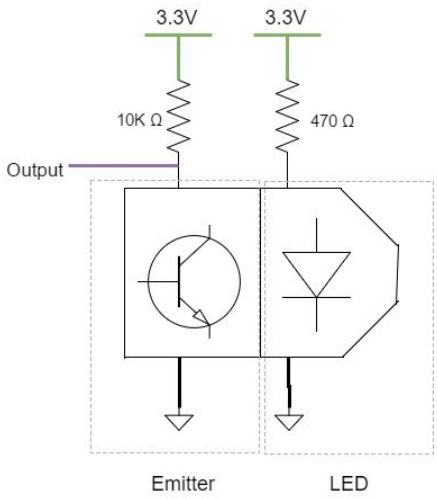
To obtain the value from our beacon detector we created a function, called Ewok_ReadBeacon(), that would read a digital value from the pin from the UNO stack. We were able to obtain a digital value versus an analog value because our detector had a comparator in its circuit. When the output pin outputted a 1, the signal was high which meant the beacon detector is on. When the output pin outputted a 0, the signal was low meaning that the beacon detector is off. This function was then used in the beacon service to detect whether an event occurred and post the event in the state machine where the state machine would handle the event just like in the bumper service. The code for the beacon is linked below.

Beacon code: https://www.dropbox.com/s/ezoeyr4i6oelxqy/BeaconService.c?dl=0

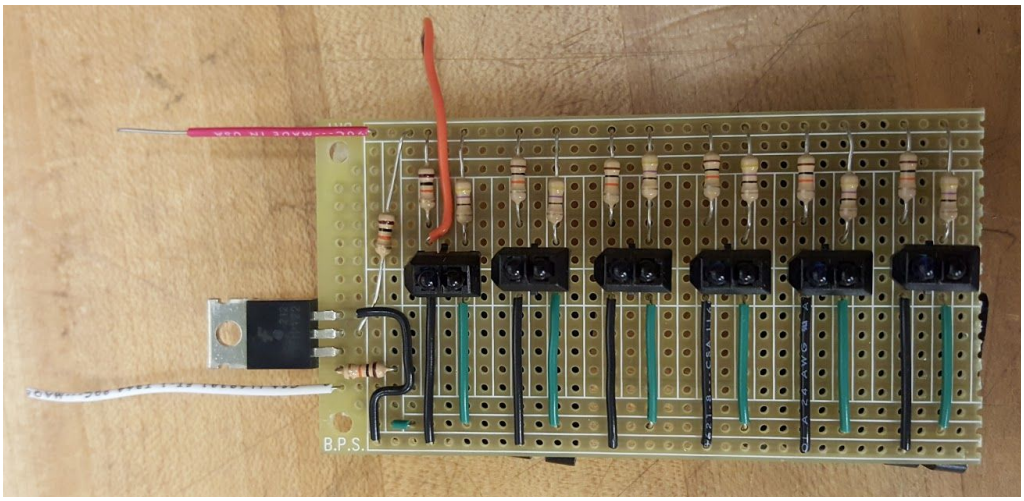
Section: Tape Sensors

Subsection: Circuit

Below is the schematic that was used for one of the tape sensors, however it is the same idea for all of the tape sensors were on a central board. The actual sensor was soldered in the perf board with long wire and leads so it was easy to move them around on our robot. We powered the tape sensors with 3.3V because the output was relatively close to that value and we would not blow a fuse on our UNO stack. We choose these specific resistors because the emitter and LED had a max current that we can not surpass.



Here is the central board for the tape sensors. The tape sensors were not soldered onto this board but were placed on the board to know where to solder the other components. Moreover, the tape sensors were connected to longer wires and soldered to this board.



We did not have any issues when constructing this circuit. The first steps we took when learning about this component was that we looked the data sheet, looked at a youtube video of how it worked, and a simple schematic of how it can be used. With these sources we were able to get this up and running fairly quickly and make the circuit schematic above.

Subsection: Code

The implementation of the tape sensors took some time. This is because we weren't entirely too sure on how to implement synchronous sampling. After some struggle we got the job done. First, each tape sensor value was obtained by their unique function, however

their data was read the same way. The functions that were used to obtain their values were; Ewok_ReadFrontLeftTape(), Ewok_ReadBackLeftTape(), Ewok_ReadMidLeftTape(), Ewok_ReadFrontRightTape(), Ewok_ReadBackRightTape(), and Ewok_ReadMidRightTape(). Each function returns an analog value which then gets handled in the tape sensor service. The tape sensor service reads in all of these values for each of the functions and puts them into an array. Next, we look at the difference of the previous value and current value. The difference of the previous value and the current value are then put into an array. Next, the values in the array get averaged to detect if the signal is high or low. We take the average of these versus just comparing the previous and current value because we want to reassure that the signal is consist, whether it is high or low. Moreover, the average value gets compared to a threshold. If the average value is below the low threshold then the signal is off, and if the average value is above the high threshold then the signal is high. Then we post the event to the state machine and the state machine handles the event in there.

At some point we were having a hard time seeing which sensors got triggered, so we decided to incorporate the LED's into our code to check which tape sensors were getting triggered. This was very helpful and quick to implement.

Tape sensor code: https://www.dropbox.com/s/36oophnv53rd5qq/LED_Service.c?dl=0

Section: Track Wire

Subsection: Circuit

Below is the circuit that was used to detect the 24 -26 kHz signal, also known as the track wire and hyper portal. We soldered two separate track wire circuits for our robot, a long range and a short range circuit.

