

RicKy JoVi Final Report

EE Senior Design 2026



José Aceves, Eric Foreman, Kyle Pulido, and Vivek Attupuram

Table of Contents

1 Introduction.....	3
Problem Statement.....	3
Solution.....	3
Evaluation Against Expectations.....	3
2 Key System Requirements.....	4
2.1 Overall System Requirements.....	4
2.2 Power and Electrical Requirements.....	4
2.3 Software Requirements.....	5
3 Detailed Project Description.....	5
3.1 System Theory of Operation.....	5
3.2 System Block Diagram.....	6
3.3 Hardware Design and Operation.....	7
3.3.1 Overall PCB Design and Main Controller.....	7
3.3.2 Motor Driver and Motor System.....	10
3.3.3 Wall Sensing Hardware and Behavior.....	12
3.3.4 Power System.....	16
3.3.5 User Interface and Debug Hardware.....	17
3.3.6 Mechanical Design and Chassis.....	19
3.4 Software Design and Operation.....	21
3.4.1 Overall Software Structure.....	21
3.4.2 Encoder-Based Pose Tracking and Motor Control.....	23
3.4.3 Maze Mapping and Flood-Fill Navigation.....	24
3.4.4 Software Debugging and Visualization.....	25
4 System Testing.....	27
5 Conclusions.....	29

1 Introduction

Problem Statement

The goal of this project was to design, build, and race an autonomous robot mouse capable of solving a maze and then making the fastest run from start to finish. Our team was expected to complete the design of the mouse, build a working prototype, and produce final design documentation. The main challenge was creating a small robotic system that could navigate a maze autonomously while detecting walls, tracking its position, and making navigation decisions in real time.

For a maze-solving robot to work, the hardware and software must operate together reliably. The robot mouse has to drive straight, turn accurately, sense and respond to walls, update its knowledge of the maze, and choose where to move next. Even small errors in sensing, motor control, or turning can cause the robot to lose alignment and eventually crash into a wall or lose track of its position. Because of this, the project required efficient integration of motors, sensors, embedded software, power management, and mechanical design into one functional system.

Solution

Our team designed an autonomous robot mouse using an ESP32-S3 microcontroller as the microcontroller. The robot used N20, 150 RPM, 1:100 ratio DC motors driven by a DRV8847RTE motor driver for movement, and VL53L4CD time-of-flight sensors for distance sensing. The software was designed to control motor movement, read sensor data, track the mouse's position in the maze, detect walls, and make decisions using a flood fill maze-solving algorithm.

To allow for successful solving of the maze, the robot had to explore the maze while navigating to the centre, record wall information, return to the start, and eventually perform a faster run using the most optimal path it learned. The system combined low-level motor control with higher-level navigation logic. Motor routines controlled driving straight, turning 90 degrees, and making U-turns, while the sensing and maze mapping logic helped determine which action the robot should take next.

Evaluation Against Expectations

The final robot did not complete the maze on race day. However, one of the main successes of the project was getting the full system to work together as an integrated prototype and making it halfway through the maze. By the end of the project, the ESP32-S3, motor driver, N20 motors, encoders, time-of-flight sensors, and navigation software were able to operate together in one

system. This was a big accomplishment because many of the challenges came from integration rather than from individual systems themselves

Reflecting on our performance, we were able to realize some important issues. One hardware issue was that the ISENSE pins on the DRV8847RTE motor driver were left floating. This led to faulty shut downs of our motor driver. We corrected this issue by soldering these pins to nearby ground pads using bodge wires. Another issue was related to pin 45 on the ESP32-S3, which was pulled high and created problems when uploading and running code due to us using this pin as a motor encoder input. Because of this, the right motor had to be unplugged every time we wanted to upload code or to reset the mouse. There was only one ToF sensor in the front of the mouse in the final design. Due to only relying on the two side sensors and our motor encoders, the mouse could straighten itself as it ran through the maze, but not as accurately as could have been. This in turn, negatively impacted our turns, causing the mouse to turn too early or turn too late during critical moments. We believe an additional front sensor could have enabled us to eliminate more edge cases and have our mouse get lost less often.

Overall, the project showcases the difficulty of building a reliable autonomous maze-solving robot mouse. Even though the robot did not complete the maze, the final prototype still showed meaningful progress in system integration, embedded control, and autonomous navigation. The design process also revealed clear future improvements.

2 Key System Requirements

The robot mouse was designed around the main requirement of autonomously navigating a maze, learning the wall layout, and attempting a fastest run from start to finish.

2.1 Overall System Requirements

The full system must be compact, stable, and reliable enough for repeated testing. The system also needs to provide debugging feedback through serial output, LEDs, and buttons so problems can be identified during testing.

2.2 Power and Electrical Requirements

The robot requires separate voltage levels for the motor system and logic electronics. The motor driver and N20 motors require a higher voltage rail, while the ESP32-S3, VL53L4CD sensors, and logic components require a regulated 3.3 V supply. The PCB design had to include a battery input, USB-C connection, a 6 V power path for the H-bridge motor driver, and a 3.3 V regulator for the microcontroller and sensors.

The board needs to support programming and debugging through USB-C while also allowing the robot to run from battery power during testing and competition. The design also requires reset and boot buttons for programming the ESP32-S3, connectors for the motors and encoders, connectors for the ToF sensors, and indicator LEDs for basic system feedback. An additional button for mode switching and starting the algorithm would also be required.

2.3 Software Requirements

The software needs to connect the low-level hardware functions with the high-level maze-solving behavior. At the low level, the software has to read encoder counts, measure distances from the ToF sensors, control motor speed and direction through the DRV8847RTE motor driver, and perform straight-line and turning movements. Effective straight line navigation requires well tuned PD controllers based on encoder counts and sensor data and accurate 90 and 180 degree turns require callable functions given the desired turn angle.

The software needs to track the robot's current cell and orientation, record wall information, and use a flood-fill algorithm to choose the next movement. The robot has to explore the maze, reach the center, return to the start, and then attempt a faster run using the learned path. The software also needed to provide debugging information through serial output and LED indicators and potentially even WiFi links for ease of testing.

3 Detailed Project Description

3.1 System Theory of Operation

The mouse is an autonomous maze-solving robot controlled by an ESP32-S3 microcontroller. The ESP32-S3 acts as the main control unit by reading distance sensors, tracking encoder counts, controlling the motor driver, and running the maze-solving algorithm. The robot uses a differential drive system, meaning the left and right N20 DC motors are controlled independently. By changing the speed and direction of each motor, the mouse can drive straight, turn left, turn right, or make a U-turn.

When the robot is powered on, the ESP32-S3 initializes the motor control pins, encoder inputs, I2C communication for the time-of-flight sensors, and the maze-solving software. The mouse begins at cell (0,0) facing north. Once the run begins, the software initializes the wall map, pre-loads the outside boundary of the maze as walls, and runs the flood-fill algorithm to calculate distances from the goal cells.

During operation, the robot continuously runs two main software tasks. The sensor task reads the VL53L4CD time-of-flight sensors and updates the most recent left, front, and right wall distances. The motor task updates encoder-based distance and angle estimates, controls motor

speed, records new wall information, and decides when to move straight, stop, or turn. This task-based structure allows sensing and movement control to happen repeatedly without waiting for one process to fully finish before the other begins.

