ME 588 – Mechatronics
Instructor: Laura Blumenschein

Team 14 Final Report

Robotic Whack-A-Mole Competition

'Robert'

Date: May 2nd, 2022

Report Prepared by:

Rithvik Pillai, Yigit Karatas, Dylan Foster, Samvit Valluri

Contents

Abstract

The robotic system designed in this class participated in the ME 588 Robotic Whack-A-Mole competition and won third place out of twelve teams competing. The goal of this competition was simple: build a robotic system that can autonomously navigate the four-by-four multi-colored playfield pictured below in Figure 1, starting on the white square with a pre-set color input. The robot must deploy foam cubes on all of the selected color squares and return to the white square. The robot that deploys the most cubes correctly and does so the quickest wins the competition.

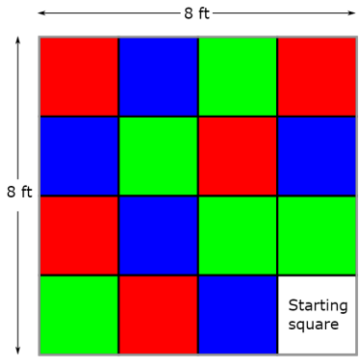

Figure 1: Top View of the Playfield

To accomplish these objectives, our team performed a functional decomposition and designed sub-systems for each of the major functions. The necessary functions were straight-line constant velocity travel, precise left/right rotation, square color detection, foam cube deployment, selection of target color, an input to start the navigation, and visual indication of specific functions. To accomplish each of these functions robustly, four sub-systems were designed by each team member, and integrated together in the final design. These four sub-systems are the drive sub-system which would perform straight line travel and precise rotation; the dispenser sub-system which would perform color detection and foam cube deployment; the input/lighting sub-system which would allow selection of target color, an input to start the motion scheme, and visual indication of these functions; and finally, the finite state machine which would control each of the sub-systems to achieve the design objectives.

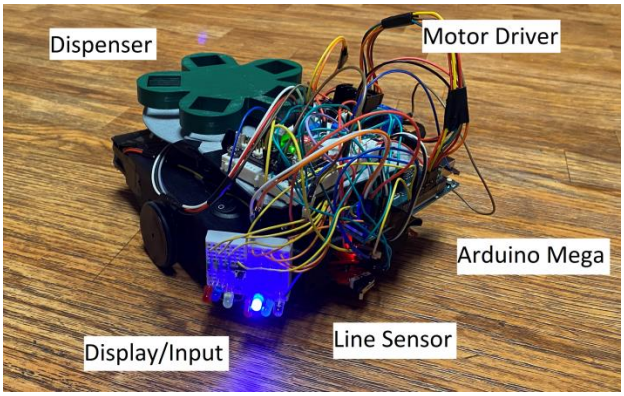

Figure 2: Final Robot Prototype

Robot Design

Drive Subsystem

With the strategy designed to attain the objectives above, it was very clear that the main functions that the drive subsystem would need to achieve were straight line constant velocity travel and precise left/right rotation.

The hardware selected for this 2-wheel differential drive system were: x2 AndyMark 12VDC motor-encoders (am-4338), x1 L298N H-bridge motor driver, x2 10k potentiometers, x2 PLA wheels with rubber O-rings, x1 castor ball, an Arduino Mega micro-controller, and a 12V Li-Ion battery. All of these parts were attached to the ME375 Skitter Robot Chassis as seen below in Figure 3. For exact wiring and connections, refer to the motor driver & motor encoder sub-systems in the main electrical schematic in Appendix A.