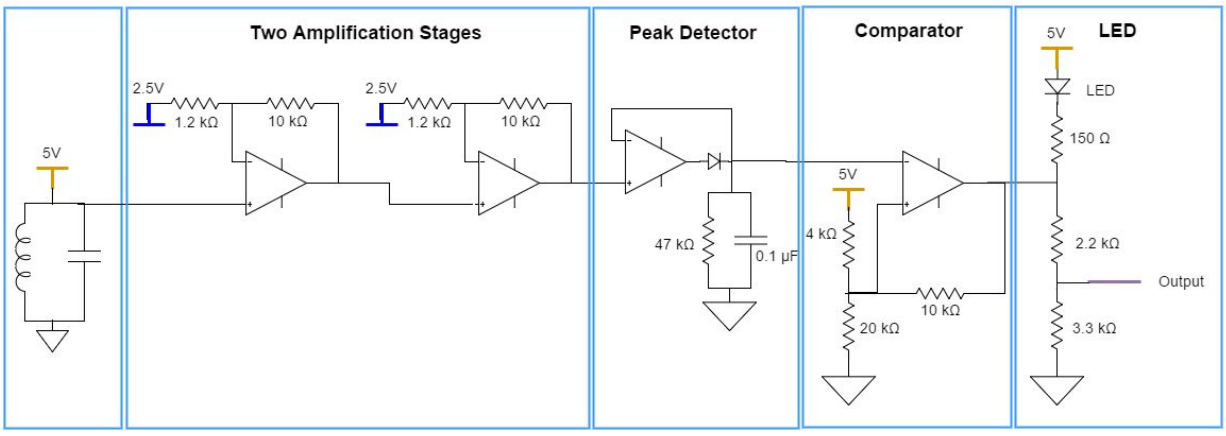
We initially constructed our state machine to use 3 track wire sensors, one long range and two short range. We decided to build and solder 4 track wire sensors just in case one sensor failed. From the circuit schematic below, you can see the track wire sensor was built using 5 modules. The first is a tank circuit, followed by a gain stage with two sub gain states. We designed two non-inverting amplifiers, both with a gain of 8.33. The long range sensor had the second gain stage with a gain of 10. Next, we built a peak detector with a 47k ohm resistor in parallel with a 0.1microF capacitor. Following the peak detector was a comparator which set the thresholds. Lastly, we built a voltage divider to lower the voltage of the output of the peak detector to 3.2V or ~0V.

Before we decided on the circuit described above, we incorporated a filter stage where we used a passive bandpass filter to filter frequencies below 24KHz and above 26KHz. However, we heard from a fellow team that incorporating a passive filter in the track wire sensor was a terrible idea because the op amps were not fast enough to filter frequencies at such high frequencies. In addition, there were little to no objects that

radiated light at such high frequencies in the room. This led us to dropping the filter implementation and keeping the circuit described above.

We came across issues with the track wire sensors after we soldered them. One issue was a strange output magnitude from the peak peak detector. We soldered the first track wire sensor without any issues but for some reason, the second sensor magnitude (after the peak detector) would immediately jump to 5V when the inductor came a certain distance close to the track wire. The spike was temporary and exponentially decayed if the inductor was static where the spike in voltage occurred. However, if the inductor was moved closer to the track wire again, the spike would occur again. Neither one of us have ever seen such a behavior in any of the track wire sensors we build. The spike would occur every time the inductor came close to the track wire. The output of a normal peak detector was a consistent and linear rise and fall as the inductor moved to and away from the track wire. However, the second track wire would spike several times if moved to the track wire and not moved away. We knew that having a track wire sensor with that behavior would cause false distance readings from the track wire. We build the third track wire sensor and to our surprise, it behaved the exact same way as the previous sensor did. We were very confused about the situation so we brought it up to our mentor, Oliver, and asked him for his opinion about the circuit. He inspected the circuit and did not find any soldering mistakes or shorts in the soldering. His advice was to walk through the circuit and debug it one module at a time. We debugged the circuit and found that the problem was only the peak detector.

We decided to build a new sensor and to our surprise, it had the same problem as our previous sensor. The peak detector would spike as the inductor got close to it. We soldered a fourth sensor and it worked without any issues. In the end, we ended up using only one peak detector in our robot and figured out an algorithm that would land our robot inside the parameter of the trackwire every time.



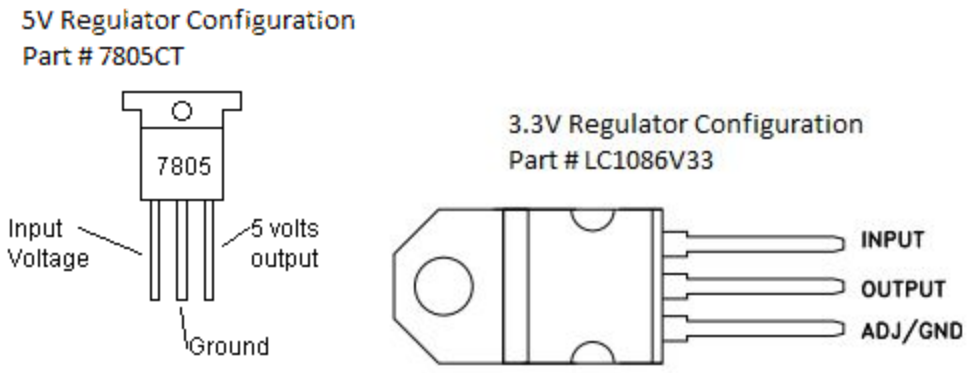
Subsection: Code

The implementation for the track circuit was similar to the tape sensors because we were reading an analog value instead of digital value like the beacon sensor. We had the option of putting a comparator on the track wire circuit, but if we did that we would not have much control on the hysteresis values. Setting the threshold values in software is much easier and simpler to do compared to hardware. If we were to do the comparator, we would have to recalculate the threshold values by looking at the resistors that are being used and the voltage that is being put in the circuit. This process would take more time then quickly switching values in the #define. To read values from the track wires, we used the functions called Ewok_ReadFrontRightTrack() and Ewok_ReadBackRightTrack(). We then take eight readings of each track wire sensor and take an average of those eight readings. If the average of those readings are above a high threshold then the sensor has been triggered. If the average of those readings are below a low threshold then the sensor has been untriggered. Taking the average of our readings is called synchronous sampling and we use this to make sure that the signal is actually go high or low and that the signal that has been sensed isn't just a fluke in the sensor.

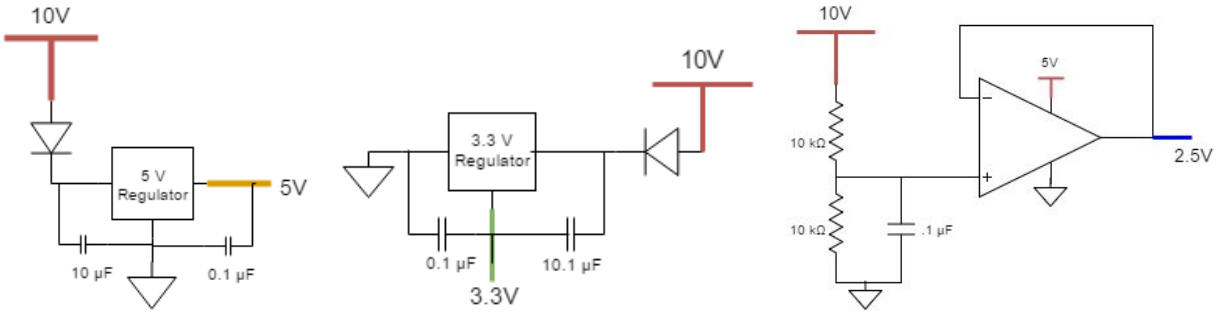
Track wire code: https://www.dropbox.com/s/ptlpwbxkiqumzbt/TrackWireService.c?dl=0

Section: Power Distribution Board

We created a power distribution board so we would have a central hub to power all of the sensors. After connecting the wires together and tested the power distribution board, the 5V regulator worked fine but the 3.3V regulator did not. The 3.3V regulator was getting really hot, so we assumed that there was a short in the circuit. We realized that we forgot to look at the data sheet for the 3.3V regulator and just assumed that the 5V regulator and the 3.3V regulator had the same configuration and they don't. We were sure that one of the teammates blew up the 3.3V regulator and had to back and re-circuit that.