As the mouse drives forward, encoder counts are converted into distance traveled. The robot uses this distance to determine when it has entered a new maze cell. After entering a cell, the software records wall information using the current sensor readings, updates the internal wall map, reruns the flood-fill algorithm, and chooses the next action. The possible actions are to continue straight, turn left, turn right, turn around, stop, or declare that the robot is stuck.

The motor control system uses encoder feedback to estimate the robot's forward distance and turning angle. While driving straight, the robot uses wall distance information and angle correction to keep itself aligned. If both side walls are detected, the mouse compares the left and right distances and adjusts the motor speeds to stay centered. If only one side wall is detected, the mouse uses that wall as a reference. When a front wall is detected, the robot slows down, stops near the target distance, records the wall information again, and then chooses the correct turn based on the flood-fill decision.

The intended full run consisted of three phases: exploring to the center, returning to the start, and performing a faster run back to the center. In the first phase, the flood-fill goal was the center of the maze. After reaching the center, the goal changed to the start cell. After returning to the start, the goal changed back to the center for the final run. This allowed the robot to use the wall information it learned during exploration to choose a better path during the final run.

3.2 System Block Diagram

Figure 1 shows the overall system block diagram for the RicKy JoVi robot mouse. The system is centered around the ESP32-S3 microcontroller, which acts as the main controller for sensing, motor control, and maze-solving logic. The ESP32-S3 receives distance measurements from the VL53L4CD time-of-flight sensors and encoder feedback from the motors. Using this information, the software determines the robot's position in the maze, records wall information, and chooses the next movement using the flood-fill algorithm.

The ESP32-S3 controls the DRV8847RTE motor driver, which drives the two N20 DC motors used for differential drive motion via PWM inputs. The power system provides a regulated 3.3 V rail for the ESP32-S3 and sensors, along with a separate motor voltage rail for the motor driver and motors. User interface components, including buttons and LEDs, provide basic control and debugging feedback during testing.

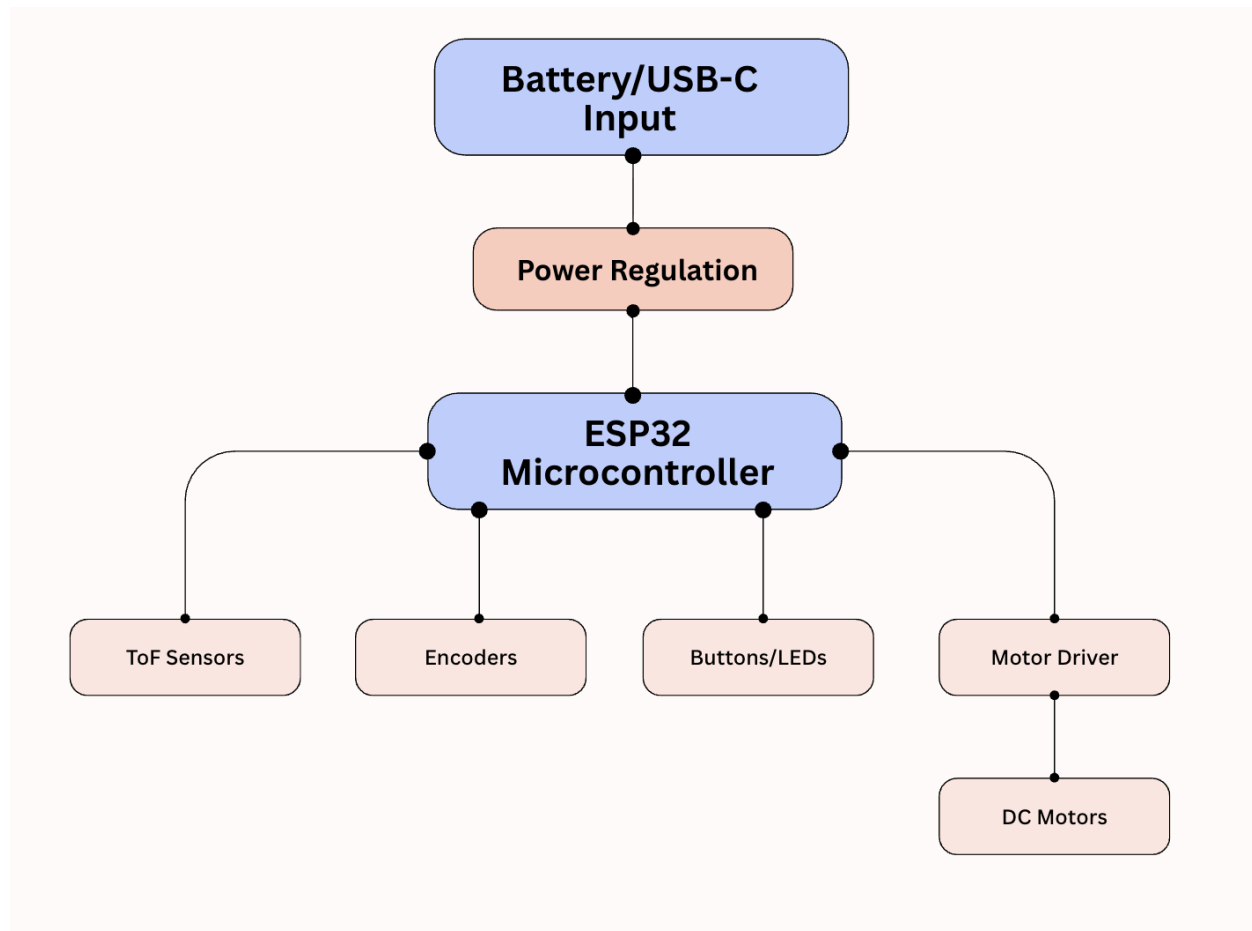


Figure 1. Overall system block diagram of the robot mouse.

3.3 Hardware Design and Operation

3.3.1 Overall PCB Design and Main Controller

The robot mouse was designed around a custom printed circuit board that used the ESP32-S3 as the main controller. The ESP32-S3 was chosen because it provided enough GPIO pins for the sensors, motor driver, encoders, buttons, and LEDs while also supporting the embedded software needed for autonomous maze solving. The microcontroller handled all major system functions,

including reading distance sensors, tracking encoder counts, controlling motor movement, and running the flood-fill navigation algorithm.

Figures 2 and 3 show the overall schematic and 3D rendering for the custom robot mouse control board. The schematic was used to organize the electrical connections between the ESP32-S3, motor driver, sensors, power system, buttons, and LEDs. The 3D rendering shows how these components were placed onto a compact board that could mount directly onto the robot chassis. Designing a custom PCB helped reduce wiring complexity and made the robot more reliable than a breadboard-based prototype. Since the mouse had to move repeatedly through the maze, loose jumper wires and breadboard connections would have created reliability problems.

The PCB provided fixed connections for the ESP32-S3, DRV8847RTE motor driver, voltage regulators, time-of-flight sensor headers, motor connectors, battery input, USB-C programming port, buttons, and indicator LEDs. The board also included boot and reset buttons, which were needed to place the ESP32-S3 into programming mode and reset the controller during testing. Indicator LEDs were included to provide basic feedback about power and operating mode.

