LockhEEd Mousetin
Autonomous Maze-Solving Robot

Andrew Collette
Ryan Horr
Cole Portman
Johan Rengifo

Table of Contents

Introduction 3

Key System Requirements 4

Detailed Project Description 6

- Power 6
- Microcontroller 8
- Sensors 9
- Motor 11
- Motor Driver 12
- Wheels 14
- Push Buttons 15
- I2C Connection 16
- Function LED 17
- Board Layout 18
- 3D Prints 21
- Uploading Code 23
- Software Development Overview 24
- Architecture 25
- Analog Sensing 27
- Motion Control 30
- Maze Search Algorithm 37

System Testing 44

- Practice Mouse and Initial Component Selection 44
- First Board Revision 46
- Second Board Revision 47
- Software Overview 48
- Analog Sensor Validation 49
- Motion Control Validation 50
- Maze Representation and Search Validation 52
- OLED Display Validation 53
- Integration Testing on the Assembled Board 54

Conclusions 55

Introduction

The objective for the electrical engineering senior design project is to create a robot, called a "mouse," which can autonomously navigate through a 16 by 16 unit maze of 180 mm squares to a 2 by 2 unit square in the center. The maze will be randomly generated on the day of the race. The mouse is not allowed to receive communications or controls during the competition, move over the walls in any way, or dislodge the walls of the maze from the base. Changes may not be made to the mouse or its code after the reveal of the maze. Mice will have 10 total minutes inside the maze, inside of which multiple runs can be made. The goal is to go from the starting square, a corner of the maze with three walls, to the end in the least amount of time possible. Time penalties will be assessed on mice which need to be picked up and replaced at the start.

It is under these requirements that LockhEEd Mousetin built their mouse for the micromouse competition. **Figure 1**, below, shows the mouse that LockhEEd Mousetin built for the competition.

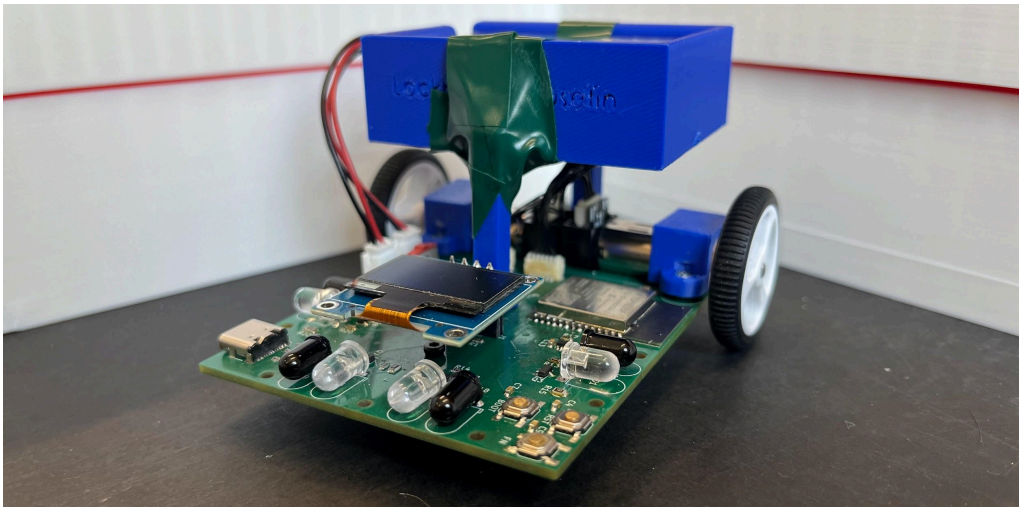

Figure 1. A photo of LockhEEd Mousetin's completed mouse

Key System Requirements

To abide by the rules of the micromouse competition described in the introduction and to give ourselves the best chance of solving the maze, we developed the following set of system requirements in different functional areas. In the next section, we will discuss the decisions we made to fulfill these requirements.

Sensing

- Mouse can sense walls and openings
- Mouse can sense distances of walls accurately up to 180 mm (1 unit length)
- Mouse can sample distances at 1000 Hz to allow quick updates in our control system
- Mouse can take inputs from the user on the board to switch between searching and solving modes

Movement

- Mouse can move
- Mouse can move a specified distance accurately up to 2880 mm (16 unit lengths)
- Mouse can move at various speeds
- Mouse can turn left or right 90°
- Mouse can turn 180°
- Mouse can stay centered between 2 walls
- Mouse can stay a set distance away from 1 wall
- Mouse can drive straight without walls to guide it
- Mouse can move quickly enough to explore the whole maze. This is about 1 unit square every 0.5s, enough for the mouse to explore the maze, drive back to the start, and drive the optimal path again
- Mouse does not hit walls
- Wheels should not slip

Debugging

- Mouse can display information for debugging
- Mouse has easily accessible test points for voltage testing
- Mouse can communicate with a computer via serial communication
- Mouse has a function LED to easily test if the board works

Code

- Mouse knows where it is in the maze
- Mouse knows when it reaches the goal of the maze
- Mouse knows where openings and walls are in the code
- Mouse can determine an optimal path from start to finish

Microcontroller

- Has a sufficient number of IO ports (both digital and analog) for the peripherals
- Has enough storage space for our program
- Has a small footprint and few required passives
- Runs at a high enough clock speed for our control system

Power

- Mouse has enough power for at least 10 minutes of runtime through the maze at maximum board power consumption
- Mouse must not overheat
- Mouse can easily show when the batteries are dead
- Power can be disconnected from the board easily
- Mouse can be powered via USB

Structure

- Mouse body is rigid
- Mouse has a place to hold batteries
- Mouse does not rock forwards and backwards

Detailed Project Description

Power

Figure 2, below, shows our power inputs. Our board can take power inputs from two sources. We have two 3.7V lithium-ion batteries in series which can fully power the board when plugged in. Lithium-ion batteries are rechargeable, which reduces total cost of testing. These batteries provide 7.4V, 1.4V above the required voltage for our motors. By only having 2 batteries, this reduces weight on the board and allows it to move faster with less rolling friction. If USB is plugged in, USB power (5V) can be used to power all the 3.3V components. When both USB and the batteries are plugged in, we have a circuit consisting of two p-channel MOSFETs that prioritize the 5V from the USB to not waste power from the batteries.

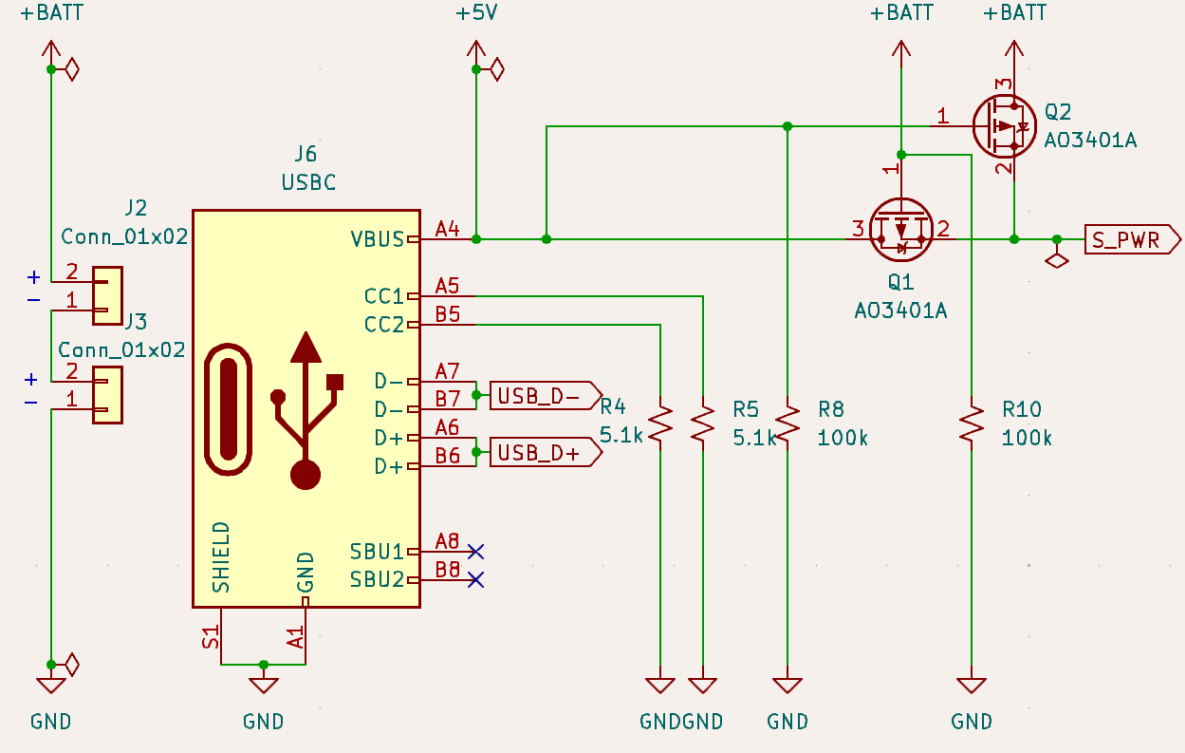

Figure 2. The power inputs of the micromouse. Inputs can come from either USB at 5V or lithium-ion batteries at 7.4V.

On the board, there are two voltage regulators, one for 6V and one for 3.3V. The 3.3V regulator will be powered by the voltage coming from the voltage switching circuit discussed above. The 6V regulator will only be powered by the 7.4V coming from the batteries. This saves us from having to use a buck-boost converter, and allowed us to only use an LDO regulator for reduced complexity and size. This means that the motors can only be turned on when the

batteries are connected, but this was acceptable for our use. As a part of the regulator circuit, there is a switch which can disconnect the batteries from the rest of the board. This is a quick way to shut off power to the motors during runtime, if needed. Finally, there is a power indicator LED connected to the 3.3V power plane. This is a quick way to tell if our board is receiving power or not. This also tells us if our batteries died while the mouse is running. **Figure 3**, below, shows the regulator, switch, and indicator LED circuitry.

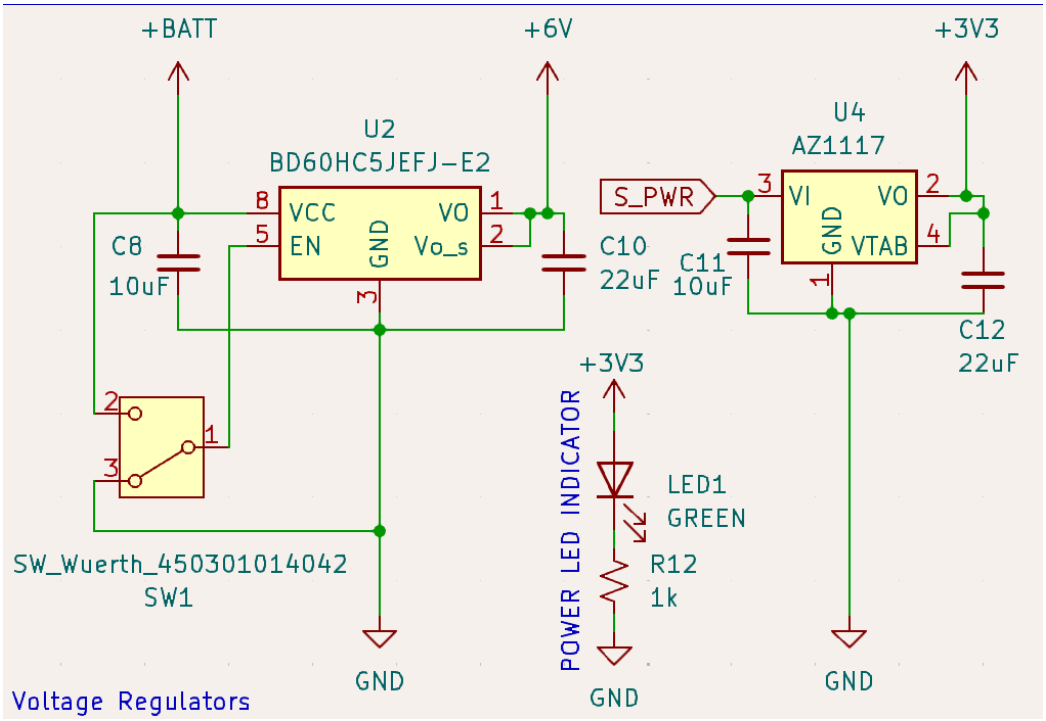

Figure 3. Voltage regulator, indicator, and switch circuitry. This section converts input power from the batteries or USB into usable 6V and 3.3V for the rest of the board

Battery charging is handled off-board via a dedicated lithium-ion battery charging station. This was done for two reasons. Firstly, battery charging circuitry would have taken up extra room on the PCB, causing the board to be bigger. Secondly, if we had a soldering or design mistake with our charging circuitry, we would have a chance of damaging the batteries. Damaged lithium batteries can erupt in flames, posing a danger to all in the building. To avoid safety concerns, we will charge our batteries with an external, previously tested, battery charger.

