Mechatronics 2016 Final Project Report

Jamie Dieckman
Vishnu Surya
Leya Breanna Baltaxe-Admony

8 December 2016

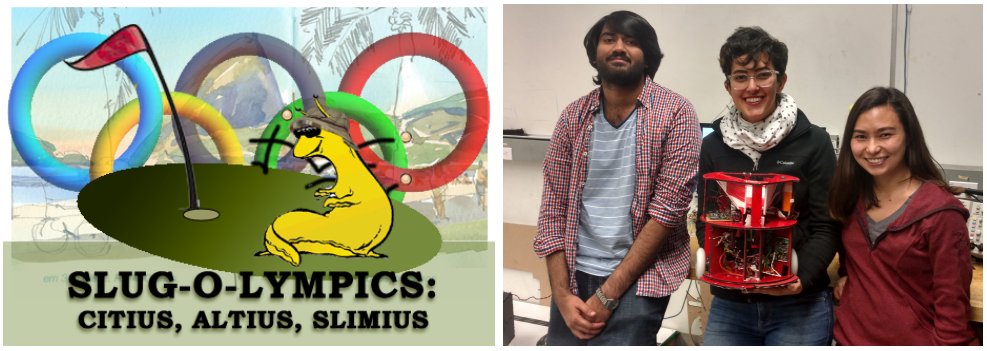


Figure 1: Our SLUG-O-LYMPICS family photo

1 Introduction

For the first time in 112 years, Golf returned as an Olympic sport at the Rio de Janeiro Summer Olympics. This year's CMPE-118 challenge celebrates its return. Given only four weeks, our objective was to create a functional robot from the ground up to complete the following tasks:

- Collect ping pong balls from ball loaders (Figure 2)
- Deposit two balls in one (of three) targets (Figure 2)
- Deposit one ball in another target

These tasks needed to be completed within two minutes, while avoiding obstacles and staying on the field (driving off the field would result in a long drop to the floor if no one caught it). The field and targets are lined with two inches of black tape on all sides. We used four total Vishay optical sensors (TCRT5000) for tape following around the field and stopping on targets. Ball loaders each dispense three balls using a plunger system (Figure 2). Ball loaders are active and usable only when its corresponding trackwire, which emits and electromagnetic signal, is on. Targets are each topped with infrared 2KHz beacons. Obstacle positions, active trackwires, and target positions are randomly generated between attempts.

In this document we will discuss our design considerations, expected results, and many many revisions after those initial expectations. Section 2 covers all mechanical aspects. Section 3 covers electrical, processing, and sensors. Our state machine design and code for sensor integration can be found in Section 4. Administrative notes such as scheduling and a Bill of Materials can be found in Section 5.

2 Mechanical Design

At the beginning of the project, we wanted to get our robot moving as quickly as possible so that we could begin working on integration and code. We began with a simple design that would allow us to make changes later as we saw fit and allow for a lot of room to move components around.

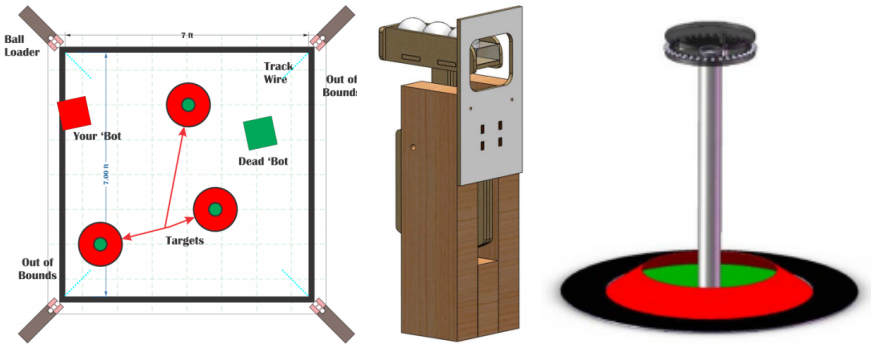


Figure 2: A bird's eye view of the field (left) is shown with some of the key features of the field: a ball loader (center), and target (right)

2.1 Strategy

We chose to have three tiers to give us the maximum amount of space for component placement.The majority of the bot was rounded to minimize the chance of becoming stuck on obstacles, as well as to allow our bot to make tank turns on a central axis. The top layer of the bot needed to come to a point because it routinely got stuck on the upper front face of the ball loaders. A circular portion of the base was extruded to help align with the target as well as allow room for our ramp to drop balls over the lip of the target. The third level has an extrusion with the inside width of the ball loader to help the robot line up to the loader if it is at a slight angle. A funnel of foamcore passed the balls to a ramp where their release was controlled by a servo.

2.2 Plunger Mechanism Evolution

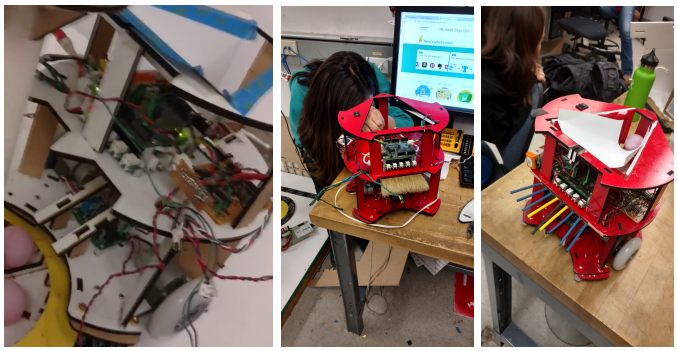


Figure 3: A few of the many iterations of our plunger mechanism

Development of a robust plunger mechanism required experimentation. We began with a V shaped design that we hoped would guide the bot into the ball loader (Figure 3 Right). When this design did not work well due to its inflexible nature, we decided to use a paint brush instead (Left). Since the brush was slightly too stiff we trimmed it down to reduce its resistance. After finding that the brush was not as reliable as we had hoped, we glued heat shrink in its place and found that to be the best mechanism for releasing the balls (Center). Additionally, we also changed our strategy for lining up to the ball loader. We initially planned to use two track wire sensors; the first would be placed between the motors, and when sensed, the bot would tank turn until the second sensor was detected, indicating that the bot was lined up with the loader. Unfortunately this method did not work as well as we had hoped, and often resulted in misalignment. When it was clear this method was not reliable enough, we decided to use just the track wire in the center of the bot and a timer to make a 45 degree tank turn and align with the loader. We found this method to work very well and integrated it into our final design.

2.3 Ramp

The ramp was made of two parallel pieces of MDF, placed so that only one ball at a time could pass. A servo was placed near the top of the ramp for releasing and retaining the balls. The angle of the ramp had to be adjusted to be much shallower than we had originally expected in order to prevent the balls from bouncing out of the hole. Additionally, we found that we needed to force the balls slightly to the right in order to drop more than one ball in a target without readjusting the entire bot. We did this by adding a small piece of heat shrink to the left side of the ramp to guide balls right. we also added a piece of felt on the right side of the ramp to slow down the balls and prevent them from bouncing out.

2.4 Component Placement

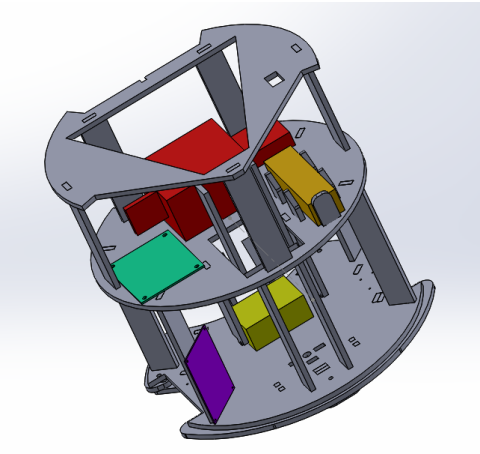


Figure 4: Components (shown in color) on the 3D modeled bot

We chose to place most of our components on the bottom most platform. We placed our bumpers on the lowest platform since we hoped to be able to detect collisions with the targets. To make space on the top for circuits we placed the bumpers on the underside of the platform, and fed wires through to the circuit on top. The bump sensors shared a circuit board with the tape sensors since they had similar placements. The tape sensors were place three in the front in a triangular formation for the purpose of tape following. A fourth was placed about two inches in from the back of the bot for use in detection of the target. The inductor of the track wire detector was placed directly between the motors so that the bot could make a 180 degree turn on its most central axis. The circuit board for the track wire detector was attached to the front left pillar to make the LED visible and to make good use of space (Figure 4 Purple). The servo was placed on a shelf that hung from the middle platform, just under the ramp. The circuit for the servo was placed on the front right pillar so that it could be near the servo and to maximize space. The bottom platform also held the H-Bridge for the motors (Yellow).

On the second layer, we placed our UNO32 (Red), battery (Orange), and our beacon detector circuit (Green). The wires from the bottom layer were fed through two large holes to the UNO32. The phototransistor of the beacon detector was placed on the uppermost platform, and was mechanically shielded with a slit made of 4 layers of MDF. The heat shrink for depressing the plunger was glued to the underside of the second platform as well.