The ESP32-S3 was placed near the center of the PCB so that signals could be routed to the different subsystems around it. The ToF sensor connectors were placed along the top and sides of the board to allow sensors to be mounted facing the front, left, and right sides of the robot. The motor connectors and motor mounting locations were placed near the bottom of the board, close to the wheel locations. The battery and USB-C input were placed toward the side of the board so power and programming connections would be accessible during testing.

One important issue discovered during testing was related to the ESP32-S3 pin selection. Pin 45 was pulled high during startup, which created problems when uploading code and running the mouse. Because of this, the right motor had to be unplugged whenever code was uploaded or when the mouse was started. This showed the importance of checking ESP32-S3 boot and strapping pin behavior before assigning pins to motor control signals. In a future revision, pin 45 should be avoided for motor driver control or any signal that could interfere with startup and programming.

Another hardware issue involved the DRV8847RTE motor driver ISENSE pins. In the original PCB design, the ISENSE pins were left floating. During testing, this caused motor driver issues and required a manual solder fix. In a future board revision, the ISENSE pins should be connected properly according to the motor driver requirements instead of being left floating. This would make the motor driver behavior more reliable and avoid needing manual rework after fabrication.

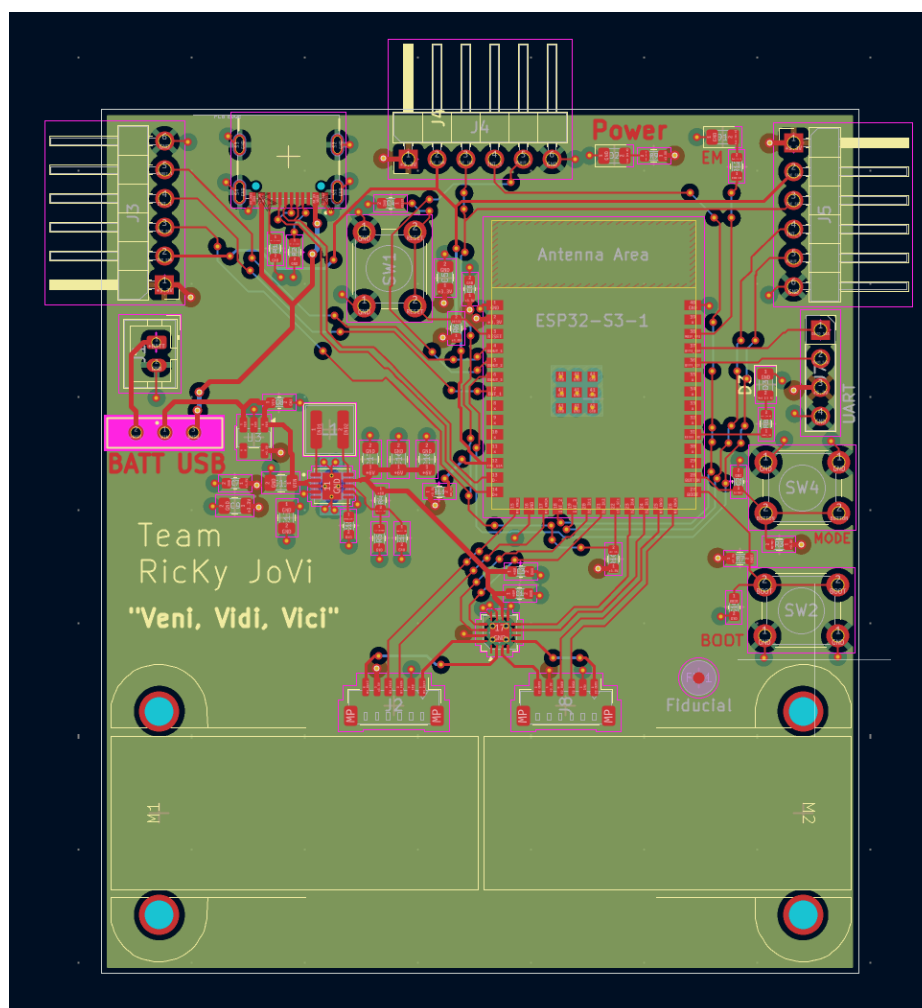


Figure 2. Overall PCB schematic for the robot mouse, showing the ESP32-S3 main controller, power regulation, motor driver, ToF sensor headers, USB-C programming circuit, buttons, and indicator LEDs.

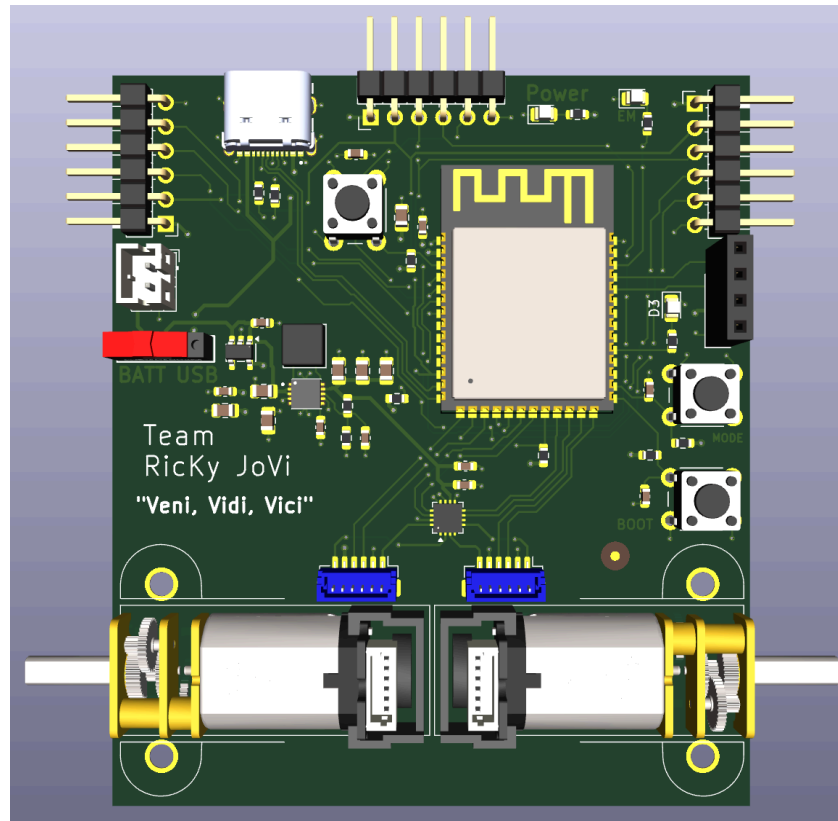


Figure 3. 3D rendering of the custom RickY JoVi robot mouse PCB. The rendering shows the ESP32-S3 module, USB-C connector, sensor headers, buttons, LEDs, motor connectors, and mounting locations for the two N20 motors.