Here are the circuit schematics that were used for the each part of our power distribution board; 5V board, 3.3V board, and 2.5 board.



Section: Motors

Subsection: Wheels

The regular motors were used to drive the robot around. The motors were connected to the wheels of the robot and a shaft. The functions used to drive these motors were Ewok_LeftMtrSpeed(SPEED) and Ewok_RightMtrSpeed(SPEED). These motors were powered at 5V and they were just powered via the power distribution board.

Subsection: Ball Launcher

The motor that we used for the ball launcher was a RC servo. The servo would see-saw back and forth to release on ball one at a time when the opponent's beacon was detected. We used the function Ewok_SetServo(value) to create that see-saw effect in the state machine. This function was calibrated to the specific degree we wanted it. There was a lot of trial and error to obtain the correct degree value, but got it work in the end. This motor was powered with 3.3V via the UNO stack. A regular motor was used to shoot the ball. The functions Ewok_LaunchMotorOn() and Ewok_LaunchMotorOff() were used to turn the wheel motor on and off. This motor was powered at 5V.

Section: Drivers

We made a drivers source file which contains the initialization of ports and pins for different components used throughout the project. Additionally, this contained basic functions that were used throughout the project.

The first function that is in our drivers source file is Ewok_Init(), where it sets the I/O ports for the track wire, tape sensors, bumpers, motors, and the RC servo. Additionally it initializes AD_Init(), PWM_Init(), and RC_Init(). The next function in our drivers file are functions that control our motors for the wheels which were called Ewok_LeftMtrSpeed(SPEED), Ewok_RightMtrSpeed(SPEED). This function is used to set the speed and direction of the left/right motor. These functions take in a value between -10 and 10, where a negative value reverses the direction of the robot and a value of 0 stops the robot. Next we have Ewok_ReadFrontLeftBumper() and Ewok_ReadFrontRightBumper()

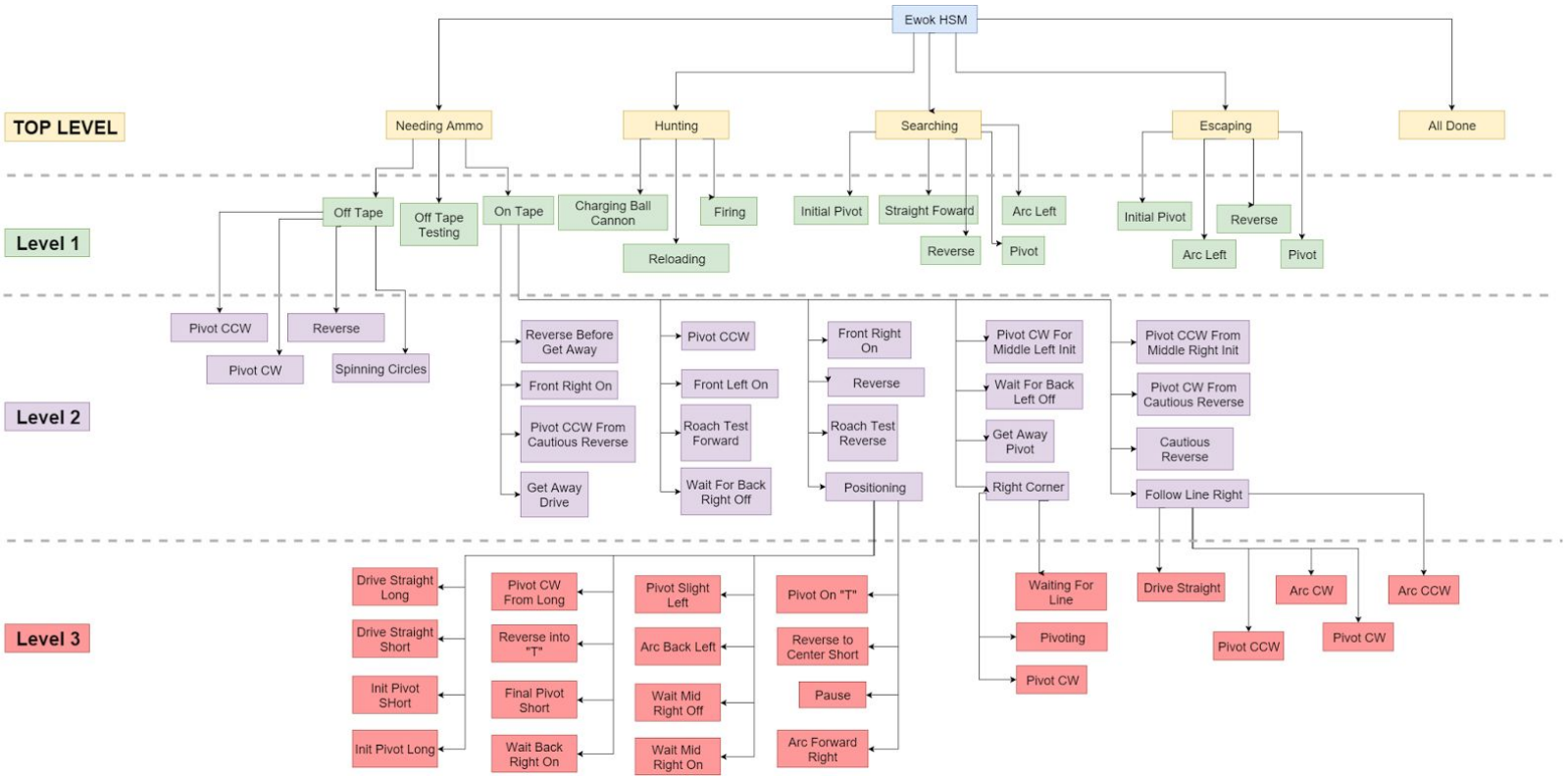
which just reads the signal from the mechanical which switch. These functions return a 1, if the bumper has been triggered, and 0 if the bumper has not been triggered. This next function Ewok_ReadBumpers() returns a 4-bit value representing both bumpers in the following order: left bumper, right bumper. So if the returned value is 0x2 then the left bumper has been triggered, if the returned value is 0x1 then the right bumper has been triggered, if the returned value is 0x3 then both bumpers were triggered, and lastly if the returned value is 0x0 then no bumpers were triggered. Next the functions Ewok_ReadFrontLeftTape(), Ewok_ReadBackLeftTape(), Ewok_ReadMidLeftTape(), Ewok_ReadFrontRightTape(), Ewok_ReadBackRightTape(), and Ewok_ReadMidRightTape() all just read in an analog value that's coming from the tape sensors and these values are then handled in the tape sensor service.