The third layer held the funnel, made of foamcore lined with felt. the remote power switch was also on the top most layer for easy access.

2.5 Conclusion

It was very helpful that the first thing we did was CAD our bot. It made us think about the objective as a whole, and see how we could really use the field to manipulate the position of our bot. It was also helpful for brainstorming all of the electrical components we would need and sorting out where to put them early. Because we had the models readily available, we were able to print and re-print pieces of the bot over and over whenever we wanted to make slight changes. If

we could do it again, it would be nice to make several prints of each part initially so that we could alter them in the fab lab and swap them out easily.

3 Electrical Design

Our design required circuitry for the Beacon Detector, Tape Sensors, Bump Sensors, Trackwire Sensors, and Servo. Also considered in this section is the integration of all electrical components with our software via the UNO stack (Figure 5).

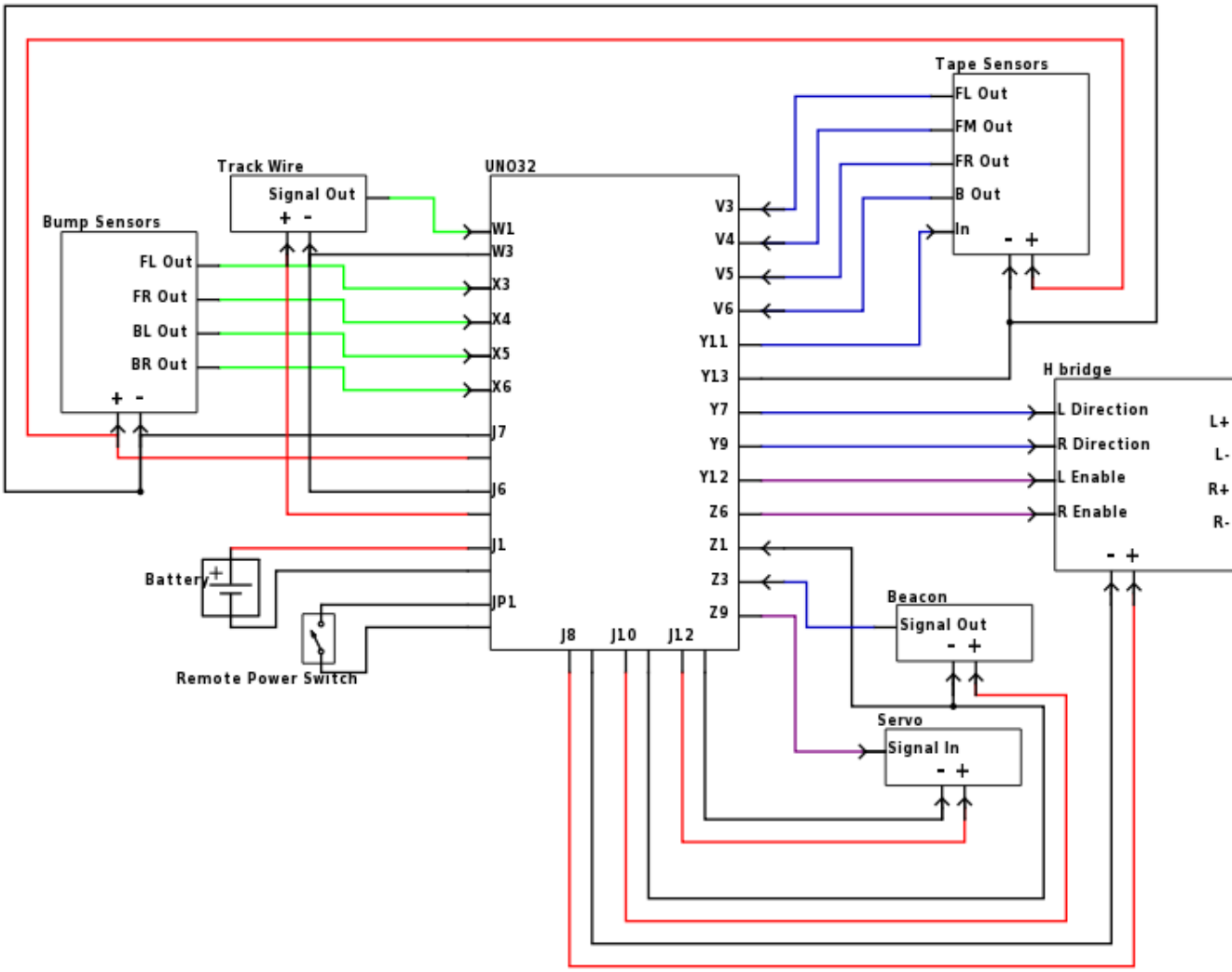


Figure 5: Our block diagram for the PIC32

3.1 Strategy

After completing individual electrical components, we would immediately come together to begin integrating that part before moving on to the next task. Making a test harness for each component was helpful for debugging the electrical system later on in the project. By each working on separate circuits, but coming together to debug when issues arose, we were able to complete our soldering in a timely manner and with little difficulty.

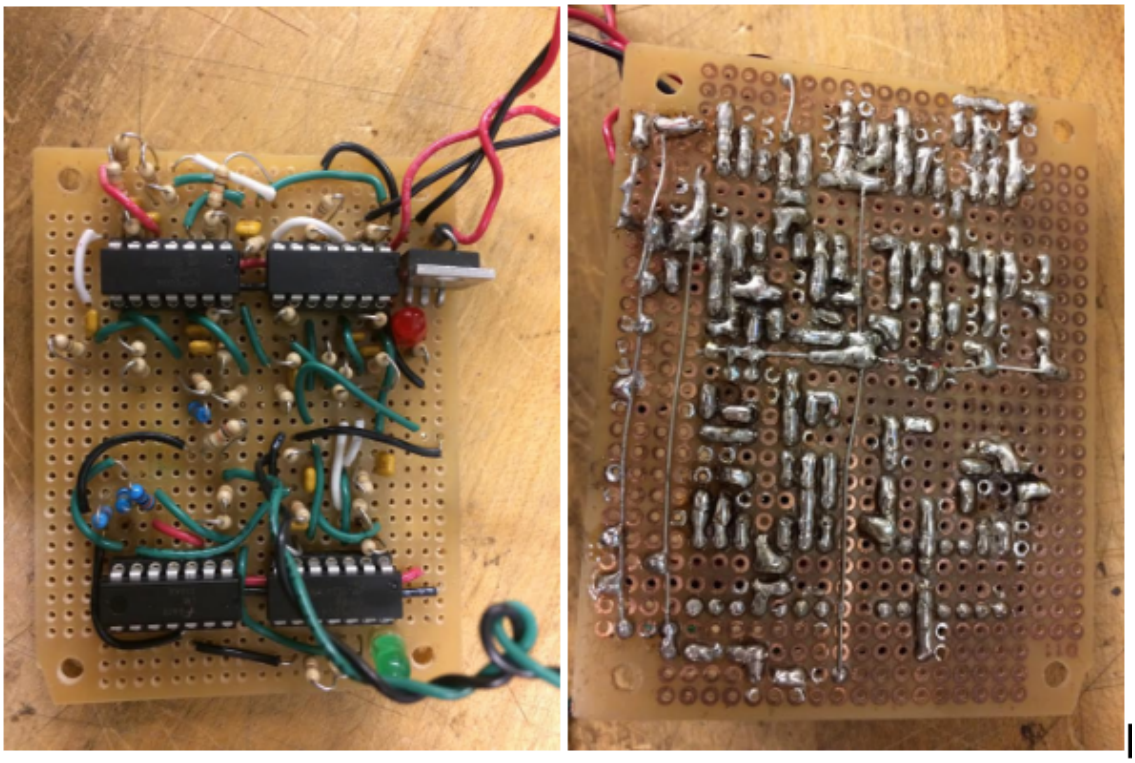


Figure 6: Beacon detector Schematic

3.2 Beacon Detector

We decided to use a band pass filter in our beacon detector. We used a 6^{th} order Butterworth filter. We had two LEDs on the board, a red LED to indicate whether our board was getting power or not and a green LED for Detecting a beacon event. We used a total of four OpAmp ICs (MCP6004) on our board. Our beacon was able to detect a $2kHz$ from a distance of 7 feet and was able to neglect $1.5kHz$ and $2.5kHz$ signals at a distance of 1 foot.

Our Beacon detector has two functions. Because our bot could tape follow in either direction, one task was to detect which side the field was located on. We needed to know which side the field was so that we could differentiate between a bump event from a loading tower or from a dead bot on the tape. The other task is to find the targets on the field so that our bot could move towards to score.

After we had a fully working beacon detector, our main task was to make the detector more precise and directional. We tried shielding our photo transistor plastic case and electrical tape, but doing so reduced its range from 7 feet to 2 feet. As a result we need to come up with a different solution for making the detector more precise. Then we tried using layers of MDF with a small at the center so the photo transistor can receive the signal. This seemed to be working well as we got back our range of 7 feet. But the photo transistor was not able to receive the signal when it is nearing the beacon, this is because we used a circular hole which is shielding the signals coming from the top resulting in an decrease in our range in vertical direction. Then we came up with an idea of having a vertical slit that goes all the way to the top the MDF such that there are no obstructions blocking our detector's view in vertical direction.