3.3.2 Motor Driver and Motor System

The robot mouse used two N20 DC motors in a differential drive configuration. In this type of drive system, the left and right motors are controlled independently so that the mouse can move forward, turn left, turn right, and perform a U-turn without the need for a separate steering mechanism. This was a good choice for maze navigation because it allowed the robot to turn within the limited space of a maze cell while keeping the mechanical design relatively simple.

The two motors were driven by a DRV8847RTE dual H-bridge motor driver. This motor driver was selected because it is capable of controlling two DC motors from a single compact integrated circuit, which helped reduce board area and simplified the overall PCB design. The

DRV8847RTE received logic-level input signals from the ESP32-S3 and used the 6 V motor supply to drive the outputs connected to the motors. By controlling the input signals with pulse-width modulation (PWM), the ESP32-S3 could vary the motor speed and direction as needed for forward motion and turning.

Figure 4 shows the motor control portion of the schematic. The DRV8847RTE is powered from the 6 V motor rail, with local decoupling capacitors included to stabilize the supply. The outputs from the motor driver are routed to connector J2 for one motor and connector J8 for the second motor. These same connectors also include the encoder signals and 3.3 V supply needed by the motor encoders. This allowed the motors and encoders to be connected to the board through a single connector for each side of the mouse.

The N20 motors were chosen because they were compact enough to fit on the mouse while still providing enough torque and speed for maze movement. Initially, we bought 500rpm N20 motors to enable our mouse to move quickly. After driving testing with our practice mouse however, we discovered that the low gear ratio of these motors made it incredibly challenging to drive slowly and perfect our straight line navigation. The wheels wouldn't get turning until they had an applied PWM of near 70, which led to imbalanced, inconsistent driving. We then decided to switch to 150rpm N20 motors with a higher gear ratio to allow our robot to move in a more controlled manner at lower speeds. During testing, the minimum PWM values required to move the motors were approximately 37 for the left motor and 40 for the right motor. This gave the system a useful operating range, since the motors could move at relatively low speeds while still allowing higher PWM values for faster motion later.

Motor encoder feedback was an essential part of the system because the software used encoder counts to estimate both linear distance and turning angle. As the robot moved forward, encoder counts were converted into traveled distance so the mouse could estimate when it had crossed into a new maze cell. During turns, the encoder counts from the two motors were also used to estimate the robot's rotation. This made encoder consistency very important, since any variation in the counts per revolution would cause position and angle error to accumulate over time.

One of the most important hardware issues discovered in this subsystem involved the ISEN12 and ISEN34 pins of the DRV8847RTE. In the original PCB design, these current-sense related pins were left floating. During testing, this caused motor driver problems and required a manual solder fix in order for the driver to behave properly. This issue showed that support pins on the motor driver are just as important as the main input and output pins. In a future revision, these pins should be connected correctly in the original design rather than fixed after fabrication.

Overall, the motor driver and motor system provided the foundation for all robot movement. The subsystem successfully allowed the ESP32-S3 to command straight driving and turning motions, while encoder feedback gave the software the information needed to estimate motion through the maze.

3.3.3 Wall Sensing Hardware and Behavior

In our original design, we used the VL53L0X sensor, which reportedly can sense up to 2 meters away. We purchased breakout boards for practice mouse testing with this sensor, however we found it to be quite unreliable. The distance the sensor reported wasn't very consistent, and they

really struggled to measure distances less than 40mm. Because of this, we explored for an alternative sensor to use before populating our own hardware. Fortunately, we found the VL53L4CD sensor, with the same footprint and pinout that we could drop directly onto our existing PCB with no changes. This sensor is geared towards short range detection and is designed to detect distances between 1mm and 1200mm away, which is much better suited for our functionality. Upon testing our breakout boards with the VL53L4CD sensors, we saw that the readings were much more stable and that we were getting accurate sensor readings down to 10 mm.

The original sensing design used three time-of-flight sensors: one facing forward, one facing left, and one facing right. The front sensor was used to detect walls in front of the mouse and determine when the robot needed to stop or turn. The left and right sensors were intended to detect side walls and help the robot stay centered while driving through the maze. By comparing the left and right distance readings, the software could make small steering corrections to reduce drift.

Figure 5 shows the ToF sensor header connections in the PCB schematic. Each sensor connector included 3.3 V, ground, I2C SDA, I2C SCL, interrupt, and XSHUT connections. The sensors shared the same I2C bus, while the XSHUT pins allowed the ESP32-S3 to control when each sensor turned on. This was important because multiple VL53L4CD sensors start with the same default I2C address, so the sensors need to be enabled one at a time before assigning or using separate addresses.

The VL53L4CD sensors operate from 2.6 V to 3.5 V, so they were powered from the same regulated 3.3 V rail as the ESP32-S3. The main board also included I2C pull-up resistors on the SDA and SCL lines so the sensors could communicate reliably with the microcontroller. The software was designed to read updated sensor measurements every 10 ms, giving a sampling rate of about 100 Hz. This update rate was fast enough for wall detection and alignment correction while keeping the software workload manageable.

During testing, the VL53L4CD sensors produced reliable readings between about 10 mm and 600 mm. The software used distance thresholds to interpret these readings. A short front distance indicated that the mouse was approaching a wall and needed to stop or turn. In our program, our mouse detected a wall ahead when the front sensor detected a wall within 200mm away. It then decelerated and approached the wall until the sensor read a distance between 45 and 55 mm away. Our left and right sensors had a wall detection threshold of 110 mm. If the sensor reading was less than this threshold, our mouse would log a wall in the wall bit map, and if the sensor reading was greater than this 110mm threshold, an opening was detected. Based on our testing, this threshold was sufficient to not miss any walls and in the unlikely event that the car was against one wall, it would still record the opposite wall

The final sensor arrangement was one of the limitations of the project. While we were able to detect whether there was a wall to the left, right, or in front of the mouse quite well, this logging required the mouse to be well placed within a cell. Upon reflection, we believe that if we had a front right and front left sensor, as well as potentially another sensor on each side of the mouse, we would be able to determine how well centered our mouse was within a cell with more accuracy.

Overall, the time-of-flight sensing system gave the robot the ability to detect nearby walls and openings in the maze. However, the final performance showed that sensor placement and side-wall feedback are critical for a maze-solving robot, especially when the robot must stay aligned over multiple cells.