Driver code: https://www.dropbox.com/s/g1rr58j7pkbz9vk/Ewok.c?dl=0

Section: State Machine

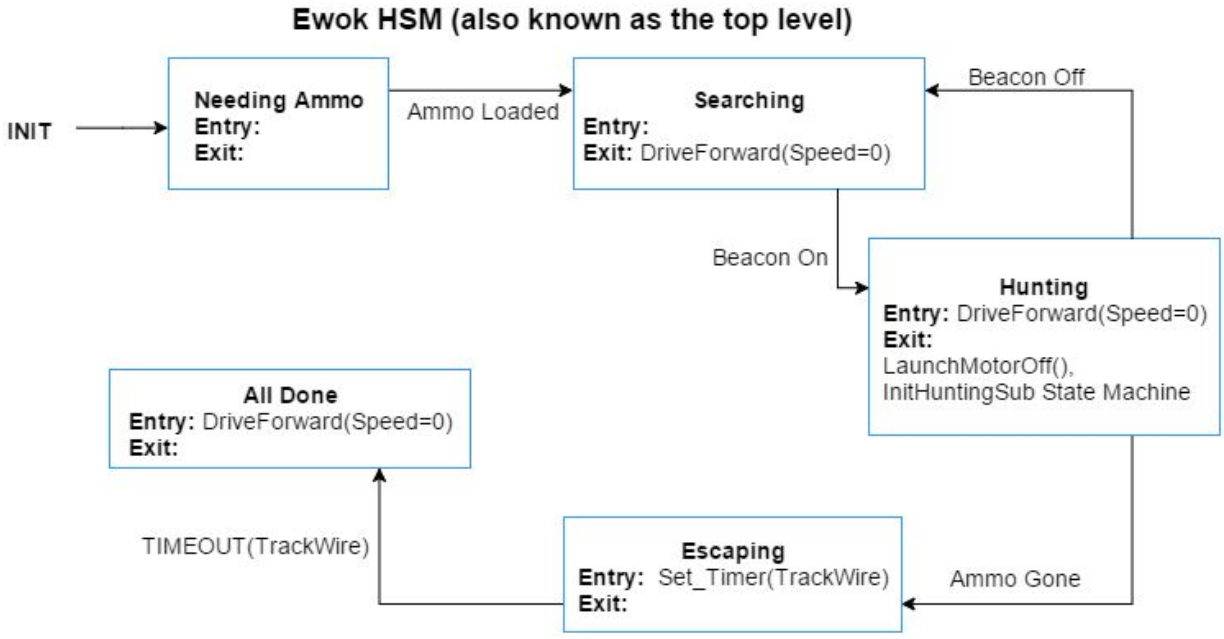
Subsection: General State Machine

Before getting into the details of the state machine for our robot, below is the layout of the hierarchy of our state machine. For now ignore all of the words within the flow chart. The main concept to take out of this diagram is that there are four levels in our hierarchical state machine. We have our top level which is in yellow, our first sub state machine in green, our second sub state machine in purple, and our third sub state machine which is in red. The diagram does not show how each state is triggered, it just shows that some states can have it's own state machine and the flow of the states.



Subsection: Top Level

The top level of the state machine consisted of five states; needing ammo, hunting, searching, escaping, and all done. When turning on the robot, the first state in the state machine that's executed is needing ammo. Once the ammo has been retrieved, we go into the searching state. While in the searching state, we try to find our opponent until the beacon detector goes on, then we go into the hunting state. In the hunting state, if the beacon goes off then we go back to the searching state. When in the hunting state and there is no more ammo left, we go into the escaping state which tries to find the hyper portal. Lastly, once the robot has found the hyper portal that is on then we enter the last state all done which just stops the robot. Below is the state machine diagram and code for the top level.



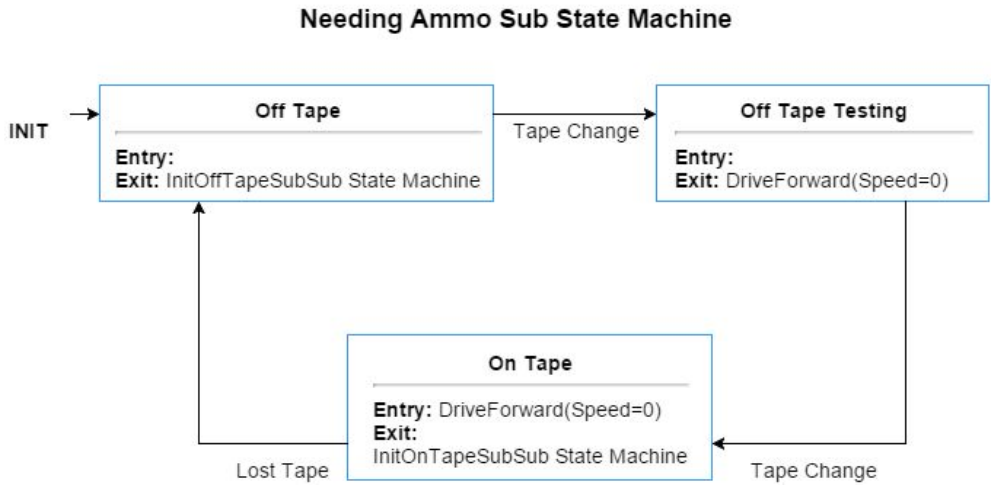
Top Level code: https://www.dropbox.com/s/muh9f0pzp4ftzz7/EwokHSM.c?dl=0

Subsection: Level 1

There are four separate state machines for the first level in the hierarchical state machine. These four separate state machines are for needing ammo, hunting, searching, and escaping.