Microcontroller

For the microcontroller, we chose an ESP32-S3-WROOM1-N8. This was chosen for multiple reasons. Firstly, Professor Schafer has extensive experience with the ESP32-S3 and can answer questions when we run into problems. This significantly cut down on debugging time. Secondly, this chip runs at up to 240 MHz, high enough for the processing of our control system. The chip only has three required capacitors and one required resistor on the power pins, reducing the board space that it takes up compared to other chips. The "N8" means that the chip has 8 MB of on-chip flash memory. This is enough to store our code for the program. Finally, there are enough IO ports on this chip for our uses. These include analog inputs with on-chip ADCs to convert the analog voltages from our IR phototransistors into digital signals for us to use in our control system. **Figure 4**, below, shows the microcontroller and passives on our schematic. The chip selected does include bluetooth and wifi capabilities that we thought could be helpful in debugging, but we did not end up utilizing this function.

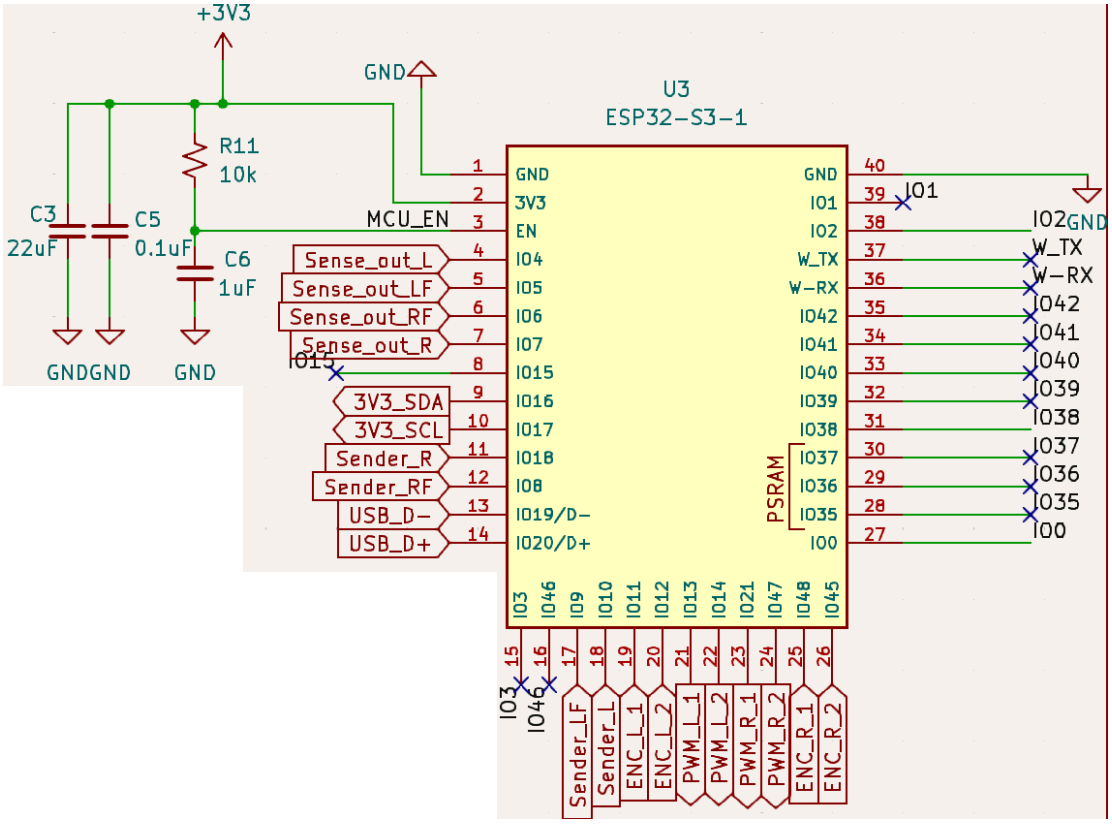

Figure 4. ESP32-S3 microcontroller on the schematic. Many of the IO ports are labeled with their uses on the board.

Sensors

The sensors chosen for this board are a pair of an infrared LED and a phototransistor. The LED selected was the INL-5AMIR15, and the phototransistor selected was a SFH 313FA. These sensors work together to get distances. The IR LED emits infrared light which bounces off the nearest wall and reflects back into the phototransistor. Based on the amount of light received by the phototransistor, an analog voltage is returned to the microcontroller. The more light that the phototransistor collects, the closer the wall is to the mouse. In software, an equation can be fitted to the readings to turn the voltages into distances in millimeters. **Figure 5**, below, shows the schematic view of two of the sensor pairs.

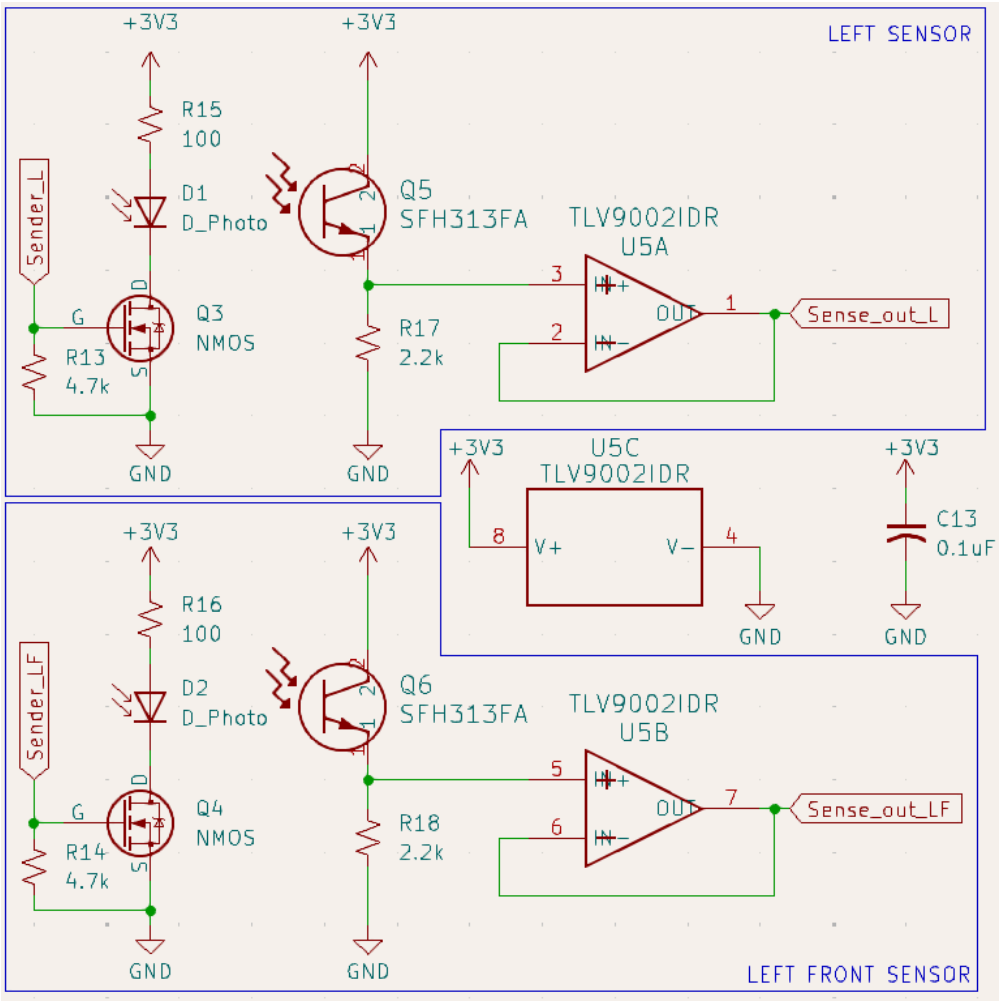

Figure 5. Distance sensors on the board. Each sensor circuit consists of an IR LED and a phototransistor. An opamp is used to boost the signal from the phototransistor to the board. There is an identical circuit for the right and right front sensors.

Our two main options for distance sensing were these analog sensors and time of flight sensors (TOF). TOF sensors are much more accurate at distance and are easier to setup. They run on I2C and come pre-calibrated. The main drawback of TOF sensors is that they run much slower than analog sensors. During testing, we were only able to get readings from all four TOF sensors at 20 Hz. 20 Hz is too slow for our control system that we tested on Dr. Schafer's practice mouse. When we tried the analog sensors, we were able to get readings from all four sensors at 3000 Hz. We felt much more confident in our ability to run a control system at 3000 Hz than 20 Hz. Even though the analog sensors required additional setup in the form of extensive calibration to convert the analog readings to distances, we chose to use the analog sensor setup over a TOF sensor setup. Additionally, we did not view the lower maximum distance of analog sensor readings as a drawback. As long as the mouse can see one unit square in any direction, we can create our control algorithm. Since the analog sensors can read distances greater than 180 mm, they will work.

There are four pairs of sensors on this board. There are two sensors on the front of the board. These could be used to square the board up to a wall, ensuring that the mouse is perfectly perpendicular to the wall. This is done by changing the angle of the mouse until both sensors read the same. There is also a sensor on the left and right sides of the board. These would be used to sense side openings as well as keep the mouse centered in a corridor.

One final part of the sensor circuits are the opamps. Two pairs of sensors use one chip, the TLV9002IDR, to help strengthen the signal from the phototransistor. Without the opamps, the signal may be too weak or noisy to be reliably read at the microcontroller. Each opamp chip contains two independent opamp circuits, so only two opamp chips were needed on the board for the four sensors.

Motors

A two-wheel system seemed like the obvious choice for movement for the mouse. This simple approach reduces complexity and allows for the turning and driving capabilities we were wanting. We chose the Pololu 5135 motor, shown in **Figure 6** below, to drive the wheels on our mouse. The first consideration was motor voltage. We wanted a 6V motor for a blend of power and weight. With a 12V motor, we would have needed 4 batteries atop our mouse. Potential gains in speed with a 12V motor would be offset by the additional battery weight. A 6V motor could provide enough speed and torque for our mouse from preliminary testing. This motor also has 50:1 gearing. Through testing multiple motors before selecting this one, a 50:1 gear ratio seemed to provide the range of speeds we were looking for in our mouse. These speeds will allow us to be able to fully search a maze, drive back to the start, and solve the maze again.