3.3 Tape Sensors

We decided to use 4 tape sensors, 3 in the front for tape following and one in the back for target detection. The 3 in the front were in a triangle formation so we could distinguish tape at a 90 degree angle to the bot. We used male-female connectors to make the connection between the sensors and board removable. To ensure that the output to the Pic was below 3.3V, a 3.3V regulator was used on the 9.9V input. To enable synchronous sampling, the tape sensors were turned on and off using a digital output from the UNO. Since a single pin from the UNO could not supply enough current

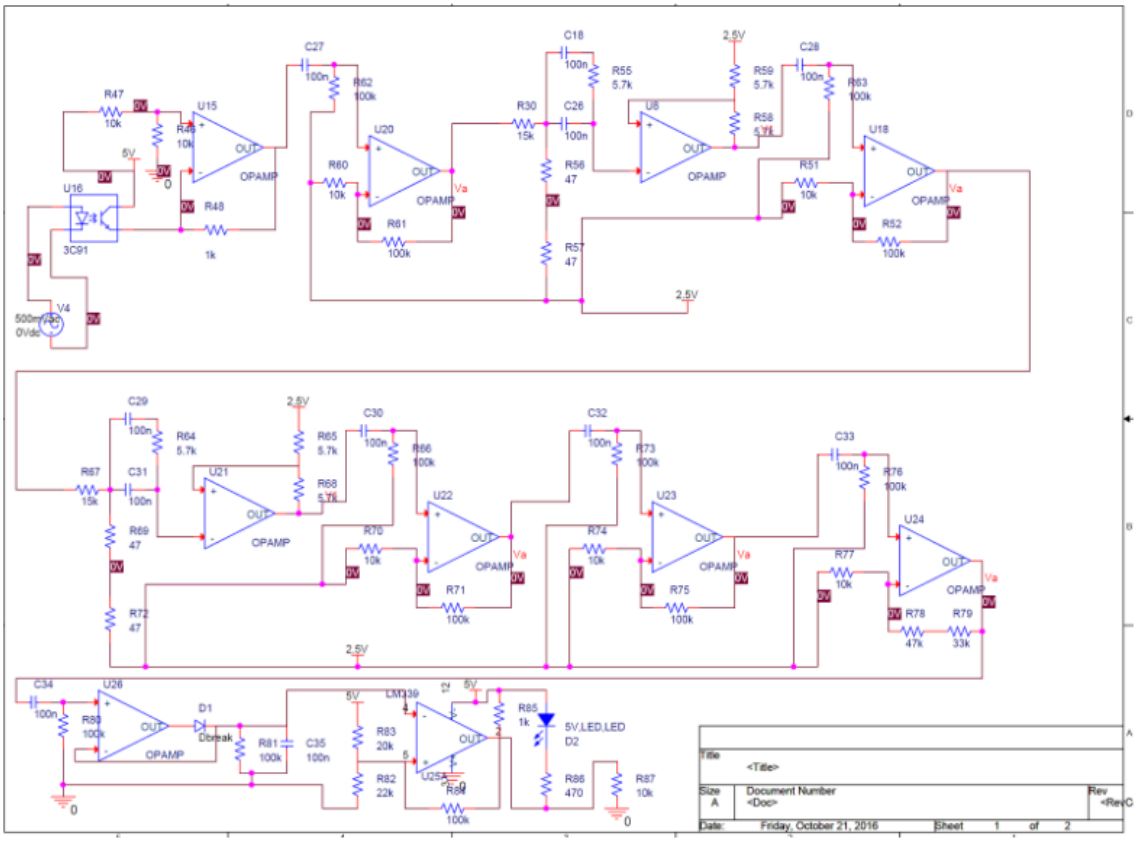


Figure 7: Beacon detector photo

to drive the tape sensors, a darlington(TIP122) was used to amplify the current. There were also four LEDs on the circuit that indicated if the tape sensors saw tape or not. We encountered many issues with our tape sensors, that often were a result of poor solder connections. Because the signal was variable, we used analog ports V3,V4,V5,and V6 (Figure 5). X11 was used to generate the digital signal to turn the LEDs on and off.

3.4 Bump Sensors

We used 4 bump sensors, for front left, front right, back left, and back right bumpers. We used male-female connections between the bump sensors and the circuit board so the connection would be removable. The bump sensor circuit was very simple, consisting of a current limiting resistor and the switch of the bumper. The signal was high when the bumper was pressed, and low when the bumper was depressed. This allowed us to use IO ports, X3, X4, X5, and X6. (Figure 5).

3.5 Trackwire

Initially we had planned to use two trackwire sensors to line up with the ball loader. In fact, we built and coded for two trackwire sensors. Only after 3 days of trying to make the bot line up properly did we look up and wonder how everyone else was doing it so perfectly. 15 minutes after deciding to try it out with one trackwire sensor and a timer, the bot was lining up to load perfectly. The final bot only used one trackwire sensor going into IO port Z1 (Figure 5).

The only issue we had with the circuit (Figure **??**) was the inductor itself. For some reason, 2/4 inductors we tried from BELS had an issue where they would only work if squeezed.

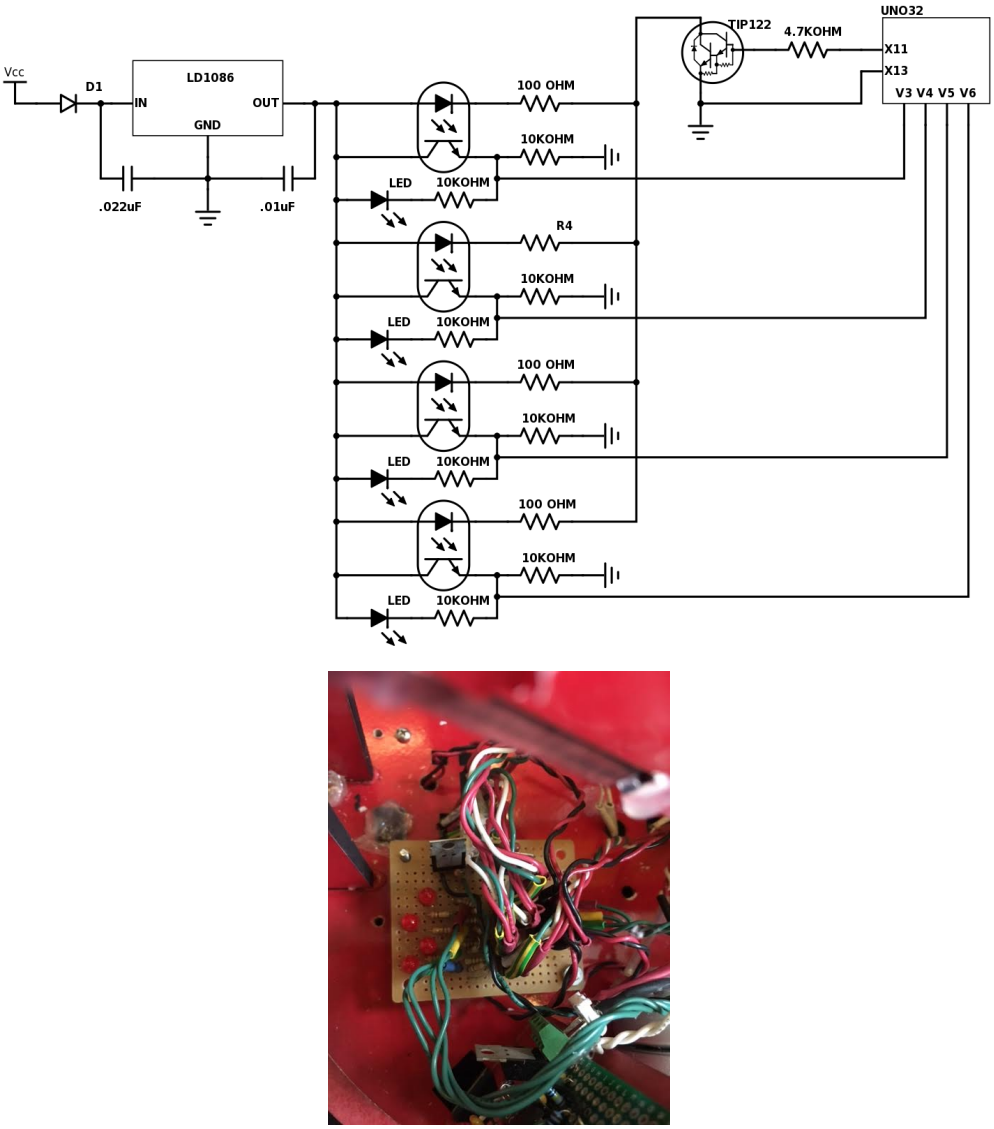


Figure 8: Tape Sensor Schematic (top), Photo of tape sensor and bump sensor circuit(bottom)