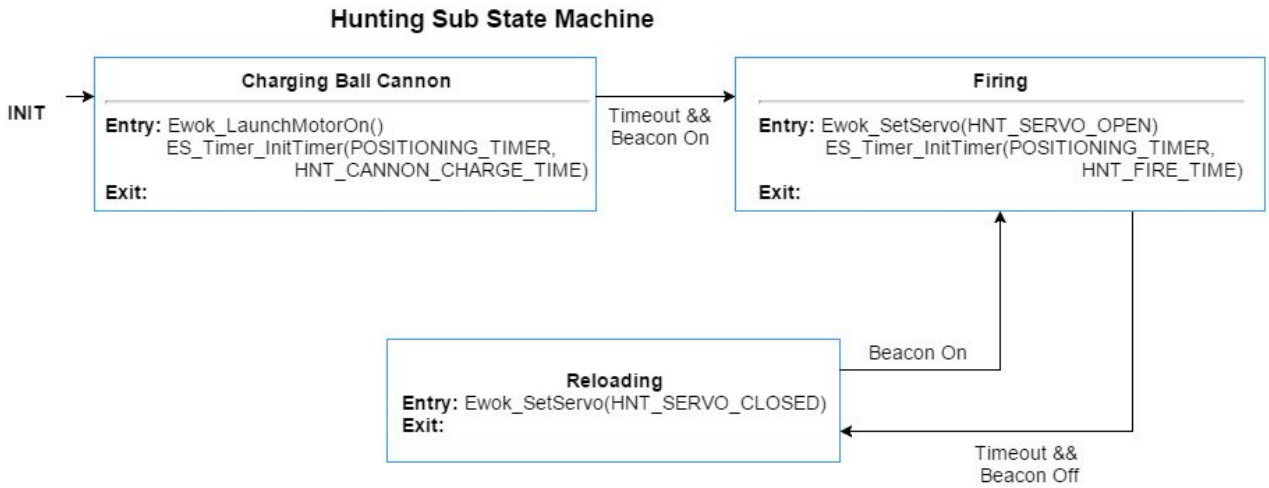
Needing ammo consists of three states; off tape, on tape, and off tape testing. Off tape goes into a sub state machine until a tape change has been detected, then it goes to off tape testing. Off tape testing makes sure the robot is off the tape and goes to the state on tape when a tape change has been detected. And on tape allows the robot to follow the tape

once tape has been found and goes back to off tape when tape has been lost. Below is a diagram and code for the needing ammo state machine.



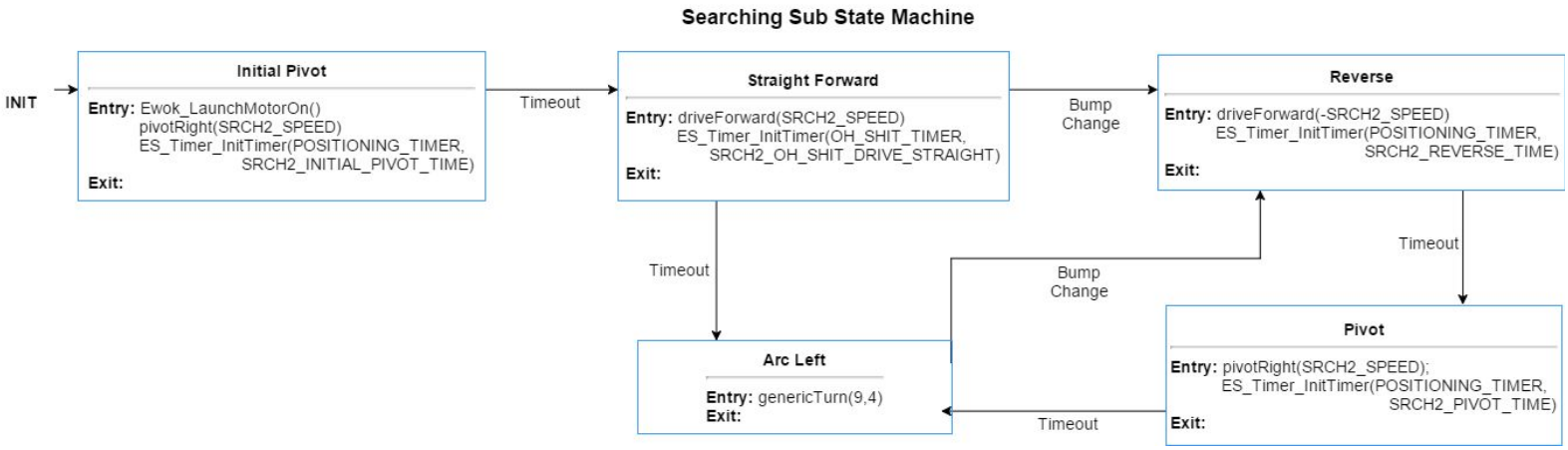
Needing Ammo code:
https://www.dropbox.com/s/dhcbmyw6yfbquwt/needingAmmoSubHSM.c?dl=0

Hunting has three states as well; charging ball cannon, firing, and reloading. Charging ball cannon state turns on the ball launcher motor and starts a timer. When the timeout for the charging ball cannon happens and the beacon detector is on then we go into the firing state which shoots the ammo to the opponent. In the firing state, the servo is open and a second timer starts. When the second timer expires and the beacon goes off then robot goes into the reloading state, where in the reloading state the servo is switched to closed until the beacon is on. Below is the state diagram and code for this state.