Figure 6. Pololu 5135 motor with side encoder. Two of these are on the LockhEEd Mousetin board as the method of movement

Another aspect of this motor we needed was an onboard encoder. The encoder sends signals to the microcontroller to tell it how much it is turning. This motor provides 12 counts per rotation, which is about 600 counts per rotation of the actual wheel. This gives us precision to the nearest half degree rotation of the wheel, which is sufficient for our use case. This corresponds to about 0.1 mm, which is negligible. The encoder connection on this motor points perpendicular to the axis of the wheel. This is ideal because it allowed us to place the motors closer together, reducing the width of the mouse. Two motors back-to-back is the minimum width of the mouse which we were able to achieve. If the encoders were parallel to the axis of rotation, there would need to be additional room on the board to accommodate the cables coming out of the motors.

Motor Driver

The motor driver selected for the mouse was the Texas Instruments DRV8411A. This motor driver has two independent channels for motor control, perfect for the two motors on the mouse. It takes both 6V power, for the motors, and 3.3V, for the driver and encoder logic, from the regulators. It also takes PWM signals from the microcontroller which controls the voltage level to the motors. Running at less than max PWM levels allows our motors to run at variable speeds. This chip is very small at only 3mm by 3mm, saving space on the board. **Figure 7**, below shows the motor control circuitry, including the DRV8411A.

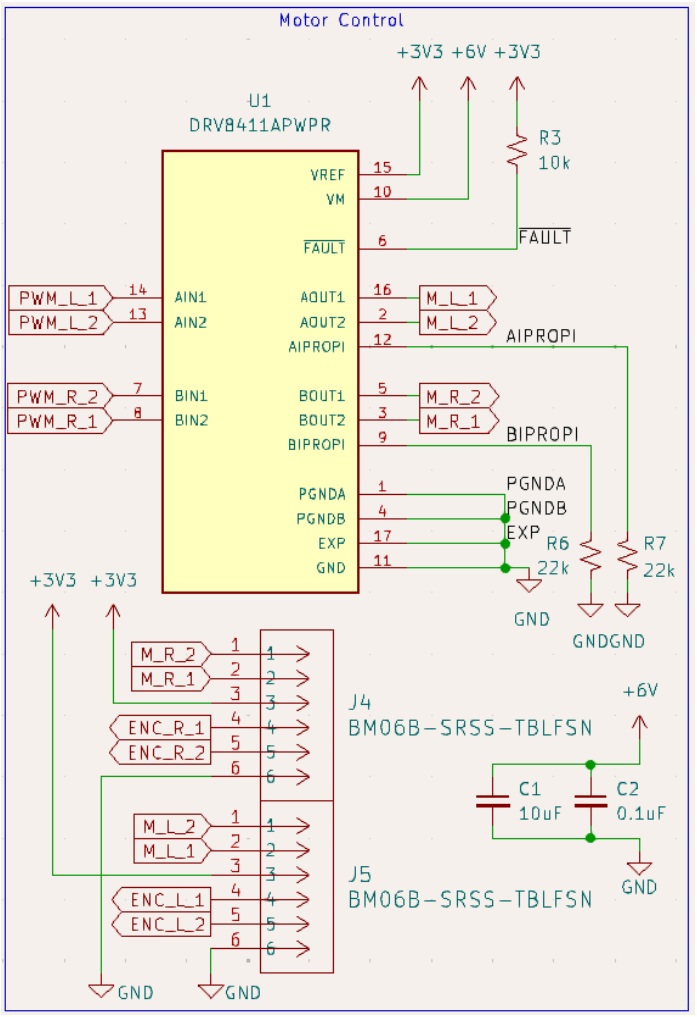

Figure 7. Motor control circuitry on the board. This includes the DRV8411A motor driver, required passives, and the connectors to the motors.

One other reason we went with this chip is due to its manufacturer, Texas Instruments. Texas Instruments chips are very widely used, and thus they have great documentation. If we were to run into any issues with implementing this chip, it is likely that others have also run into the same issues and it would be more easily solved.

The motors connect to the board through a JST-SH-6 connector. This is a standard connector that allows us to connect and disconnect the motors from the board easily.

Wheels

Figure 8, below, shows the wheels chosen for this mouse. Their small sizes gives us more control over movement. They also contain a rubbery-textured tire around the plastic spokes. This gives our mouse more traction, preventing slipping. Slipping would be detrimental because slipping causes the motor counts from the encoder to be inaccurate. The mouse would think it had driven further than it actually had.

Figure 8. 32 mm wheels used on the micromouse. The wheels have a rubbery tire.

Push Buttons

There are three push buttons on the board. Two of the buttons are required by the ESP32-S3 microcontroller. One is for resetting the board, and it sets the code to run from the start again. The other button is called “boot,” and this button is used to help put the board into code upload mode. This happens automatically most of the time on code upload, but it can be useful in times when the upload fails for various reasons. The final button is an elective button we added to the board. This button, called “function,” goes to an IO port on the microcontroller. This can be used for anything in the code, but the intention was for it to be used to switch the mouse from searching mode to solving mode. The rules of the competition allow for switches and buttons to be used during runtime. **Figure 9**, below, shows the button circuitry.

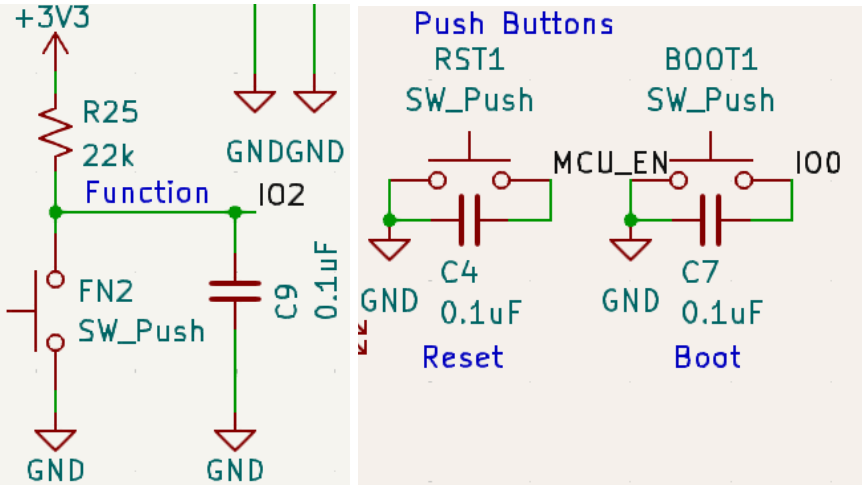

Figure 9. Push button circuitry. There are three push buttons on the board.

The buttons all have a capacitor in parallel with them. This is to reduce switch bouncing when pressed.

I2C Connection

Figure 10, below, shows the I2C circuitry. A pinheader allows any I2C compatible board to be plugged into our board. We planned to use this for a small screen. This would allow us to print information during debugging and during runtime. In our final mouse, we used an I2C OLED screen to display the mouse's coordinates in the maze.

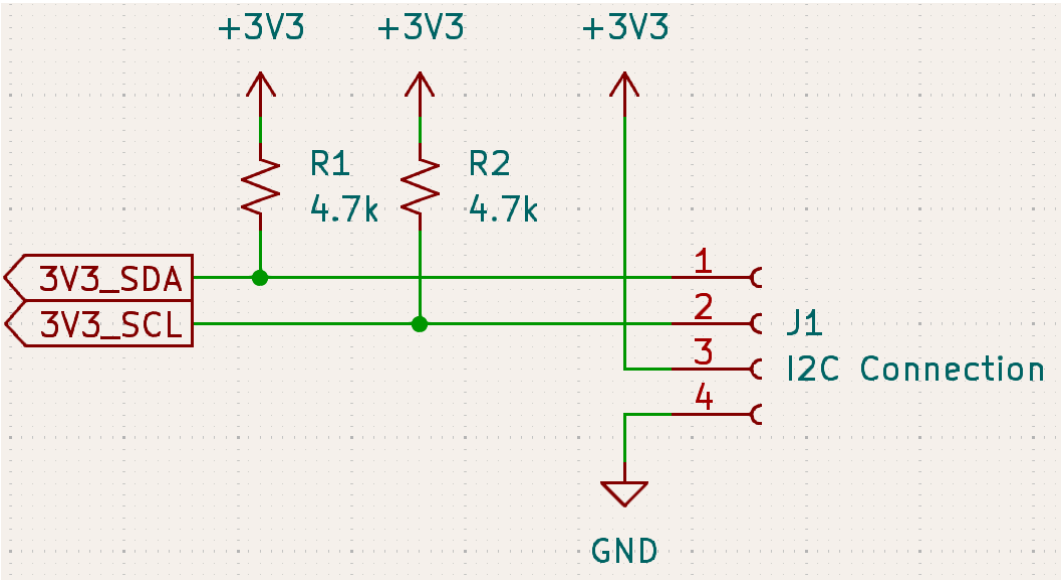

Figure 10. I2C circuitry. Any I2C compatible device with a pinout can be connected here.

Function LED

In addition to the power LED discussed in the power section, a second LED is on the board. This can be used at any time in the code to communicate something to the user. It is connected to an IO port on the microcontroller and can be turned on or off. We used this LED as a first code test to make sure our microcontroller could actually run code. Seeing the LED blink on and off was a very quick test that our mouse was working. **Figure 11**, below, shows the very simplistic LED circuitry.

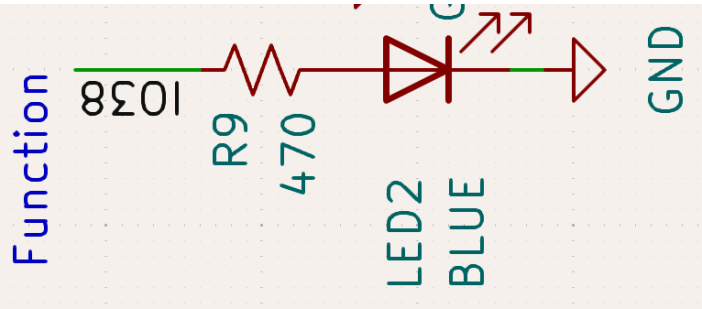

Figure 11. Function LED circuitry.

Board Layout

Figure 12, below, shows the final PCB layout. This is a four-layer PCB with a dedicated ground plane and 3.3V power plane. The top and bottom planes are reserved for signals. All above described circuitry is on the top side of the board.