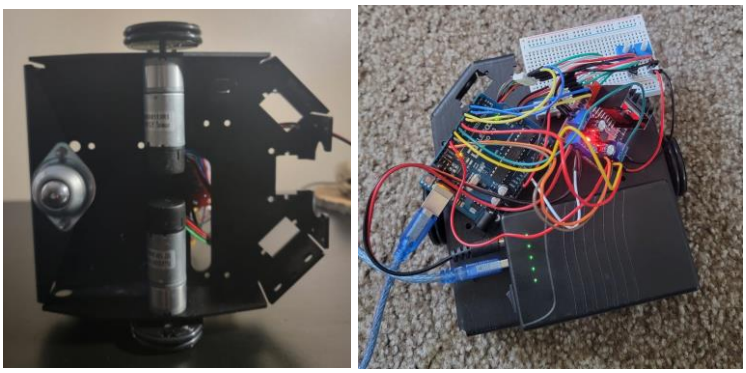

Figure 3: Early Drive System Prototype (left – bottom view, right – top view)

The first step in controlling the motors is to *control the actuators* by building out basic open-loop control functions to control direction and speed of each motor using the motor driver. The functions were written in Arduino IDE, take a PWM input from -255 to 255, and output the correct values to the motor driver pins.

```
void motorL_drive(int pwm){
  if (pwm > 255){
    pwm = 255;
  }
  if (pwm < -255){
    pwm = -255;
  }
  int pwm_val = (int) fabs(pwm);
  if (pwm == 0 || vt == 0){
    digitalWrite(in1,LOW);
    digitalWrite(in2,LOW);
  }
  if (pwm > 0){
    digitalWrite(in1,LOW);
    digitalWrite(in2,HIGH);
    analogWrite(enA,pwm_val);
  }
  if (pwm < 0){
    digitalWrite(in1,HIGH);
    digitalWrite(in2,LOW);
    analogWrite(enA,pwm_val);
  }
}
```

```
void motorR_drive(int pwm){
  if (pwm > 255){
    pwm = 255;
  }
  if (pwm < -255){
    pwm = -255;
  }
  int pwm_val = (int) fabs(pwm);
  if (pwm == 0 || vt == 0){
    digitalWrite(in3,LOW);
    digitalWrite(in4,LOW);
  }
  if (pwm > 0){
    digitalWrite(in3,HIGH);
    digitalWrite(in4,LOW);
    analogWrite(enB,pwm_val);
  }
  if (pwm < 0){
    digitalWrite(in3,LOW);
    digitalWrite(in4,HIGH);
    analogWrite(enB,pwm_val);
  }
}
```

Figure 4: Motor Driver Control Functions

The next step in controlling the motors is to *measure feedback* from the quadrature encoders attached to the motor shafts. To do this, one of the channels of each encoder is connected to a system interrupt pin on the Arduino. This allows for quicker reading of the channels and consequently higher-quality control.

```
// Library Declarations
#define ENCODER_OPTIMIZE_INTERRUPTS
#include <Encoder.h>
```

Figure 5: Encoder Library Declaration

An encoder library 'Encoder.h' is used for this application as it uses atomic blocks and interrupts to ensure fast and accurate readings. The 'Encoder_Optimize_Interrupts' definition enables the library to use an optimized version of the library specifically for interrupt-enabled reading.

```
// Left Motor (A)
#define enA 11 // ENA is the PWM input to Left Motor (A)
#define in1 10 // This actuates the backwards direction of the Left Motor (+)
#define in2 9 // This actuates the forward direction of the Left Motor (-)
float motorL_pos = 0;
float motorL_posPrev = 0;
float motorL_vel = 0;
float motorL_velPrev = 0;
float motorL_VFilt = 0;

// Right Motor (B)
#define enB 6 // ENB is the PWM input to the Right Motor
#define in3 8 // This actuates the forward direction of the Right Motor (+)
#define in4 7 // This actuates the backwards direction of the Right Motor (-)
float motorR_pos = 0;
float motorR_posPrev = 0;
float motorR_vel = 0;
float motorR_velPrev = 0;
float motorR_VFilt = 0;
```

```
// Encoder Declarations
Encoder leftenc(2,4);
Encoder rightenc(3,5);
volatile float motorL_encoder = 0;
volatile float motorR_encoder = 0;
long prevT = 0;
long motorL_enc_curr = 0;
long motorR_enc_curr = 0;
long enct_left = 0;
long enct_right = 0;
```