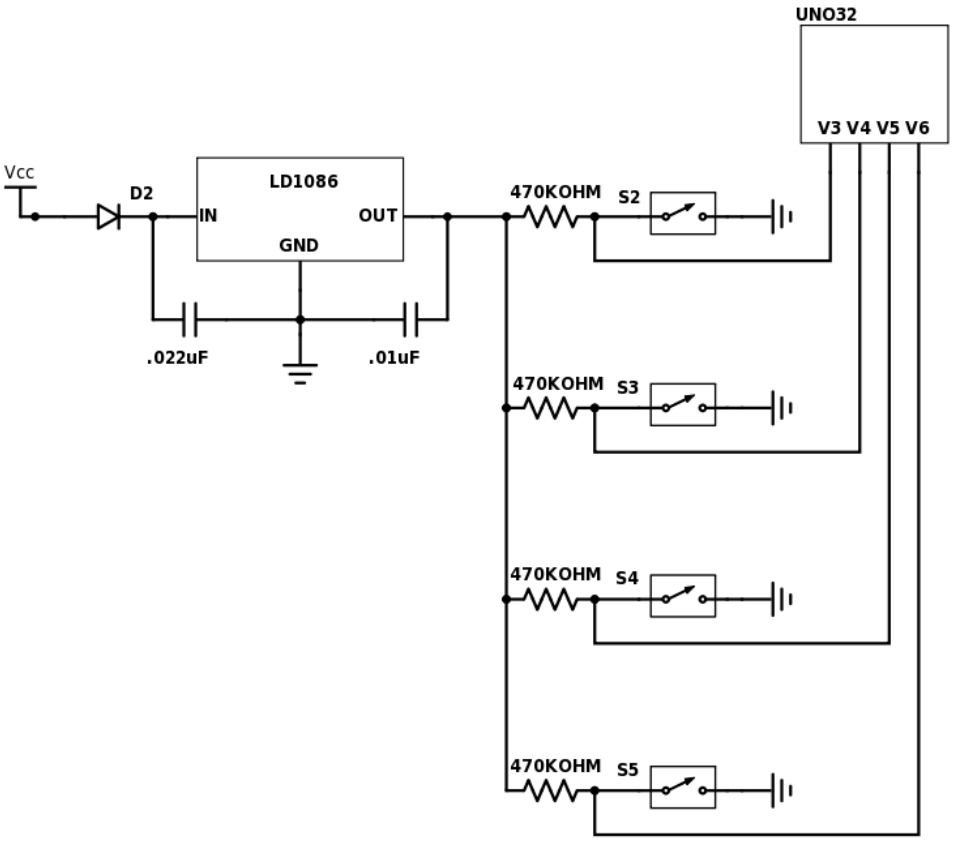


Figure 9: Bump Sensor Schematic

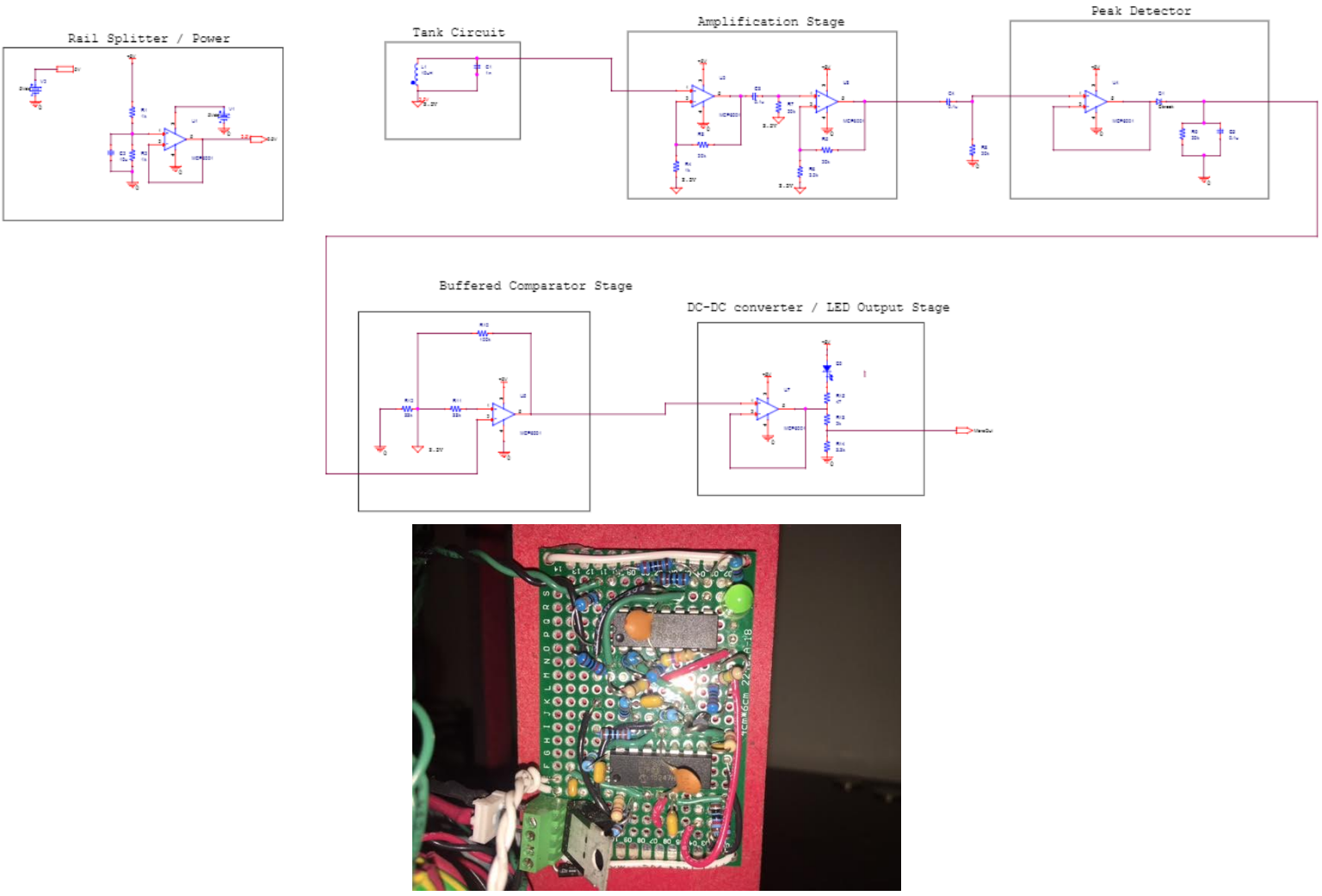
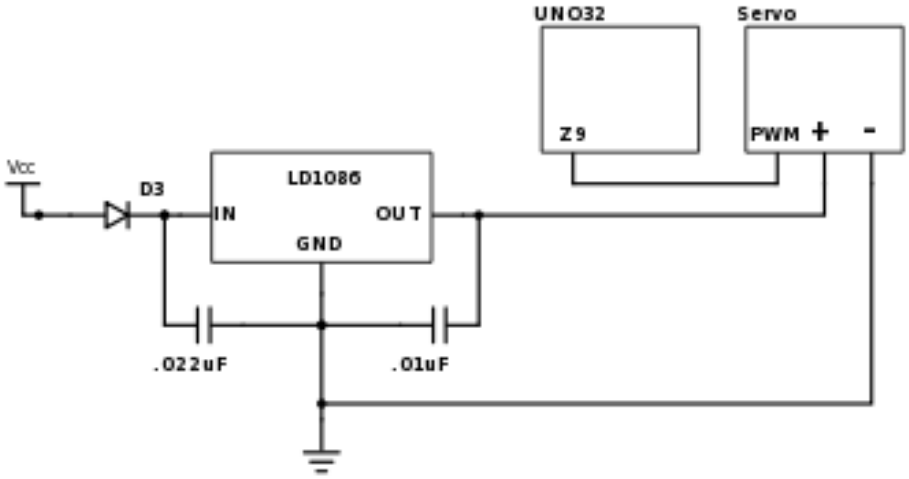


Figure 10: Trackwire Sensor Schematic and photo

3.6 Servo

For the servo we built a small circuit. This was necessary because not only did the Pic supply 9.9V of power where the servo could only take 5, but the pic also could not supply enough current to drive the servo without the addition of a voltage regulator.