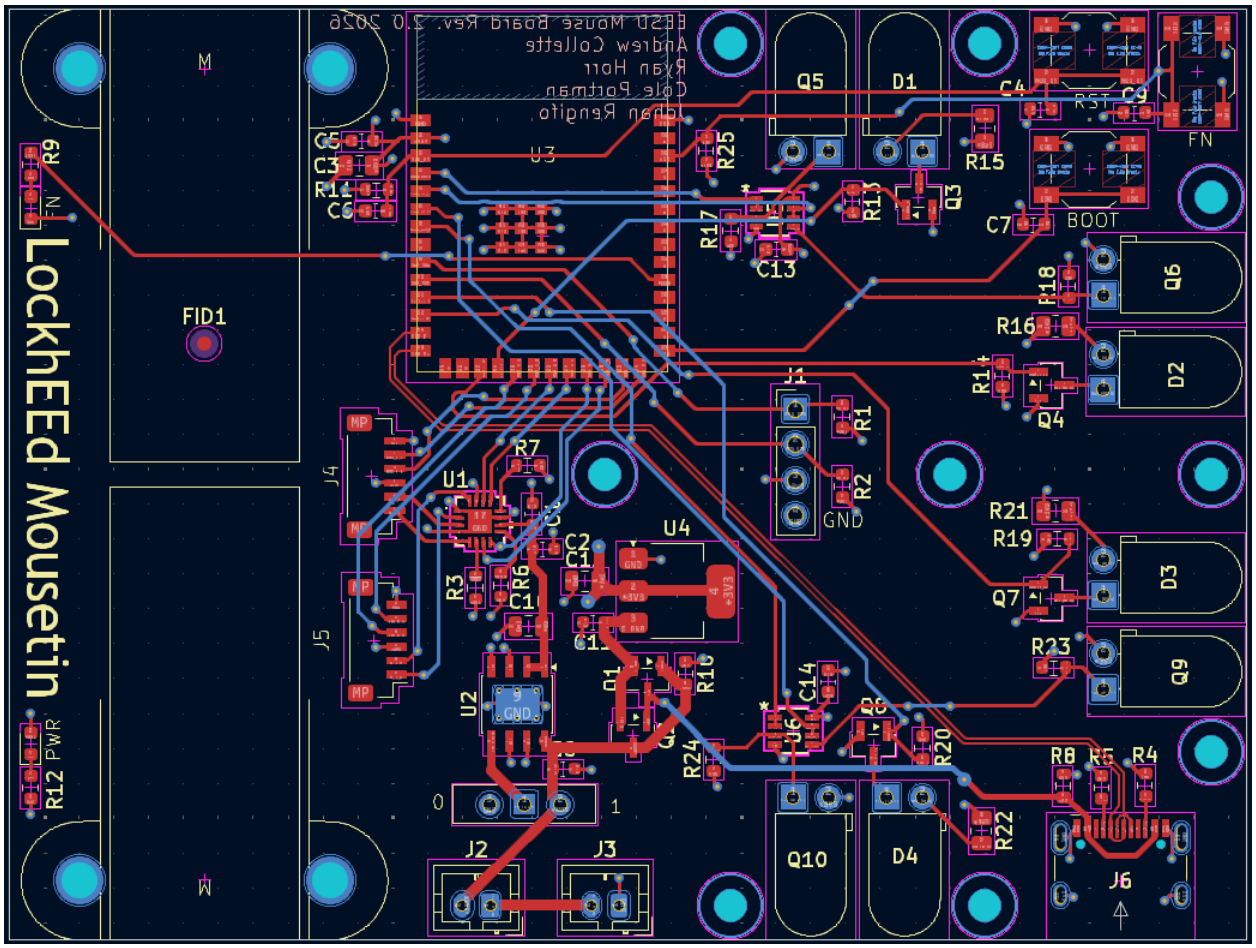

Figure 12. LockhEEd Mousetin's final PCB layout.

One major design decision we decided on was to have the motors in the back of the board, not centered like it was on the practice mouse. This was mainly done to make balancing the board much easier. With the pivot point in the back, the mouse can only rock to the front side. With a skid in the front, we can tune the height of the front of the mouse and make the mouse perfectly level much more easily. On the practice mouse, which had the wheels in the middle, we often encountered problems when the mouse would start, stop, and turn. The mouse would flop back and forth on the pivot, and the skids we had on that mouse were much harder to tune. By having the motors in the back, we solved this issue. The motors are connected to the board with a motor mount and M2 screws in the M2 screw holes on either side of the motors.

The board size was defined by necessity. The width of the board is defined by the width of two motors placed back-to-back. The length of the board was defined by the number of components which had to be on the edge of the board. Since the back of the board was taken up by the motors, a whole edge was unable to be used. The sensors had to be placed on the edges of the board so that nothing blocked their path to seeing the walls. Two sensors had to be placed on the front edge, while the other two sensors had to be placed on the left and right sides of the mouse. Buttons had to be near an edge to be easily accessible. If they were towards the middle of the mouse, they would be blocked by components and 3D prints and difficult to access. The USB connector had to be on an edge to have room for the USB cable to come out of it. We also wanted the battery connectors to be on an edge for ease of removal. If, during testing, the board began to overheat, we wanted to be able to disconnect the batteries as quickly as possible for safety reasons. All of these edge components, along with the motors being on the back of the board, necessitated a rather long PCB. The final dimensions of the PCB was 88mm by 67mm. This is narrower, but longer, than the practice mouse.

Other important component placements included the M2 mounting points and the I2C connector. M2 mounting holes were placed on the board at various spots. Each set of analog sensors had a hole on either side for a potential sensor mount. The idea with the mount was, if it was deemed that the analog sensors changed readings often due to getting accidentally bumped and moved, a housing around them would help increase reading consistency by keeping the sensors in place. During testing, we decided that this was not necessary and the sensors were fairly resistant to getting accidentally bumped, so these M2 holes were not used. Other M2 mounting points were designed to be used for the battery holder and a skid in the front. The battery holder mounting point is below the microcontroller. This is so the battery holder can be mounted above the motors, out of the way of other components on the board that need to be accessed more often, like the I2C connector and the buttons. The last M2 mounting hole is for the skid. This rests near the front of the board, away from the axis of rotation of the mouse.

The I2C connector rests in the middle of the board. It is oriented such that a screen attached to the connector would go in the direction of the front of the board. It would rest over the skid M2 hole since the skid goes on the bottom of the board. The battery holder would not be over the screen, making the screen easy to read.

On the board design, care was taken on trace widths. The default trace width for signals was 0.25 mm. Traces that carry higher currents are thicker. The 6V and battery traces are 0.6 mm and 0.75 mm respectively. The 5V traces are 0.4 mm.

On the board layout, a fiducial was added. This was required for the pick-and-place machine to orient itself and place parts accurately. Without this, all parts would have been needed to be placed manually. The fiducial was also placed under one of the motors. It is out of the way of other components and gets covered by the motor after the pick-and-place machine had placed all components.

3D Prints

We designed two 3D printed parts. The first part was the battery holder, shown in **Figure 13** below. This attaches to an M2 mounting point on the PCB. It is tall enough to clear the cables coming out of the motors, but not too tall to cause the mouse to tip over during regular use. The place where the batteries sit is shaped like a window with large cutouts. These cutouts are to reduce weight. The other side of the holder rests on the back of the board without an actual mounting point. This support is just to hold the weight of the batteries. The one M2 connection is enough to hold the battery holder onto the board. One side of the battery holder has cutouts for the battery wires to run through. These cutouts are filleted on all sides to prevent damage to the wires from sharp corners.

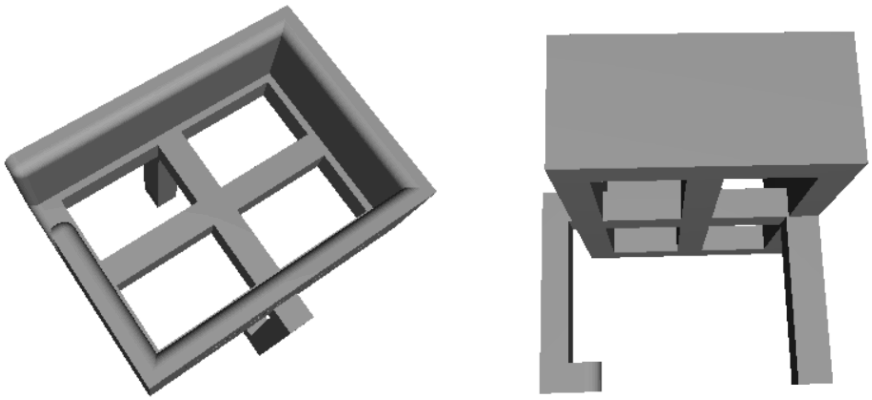

Figure 13. Two views of the battery holder. The L-shaped foot contains a M2 screw hole to attach to the board.

Figure 14, below, shows the other 3D printed part we designed. This part is a skid attached to the bottom of the board. Since we have our wheels in the back, the front of the board is supported by this one skid. To keep the board level, we only needed to adjust the height of this one part, which was easily done with washers. This skid is attached to the board at one mounting point using an M2 screw. The part making contact with the ground is a wide hemisphere for a wide area of contact. A cylinder comes out of the top to clear some of the through-hole connections on the bottom of the board. The spherical part was sanded to create a smooth and consistent contact with the ground.

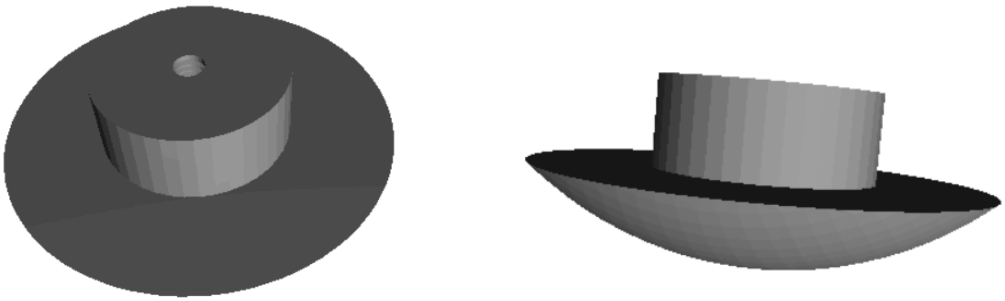

Figure 14. Two views of the 3D printed skid.

Uploading Code

Uploading code is done through USB. By plugging into the board through a USB cable attached to a computer, one can upload code over USB. After detaching the USB, code remains on the mouse, even when power is disconnected. It is through the USB that serial communication with the board can also be monitored.

The ESP32-S3 can also handle uploading through UART instead of USB protocol. We chose to implement USB code upload because UART communications requires additional components on the board. To eliminate complexity and reduce cost through using fewer components, we decided to only implement code uploading through USB.

Software Development Overview

The micromouse robot developed for this project was designed to autonomously explore and solve an unknown maze using onboard sensing, motion control, and route finding. The software system coordinated all aspects of robot behavior, including sensor reading, motor control, maze mapping, and solving. Because the robot operated in a constrained physical environment with limited sensing, the software architecture needed to be both modular and robust to ensure reliable functionality.

To achieve this, the software was divided into several subsystems, each responsible for a specific aspect of maze solving. Low-level modules handled direct hardware interaction, including infrared distance sensing, encoder tracking, and PWM motor control. Additional to these hardware interfaces, higher level algorithms managed maze mapping, exploration methods, and shortest path navigation. This layered architecture allowed the robot to both move consistently throughout the maze and translate raw sensor measurements into intelligent movement.

A major design objective of the software was reliability in real-world conditions. We experienced issues with motors moving at different speeds at the same PWM control, not returning identical encoder counts, and IR sensors returning different values and identical distances. To address these issues, the control software incorporated multiple corrective checks, including PID based wall following, encoder based motion feedback, majority vote sensor filtering, wall hysteresis logic, and recovery procedures for incomplete maze exploration. These features improved the consistency of robot behavior while letting the system adapt to real maze conditions.

The following sections describe the software implementation from the high level overall structure, to the low level motion and control functions, ending with the maze exploration and path planning algorithms.

Architecture

The software was divided into modular subsystems, which was done for a variety of reasons. Each subsystem handled a separate responsibility and contained a multitude of functions. This was critical for debugging and testing, and was important when updating code or adding additional functionality. The code implements the following files to achieve this modularization.