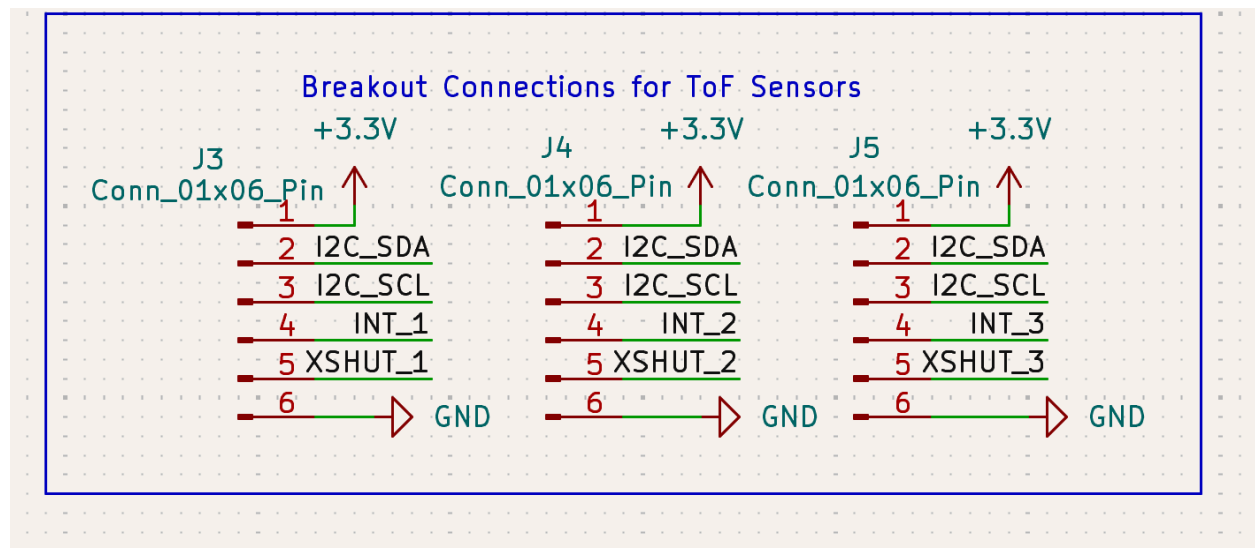


Figure 5. Time-of-flight sensor header schematic showing the front, left, and right VL53L4CD sensor breakout connections

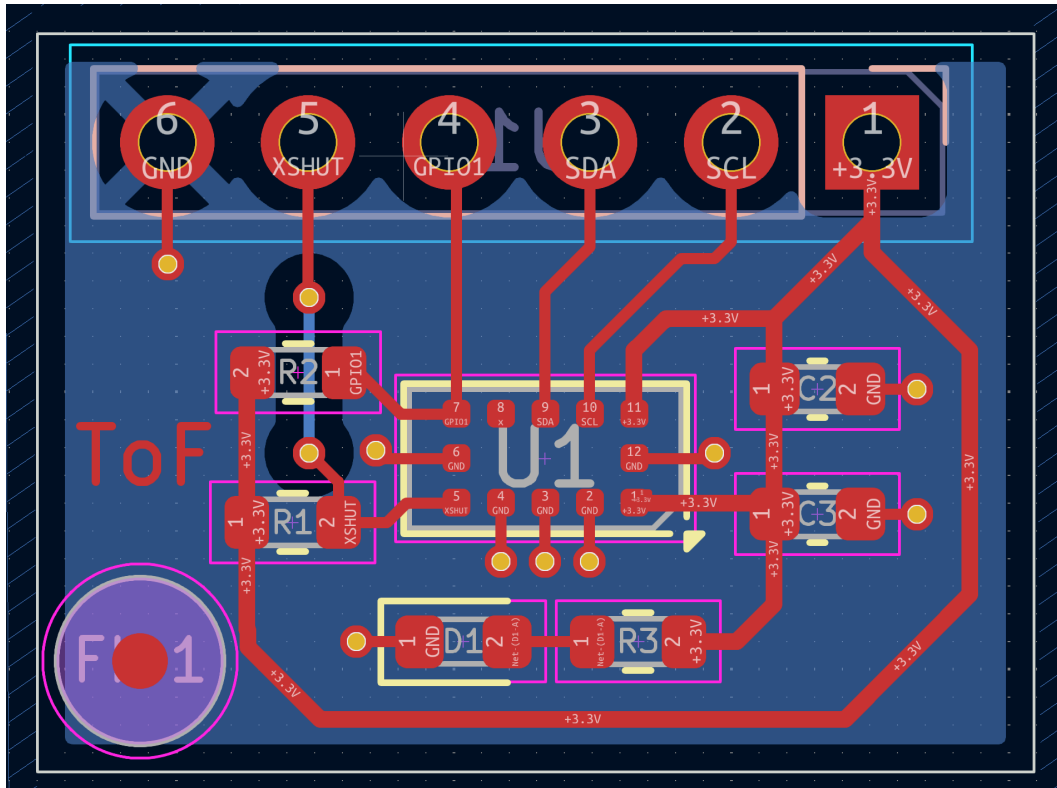


Figure 6. Time-of-Flight VL53L4CD Breakout board PCB layout

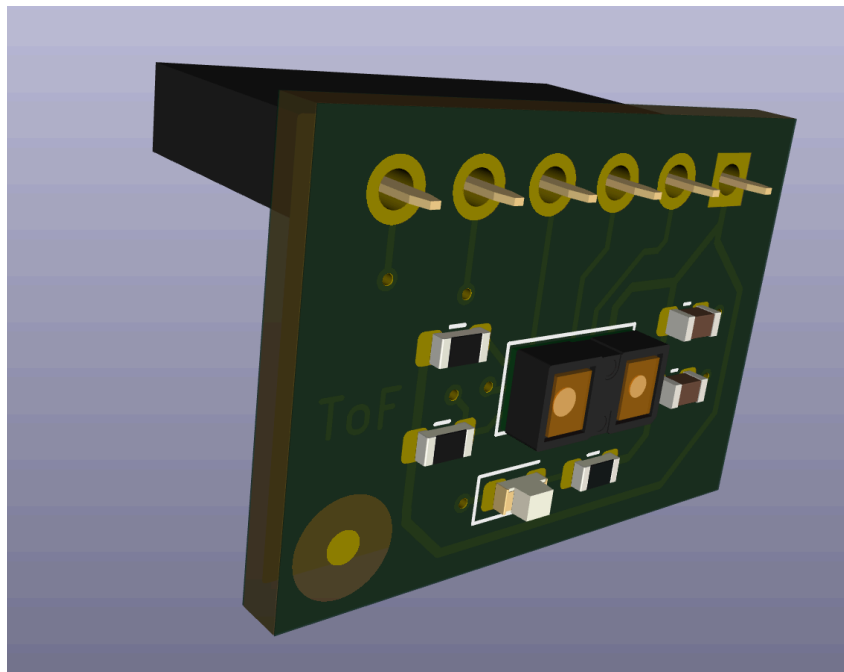


Figure 7. Time-of-Flight VL53L4CD Breakout board 3D populated visual

3.3.4 Power System

The robot mouse required two main voltage rails: a 6 V rail for the motor system and a 3.3 V rail for the ESP32-S3, VL53L4CD sensors, logic signals, buttons, and LEDs. Separating the motor and logic power was important because the motors draw more current and can introduce noise onto the supply line. The logic electronics needed a stable 3.3 V supply so that the microcontroller and sensors could operate reliably while the motors were running.

Figure 8 shows the power path portion of the PCB schematic. The board was designed to accept power from either a battery connector or a USB-C connector. The USB-C connection was mainly included for programming and debugging the ESP32-S3, while the battery input was intended to power the mouse during maze testing and competition. A power switch was included so the user could turn the robot on and off without disconnecting the battery.

The TPS63060 buck-boost regulator was used to create the 6 V rail for the motor driver and N20 motors. This allowed the motor system to receive a consistent voltage even as the battery voltage changed during use. The 6 V rail was routed to the DRV8847RTE motor driver, which then controlled power delivery to the two motors. The buck-boost converter proved to be an effective design choice as our system only required one 3.7V LiPo battery. Using a simple LDO, our system would have required multiple 3.7V batteries in series. This would lead to a bulkier, more expensive mouse.