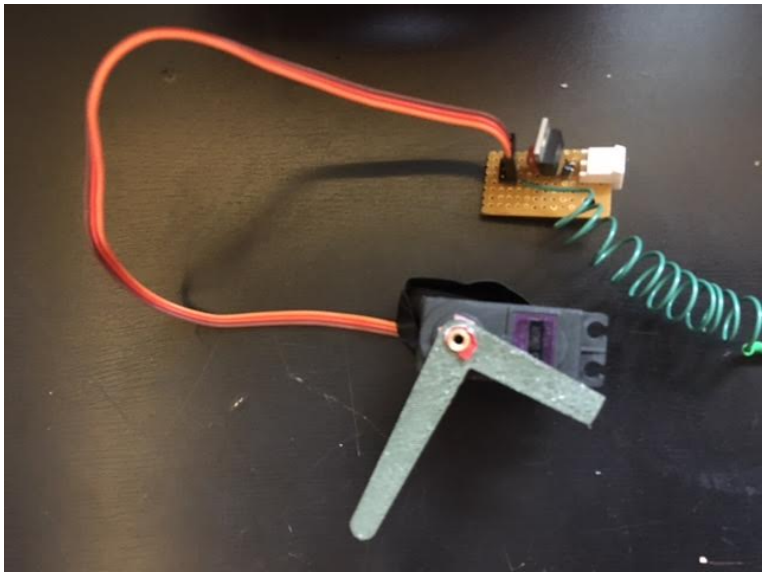


Figure 11: Servo Schematic and photo

3.7 Conclusion

We relied heavily on the functionality of our sensors and actuators in our state machine design. At times this caused us problems, such as finding our tape sensors weren't working, and realizing it was due to broken solder connections. We also encountered hardware issues when we attempted to develop a ball loading method where we used two rack wire sensors to line up with the loader. We found that this would not work, and ended up using only one track wire detector. We also found that our motors would cause our track wire detector, beacon detector, and servo to act unpredictably due to the motors' magnetic field. Through strategic wiring and placement of components we were able to work around these issues to robustly complete the task.

4 Software Design

Software design required converting the electrical signals input to the PIC32 into meaningful data. We also needed to create a multilevel state machine (Figures 12-17) in order to make the robot autonomously navigate the field without outside input. Our state machine waits for events and services to be triggered and then will change its "current state" based on them (i.e. if this, then do that). We used the Events and Services Framework provided by this course to implement our hierarchical state machine.

4.1 Strategy

Our strategy relied heavily on sensors, actuators, and timers. We decided to begin by having the bot spin 360 degrees and search for tape using the front three sensors. If it does not see tape while spinning, it will drive forward until it sees tape. Once the bot has found tape, it will begin tape following. If the bot has tape followed for two seconds, it will do a 360 turn to look for beacons and register if the field is to the left or right side. This information is used to determine if a bump event is the result of collision with the dead bot or a ball loader. If the bumper that is triggered is towards the outside of the field, then the object must be a ball loader. If both front bumpers are hit, or the bumper hit is towards the inside of the field, then the object hit must be a dead bot, since dead bots will always be at least half in the field. If the object hit is a dead bot, we turn around and continue to tape follow the other direction. If it is a ball loader, we look for a track wire. If a track wire is detected, the bot does a 45 degree turn and backs up to load the balls. if we do not see a track wire, we will begin to tape follow and continue to the next loader. To ensure that we always successfully loaded, we pull forward and back up three times. Once we have loaded, we pull away from the loader and turn 360 looking for beacons. If a beacon is spotted, the bot will back up towards the beacon until the back most tape sensor is on the tape around the target. If it loses the beacon signal, the bot turns back and forth until it finds the signal again. Once the target has been found, the servo will dispense two balls, then begin to look for a second target. To ensure that the bot does not see the same beacon and go to that target again, it turns 45 degrees and orbits the target. Once it sees a new target, it will go to that one and drop two balls. We decided to dispense 2 balls in the second hole just in case they had gotten stuck and one had not been dispensed in the first target. After the second target, the starts over by looking for tape and then following it to a ball loader.

4.2 Hierarchical State Machine Design

Our hierarchical state machine (HSM) was two layers deep. Figure 12 shows the top level of our HSM. States with their own sub state machines are depicted with Xs under their names.

Although not all states have sub state machine files, some of them have faux sub state machines using timers. For example: "Dead Bot," a state for avoiding a dead bot while line following, uses timers and static variables to retain memory of what stage of avoidance it was in each time the state is re-entered.Figures 13-17 show our true sub state machines.

4.2.1 Finding Tape

When the bot is first placed on the field, it goes into the "FindingTape" state machine (Figure 13). Here the bot does a full 360 degree turn while waiting for tape events. If it finds tape while turning, it will transition into "FollowingTape." This rules out the possibility that the tape sensors were placed outside the tape on the field. If no tape is found, it is safe to assume that the bot is somewhere in the middle of the field, so it continues forward to find tape. Because of the location of our tape sensors, we did not need to worry about accidentally following the tape around the targets.

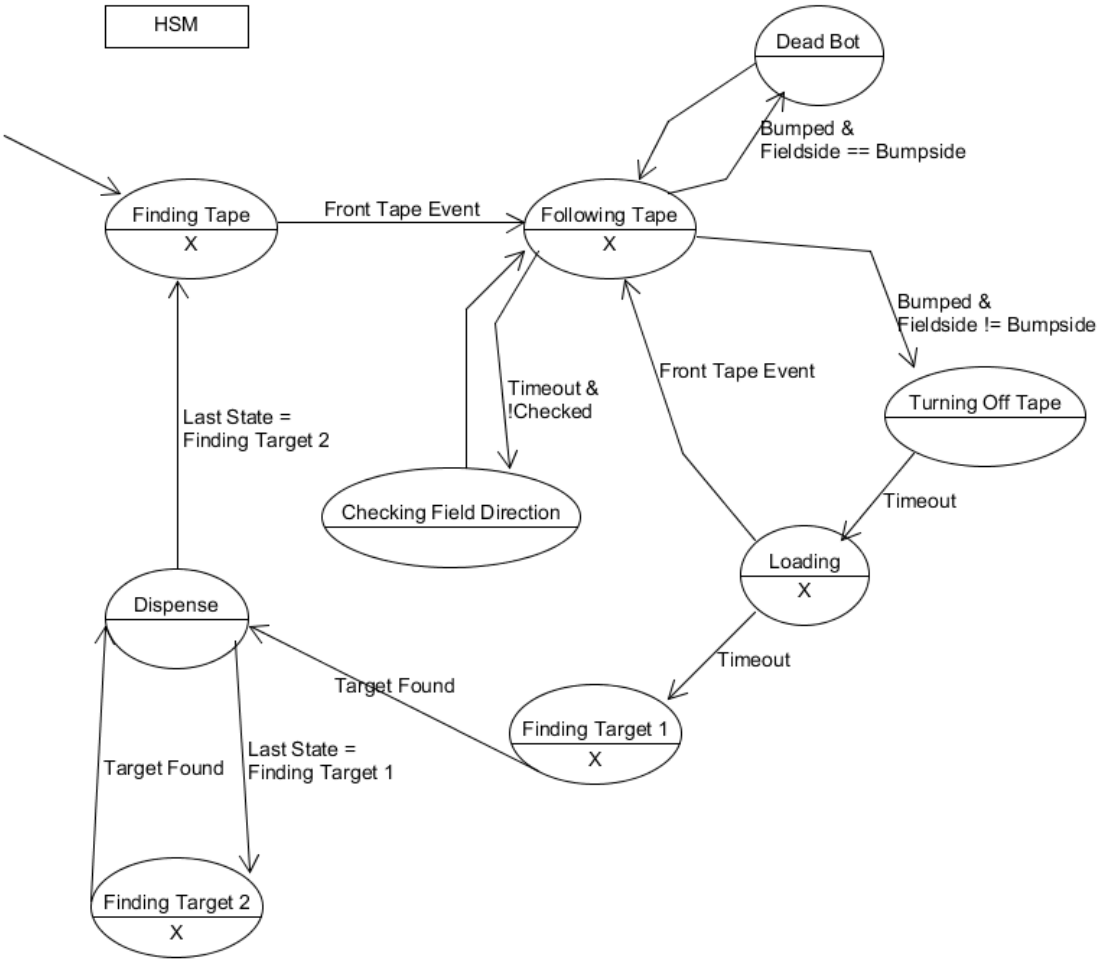


Figure 12: The top level of the Hierarchical State Machine

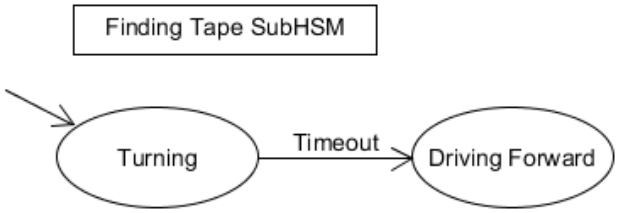


Figure 13: SubHSM for initially finding the tape when first placed on the field

4.2.2 Following Tape

This state machine simply tries to keep all 3 of the tape sensors on the tape. If all 3 are on, it tells the bot to drive forward. This state machine is interesting because any of the 5 states (Figure 14) can transition into any other state. Transitions are based on events from the three tape sensors. 111 indicates all three are on tape, while 101 indicates only the front sensor is off the tape. Our