Table 1. Code files and functionality

Module	Primary Responsibility
main.cpp	System setup and execution
motion.cpp/h	Motor control, encoder tracking, and PID steering
analog_sensors.cpp/h	Infrared sensor setup and distance measurement
maze.cpp/h	Maze representation, exploration, and path planning

Upon turning on, the mouse begins to execute code in main.cpp. This file is relatively short since it is only in charge of initializing and calling system functions, which are executed in the other modular files. The functions that are called within main are listed below, in order

```
void setup() {
    motor_setup();
    analog_sensor_setup();
    oled_init();
    maze_init();
    maze_search();
    maze_print_serial();
    maze_return_to_start();
}

void loop() {
    //maze_solve();
}
```

We can see that the vast majority of functions are executed once, in the code setup. Only the maze_solve is called in the loop. The functions perform the tasks that they are named after. We see that first the code sets up the motors and sensors, then the OLED display. It then initializes a blank maze, ready for mapping. From there it calls the search function to search the maze. Once the maze is searched it prints the map over the serial monitor and returns to the start, ready to do a speed solve.

Analog Sensing

The next sections examine the lower-level subsystems responsible for direct hardware interaction. These subsystems include infrared distance sensing, motor control, encoder feedback, and closed-loop steering correction.

As explained previously, the board has 4 IR sensors on it to measure where the mouse is in relation to the maze walls. All of the code to use the sensors was kept within the analog_sensor.cpp/h files. We used the four for the following purposes:

Table 2. Sensor purposes

Sensor	Purpose
Left	Left wall tracking
Right	Right wall tracking
Left Front	Primary front wall detection
Right Front	Front wall alignment

analog_sensor.cpp contained the following functions:

```
analog_sensor_setup()
readIR(int emitPin, int recvPin)
readLeft()
readLeftFront()
readRight()
readRightFront()
```

`analog_sensor_setup()` purpose was to initialize the pins that were to be used for the analog sensing. This function configures the infrared emitter pins as outputs and the receiver pins as inputs. At the beginning of the function, each emitter is set low to ensure that the sensors begin in a known off state. The receiver pins are then configured as inputs so the microcontroller can read the analog voltage produced by each infrared receiver.

After the pin modes are configured, all four emitters are turned on. This prepares the sensors for use during maze navigation. Since this function only handles hardware setup, it is called once at the beginning of the program from main.cpp.

`readIR(int emitPin, int recvPin)` is the function that directly reads in the analog voltage from the receiver pins. It takes in the specific emitter and receiver pin to modulate and

read from, and then performs the low level sensor reading. It actually takes in two different readings from the receiving pin, one when the emitter is turned off, and a second when the emitter is turned on. It subtracts the reading when the emitter is turned off, what we call the "ambient light reading," from the value when the emitter is on. It returns that raw voltage value as its reading.

Lastly in the file we have all of the read functions (**`readLeft() readLeftFront() readRight() readRightFront()`**). These functions all follow the same four line structure, each tailored to communicate with the respective sensor:

```
int analog = readIR(IR_EMIT_PIN, IR_RECV_PIN);
if (analog <= 0) return -1.0;
double x = 1.0 / analog;
return 3e9*x*x*x - 2e7*x*x + 62071.0*x - 32.132;
```

(Unbolded sections tailored to specific sensor)

There is an **`IR_EMIT_`**`PIN` and **`IR_RECV_`**`PIN` for each of the four sensors. The code passes that information to **`readIR()`** to get a raw voltage value. If **`readIR()`** returns a zero, the read function automatically returns a distance of negative one, signifying that there is a reader error. If the reading is valid, the function converts the raw analog value into an estimated distance. The code first computes the inverse of the analog reading using the third line. This is done since the infrared distance sensors have a nonlinear relationship between reflected intensity and distance. As distance increases, the reflected signal decreases in a nonlinear way. Using the inverse of the analog reading helps linearize the relationship before applying the calibration equation.

The code has unique calibration equations for each of the sensors. This was done by measuring the raw voltage values as specific distances, inverting those values, and then running a third degree polynomial regression. The calibration formula is applied in the code to the inverse analog value, and then that number is returned as the measured distance in millimeters.

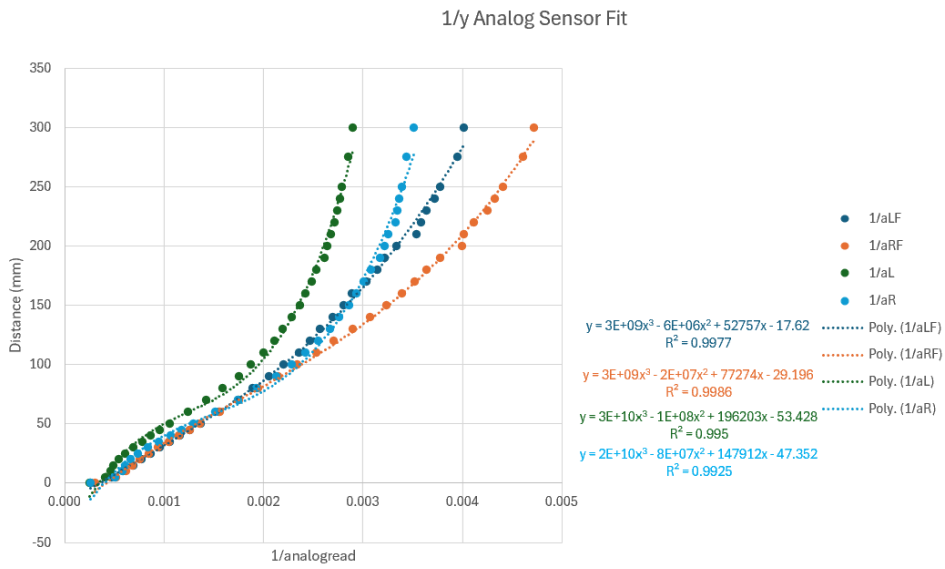

Figure 15: Third degree regression for the four analog sensors

All together, analog_sensors.cpp converts raw infrared reflections into calibrated distance measurements that the rest of the software uses. These distance measurements are used throughout the rest of the system for wall detection, PID steering, and maze mapping. This sensor interface forms the lowest software layer of the robot, allowing all motion and navigation algorithms to make decisions based on the mouse's physical surroundings.

Motion Control

The motion control subsystem was responsible for all physical movement performed by the mouse. This subsystem interfaced directly with the motor driver hardware and wheel encoders to generate controlled forward motion, turning maneuvers, and braking behavior. In addition to basic motor control, the motion subsystem also provided the foundation for higher level steering and maze navigation algorithms.

The robot used independently controlled left and right motors. By varying the speed and direction of each motor, the robot was able to move forward, backward, pivot, and precisely turn within the maze. The motion subsystem implemented these behaviors through a combination of PWM motor commands and encoder-based feedback.

The foundation of motion.cpp/h are the five basic motion functions

```
motor_setup()
set_left_speed()
set_right_speed()
stop()
idle()
```

These give the basis for all other functions. Similarly for the analog sensing, the first function, **`motor_setup()`**, initializes all motor driver and encoder pins used by the robot. The four PWM pins connected to the motor driver are configured as outputs, and the encoder pins are configured as **`INPUT_PULLUP`**, which enables the microcontroller's internal pull-up resistors and configures the pin to read in encoder counts. This function is called once at the beginning of main.cpp.

The **`set_left/right_speed()`** functions are identical in logic, and different only in which pins they communicate with. The function accepts a signed integer representing the wanted motor speed. Positive values are for forward motion, negative for reverse. The speed command is first constrained to the valid PWM range of -255 to 255. If a higher value than 255 is sent to the function, this converts that value to 255. This prevents invalid motor commands from being sent to the driver hardware. When the speed is positive, the function sets one motor driver pin low while the other receives a PWM signal. This causes the motor to rotate in the forward direction. When the speed is negative, the opposite pin configuration is used, reversing the direction of motor rotation. The magnitude of the PWM value determines the motor speed. There is also a check to see if a speed value of zero was sent to the function. If so, both control pins are driven low, allowing it to coast freely.

`stop(), idle()` are simple functions that do what their name implies. **`stop()`** sets all four PWM pins high to actively break the motor while **`idle()`** sets all four PWM pins low, letting the motor coast. From these five, all other motion is built.

After the low-level motor functions were implemented, the next layer of the motion subsystem needed to be to use encoder feedback to measure wheel rotation and implement that into motion functions. While the motor speed functions command the wheels to spin, the encoder functions allow the software to estimate how far each wheel had actually traveled. This was necessary because PWM commands alone do not guarantee accurate motion due to mechanical differences between the motors.

The motors had built-in encoders. Each encoder produced two digital signals, labeled A and B, that changed state as the wheel rotated. By monitoring the signals, the software could determine both the amount of rotation and the direction of rotation.

There are two functions to read and count the motor encoders: **`poll_left_encoder(), poll_right_encoder()`**. They were designed as follows:

```
static void poll_left/right_encoder()  {
    static uint8_t last = 0;
    uint8_t a = digitalRead(ENC1_A);
    uint8_t b = digitalRead(ENC1_B);
    uint8_t state = (a << 1) | b;
    switch ((last << 2) | state) {
    case 0b0001:
    case 0b0111:
    case 0b1110:
    case 0b1000:
        enc_left_count++;
        break;
    case 0b0010:
    case 0b0100:
    case 0b1101:
    case 0b1011:
        enc_left_count--;
        break;   }
    last = state;
}
```

These functions store the previous encoder state in the static variable **`last`**. Since **`last`** is static, it retains its value between function calls. Each time the function runs, it reads the current values of encoder channels A and B. The two digital signals are combined into a two-bit value called state. The function then combines the previous state and current state into a four-bit value through **`(last << 2) | state`** which is used in the switch statement. This four-bit pattern represents the transition that occurred since the last reading. Certain transitions correspond to forward wheel rotation, while others correspond to reverse wheel rotation. If the transition matches a forward pattern, enc_left_count is incremented. If the transition matches a reverse pattern, enc_left_count is decremented.

There are then four motion functions that directly use the encoder counts to move the mouse. Listed, they are: **`drive_straight(), turn_left(), turn_right(), turn_180().`** Starting with **`drive_straight()`**, it performs that exact function, moving the mouse forward a specific distance using the encoder feedback. It takes in four variables: **`float dist, int speed, float yaw, bool coast.`** The **`dist`** parameter specifies the desired travel distance in millimeters. The **`speed`** parameter determines the PWM motor command. The **`yaw`** parameter provides a feedforward trim between the left and right motors, and the **`coast`** parameter determines whether the robot should coast or actively brake at the end of the movement.

At the beginning of the function, the desired distance is converted into an expected encoder count using the wheel diameter and encoder resolution. The left and right encoder counts are then reset to zero so the movement starts from a known reference point. The function commands both motors to begin moving. The left motor receives speed, while the right motor speed is multiplied by the yaw correction factor. While the robot is moving, the function continuously polls both encoders. It calculates the absolute value of both encoder counts and averages them: **`long avg = (left_abs + right_abs) / 2;`** Using the average of both wheels provides a more balanced estimate of how far the mouse has traveled. Once the average encoder count reaches the expected count, the loop exits and the robot stops. At the end of the function, the mouse either coasts or brakes depending on the coast parameter. If coasting is enabled, the **`idle()`** function is called. Otherwise, the mouse briefly applies reverse motor power before calling **`stop()`**. This reverse pulse helps slow the mouse quickly and improves stopping accuracy.