Figure 6: Feedback Variable Declarations (left) – Encoder Instance Declarations (right)

In Figure 6, the feedback variables and encoder instance declarations are shown. Note that the encoder readings are specified as 'volatile float' variables to ensure maximum optimization. The 'motorX_enc_curr' and 'enct_X' variables are used in the gain scheduling techniques for switching between straight line travel and rotation and will discussed in the Finite State Machine section.

The position and velocity variables are used to store current information about the motors. Note that there are also variables set for their previous values and filtered values: these are used for the velocity state estimation and filtering that will be discussed below.

```
// ************** MEASUREMENT ********************** //

// Read Feedback from Encoders
motorL_encoder = -1*leftenc.read();
motorR_encoder = 1*rightenc.read();

// Convert Position to Revolutions
motorL_pos = motorL_encoder / (230);
motorR_pos = motorR_encoder / (230);

// Convert to RPM
long currT = micros();
float deltaT = ((float) (currT-prevT))/1.0e6;
float motorL_vel = 60*(motorL_pos - motorL_posPrev)/deltaT;
float motorR_vel = 60*(motorR_pos - motorR_posPrev)/deltaT;
motorL_posPrev = motorL_pos;
motorR_posPrev = motorR_pos;
motorL_velPrev = motorL_vel;
motorR_velPrev = motorR_vel;
prevT = currT;

// Discrete Low-Pass Filter (25 Hz Cutoff)
motorL_VFilt = 0.854*motorL_VFilt + 0.0728*motorL_vel + 0.0728*motorL_velPrev;
motorR_VFilt = 0.854*motorR_VFilt + 0.0728*motorR_vel + 0.0728*motorR_velPrev;
```

Figure 7: Measurement of Encoders – Velocity State Estimation & Filtering

At the beginning of the main loop in the Arduino code, the encoder feedback is read using the ‘Encoder.read()’ function from the ‘Encoder.h’ library. With some experimentation in the Serial Window, it was found that the encoders read in opposite directions, hence the sign flip of the left encoder to ensure correct directionality. To obtain a conversion from encoder ticks to revolutions, the number of ticks per revolution was experimentally measured using the Serial Window and was found to be 230 ticks/rev.

To estimate the velocity state, a microsecond timer is started using the Arduino IDE’s ‘micros()’ function and the elapsed time at the beginning of the loop is recorded in the variable ‘currT’. At the end of the computation, the current time value is stored as the previous time in ‘prevT’, allowing us to determine the change in time over the computation. Similarly, the current/previous storage process is completed for both the position and velocity variables for each motor every computation. The computation itself simply computes the current motor velocity as a function of the current and previous motor positions and elapsed times.

This method is very intuitive and common in velocity state estimation; however, it can introduce large amounts of noise in the velocity signal due to the encoder pulsing. To solve this problem, a simple averaging filter is introduced to filter out the high-frequency noise components. This works extremely well but introduces a fair amount of delay into the system which may have repercussions in the controls design.

```
// *** PI Control *** //
// PI Controller for Left Motor
e_int_L = e_int_L + (e_L * deltaT);
e_der_L = (e_L - e_prev_L)/deltaT;
e_prev_L = e_L;
motorL_pwm = (Kp_L * e_L) + (Ki_L * e_int_L)+ (Kd_L * e_der_L);
motorL_drive(motorL_pwm);

// PI Controller for Right Motor
e_int_R = e_int_R  + (e_R * deltaT);
e_der_R = (e_R - e_prev_R)/deltaT;
e_prev_R = e_R;
motorR_pwm = (Kp_R * e_R) + (Ki_R * e_int_R) + (Kd_R * e_der_R);
motorR_drive(motorR_pwm);
```

Figure 8: PID Control Algorithm

A simple PID control algorithm is developed to control the motors according to a measurement value and a setpoint. The difference between the setpoint and feedback measurement value is stored in the variable 'e_X'. Note that in the main loop, the error variable 'e_X' has not been set. This is because it is defined in the functions below to be called in the finite state machine.