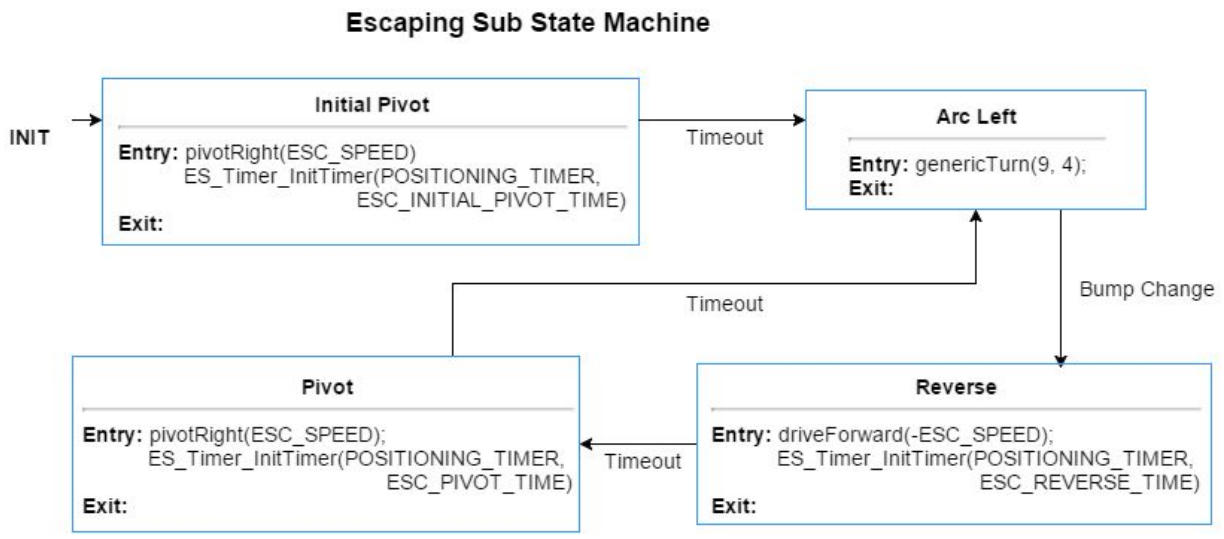
Hunting code: https://www.dropbox.com/s/oikgz26i1j55xop/huntingSubHSM.c?dl=0

Searching has five states; initial pivot, straight forward, reverse, arc left, and pivot. The initial pivot turns the launcher motor on, the robot pivots, and a pivot timer starts. When the pivot timer expires then we go into the straight forward state. In the straight forward state the robot drives forward and an “oh shit” timer starts. When the “oh shit” timer expires then we go the arc left state and the robot turns. When in the straight forward state or in the arc left state and there is a bump change then we go into the reverse state. In the reverse state the robot drives backwards and a new timer is started. When that timer expires we go into the pivot state where the robot pivots right and another timer starts. When the timer in the pivot state expires the we go back to the arc left state. Below is the state diagram and code for the searching state.



Searching code: https://www.dropbox.com/s/0jhc0hlpa4h9j96/searching2SubHSM.c?dl=0

Escaping, has four states; initial pivot, arc left, pivot, and reverse. When first entering in the escaping sub state machine you start in the initial pivot state. The robot pivots and starts a timer. When the timer expires, the robot goes into the arc left state. In the arc left state the robot does a generic turn and keeps on turning until a bump change is detected and then enters the reverse state. In the reverse state a new timer is started. When the timer in the reverse state expires the robot enters the pivot state. The pivot performs the same task as the initial pivot state. Below is the diagram and code for this sub state machine.