`turn_left/right()` takes in variable `int degree` into the function. It then calculates the amount of wheel travel required to rotate the robot by that angle. This is based on the wheel spacing and desired turn angle: `float wheel_travel = (PI * WHEEL_SPACING * abs_deg) / 360.0;` This equation estimates the arc length that the moving wheel must travel around the robot's turning radius. The wheel travel distance is then converted into encoder counts using the same wheel circumference and encoder resolution relationship used for straight driving. For a left turn, the code drives only the right wheel while the left wheel remains stationary, and the opposite for a right turn. For a left turn, the right encoder count is reset to zero, and the right motor is commanded to move. The direction of motion depends on whether the requested angle is positive or negative, if a negative angle is requested the right motor drives reverse. The function continuously polls the right encoder until the encoder count reaches the calculated target. Once the target count is reached, the robot calls `stop()` to brake the motors. The code also includes an empirical multiplier in the encoder target calculation. This correction factor compensates for real-world turning errors such as wheel slip, friction, and the fact that the ideal geometric turn calculation does not perfectly match the physical mouse. In the code the multipliers are set to around 1.5, and can be tuned to accurately get a precise turn. It is worth noting that we only ever call these functions to turn 90 degrees, so the multiplier was set to perfect that angle specifically.

The last encoder based function is `turn_180()`. In `turn_180()` the robot first uses its side sensors to compare the distances to the left and right walls. If the left wall is closer than the right wall, the mouse begins to turn toward the further wall. If the right wall is closer, or both readings are equal, it begins the turn in the opposite direction. The turn itself is split into two smaller movements rather than one continuous 180-degree rotation. This allows the robot to pivot more carefully within the limited space of a maze cell and helps avoid contacting the walls. The motors are in the back of the mouse, and the mouse itself is relatively long, causing typical turn-in-place 180s to swing the front wide. Each phase uses the existing `turn_left()` and `turn_right()`. It is worth noting that we only call 88.75 degrees in those turns. There is a short overshoot between turns, so this lets our 90 degree turns be strictly 90 and the 180 still be 180. Short delays are inserted between each turn segment to allow the robot to settle mechanically before beginning the next movement.

`driveStraightGuided()` is another function and is the first PID controlled motion function we implement. The function is the primary straight-line movement function used during maze traversal. Unlike **`drive_straight()`**, which mainly relies on encoder distance and a fixed **`yaw`** trim, **`driveStraightGuided()`** uses closed-loop feedback from the side wall sensors, front sensor, and wheel encoders to keep the robot centered while driving through a cell. This made it more reliable for actual maze navigation.

The function takes in variable **`float dist_mm`**. The function begins by resetting both encoder counts to zero. This gives the robot a fresh distance reference for the current movement. It also checks whether the robot has just completed a turn using the **`g_just_turned`** flag. If this flag is set, the function temporarily increases the PID correction strength, or K value, at the beginning of the drive. This helps the robot recover from small alignment errors immediately after turning, but this strength is not necessary once the robot is relatively aligned.

The target travel distance is then converted into encoder counts using the same wheel diameter and encoder resolution relationship discussed earlier. The function subtracts a small amount from the commanded distance to account for physical stopping distance and positioning offsets. This allows the robot to stop more accurately within a maze cell instead of overshooting. The PID state variables are reset before movement begins. These include the integral term, previous error, filtered error, and timing reference. Resetting these values prevents correction values from a previous movement from affecting the next drive command.

The function then enters its main control loop. During every loop iteration, both encoders are polled and the front sensor is checked. The front sensor acts as a safety stop: if the robot detects a wall too close in front of it, the function exits even if the target encoder distance has not been reached. The function also exits once the average encoder count reaches the desired travel distance.

The PID correction itself runs on a timed interval rather than every loop iteration. This is controlled by the **`loop_ms`** value in the PID tuning structure. At each PID update, the robot first reads the left and right side sensors and determines what kind of wall reference is currently available. There are four different wall modes which are important for the PID control:

Table 3: Wall Modes

Mode	Meaning
MODE_CENTER	Both walls are available, so the robot centers between them
MODE_LEFT	Only the left wall is reliable, so the robot follows the left wall
MODE_RIGHT	Only the right wall is reliable, so the robot follows the right wall
MODE_NONE	No reliable wall reference is available, so encoder correction is used

This mode-based design is important because the robot cannot always rely on both side walls. In intersections or openings, one or both wall readings may disappear. Instead of treating this as a steering error, the function detects the change and switches control strategies.

The code also uses wall hysteresis. A wall is considered present when its distance reading is below a lower threshold (75 mm), but it is not considered gone until the reading rises above a higher threshold (90 mm). This prevents rapid switching between modes due to noisy sensor readings. The function also detects sudden jumps in side sensor readings, which usually indicate that a wall has ended at an opening or intersection.

Once the mode is selected, the function calculates the steering error. In center mode, the error compares the left and right wall offsets. In left-wall mode, the robot tries to maintain a desired left-side distance (~50 mm). In right-wall mode, it does the same using the right wall. In no-wall mode, the function compares left and right encoder travel since entering open space, allowing the robot to continue straight through an intersection without wall feedback.

After the error is calculated, by comparing current distance-from-wall reading to ideal distance-from-wall values, small errors are ignored using a deadband. The PID output is calculated from proportional, integral, and derivative components. The proportional term responds to the current error, the integral term accounts for accumulated error, and the derivative term responds to how quickly the error is changing. The final correction value is clamped to a maximum output limit so the robot does not overcorrect. The correction output is then applied differentially to the motors. One motor is sped up while the other is slowed down, causing the robot to steer back toward the desired path. This allows the robot to continuously correct its heading while moving forward.

`driveStraightGuided()` is the most advanced motion function in the code. It combines encoder-based distance tracking, front-wall stopping, side-wall following, PID steering, wall detection hysteresis, post-turn correction boosts, and open-space encoder correction. This made it the main movement function used by the maze navigation code when driving from one cell to the next.

There are a few special movement functions that are used in specific situations. They were affectionately named **`goofyStraight(), goofyLeft(), goofyRight()`**. These functions were only called in the following scenario: the mouse wants to turn left or right into an opening, and the wall in front does not exist. In this case, we do not have a front wall to line up the mouse before turning, so the code implements other strategies.

The **`goofyStraight()`** function performs a PID-guided forward drive similar to **`driveStraightGuided()`**, but instead of stopping after traveling a fixed encoder distance, the movement termination is controlled by sensor detection rather than by a fixed travel distance. The function takes in a variable **`bool side`**, to represent left or right. Depending on the side it then drives until a wall is detected on that side. Essentially allowing the mouse to drive past the opening while PID controlled, then stop at the edge of the cell to gain a reference location.

`goofyLeft/Right()` Are called after **`goofyStraight()`**. All they do is call a -90 reverse turn, then drive forward to place the mouse in the correct position for the next function to be called.

These functions provided additional positioning accuracy beyond what could be achieved using our regular motion functions. By combining sensor-triggered movement, turning primitives, and short guided correction drives, the robot was able to achieve more reliable alignment when transitioning between maze corridors.

Maze Search Algorithm

The maze search subsystem represents the highest level of the software stack and is responsible for the actual problem the mouse is meant to solve: discovering an unknown maze, building an internal map of it, and computing efficient paths through that map. This subsystem is implemented entirely within maze.cpp and maze.h, and it sits on top of the analog sensing and motion control layers described in the previous sections. Where the lower layers expose primitives such as **`readLeft()`** or **`driveStraightGuided()`**, the maze module operates in the abstract domain of cells, walls, headings, and graph search.

The mouse stores its understanding of the maze in a two dimensional array **`g_cells[MAZE_H][MAZE_W]`**, where each cell is a single byte packed with wall information and metadata flags. The four lowest bits of each cell represent the presence of walls on the north, east, south, and west faces of that cell, and additional flag bits record whether a cell has been visited and whether it was discovered during the main exploration pass or during a later mop-up pass. Wall information is OR'd into cells rather than overwritten, which means that once a wall is recorded it is never erased. This design choice was made deliberately because false negatives, that is, missing a real wall, are far more dangerous than false positives, since a missing wall could cause the mouse to attempt to drive through a physical barrier. The cost of this conservatism is that any noisy sensor reading that registers a phantom wall becomes permanently embedded in the map, which motivates several layers of defensive logic described later in this section.

The maze module exposes a small public API consisting of **`maze_init()`**, **`maze_search()`, `maze_return_to_start()`, `maze_solve()`, `maze_print_serial()`,** and a handful of accessors. Internally, however, the module is built around three core engines: a median filtered sensing routine, a depth first search exploration engine, and a breadth first search navigator that operates on already discovered cells.

`maze_init()` is the entry point for the maze subsystem and is called once from **`main.cpp`** after the motor and sensor setup routines have completed. The function clears the entire g_cells array, places the mouse at the starting coordinates (0, 0) with a north heading, and stamps border walls around the perimeter of the maze so that the search algorithm cannot attempt to drive off the edge of the grid. After initialization, the OLED is updated to show the starting

cell coordinates so the team has immediate visual confirmation that the firmware has booted into a known state.

Before any exploration logic can run, the mouse must be able to look at its current cell and reliably record what walls are present. This is the job of **`sense_and_record()`**, and it is one of the most critical functions in the file because every wall in the final map is written by this routine. The function first waits **`SENSE_SETTLE_MS`** milliseconds to let the chassis stop rocking after a move, since residual motion produces inconsistent IR readings. It then takes **`SENSE_SAMPLES`** readings from each of the three relevant sensors, with a short gap between samples, and applies a majority vote across the samples. A wall is registered only if more than half of the samples agree that a wall is present, which kills single sample noise spikes that would otherwise corrupt the map.

Once the front, left, and right wall detections have been resolved, the function converts these robot relative directions into absolute compass directions using the current heading. For example, when heading north, the front sensor maps to the north wall bit, but when heading east, the same physical sensor maps to the east wall bit. The detected walls are then OR'd into the current cell, and crucially, the corresponding opposite face wall is also OR'd into the neighboring cell when one exists. This propagation ensures that the map remains internally consistent: if cell (2, 3) has a north wall, then cell (2, 4) must have a south wall, and the sensing routine enforces this invariant automatically.

The motivation for this sensing design came directly from a failure mode observed during validation. On early runs, the mouse would frequently stop exploring after reaching the upper left of the maze and skip the lower right entirely. The root cause was a single noisy reading at cell (0, 1) that stamped a phantom east wall onto the map. Since walls are never erased, the depth first search treated that wall as real for the rest of the run and never attempted to enter the cells beyond it. The median filter was added specifically to reduce the probability of these spikes making it into the map.

Above the raw motion control functions exposed by **`motion.cpp`**, the maze module defines its own movement primitives that combine motion calls with bookkeeping updates to the mouse's pose. **`do_turn_left(), do_turn_right(), and do_turn_180()`** execute the corresponding rotation, update the heading variable **`g_hdg`**, and set a **`g_pending_offset_mm`** value that records how much forward distance was effectively traveled by the pivot itself.

`do_move_forward()` then uses this pending offset to shorten the next forward drive so that the mouse ends up centered in the new cell rather than overshooting by the width of the pivot.
The turn primitives also implement a maneuver the team referred to internally as the goofy turn, which addresses a geometry problem unique to this mouse. In certain intersection configurations, a standard in place turn would clip a wall because the mouse's body extends past the cell boundary. The goofy turn detects this situation by reading the front sensor and, when the front wall is too close, drives a small additional distance forward, then performs the rotation, then backs up into proper alignment. This behavior is invoked transparently inside `do_turn_left()` and `do_turn_right()` based on the front sensor reading, so the higher level search algorithm never has to know which variant was used.