Gain scheduling is a technique where the main control algorithm runs consistently in a main loop, while the characteristics of the algorithm are modified according to the current state (in a finite state machine). This is an efficient way to switch between modes of travel, in this case, straight line velocity PI control and precise rotation encoder PID control.

```
void resetError(){
  e_L = 0;
  e_int_L = 0;
  e_der_L = 0;
  e_prev_L = 0;

  e_R = 0;
  e_int_R = 0;
  e_der_R = 0;
  e_prev_R = 0;
}
```

```
void straightlineControl(){
  e_L = vt - motorL_VFilt;
  e_R = vt - motorR_VFilt;
  vt = 150;
  Kp_L = 0.5;
  Ki_L = 4.5;
  Kd_L = 0.02;

  Kp_R = 0.5;
  Ki_R = 4.5;
  Kd_R = 0.02;
}
```

```
void turnControl(){
  Kp_L = 1.5;
  Ki_L = 0.2;
  Kd_L = 0;
  Kp_R = 1.5;
  Ki_R = 0.2;
  Kd_R = 0;
  if (turnDir[turnIndex] == 0){
    enct_left = motorL_enc_curr - 190; // Target Encoder Value corresponding to +90 deg
    enct_right = motorR_enc_curr + 195;
  }else if(turnDir[turnIndex] == 1){
    enct_left = motorL_enc_curr + 190; // Target Encoder Value corresponding to +90 deg
    enct_right = motorR_enc_curr - 190;
  }

  e_L = enct_left - motorL_encoder;
  e_R = enct_right - motorR_encoder;
}
```

Figure 9: Gain Scheduling Functions

Dispenser Subsystem

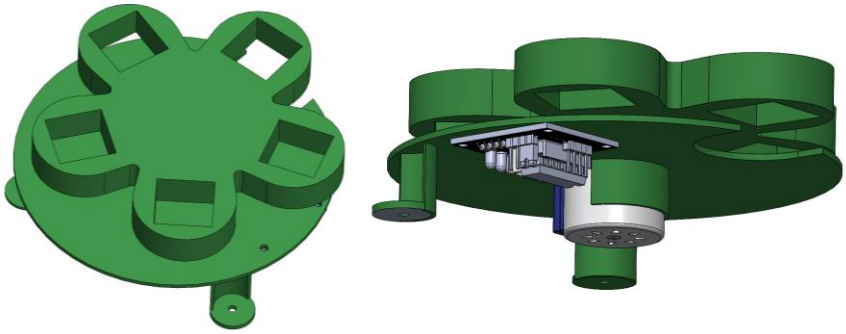

Figure 10: Dispenser Design, CAD (left – iso view, right – rear iso view)

The biggest motivation behind the design of the robot's mole-whacker dispenser was to maintain as much simplicity as possible, so as to ensure a mole-whacker was deployed whenever called upon to do so. The result of this was the design seen above, referred to as the *carousel*. The carousel consists of two custom pieces, and two pieces of hardware.