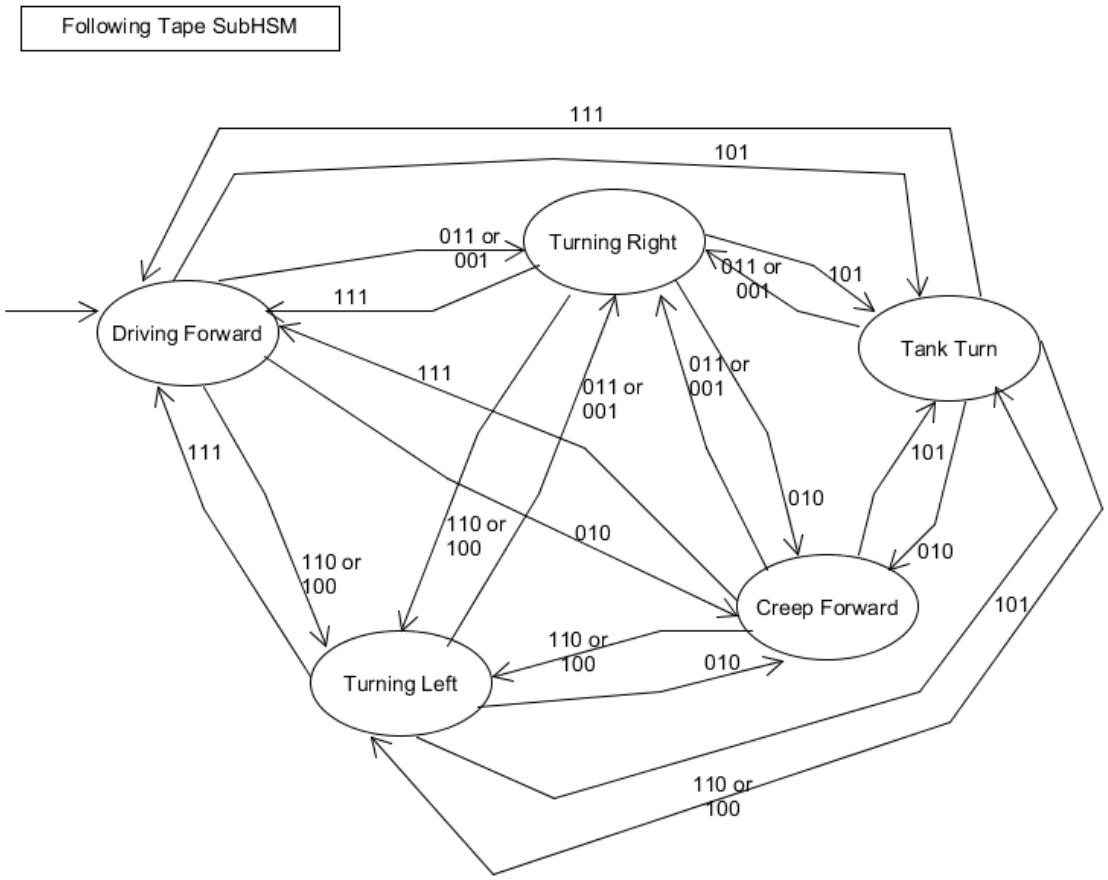


Figure 14: SubHSM for following tape

method for doing this in code is simply to have a case for each possible sensor variation within each state, shown in the code below (taken from the driving forward state):

```
switch (ThisEvent.EventParam & 7) {
case OOI:
case OII:
nextState = TurningRight;
makeTransition = TRUE;
ThisEvent.EventType = ES_NO_EVENT;
break;
case IIO:
case IOO:
nextState = TurningLeft;
makeTransition = TRUE;
ThisEvent.EventType = ES_NO_EVENT;
break;
case IOI:
nextState = TankTurn;
makeTransition = TRUE;
ThisEvent.EventType = ES_NO_EVENT;
```

```
break;
case OIO:
nextState = CreepForward;
makeTransition = TRUE;
ThisEvent.EventType = ES_NO_EVENT;
break;
case III:
nextState = DrivingForward;
makeTransition = FALSE;
ThisEvent.EventType = ES_NO_EVENT;
break;
default:
break;
}
```

4.2.3 Loading

Before arriving in loading we check if the ball loader is active. To do this, we drive off the tape at a 45 degree angle. If the trackwire sensor is not triggered, we return to following tape when there is a tape event (which would occur after the bot passes the ball loader). Only when the trackwire sensor is triggered does the bot enter the "Loading" subHSM (Figure 15)

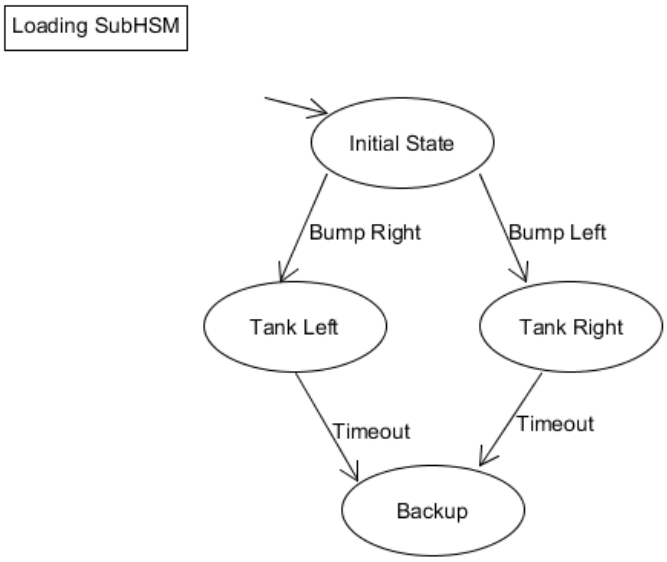


Figure 15: SubHSM for Loading

4.2.4 Approaching Targets

Our method for approaching each of the targets was essentially the same (Figures 16 & 17). The bot would spin in a circle, locating the beacon. If it could not locate the beacon from its current position, it would drive forward to a different vantage point and try again. When a beacon was detected, it simply backed up to it to dispense balls.

The only difference in finding the second target was that we had the bot "orbit" the initial target, by driving in a perfect arc (it took a few tests to get the exact motor speed correct). The bot would drive forward and tankturn to line up for orbiting. We created a static variable to keep track of if the first target's beacon had been placed out of view in order to avoid approaching the same target multiple times. The variable in conjunction with orbiting made it virtually impossible to approach the same target twice. Initially we had been using almost the same algorithm for the second target as the first, and we would frequently approach the same target twice. Orbiting also cut down significantly on the time needed to find the second target.