`face_heading()` is a small helper that turns the mouse to face an arbitrary absolute heading, computing the shortest rotation, whether straight, single right turn, single left turn, or 180 degree turn, and dispatching to the appropriate primitive. Together with `do_move_forward()`, this gives the search engine a clean two function interface for executing any path: face the desired direction, then step forward one cell.

The main exploration engine is implemented in `run_dfs()`, which performs an iterative depth first search using an explicit stack of visited cells rather than recursion. Recursion was avoided because the worst case stack depth on a 16x16 maze could exceed the microcontroller's available stack space, and the iterative version makes the path history directly available for backtracking.

At each step, the algorithm builds a list of candidate directions from the current cell. A direction is considered valid only if it stays within the maze bounds, has no recorded wall blocking it, and leads to a cell that has not yet been visited. The candidates are then sorted using a two part scoring function. The primary score is the Manhattan distance from the candidate cell to the nearest cell in the goal zone, which is the 2x2 block at the center of the maze. This biases exploration toward the goal so that on a typical maze the mouse reaches the center early in the run rather than wandering through unrelated branches first. The secondary score is the rotational cost of facing the candidate direction, where moving straight ahead costs zero, turning left or right costs one, and a 180 degree reversal costs two. This tie breaker prefers smoother paths when multiple directions are equally close to the goal, which reduces unnecessary turns and saves time.

When a candidate direction exists, the mouse executes the move, marks the new cell as visited, runs **`sense_and_record()`** to capture the walls of the new cell, and pushes the new cell onto the path stack. An early exit check then fires: if the mouse has entered the goal zone, the search halts immediately rather than continuing to explore, on the assumption that reaching the center is sufficient for a search run.

When no candidate direction exists, the cell is treated as a dead end and the algorithm enters its backtrack logic. Before backtracking, the algorithm checks whether the current cell has a front wall, indicating a cove like configuration. If so, it re-runs **`sense_and_record()`** to catch any side walls that might have been missed on the approach. If the cell turns out to be a fully enclosed three walled cove, an optional squaring routine can be invoked to align the mouse against the front wall before turning. In contrast, when the cell is a plain corridor or branch backtrack point, re-sensing is deliberately skipped, because re-sensing while the mouse is poorly aligned, that is, after it has just entered a corridor between two visited cells, risks stamping phantom walls onto cells whose true walls were already known. This was learned during validation: an over-aggressive re-sense policy was found to corrupt the map by inserting walls between visited cells, which then caused subsequent backtrack moves to fail because the navigator believed it could not pass through walls that did not physically exist.

Once the dead end logic completes, the algorithm pops the current cell off the path stack and uses **`move_to_adjacent()`** to drive back to the previous cell in the path. Because every entry on the path stack is by construction adjacent to the next, this single step move is always valid.

Even with median filtered sensing, occasional false walls can still slip through and leave reachable cells unvisited at the end of the depth first search. The mop-up pass implemented in **`maze_mop_up()`** is the safety net for this case. After the main DFS completes, the function scans the entire grid looking for any visited cell V that has an unvisited neighbor U with no wall recorded between them. If such a pair exists, it means the original DFS sensed no wall between V and U but never visited U, which can only happen if a different, false wall blocked the approach to U from some other direction during the original search.

When such a bridge cell is found, the algorithm uses the BFS navigator to drive the mouse to V, faces it toward U, steps forward into U, marks the cell as both visited and rescued via the MOPUP flag, runs **`sense_and_record()`** from this new and presumably better aligned

position, and then runs a mini DFS from U to sweep any pocket of cells connected to it. The whole grid scan then restarts, since exploring a pocket may have revealed walls that change which cells are now reachable. The loop terminates when a full pass over the grid finds no more bridge cells. The MOPUP flag is preserved in the diagnostic output so that the team can identify which cells required rescue, which serves as a direct tuning signal: a high mop-up count after a run indicates that the sensing thresholds or settle time need adjustment.

In the version of the firmware used for the final demonstration, the mop-up call is commented out inside **`maze_search()`** because the median filter alone proved sufficient on the test mazes used during validation, but the implementation remains in place as a safeguard for harder mazes.

Once the maze has been explored, the mouse needs a way to navigate efficiently between known cells without re-running the full search. This is the role of **`bfs_navigate()`**, which performs a standard breadth first search over the graph of already visited cells. The search expands outward from the mouse's current position, stepping only through cells that have been marked visited and only across edges that have no recorded wall, recording the direction of arrival into each cell in the **`came_dir`** array. When the target cell is reached, the path is reconstructed by walking backward through came_dir from the target to the source, reversed into source to target order, and then executed step by step using **`face_heading()`** and **`do_move_forward()`**.

This navigator is reused in three different contexts. During the mop-up pass, it drives the mouse to bridge cells. In **`maze_return_to_start()`**, it drives the mouse from wherever the search ended back to cell (0, 0). In **`maze_solve()`**, it drives the mouse from the start to the goal coordinates after exploration is complete, executing what is effectively an optimized run over the discovered map. The same code path serves all three use cases, which keeps the navigation logic concentrated in a single, well tested function.

A specific exception path is built into **`maze_solve()`** to handle the case where the goal cell cannot be reached using the current map. After the function passes its initial bounds and visited checks, it delegates the actual driving to **`bfs_navigate()`**, which returns a boolean indicating whether a valid path was found and executed. If **`bfs_navigate()`** returns false, the goal exists in the grid and was nominally visited at some point, but the breadth first search was unable to construct a connected path from the mouse's current position to the goal using only

edges with no recorded wall. This typically indicates that a false wall was recorded somewhere along what should have been a viable corridor, effectively partitioning the map into disconnected regions. When this occurs, the function prints a failure message over serial directing the user to the diagnostic output, calls **`stop()`** to bring the motors to a halt, and returns false to the caller. The top level program in main.cpp is then responsible for handling the failed solve attempt, and the standard recovery behavior is to invoke **`maze_return_to_start()`**, which uses the same BFS navigator to drive the mouse back to cell (0, 0) along whatever connected portion of the map still includes the start. This guarantees that the mouse always ends a failed solve in a known, recoverable position rather than stranded in the middle of the maze, and it gives the team an opportunity to inspect the printed map, identify the offending false wall, and either rerun the search or adjust sensing parameters before attempting another solve.

`maze_solve()` is the public function that performs an optimized run from the current position to the goal coordinates. Before invoking the BFS navigator it performs two sanity checks: it confirms that the goal coordinates are inside the maze bounds, and it confirms that the goal cell was actually visited during exploration. The second check is important because if exploration was halted early or the goal was walled off, attempting to navigate to an unvisited cell would silently fail. When either check fails, the function prints a descriptive error over serial and returns false, directing the user to the diagnostic output for further inspection. **`maze_return_to_start()`** is a thin wrapper that calls bfs_navigate(0, 0) and then faces the mouse north so that it ends in a known orientation, ready for a subsequent solve run.

Two diagnostic functions, **`maze_print_serial()`** and **`maze_print_diagnostics()`**, played a central role during validation. The first prints an ASCII art map of the maze with wall characters between cells, dots for cells visited during DFS, M characters for cells rescued by mop-up, and a directional arrow at the mouse's current position. This printout is generated after every sensing operation, which gave the team a continuous view of how the internal map was evolving in real time as the mouse moved through a physical maze. Direct cell by cell comparison between this serial output and the known maze layout was the primary method used to validate that the search algorithm was correctly recording walls and reaching all reachable cells.

The second function, **`maze_print_diagnostics()`**, produces a structured per cell breakdown reporting wall bits, visit status, and rescue status, along with summary counts of total,

visited, mop-up, and unvisited cells. It also emits suggested tuning actions based on the counts, for example recommending a larger settle time when mop-up cells are present, or suggesting threshold adjustment when unvisited cells remain. This function was used both during development and as part of post run analysis after every test on the 5x3 and 16x16 mazes.

System Testing

Practice Mouse and Initial Component Selection

Before we finalized the design of our PCB, we tested each of our external sensors and components. The first design decision we made was the motor. We had access to an assortment of motors of different voltages and gear ratios. After testing each of them at different speeds and under loads we thought would be similar to the loads experienced on the mouse, we chose a 6V, 50:1 gear ratio motor. We ordered this before ordering our PCB. Dr. Schafer had provided a practice mouse which had the capability to test many basic functions before committing to a design. We tested these motors on the practice mouse, and saw that it could perform the movement we wanted at acceptable speeds.

The next design decision was the type of sensor used on the board. Originally, we had thought time of flight (TOF) sensors would work well on our mouse due to their simplicity. We chose a TOF chip that had a long range (~2000 mm) and the fastest possible update rate (~100 Hz). Before constructing our own PCB with this TOF sensor on it to test, we purchased a set of TOF breakout boards to run using the practice mouse. It is here that we determined that a TOF sensor would not be fast enough for our use case. When running all four TOF sensors on the practice board, we were only able to read all four sensors at around 25 Hz. Updating our control system at this rate would not be fast enough for our final mouse. When running the practice mouse in a test maze, it was unable to reliably stay centered between two walls in a corridor when updating the control system at 25 Hz. It is for this reason that we began to explore the use of other sensors.

Similar to the TOF sensors, we obtained the IR LEDs and the phototransistors to test using the practice mouse before ordering our own board. After removing the TOF sensors, we installed the analog sensors. We could now obtain readings from all four analog sensors at 3000 Hz, much higher than the 25 Hz provided by the TOF sensors. When running the same control system in the same corridor as when the practice mouse had TOF sensors, the mouse was now able to stay centered between the two walls.

The practice mouse, with the analog sensors and motors installed, could drive forward a set distance reliably, turn 90° left or right, turn 180°, stop at a set distance from a wall in front of it, and stay centered between two walls. We ran the mouse on a S-shaped portion of track that went forward 5 units, did a U-turn, went forward another 5 units, did another U-turn, and went

forward a final 5 units. The practice mouse with our intended motor and sensors, traversed the whole S-curve back and forth over 20 times before it crashed. This was proof to our group that our hardware decisions were good enough to move forward with designing and ordering our own PCB. The code that caused the crash would be tweaked once run on our own mouse to prevent crashes from ever happening.

First Board Revision

Upon receiving the first board, assembly began using the automatic pick and place machine for stock parts and manual pick and place for non-stock parts, with solder paste applied beforehand using an ordered stencil mask. The populated board was then inspected under a microscope to verify component positioning before being sent through the reflow oven. After reflow, any visible shorts were corrected manually, and a smoke test was performed by plugging in the board without any code loaded in order to observe how it handled power. The 3.3V LDO was noted to heat up rapidly, indicating a short, and the board was immediately unplugged.

To isolate the fault, an unpopulated board was probed to confirm layout integrity, which checked out. The LDO was then removed from the populated board and wires were soldered to its pads so that a bench power supply could be connected with current limiting enabled. Probing of suspected components revealed that current draw fluctuated when contact was made with the motor driver. The motor driver was removed using a heat gun and resoldered by tinning all the pads, placing the board on a PCB heater with the component positioned on top, and using hot air from above to reflow the joint.

After successful resoldering, the 3.3V short was eliminated, but no output voltage could be measured on either motor PWM line. Probing every pad on the driver revealed that the schematic symbol used for the motor driver had been associated with an incorrect footprint, leaving many pins misaligned with their intended pads. With a board redesign now required, the remaining components were tested for functionality. While attempting to read analog IR values, it was discovered that the NMOS used to enable the IR diode also had flipped pins in the layout, preventing it from turning on. The pushbutton was found to be wired incorrectly as well, with one terminal tied to ground and the other to an MCU pin, meaning a press would only ground the pin and never produce a logic high. A pullup resistor was added in the next revision to resolve this. Finally, the footprint chosen for the OLED I2C connector had too fine a pitch to mate with the actual OLED connectors, so a corrected footprint was selected for the next revision.