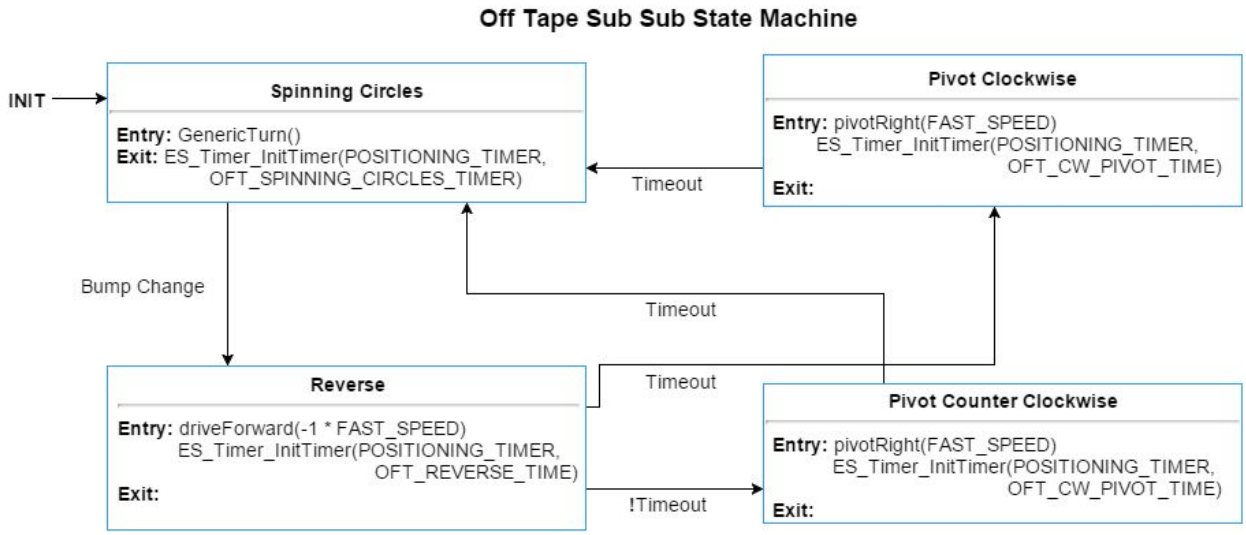


Escaping code: https://www.dropbox.com/s/6c61g5itecta2u3/escapingSubHSM.c?dl=0

Subsection: Level 2

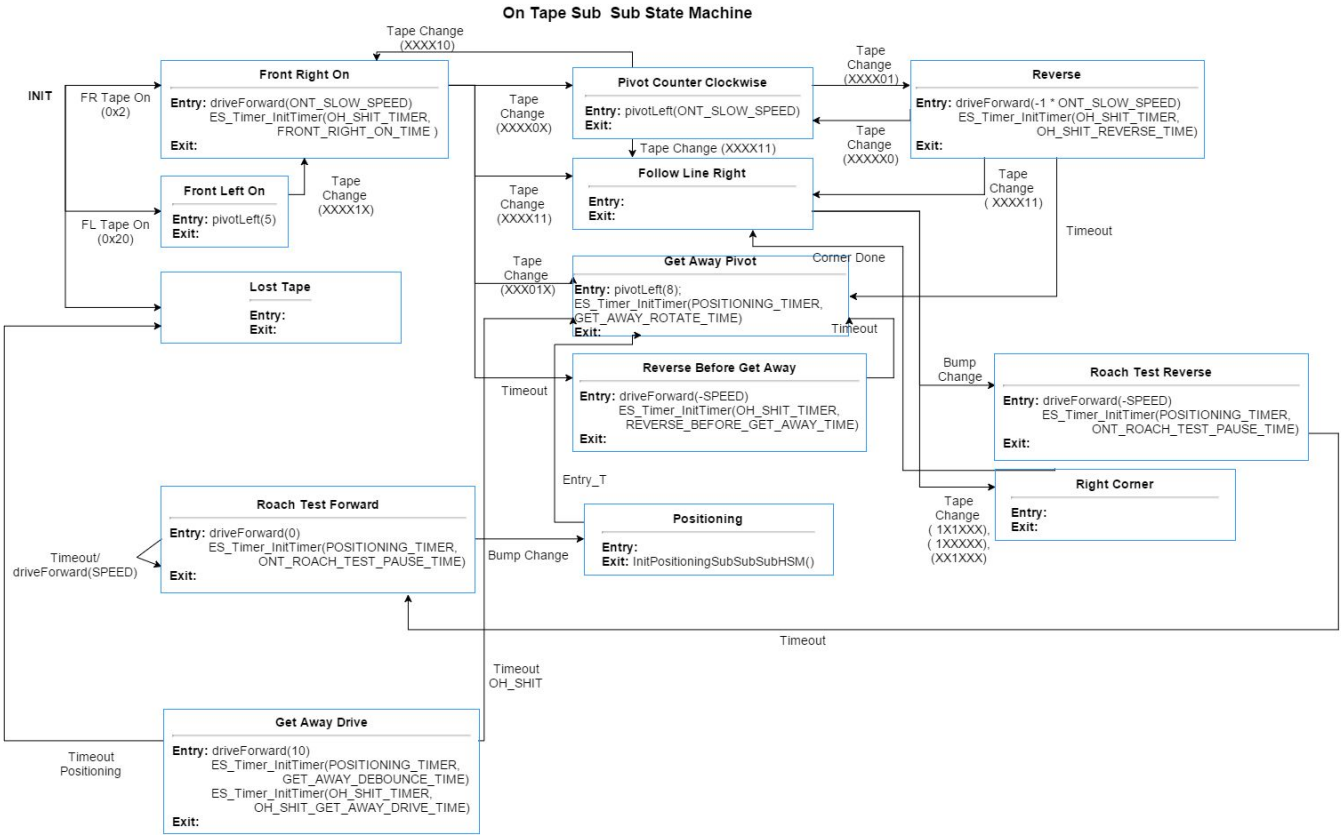
There are two states from the first level of the sub state machine that go into the second level of the sub sub state machine; off tape and on tape.

The off tape sub sub state machine has four states; spinning circles, pivot clockwise, pivot counterclockwise, and reverse. Spinning circles is the initial state in the off tape sub state machine and does a generic turn until a bump change has been detected. When a bump change has been detected, the next state is reverse. The reverse state drives the robot backwards and a timer starts. When the timer expires the robot goes into the pivot clockwise state, but if the timer does not expire the robot goes into the pivot counterclockwise state.



Off Tape code:
https://www.dropbox.com/s/xbpmqthk2rbv4ua/offTapeSubSubHSM.c?dl=0

On tape has 20 states. Refer to the diagram for the all of the states. The main idea of this state machine is to align the robot accordingly once the robot is on the tape. This state machine attempts to take every corner case when the robot detects tape.

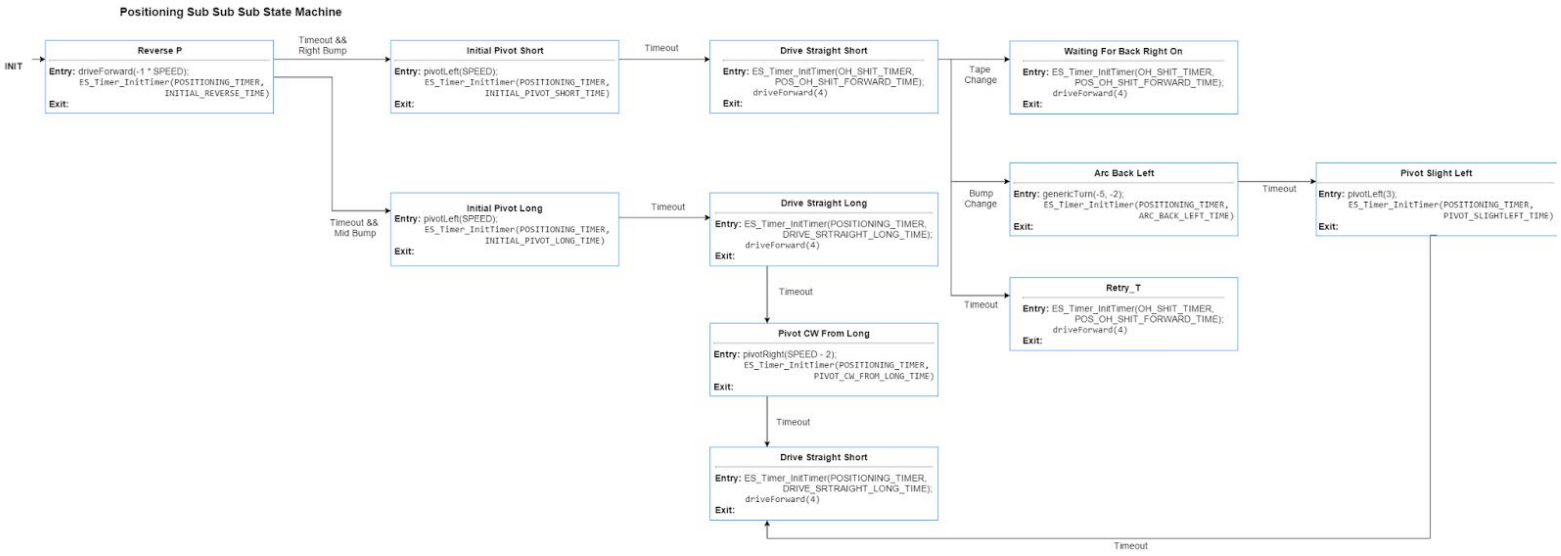


On Tape code:
https://www.dropbox.com/s/3ntdek64wp36id8/onTapeSubSubHSM.c?dl=0

Subsection: Level 3

There are three states that go into the third level also known as a sub sub sub state machine which are positioning, right corner, and line following.