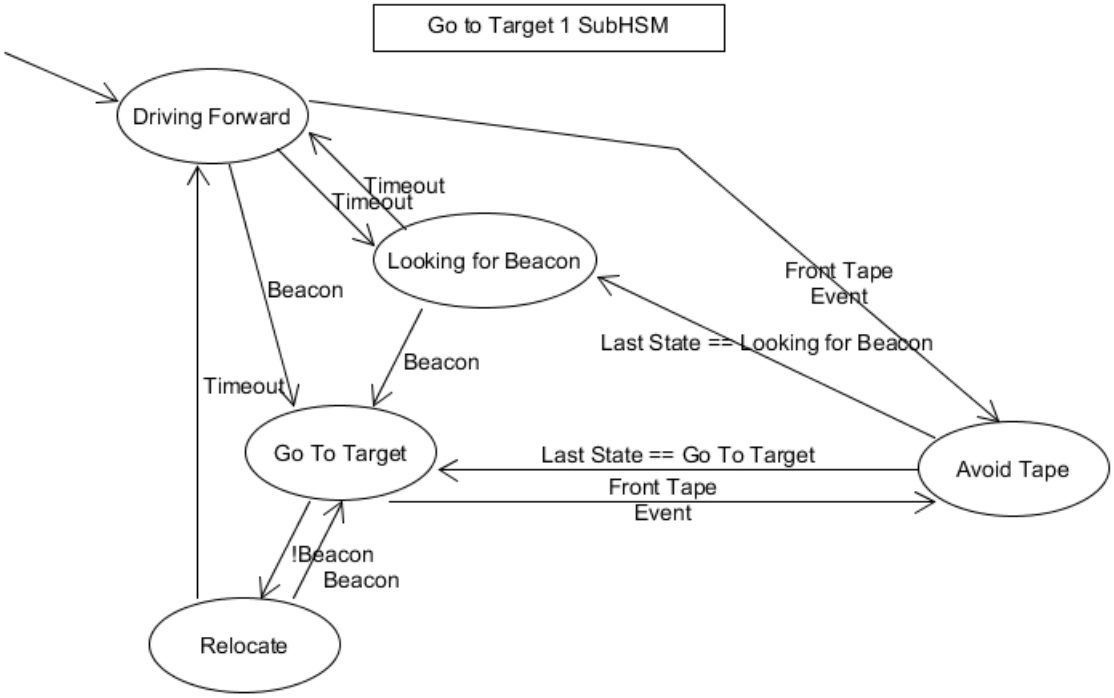


Figure 16: SubHSM for finding the first target

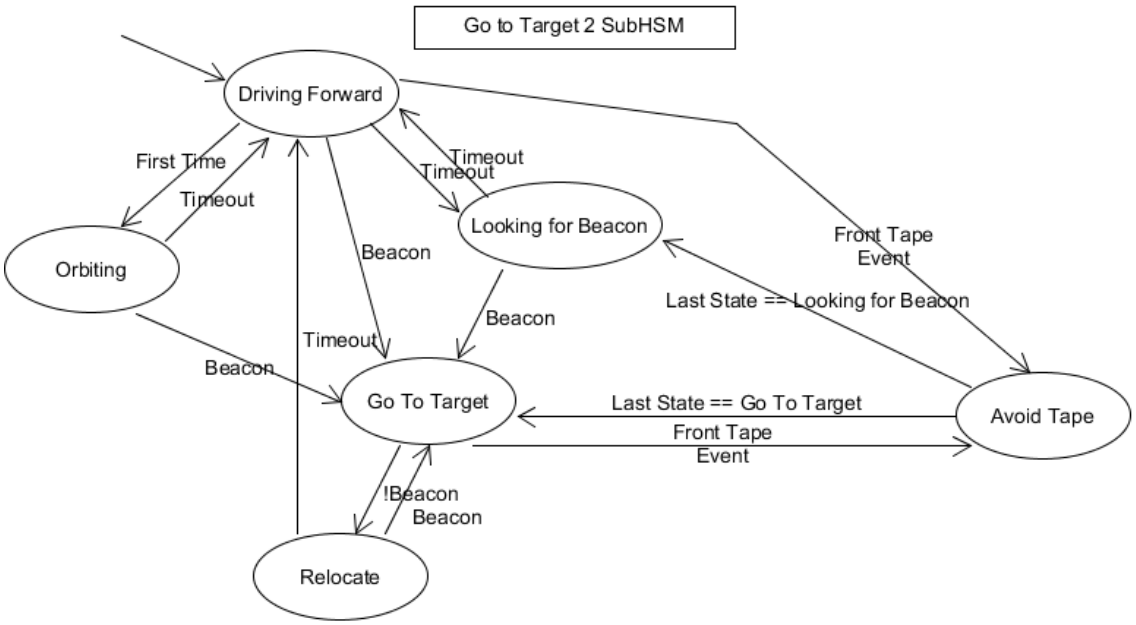


Figure 17: SubHSM for finding the second target

4.3 Sensor and Hardware Integration

We begin testing our sensors and motors by making test harnesses for them. Our tape sensor test harness proved to be very useful, since we often encountered problems while integrating them. The servo test harness was useful for determining the ball dispensing mechanism. We also made test harnesses for the motors, bump sensors, beacon detector, and track wire detector, but did not use them extensively.

4.3.1 Motors

To control the motors, we made a file, Motors.c that contained the functions for making the motors move forward, backward, tank turns etc. We also adjusted the motor speed according to the battery voltage using the helper function shown below:

```
unsigned int AdjustPWM(unsigned int speed) {
    float newSpeed = (float) speed;
    float thousand = 1000;
    float nine = 9.9;
    float tentwothree = 1023;
    float thirtythree = 33;
    float six = .6;
    float simVoltage = (speed / thousand) * nine;

    float batVolt = AD_ReadADPin(BAT_VOLTAGE);

    float newDuty = simVoltage / (((batVolt / tentwothree) * thirtythree)
    - six);
    newDuty = newDuty * 1000;

    unsigned int u_newDuty = (unsigned int) newDuty;

    if (u_newDuty > 1000) {
        u_newDuty = 1000;
    }
    return u_newDuty;
}
```

This function is called within each of our Motor functions to adjust the PWM that is sent in. For example, our "Drive Forward" function:

```
void DriveForward(unsigned int speed) {
    unsigned int newDuty = AdjustPWM(speed);
    PWM_SetDutyCycle(ENABLE_PIN_A, newDuty);
    PWM_SetDutyCycle(ENABLE_PIN_B, newDuty);
    IO_PortsClearPortBits(DIRECTION_PORT, DIRECTION_PIN_B);
    IO_PortsSetPortBits(DIRECTION_PORT, DIRECTION_PIN_A);
}
```

4.3.2 Tape Sensors

For the tape sensors, we used synchronous sampling to enhance their reliability and wrote our code as a simple service running every 5ms.The LED of the sensor was turned on and a sample was taken before turning off the LED and taking another sample. The difference between these two values was compared to a threshold with hysteresis. If the difference was below 50 (on a scale of 0-1023) then the sensor was on tape. If the difference was above 200 then the sensor was on the white MDF of the field.

To alternate between turning the LED on and off,we made a call to the CheckTapeSensors() function below using a counter to know when to set the new tape sensor value.

```
unsigned char CheckTapeSensors(int timerNumber) {
    unsigned char fl = CheckFrontLeftSensor(timerNumber);
    unsigned char fm = CheckFrontMidSensor(timerNumber);
    unsigned char fr = CheckFrontRightSensor(timerNumber);
    unsigned char b = CheckBackSensor(timerNumber);
    return (fr + (fm << 1)+(fl << 2)+(b << 3));
}

//--------------------------private functions------------------------------//

int CheckFrontLeftSensor(int timerNumber) {
    static int ReturnVal= 0;
    static int sample1;
    static int sample2;
    static int diff;
    static int Threshold = WHITE_THRESHOLD;

    switch (timerNumber) {
        case 0:
            IO_PortsSetPortBits(SYNCHRONOUS_PORT, SYNCHRONOUS_PIN);
            //turn on LED
            break;
        case 1:
            sample1 = AD_ReadADPin(FRONT_LEFT_PIN); //sample 1
            break;
        case 2:
            IO_PortsClearPortBits(SYNCHRONOUS_PORT, SYNCHRONOUS_PIN);
            //turn off LED
            break;
        case 3:
            sample2 = AD_ReadADPin(FRONT_LEFT_PIN); //sample 2
            diff = abs(sample2 - sample1);
            if (diff > Threshold) {//then there is a large difference,
            //indicating you are on whiteboard
                Threshold = BLACK_THRESHOLD;
ReturnVal = 0;
            } else {//if below threshold, then there is small difference
            //indicating you are on black tape
                Threshold = WHITE_THRESHOLD;
ReturnVal = 1;
            }
            break;
    }
    return ReturnVal;
}
```

The functions for the front right, front center, and back tape sensors are all very similar to the "CheckFrontLeftSensor" function shown above.

4.3.3 Bumper Sensors

For the bump sensors, we used digital pins and wrote a simple function to return a four bit character corresponding to each of the four bump sensors - a 1 for pressed and a 0 otherwise. We also implemented debouncing to prevent our bumpers from spamming events. Following is our debouncing service, which ensures that the bumper is held for four consecutive timeouts before an event is triggered. In conjunction with bitshifting, the case statement below lets any consecutive values for bumpMem fall through to be processed as an event, if it is not the same as the previous call to this service.

```
        case ES_TIMEOUT:
            ES_Timer_InitTimer(BUMPER_SERVICE_TIMER, TIMER_0_TICKS);
            bumpMem <<= 4;
            bumpMem |= currentBumpers;
            switch (bumpMem) {
                case 0x1111:
                case 0x2222:
                case 0x3333:
                case 0x4444:
                case 0x5555:
                case 0x6666:
                case 0x7777:
                case 0x8888:
                case 0x9999:
                case 0xAAAA:
                case 0xBBBB:
                case 0xCCCC:
                case 0xDDDD:
                case 0xEEEE:
                case 0xFFFF:

                    curEvent = BUMPED;
                    if (curEvent != lastEvent) {

                        ReturnEvent.EventType = BUMPED;
                        ReturnEvent.EventParam = currentBumpers;
#ifndef SIMPLESERVICE_TEST
                        PostHSM(ReturnEvent);
#else
                        PostBumperService(ReturnEvent);
#endif
                    }
                    lastEvent = BUMPED;
                    break;
                default:
                    lastEvent = ES_NO_EVENT;
                    break;
            }
```

4.3.4 Trackwire

For the track wire we used an analog pin. If the value to the pin was below 600 then a track wire was sensed, otherwise no track wire was sensed. We created an event checker that would give a track wire event with a parameter that indicates if it is a track wire sensed or no event.

```
uint8_t TrackEvent(void) {
    ES_Event thisEvent;
    unsigned char curTracks = CheckTracks();
    static unsigned char lastTracks = 0;
    uint8_t returnVal = FALSE;

    if (curTracks != lastTracks) {
        lastTracks = curTracks;
        thisEvent.EventType = TRACK;
        thisEvent.EventParam = curTracks;
        returnVal = TRUE;
#ifndef EVENTCHECKER_TEST
        PostHSM(thisEvent);
```

```
#else
        SaveEvent(thisEvent);
#endif
    }
```

The event checker for the trackwire detector is very simple. If the sensor value has changed, trigger an event. Below is the function for reading the trackwire sensor.

```
unsigned char CheckTracks(void) {
    unsigned char mid = 0, front = 0;
    mid = CheckMid();
    front = CheckFront();
    return (mid + (front << 1));
}

int CheckMid(void) {
    int trackVal = AD_ReadADPin(MID_TRACK_PIN);
    if (trackVal > 600) {
        return 0;
    } else {
        return 1;
    }
}
```