The AP2112K-3.3 regulator was used to create the 3.3 V logic rail. This rail powered the ESP32-S3 and the VL53L4CD sensors, and it also provided logic-level voltage for other low-power components on the board. The power system included decoupling capacitors near the regulators to help stabilize the voltage rails and reduce noise.

The board also included indicator LEDs for basic system feedback. A power LED allowed the team to quickly confirm that the board was receiving power, while other LEDs were used for debugging and mode indication during testing. These simple indicators were useful because they helped identify whether the issue was related to power, code execution, or motor/sensor behavior.

Overall, the power system was designed to support both development and competition use. USB-C made it easier to program and debug the ESP32-S3, while the battery connector and regulated voltage rails allowed the robot to operate as a standalone autonomous system. The buck-boost TPS63060 enabled a lighter, one battery system. In future revisions, the power system could be improved by further separating noisy motor current paths from sensitive logic and sensor traces.

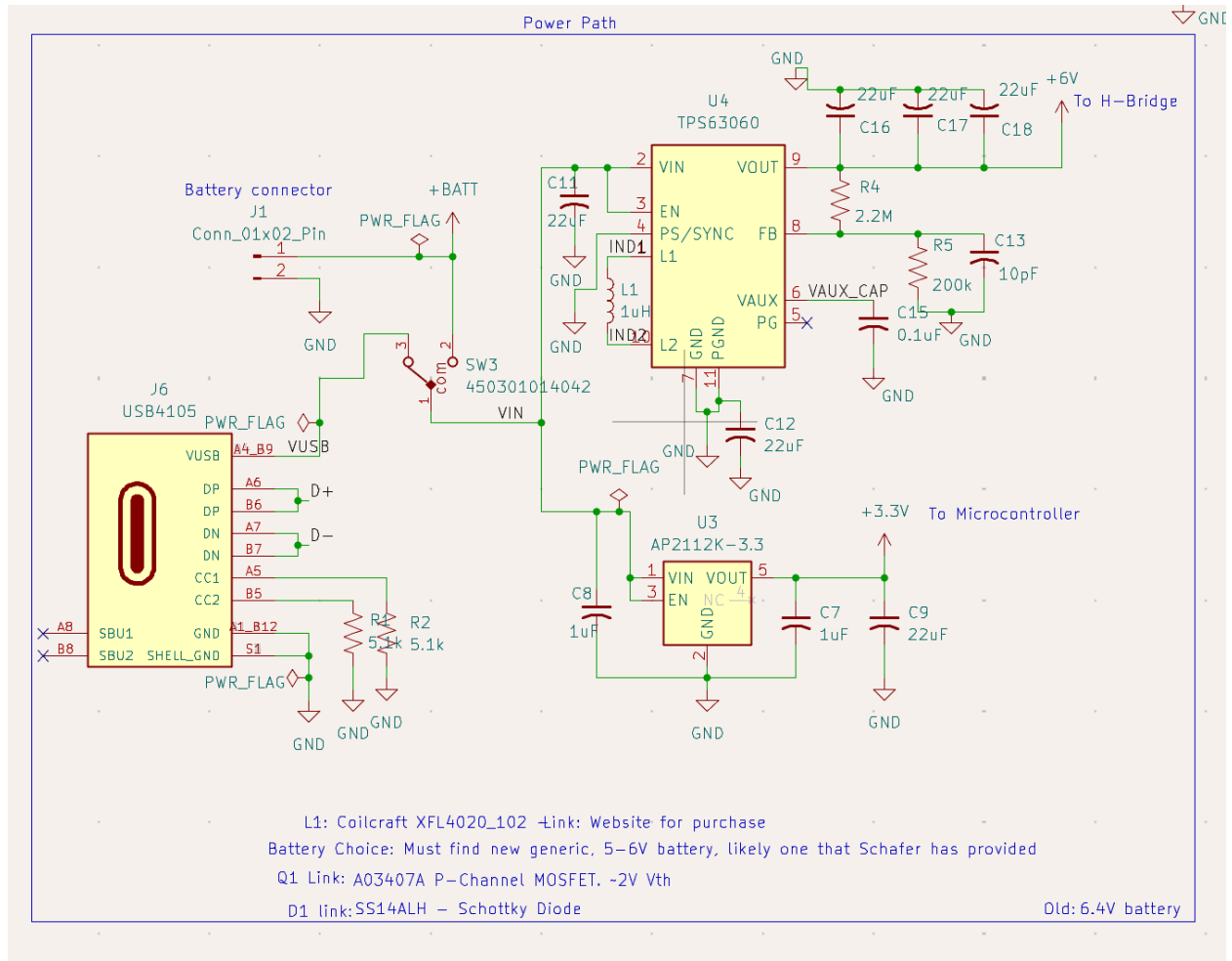


Figure 8. Power system schematic showing the battery input, USB-C connector, power switch, TPS63060 regulator, and AP2112K-3.3 regulator.

3.3.5 User Interface and Debug Hardware

The robot mouse included a small set of user interface and debugging components on the PCB. Since the mouse was designed to run autonomously, it did not need a complex user interface. However, basic buttons and indicator LEDs were still important for programming, starting the robot, resetting the system, and debugging during testing.

The PCB included reset and boot buttons for the ESP32-S3. These buttons were necessary for programming the microcontroller and recovering the board during development. The reset button allowed the ESP32-S3 to restart without disconnecting power, while the boot button allowed the board to enter the correct programming mode when uploading code through USB-C.

The board also included a push button that could be used as a user input during testing. This button was intended to start or change the operating mode of the robot without needing to reprogram the board. Having a physical button on the PCB made it easier to test the mouse repeatedly because the team could place the robot in the maze, press the button, and observe its behavior.

Indicator LEDs were included to provide simple visual feedback. The power LED showed that the board was receiving power. Other LEDs were used for blink or mode indication, which helped confirm that the ESP32-S3 was running code and that the robot had entered the expected operating state. This was useful during testing because many issues could look similar from the outside. For example, if the mouse did not move, the LEDs helped determine whether the problem was related to power, code execution, or the motor system.

Figure 9 shows the button and LED portion of the schematic. These components connected directly to the ESP32-S3 through GPIO pins and provided basic control and feedback without adding much complexity to the design.

An OLED display was used earlier in the project, to help debug issues. It was used to display the sensor statuses, sensor readings, and the PWM being written to the motors. As our project progressed, the OLED was dropped from the final design in favour of a webpage which displayed the maze layout in addition to all this information along with the buttons and LEDs. This kept the hardware simpler and reduced the number of components that needed to be integrated into the robot for the demonstration day and it reduced errors caused by the OLED hitting walls.

Overall, the user interface and debug hardware helped make the robot easier to program, test, and troubleshoot. While these components were not directly responsible for maze navigation, they were important during development because they gave the team a quick way to confirm that the board was powered, running code, and responding to inputs.