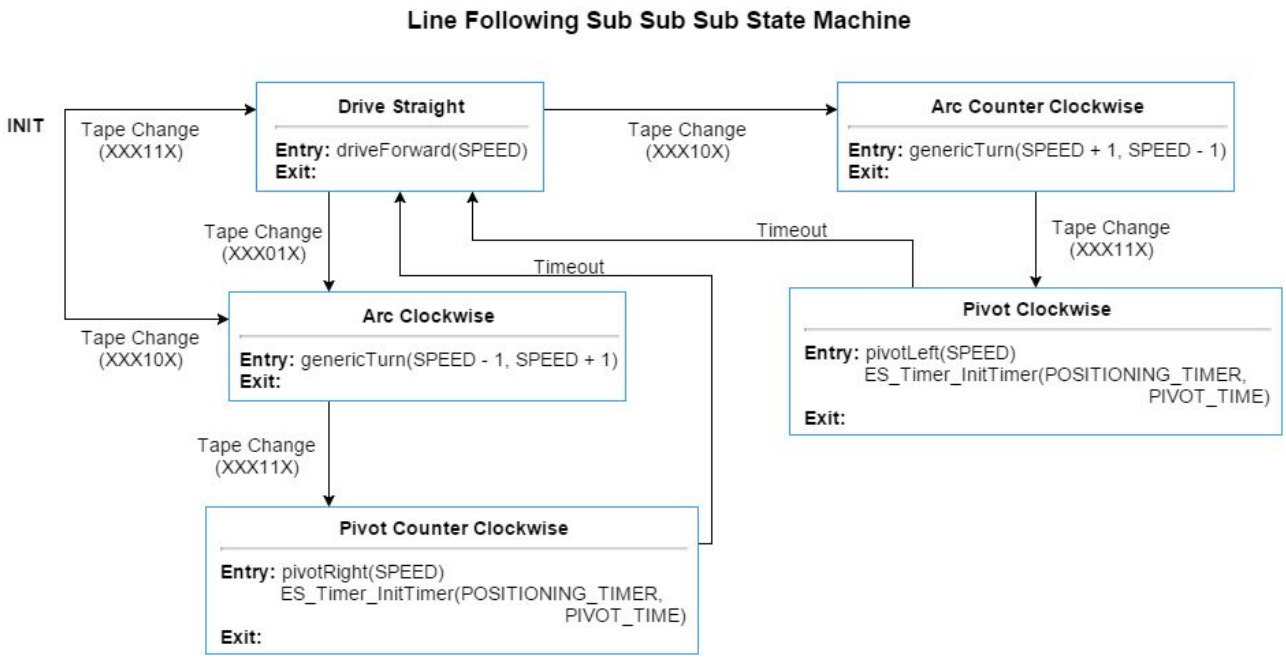
Positioning has sixteen states. Refer to the diagram for all of the states. The main of this state machine is that once the robot has detected the “T”, the robot repositioned itself to drive back into the ammo dump to get retrieve the ammo. Refer to the state diagram and code for all of the states on the next page.



Positioning code:

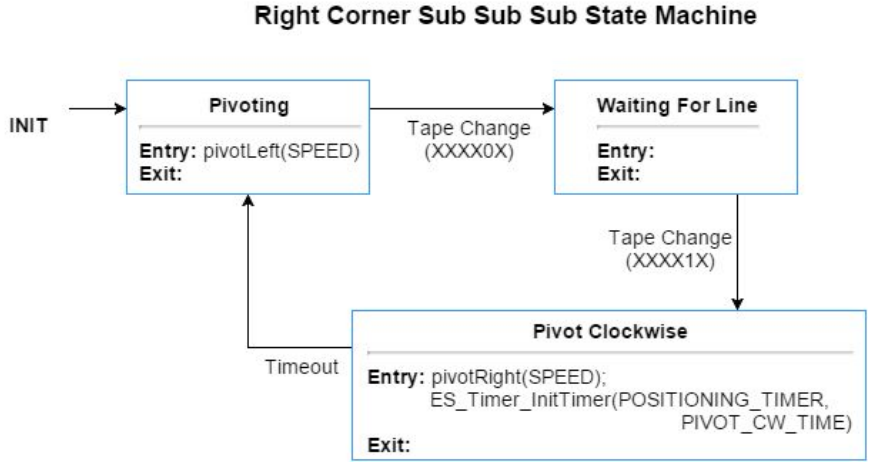
https://www.dropbox.com/s/e4i2uiisfohbwy6/positioningSubSubSubHSM.c?dl=0

Line following has five states; drive straight, arc counterclockwise, arc clockwise, pivot clockwise, and pivot counterclockwise. This state machine has two initial states depending on the parameter. If the event and parameter are tape change and XXX11X then the initial state is drive straight. If the event and parameter are tape change and XXX10X then the inital state is arc clockwise. The X's in the parameter represent that we don't care what those values are. We only care about the tape sensors that defined which is a 0 or 1 in the parameter. In the drive straight state, the robot drives forward. When in the drive straight state and the event and parameter are tape change and XXX10X then we go into the arc counterclockwise state. When in the drive straight state and the event and parameter are tape change and XXX01X then we go into the arc clockwise state. In the arc clockwise state, the robot does a tank turn and when a tape change event and parameter XXX11X gets detected you on into the pivot counterclockwise state. In the pivot counterclockwise state, the robot pivots right and a timer starts. When the timer expires the robot goes back into the drive straight state. Going back to the arc counterclockwise, when is this state the robot does another tank turn counter clockwise and if there's a tape change event and the parameter is XXX11X then the we go into the pivot clockwise state. In the pivot clockwise state the robot pivots left and a new timer starts. When the timer expires then it goes back to the drive straight state. Below is the state machine diagram and code for line following.



Line Following code:
https://www.dropbox.com/s/47flcvov7b0cdou/LineFollowRightSubSubSubHSM.c?dl=0

Right corner has three states; pivoting, waiting for line, and pivot clockwise. The first state entering the right corner sub sub sub state machine is pivoting. In pivoting the robot pivots left and changes states to waiting for line when a there is a tape change with the parameter of XXXX0X. In the state, waiting for line, there's an internal timer that just stalls until there's another tape change event with the parameter of XXXX1X. The last state in this state machine is pivot clockwise. Pivot clockwise the robot pivots right and a timer is started. Once the timer expires the robot goes back to the pivoting state. Below is the state diagram and code for the right corner state machine.



Right Corner code:
https://www.dropbox.com/s/y9wj3xszoek7ak8/rightCornerSubSubSubHSM.c?dl=0

Conclusion:

In conclusion there was a lot of learning when building this robot and a lot of “fail early and often”. Failing often and early made us achieve our desired deadline, getting checked off the day before Thanksgiving. We used the tools and skills that we gained from the previous labs to create this masterpiece. The dynamic was group worked out well because everyone had their forte, yet helped one another to get the tasks done. Next challenge, build this robot with mecanum wheels!