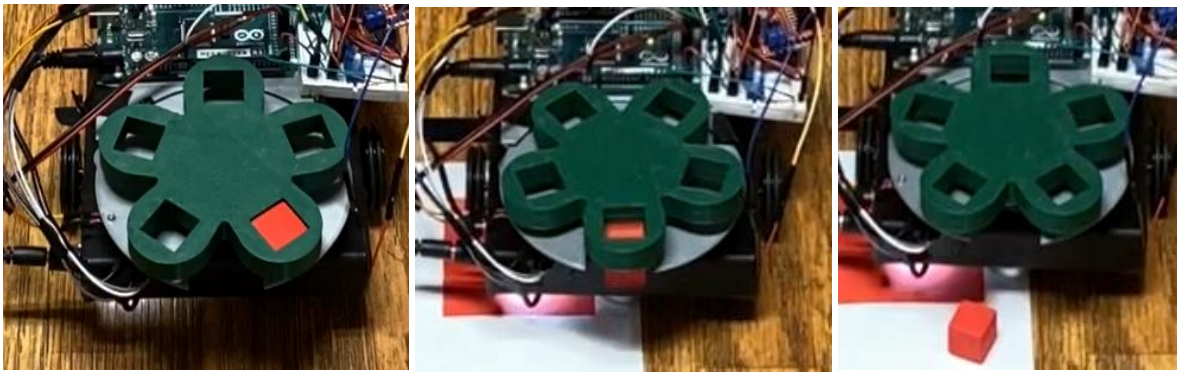

Figure 11: Dispenser before (left), during (center), and after (right) deploying a mole-whacker

Before a game is started, all five mole-whackers are placed within the square holes on the top piece, with the top piece positioned so that the moles are resting on the floor provided by the bottom piece. When commanded to deploy a mole, the top piece is driven to rotate 72°, causing only one mole to fall through the hole in the back of the bottom piece. To accomplish this, the top piece is connected to the shaft of a stepper motor (the 28BYJ-48), which is driven by a ULN2003 motor driver. By virtue of being a stepper motor, the angle of rotation was controlled by setting the number of steps taken to be approximately 72° (one-fifth of a complete revolution). The dispenser's actuation is illustrated in Figure 11.

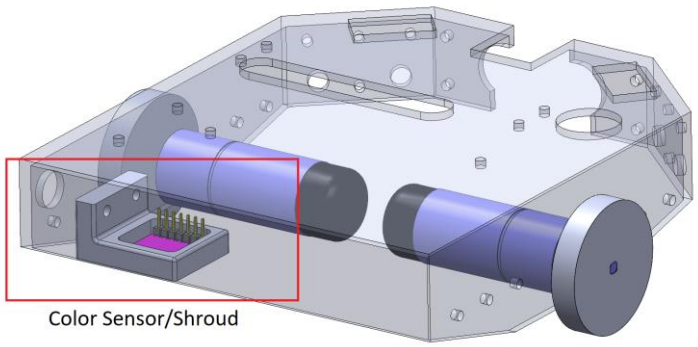

Figure 12: Placement and design of color sensor shroud

Integral to the dispenser's usage was the robot's ability to correctly detect and determine color, assumed to be either red, yellow, or blue. To accomplish this, a color sensor's (TCS34725) readings were used to set thresholds based on typical readings obtained experimentally. If a particular square's color reading passed a certain threshold, it could be reliably assumed to be the corresponding color. For example, if the amount of green reflected by a square was between a level of 3000 and 5000, it was likely the color yellow. To further ensure that readings were consistent between different environments and minor misalignments, a shroud was designed to provide optimal light and was positioned on the robot so that there was a minimal gap between the sensor and the ground.

Finite State Machine

The highest-level behavioral control of the robot is done using a finite state machine. There are 7 states that maintain the control of the robot. The problem could have been solved with less states but for the sake of simplicity and robustness there are states that only provide a waited transition between other states to ensure proper readings were taken with the sensors.

There are also helper variables that help keep track of the robot's state that doesn't fit the standard FSM model. One of these is "linecount" variable that counts how many lines have been crossed since the last turn. This helps keep the robot in minimal states while also providing an adaptive behavior. Also, there are two arrays "lineCount" and "turnDir" that defines the path of the robot. Finally, "turnIndex" keeps track of how many turns have been made so far.

The reset state of the robot is "state 0" which is the rest state of the robot where it waits for the input color and start button to be pressed. This state is there to ensure the robot doesn't start moving before the user is sure the robot should start.