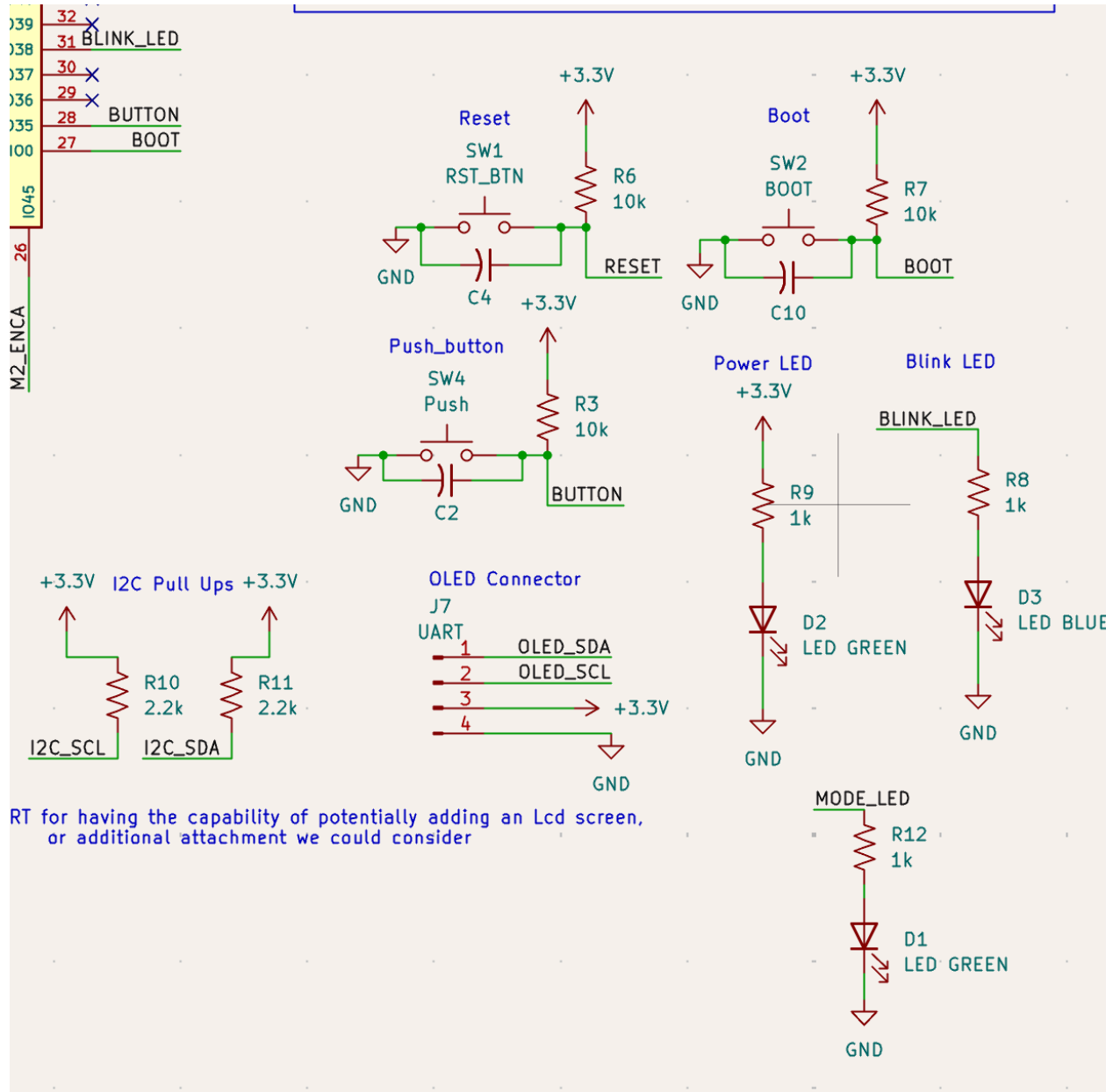


Figure 9. User interface and debugging schematic showing the reset button, boot button, push button, power LED, blink LED, and mode LED connections to the ESP32-S3.

3.3.6 Mechanical Design and Chassis

The mechanical design of the robot mouse had to support the PCB, motors, wheels, battery, and sensors while keeping the overall robot compact enough to fit inside the maze. Since the maze path width was limited, the chassis had to be small enough to move between the walls while still leaving enough clearance for alignment corrections. In addition to fitting within the maze, the

chassis also needed to hold the components in fixed positions so that motor behavior, encoder feedback, and sensor readings remained consistent during operation.

The custom PCB served as both the main electrical board and a major structural part of the mouse. The board included mounting locations for the two N20 motors near the bottom of the design, which helped create a compact differential drive layout. This reduced wiring complexity and allowed the PCB and chassis to function together as a single integrated platform. The placement of the motors and supporting components was chosen to keep the robot stable while driving and turning.

Figure 10 shows the 3D-printed chassis components developed for the robot. The chassis design process involved several rounds of iteration. One important feature that was added was a stub on the bottom of the mouse. This stub went through two iterations before reaching a design that better cleared the space between maze sections. This was important because the bottom geometry of the mouse affected how well it could move through the maze without catching or interfering with the walls.

The battery mount also required multiple revisions. The first version did not hold the battery securely, while the second version extended too far past the body of the mouse. After several iterations, a more suitable design was developed that better fit the overall robot layout. These revisions showed that the mechanical design required just as much testing and refinement as the electronics and software.

The final chassis design provided a way to mount the battery and support the robot structure while working together with the custom PCB. Although the final mechanical system was functional, the design process showed that small physical details, such as bottom clearance and battery placement can have a major effect on the usability of the robot. In future revisions, the mechanical design could be improved further by refining sensor mounting, increasing structural rigidity, and simplifying component access during testing.