Second Board Revision

The revised board was ordered and assembled using only the manual pick and place process, with positioning and shorts checked both before and after reflow. Once the shorts were addressed, code was successfully uploaded on the first attempt, and one motor channel produced the expected 6V output. The second motor channel, however, remained nonfunctional. Reflowing and fluxing the motor driver did not resolve the issue, and probing each driver pin revealed no obvious electrical fault. The test code was rewritten, after which both motor channels functioned correctly. The motor connectors were then soldered to the PCB and probed once more to confirm operation.

During a design review with Professor Schafer, the board failed to power on, and the 3.3V regulator was again found to be running hot, indicating another short on the 3.3V rail. Debugging proceeded by removing components one at a time with a heat gun, beginning with the motor driver and progressing through several passives, each of which was resoldered after being cleared. The fault was eventually traced to the microcontroller itself, whose pins tested as internally shorted. A replacement MCU was installed along with the motor driver, and full functionality was restored. After some additional probing of the motor driver outputs, the board stopped working again. It was then recognized that probing the motor connectors had been shorting the 6V and 3.3V rails together, likely damaging an MCU pin in the process. A new MCU was installed, and from that point forward, output voltages were verified using a connector with wired leads rather than direct probing on the connector pads.

Following verification, the motors were transferred from the practice board to the new board, and test scripts were written to exercise every component on the system. All subsystems were confirmed functional. The final issue encountered was intermittent code uploads, where flashing would proceed for a short time and then halt before succeeding again on a later attempt. This behavior was attributed to eFuse settings in the microcontroller related to flash supply voltage configuration on the pin one of our motor encoders was wired to (IO45). The following eFuses were burned to resolve the issue:

```
burn_efuse VDD_SPI_XPD 1
burn_efuse VDD_SPI_FORCE 1
burn_efuse VDD_SPI_TIEH 1
```

After burning these eFuses, no further hardware issues were observed and the board operated reliably for the remainder of testing.

Software Overview

The software for the mouse robot was developed in a layered fashion, with each module validated in isolation before being integrated into the full firmware stack. Development proceeded from the lowest-level peripherals up to the highest-level decision making, beginning with analog sensing, followed by motion control, then maze representation and search, and finally the OLED display and top-level program loop. Critically, all of these modules were prototyped and exercised on a practice board before the custom PCB was even assembled, which allowed early development of test code for sensing, motion, and maze solving. That early test code became the foundation of the validation suite used on the final hardware.

A consistent two-stage testing strategy was used throughout the project. New functionality was first verified on a smaller 5x3 test maze, which allowed for rapid iteration and quick visual confirmation of correct behavior. Once a feature appeared stable on the small maze, it was promoted to the full 16x16 competition maze, where edge cases and failure modes that did not appear in the smaller environment could be surfaced. When issues arose at the 16x16 scale, they were typically reproduced on the 5x3 maze for fast debugging and fixed there before being revalidated at full scale. This loop kept the iteration cycle short while still ensuring that the final firmware was exercised against the full problem domain.

Analog Sensor Validation

The analog sensing module was the first component validated because every higher-level behavior depends on accurate distance readings. Calibration was performed by placing a maze wall at a series of measured distances from each IR sensor and recording the raw ADC readings at each distance. These data points were collected in a spreadsheet, and a curve fit was applied to derive a closed-form expression mapping ADC counts to physical distance in millimeters. The resulting equations were then implemented as conversion functions inside **`Analog_Sensors.cpp`**.

Validation of the calibration was performed by reversing the process. After implementing the conversion functions in firmware, the mouse was placed at known distances from a wall and the converted distance values were displayed over the serial console. These displayed distances were then compared against physical measurements taken with a ruler. Any sensor whose computed distance did not closely match the physical measurement was recalibrated, and the curve fit was refined until all four sensors produced reliable readings across the full range of distances expected during a maze run. Wall-presence threshold logic, which is consumed by the maze module to detect openings versus walls, was validated separately by placing the mouse in known cell configurations and confirming through serial output that the correct walls were being detected.

Motion Control Validation

Motion control was validated using a hybrid approach that combined first-principles calculations with empirical tuning. Initial parameters for translation and rotation were derived mathematically from known physical quantities including wheel diameter, gear ratio, and encoder counts per revolution. These computed values gave a reasonable baseline for distance-per-tick conversions, target encoder counts for one-cell forward motion, and target counts for in-place turns. With the baseline values in place, the mouse was driven through repeated forward and turning maneuvers on the practice board and on the 5x3 test maze, and the resulting motion was observed against the expected outcome. Gains and target counts were then adjusted empirically until straight-line drives, in-place rotations, and cell-to-cell motion were consistent and repeatable.

The closed-loop wall-following PID was tuned in a similar manner. After establishing baseline gains, the mouse was driven through corridors of the test maze and observed for drift, oscillation, and centering behavior. Gains were iterated until the mouse held a stable centerline through long corridors without overcorrecting at cell transitions.

Two specific motion control issues were identified during validation and documented here as representative examples. The first was a failure mode at four-way intersections, where the PID lost both wall references simultaneously. As the mouse entered an open intersection, the side walls dropped out and the PID switched into a no-walls mode that did not apply the motor asymmetry correction normally used during wall-following. This caused the mouse to drift into a wall before exiting the intersection. The issue was further compounded by the fact that the two side walls did not always disappear simultaneously, which produced a brief single-wall lock followed by a discontinuous transition into the no-walls mode and a corresponding steering jerk. The fix involved continuing to apply the motor trim correction in the no-walls mode and avoiding a full PID state reset on transitions into and out of that mode, which preserved smooth steering through intersections.

The second issue arose from the geometry of the mouse and the timing of wall detection at intersections. In certain configurations the mouse would detect a turn opportunity only after it had partially passed the intersection, at which point a standard in-place turn would clip a wall. To handle this, a reverse turn maneuver, referred to internally as the goofy turn, was implemented. In this routine the mouse drives slightly past the intersection, backs up a measured distance, and

then executes the turn in reverse so that it ends up properly aligned in the new corridor. This maneuver was validated on the 5x3 maze before being used in the full 16x16 environment.

Maze Representation and Search Validation

The maze module was validated almost entirely through comparison of the mouse's internal map against the known physical maze. After every exploration run, the maze module printed its discovered wall map over the serial console using **`maze_print_serial`**. This serial output was then compared cell by cell against the physical layout of the test maze to verify that walls had been correctly detected, recorded, and stored in the maze data structure.

Validation began on the 5x3 maze, where the small cell count made it easy to inspect the full map at a glance and quickly identify any mismatches between the discovered and physical layouts. Once the algorithm was producing correct maps consistently on the 5x3 maze, testing was extended to progressively larger configurations and ultimately to the full 16x16 maze. At the larger scale, additional validation was performed by enabling the OLED display to show the mouse's current believed coordinates in real time during a run. This allowed the team to watch the mouse's perceived position update as it moved through the maze and to catch coordinate tracking errors that might otherwise have been masked by an outwardly correct final map.

The breadth-first search routines for returning to the start and for computing the optimized speed run path were validated by inspecting the planned path printed over serial and verifying that it matched the shortest path through the recorded map. The mouse was then commanded to execute the planned return and the behavior was observed against the expected trajectory.

OLED Display Validation

The OLED display served as both a user-facing status indicator and a secondary debugging tool. Validation focused on confirming that the display initialized correctly at startup, that text and coordinates rendered legibly during operation, and that the I2C bus initialization did not interfere with other peripherals or block the rest of the startup sequence. Initialization was verified by checking the boolean returned by the display library's begin call and by printing a status message over serial indicating whether allocation had succeeded. Runtime behavior was validated by writing the mouse's current coordinates to the display during exploration runs and confirming that the displayed values matched the coordinates printed over serial.

Integration Testing on the Assembled Board

Once all individual modules were validated, integration testing was performed on the fully assembled custom PCB. Test scripts that had originally been developed on the practice board were ported over and run against every component on the new board. These scripts exercised each motor channel independently, swept the IR emitters and read back the corresponding sensor channels, drove the OLED with test patterns, and read the user button. The scripts were used both as initial bring-up verification immediately after a board was assembled and as regression tests after any subsequent fix or hardware modification. After every meaningful change, whether a code fix or a component rework, the relevant component test scripts were rerun to confirm that the change had not introduced new failures elsewhere on the board.

The final integration test was a complete maze run on the full 16x16 maze. The mouse was placed at the start, ran a complete exploration, printed its discovered map, returned to the start using the BFS path, and was then commanded to execute the optimized speed run. Successful completion of this end-to-end sequence, with a correctly mapped maze and a clean return-and-solve, was the criterion used to declare the software validated for competition use.

Conclusions

On race day, our micromouse did not complete the full maze, but it demonstrated meaningful progress toward the original projective objective of navigating a 16-by-16 micromouse maze. In its best run, the mouse made it approximately two-thirds of the way through the maze. This was farther than it was able to travel in any of its other competition attempts. Although the final outcome was not a complete solution, the race confirmed that many of the major subsystems were functional, including power delivery, motor control, analog wall sensing, coordinate display, and basic autonomous decision-making. These were the core capabilities required for the mouse to move through the maze without external control.

Our main point of failure during race day was not completely hardware-related, but a reliability issue during navigation. The mouse generally performed best when it was guided by one or two nearby walls. In those situations, the wall-following control system could use the analog sensor readings to correct its position and keep the mouse aligned. However, the mouse struggled most during turns and transitions where it moved from being guided by a wall to having no wall reference. This was especially noticeable at openings and intersections. In several cases, the mouse was able to continue only because it happened to remain close enough to the correct path after the transition. In other words, the mouse “got lucky” multiple times and avoided becoming physically stuck, but the underlying localization error continued to grow.

As the run continued, this localization error began to compound as our OLED display, which showed the mouse’s believed coordinates in the maze, eventually became inaccurate. This indicated that the software’s internal position estimate no longer matched the mouse’s actual location. Once the coordinate tracking became offset, the maze-mapping and path-planning logic could no longer make correct decisions. Even when the mouse was still physically moving, it was effectively lost because it was making decisions based on the wrong cell location. This explains why the mouse could make progress through part of the maze but could not reliably continue to the goal.

Overall, our project was successful in producing a custom autonomous micromouse platform with functional hardware, calibrated analog sensing, encoder-based motion control, OLED debugging, and integrated maze-search software. The results from race day showed that the system was close to being capable of extended maze navigation, but that the transition between wall-guided motion and open-space motion needed more refinement. If we were to

make a LockhEEd Mousetin Board V3, there are a few changes we would like to make. Firstly, we would do everything possible to make the mouse smaller. With a smaller mouse, you have greater margins of error before hitting walls. If the mouse were much smaller, the wheel placement at the back of the mouse would not have been an issue. Having the wheels at the back and a mouse board so big forced us to develop our “goofy turns” that ate up lots of our development time. By either changing the mouse size or the motor position, these problems would be solved. Additionally, more sensors would be helpful. If we had two sets of sensors on the sides of the mouse, squaring up to walls would have been much easier. Finally, an inertial measurement unit (IMU) would have been a welcome inclusion on the board. This sensor measures forces, velocity, and orientation. This could have been used to help square up the mouse on turns, ensuring perfect 90° turns. With these three simple changes, we believe that our mouse could have completed the maze with ease and at higher speeds.