Next state or "state 1" is the forward drive state of the robot. In this state the robot sets a desired speed of 125 RPM for the PI controller for the motors and the only condition that leaves this state is the detection of a line by the line sensor. Once this is detected robot transitions to "state 2".

"State 2" is a transition state where all the errors of the PI controller is set back to 0, line count is incremented and a transition to "state 3" occurs.

"State 3" is the stopping and checking the color state. This state measured the color of the current square the robot is on. Then invokes the "dispense()" function if it is the right color. If not,

it does nothing. The transition for the state occurs after the measurement is made and checks if the "linecount" is equal to the current index in the "lineCount" array. If this check is false, the robot doesn't need to turn and transitions back to "state 1". If a turn is warranted, then the FSM transitions to "state 4".

"State 4" is the first of the turn states. This state drives the robot forward to the middle of the next square. This is done through the PI controller and exiting using a time condition. When 1.2 s of forward movement is achieved the FSM transitions to "State 5".

"State 5" is the turning state. First the gains of the motor controller are changed in preparation for the turn. Then the turning direction is checked from the "turnDir" array 0 being a left turn 1 being a right turn. When the direction is known, an error to the encoder counts is introduced, and the PID controller is given enough time to drive the error to 0. This effectively transitions the motor controller from a speed controller to position controller. When the turn is executed, the robot checks if all the turns are executed or not, indicating the path is complete. If the check is false, the robot transitions back to "state 1" if it is true, the robot transitions to "state 6".

The final state in this FSM is "state 6". This state stops the motors from moving to finish the competition. It also flashes the LEDs of the robot to indicate the end of movement.

The complete FSM all the transitions are given in the figure below. The inputs are in order: {start button, line detected, right color, all turns done}. This figure doesn't have the outputs as the outputs are a complex product of the gains of the PI controller and desired speed of it, it cannot be represented as the outputs of a FSM.

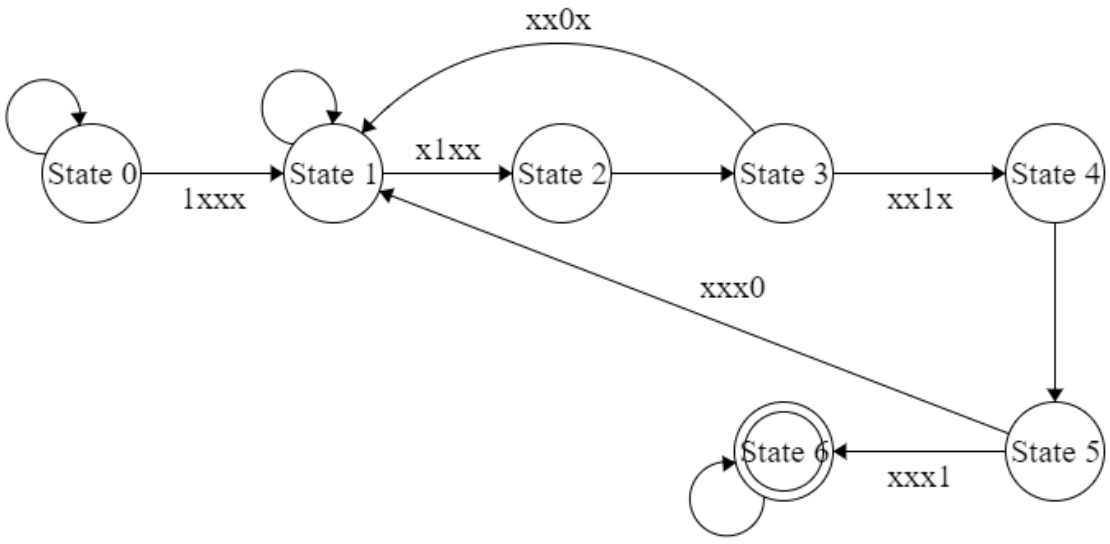

Figure 13: State Transition Diagram

Input & Display Subsystem

The inputs to the robots were separated into two different categories, the start button, and the color selector. The start button was a simple push button located on the front right breadboard. Once the button is pressed, the robot enters state 1 (drive state) from state 0 (idle state). The color selector was a 3 position DIP switch located on the top breadboard. Each pin on the DIP switch corresponds to one of the colors that will be present on the game board. The switch that is ON corresponds to the correct color that the robot will dispense on. For example, if the left most switch is turned on, the robot will dispense only on the red squares.