4.4 Beacon Detector

The integration of the beacon detector was similar to that of the track wire, except somewhat simpler since a digital port was used. We found that the beacon signal was high when it didn't see the beacon and low when it detected the beacon. We took this into account when designing our beacon helper functions, below.

```
int CheckBeacon(void) {
    if (IO_PortsReadPort(BEACON_PORT) & BEACON_PIN){
        return 0;
    } else {
        return 1;
    }
}
```

We first designed an event checker for beacon detector, but found we were getting many false events. To fix this, we debounced our beacon, creating a simple service that ran every 10ms.

```
case ES_TIMEOUT:
ES_Timer_InitTimer(BEACON_SERVICE_TIMER,TIMER_9_TICKS);
            beaconMem <<= 1;
            beaconMem |= currentBeacon;
            switch (beaconMem) {
                case 0xF:
                case 0x0:
                    curEvent = BEACON;
                    if (currentBeacon !=lastBeacon) {
                                    ReturnEvent.EventType = BEACON;
                        ReturnEvent.EventParam = currentBeacon;
                        lastBeacon= currentBeacon;
#ifndef SIMPLESERVICE_TEST
                        PostHSM(ReturnEvent);
#else
                        PostBeaconService(ReturnEvent);
#endif
```

```
                    }
                    lastEvent = BEACON;
                    break;
                default:
                    lastEvent = ES_NO_EVENT;
                    break;
            }

            break;
```

4.5 Servo

We chose to use a servo to dispense and retain balls. For this we utilized the RCServo.c and .h files, making our functions very simple. We needed a ReleaseBall() function to release one ball, and a second function RetainBall() that would turn the servo to its original position while preventing remaining balls from rolling down the ramp. The functions were called using timeout events in the HSM, since the servo needed a delay to move to each position.

```
void RetainBall(void){
    RC_SetPulseTime(RCPIN,2000);
}
void ReleaseBall(void){
    RC_SetPulseTime(RCPIN,1100);
}
```

The functions were called using timeout events in the HSM, since the servo needed a delay to move to each position. We used a static variable to determine which stage of dispensing we should go to next.

```
case ES_TIMEOUT:
   if (ThisEvent.EventParam == BALL_TIMER) {
          if (ball1Retained == 0 && ball2Released == 0) {
                RetainBall();
                ball1Retained = 1;
                ThisEvent.EventType = ES_NO_EVENT;
                ES_Timer_InitTimer(BALL_TIMER, BALLTIME);
          } else if (ball1Retained == 1 && ball2Released == 0) {
                ball1Retained = 0;
                ReleaseBall();
                ball2Released = 1;
                ThisEvent.EventType = ES_NO_EVENT;
                ES_Timer_InitTimer(BALL_TIMER, BALLTIME);
          } else if (ball2Released == 1) {
                ES_Timer_StopTimer(BALL_TIMER);
                ball2Released = 0;
                RetainBall();
                if (lastState == FindingTarget1) {
                     nextState = FindingTarget2;
                } else if (lastState == FindingTarget2) {
                      nextState = FindingTarget1;
                      ReInitFindingTarget2SubHSM();
                      ReInitFindingTarget1SubHSM();
                      ball1Retained = 0;
                      ball2Released = 0;
                }
                makeTransition = TRUE;
                ThisEvent.EventType = ES_NO_EVENT;
           }
   } else if ((ThisEvent.EventParam == LOAD_TIMER) ||
```

```
    (ThisEvent.EventParam == TURN_OFF_TAPE_TIMER)) {
          StopMotors();
    }
    break;
```

4.6 Conclusion

Software design was a very large part of the project. It was vital that we had all of our components working with their own test harnesses so that we could easily make .h files or events and services with them. Overall, our team did a really good job of individually understanding the code so that we could each add functionality to our robot, regardless of which team members were present at the computer. It allowed for a versatility which would not otherwise be possible.

If we were to do it again, it would be beneficial to spend time revisiting the State machine on a regular basis to draw some of the unforeseen stages in before trying to implement them without consulting the HSM. It would also be beneficial to buy faster motors in the beginning. A lot of our last hours of coding were spent trying to make our bot complete the task in under 2 minutes.

5 Administrative

Keeping a friendly team dynamic is something that a lot of teams struggled with. Team frustration is frequently caused by one or more members not meeting unspoken expectations. To avoid this, we kept communication open and set work expectations while discussing our When2Meet poll (Figure: 18). We decided that averaging 6 hours a day minimum on mechatronics was a good goal, though we ended up averaging much more time per day. Our dynamic was also strengthened by going out for team dinners every once in awhile. Overall we were extremely lucky to get a hard-working and good-spirited team.

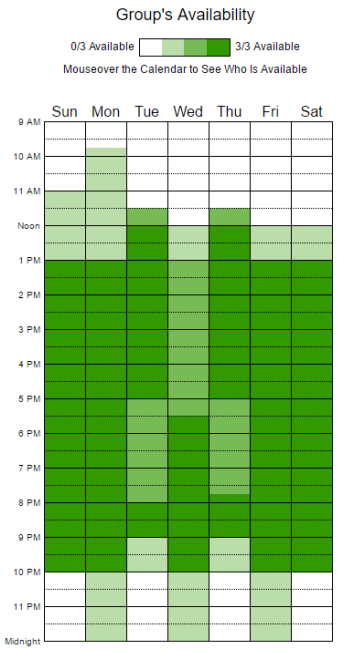


Figure 18: A When2Meet poll showing the times of day that all of us could be in lab at the same time.

For this project we were given a $150 spending limit. We ended up spending $80.96 on our final bot. Our bill of materials (Figure 19) consisted of parts purchased on Amazon Prime (great because of 2-day shipping), split with other teams buying in bulk, from BELS, and from ACE Hardware.

Name	# Needed	Notes	Unit Price	Total Price
DC Gearhead Motor	2	60rpm	13	26
Wheel	2	80mm diameter	6.5	12.99
Bump Switches	4		0.75	3
Servo	1		9.99	9.99
Assorted Protoboard	3	Price is approximate	?	2
Various Screws and Nuts	81		?	15.78
Female Power Connectors	7		0.5	3.5
MDF	3		1.5	4.5
Female 8 pin connector	4		0.4	1.6
Female 4 pin connector	4		0.4	1.6
			Total:	80.96

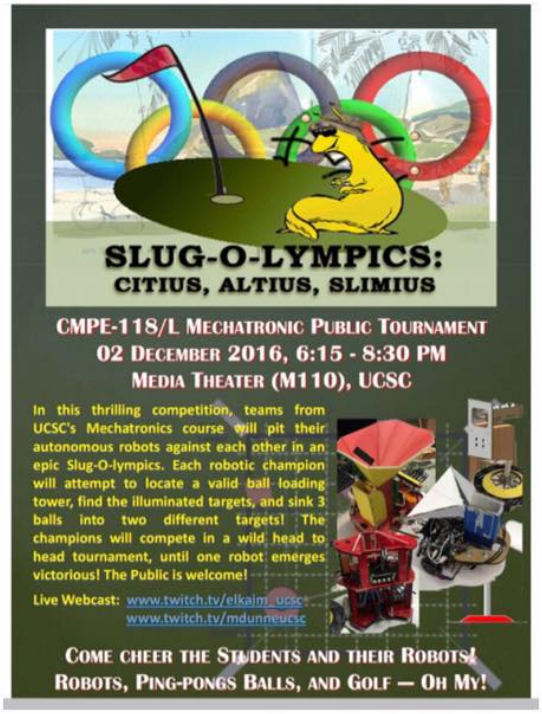
Figure 19: The bill of material for our final robot

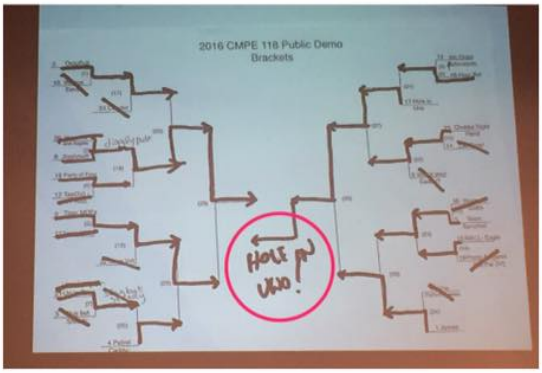
6 Conclusion

Mechatronics has done a good job of teaching us to fail - and to fail spectacularly. But, after every failure there was something to adjust to make the bot better. Hours were spent testing how the bot held up in edge cases and finding errors to fix. Even when we felt ready to check off, the bot let us down over and over. Errors were anything from not being too slow to not handling obstacles on the field correctly. Each time we failed, we were able to make the bot more robust for it. Some obstacles required more creativity to solve than any preliminary design could have offered. Many required us to backtrack and completely redesign components or code. After spending the most amount of time in lab of any class we've taken and so many late nights in lab correcting the work we thought was already finished, we came out victorious! We won the public demo and we have our failures to thank.

6.1 Thank You

Thank you to Professor Gabriel Elkaim, all of the TAs and Tutors, and the staff at BELS for making this class possible. It's been a great experience.





To watch a video of the last round at the competition, follow this link: **https://youtu.be/VNiMwP4Ju68**