The display subsystem was located on the front right of the robot and displayed information such as the color the robot was searching for, the current color the robot is seeing, and if the robot is idle state. A total of 7 LEDs were used in the display subsystem. The first three LEDs (Red, Blue, Yellow) indicated the correct color. The next three LEDs (Red, Blue, Yellow) indicated the current color the robot sees. This updates every time the color sensor is called in the FSM, hence will update every time when stopping a line. The white LED indicates the game start. If the LED is off, it shows that the robot is in its idle state, and if lit up, it indicates the robot has started its path. Once the robot reaches the initial square, this LED will turn off indicating that it is once again in idle state.

Results & Discussion

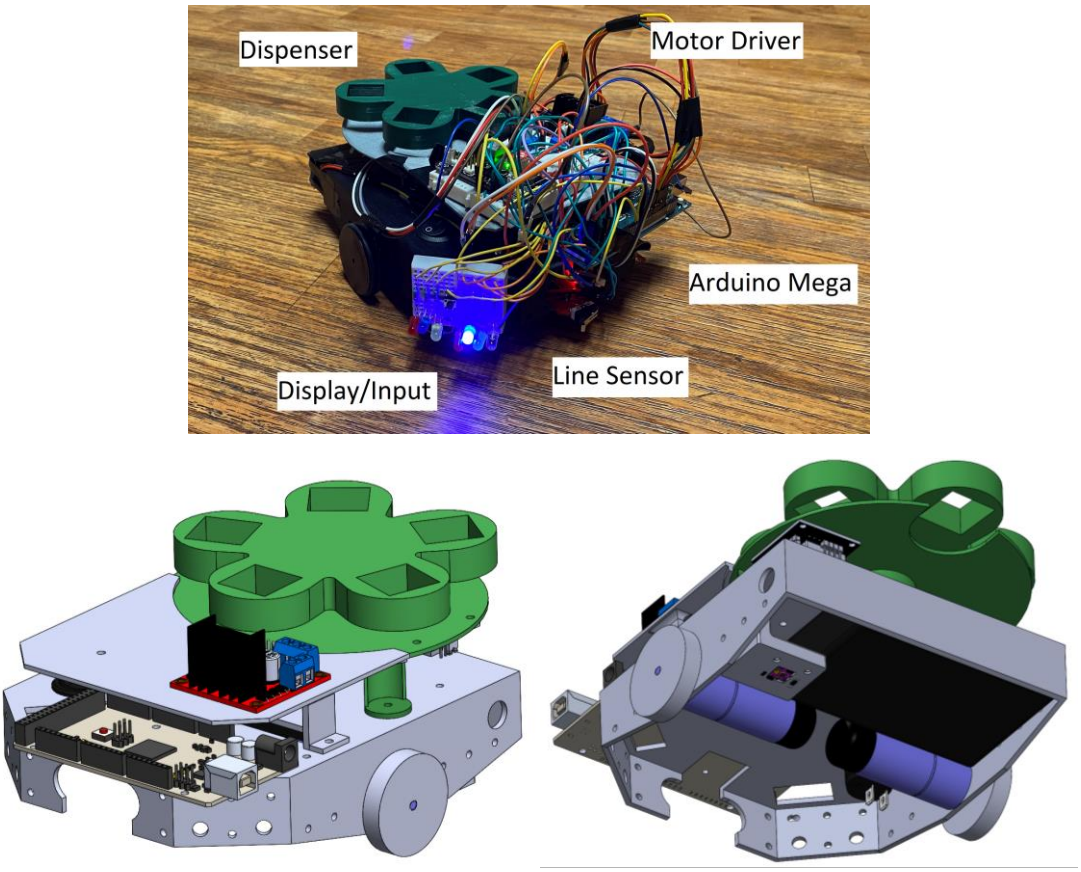

Figure 14: Final robot assembly (top) and final CAD assembly (bottom)

Combining each design element of the robot was no small feat in and of itself. This was due primarily to the constraint of reusing a robot chassis that was not designed for this application. As can be seen in the above figures, the size of the dispenser took up a considerable amount of space, which required much of the hardware to be placed at the front of the robot on the chassis and on a second level which attached the floor of the dispenser.

The team incorporated new subsystems incrementally by starting with the drive subsystem, followed by the line sensor, color sensor/dispenser, and finally with the input and display LEDs. As a result of this, the assembly process was at times tedious due to the somewhat untamed and growing collection of wiring that functionally held the robot together. One large caveat to this was that there was little to no redesign to the robot throughout the lifetime of this project. This helped to prevent the necessity for major teardown and reassembly once a subsystem was securely in place.

Functionally, the robot behaved exactly as intended from the beginning of the semester in terms of satisfying the designed behavior of the FSM. While it required a considerable time investment to tune the gains of the drive subsystem, build a reliable method in which to read colors, and create a state machine that correctly controlled the robot, the final results were more than satisfactory for the team. In an incredibly fortunate stroke of luck, the team was even able to place third in the competition, which speaks to the robustness of the robot's design and its capability to work in variable conditions. Video of one particular round of the competition can be found here, which shows how the team's success required each member's attention, understanding of their respective subsystem, and understanding of the robot's behavior overall to ensure a successful run that hit each mark. By involving each other in design choices as they were made and discussing their impact on the system as a whole, the team was able to work more efficiently than otherwise possible.

The major shortcomings of the design, which are complementary to the improvements the team wanted to make given more time, focus primarily on the reliability of the robot's driving given any starting position, and the time in which it took to finish. Specifically, the design and inclusion of a realignment state using the line sensor array at the front of the robot would have helped dramatically to prevent the robot from drifting off course, which it so often did. Getting around this in the short term required team members to account for the error accumulation by experimentally determining the optimal start position for the robot.

Another aspect of the design that had major room for improvement was the time in which it took the robot to actually complete the course, which was very close to the two-minute deadline. This was due in large part to stopping at each square to check its color, rather than checking color while continuously driving. The choice to do this was done because the team knew with a high degree of certainty that the deadline could be met successfully with a robot that stopped and checked. The additional time and possible redesigns that may have been involved in continuously driving may have pushed the team's deliverable past the project deadline. Regardless of these shortcomings, the team is more than satisfied with the robot's performance given the constraints involved with a decreased number of members on the team and a hectic semester for all involved.

Conclusion

The successful completion of the mole-whacker robot's design and fabrication was accomplished by working methodically as a team to make engineering decisions that would ensure a simple and robust system that could be assuredly delivered in the timeframe given. By decomposing the robot's functions and appropriately prescribing work to each team member, the team was able to work on each aspect of the robot concurrently. By defining the finite state machine early in the project's lifecycle, the team was able to design the drive, dispense, and input subsystems knowing exactly how they would have to functionally interface with one another. Because of this as well, the process of system integration was completed ahead of schedule, and allowed for additional time focused to testing and tuning.

The end result was a final deliverable that satisfied all requirements, was capable of operating in both lab and competition environments, and left the team with lessons learned that will be capitalized upon in future engineering efforts. These include the importance of wire management, knowing when to accept tradeoffs and constraints, and more effective methods of communicating design choices. The project's success was possible thanks to the help of the course's instructional team, the comradery among teammates, and a shared motivation for the subject of mechatronics.

Appendix A – Electrical Schematics