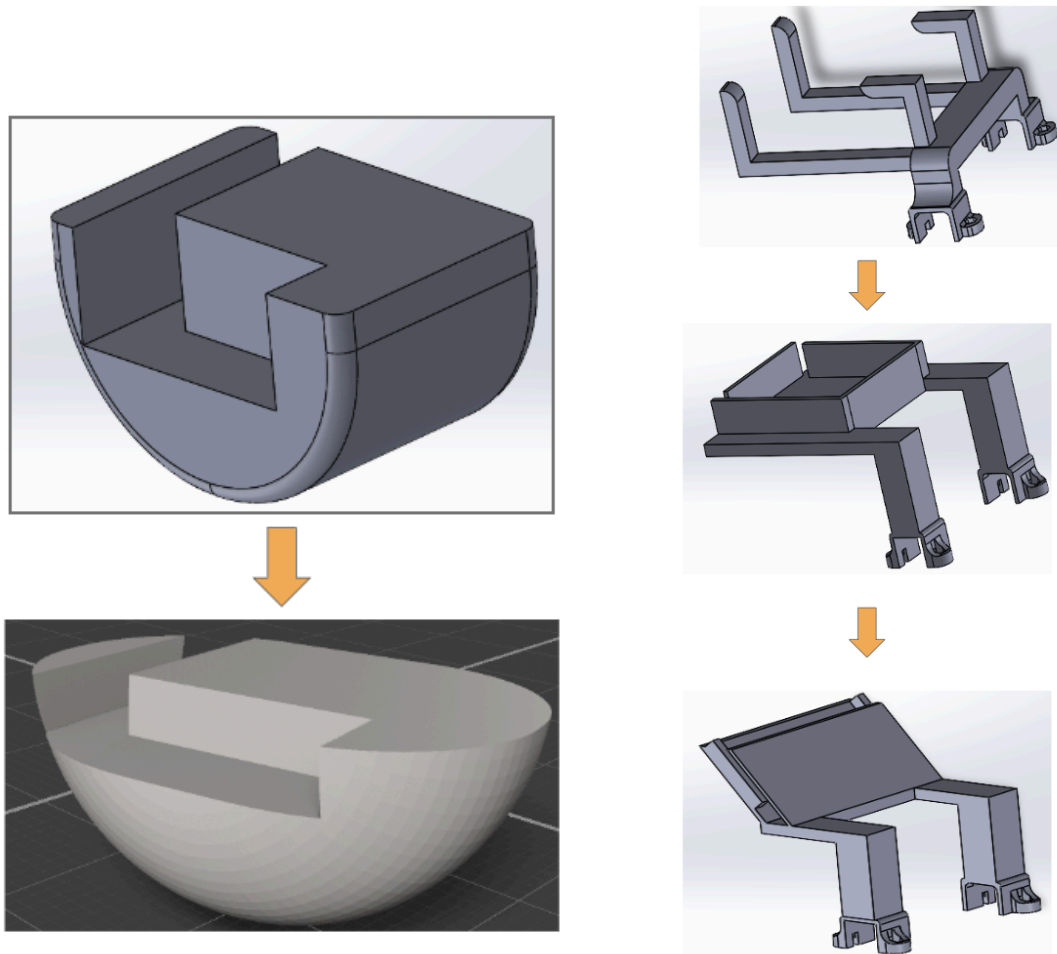


Figure 10. 3D-printed chassis and mechanical support components for the robot mouse.
 The figure shows the bottom stub design and the battery mounting structure.

3.4 Software Design and Operation

3.4.1 Overall Software Structure

The robot mouse software was written for the ESP32-S3 using the Arduino framework in PlatformIO. The software was responsible for connecting the low-level hardware systems, such as the motors, encoders, and time-of-flight sensors, with the higher-level maze-solving logic. The main goal of the software was to allow the mouse to sense walls, track its position, update its maze map, and choose movements autonomously.

The program was organized around two main FreeRTOS tasks: a motor task and a sensor task. The motor task handled encoder updates, motor control, wall recording, pose tracking, and navigation decisions. The sensor task repeatedly read the VL53L4CD time-of-flight sensors and updated the latest distance values for use by the motor task. Splitting the program into separate tasks allowed sensor readings and motor control to run repeatedly without one blocking the other. This was especially aided by the fact that the ESP32-S3 has two cores which could each run a task.

When the ESP32-S3 starts, the software initializes the motor pins, encoder inputs, I2C communication, distance sensors, telemetry/debugging functions, and start button. The mouse is assumed to begin in cell (0,0) facing north. After the start button is pressed, the encoder distance is reset, the wall map is initialized, and the flood-fill algorithm calculates an initial distance map toward the goal cells at the center of the maze.

The software tracks the mouse's position using a "pose" struct that stores the current x and y cell location, heading direction, and distance traveled since entering the current cell. Encoder counts are converted into millimeters traveled, allowing the mouse to estimate when it has moved into the next maze cell. The maze uses 180 mm cells, so the program uses this distance as the baseline for updating position.

The program uses a two-stage cell update method. At about 150 mm of travel, the software advances the mouse's internal position into the next cell and records wall information using the current sensor readings. At 180 mm of travel, the software resets the distance baseline to match the true maze cell spacing. This method was used because the mouse sometimes needed to make decisions slightly before reaching the exact center of the next cell, especially when approaching a front wall. It also helped ensure that cells were recorded with enough buffer between exiting the cell and entering it so that no cell was missed during the position tracking.

After entering a new cell, the software records any detected walls, reruns the flood-fill algorithm, and decides the next movement. The possible actions are driving straight, turning left, turning right, turning around, stopping, or declaring that the mouse is stuck. These decisions are stored as movement actions, which the motor task then executes at the correct point in the cell.

The full intended run was divided into three phases: exploring to the center, returning to the start, and performing a speed run back to the center. In the first phase, the flood-fill goal was the center of the maze. After the mouse reached the center, the goal changed to the start cell. After returning to the start, the goal changed back to the center so the mouse could use the wall information it had already learned to attempt a faster run.

Overall, the software structure allowed the robot to combine real-time sensing, encoder-based motion tracking, wall mapping, and flood-fill navigation. While the final robot did not complete the maze, the code successfully integrated the main parts needed for autonomous maze solving.

Code Evolution

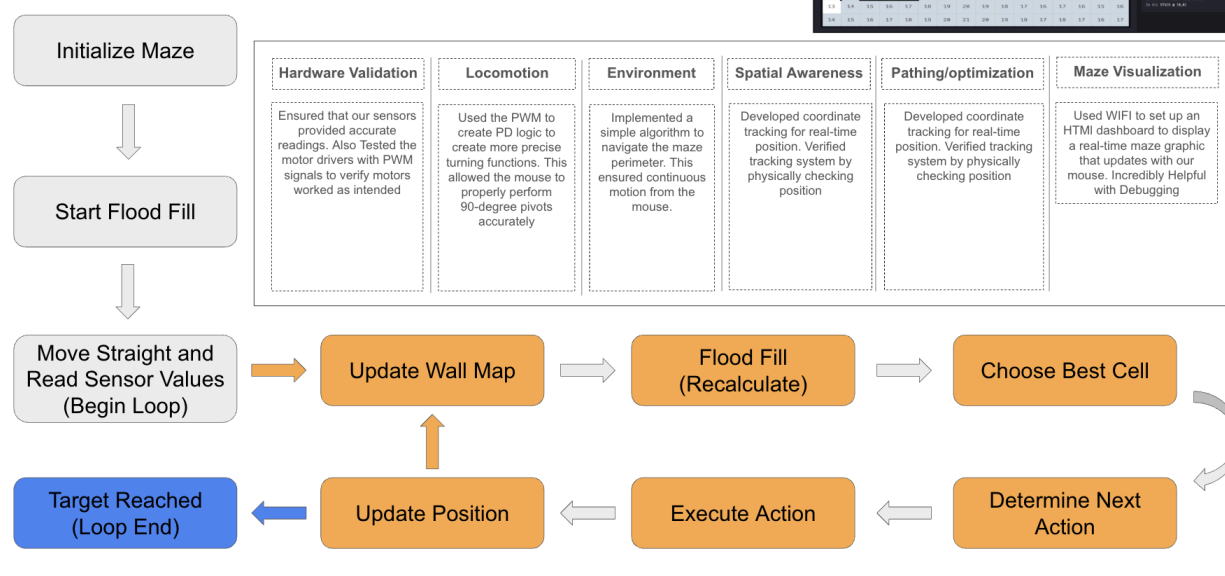


Figure 11. Software evolution and overall navigation flow.

3.4.2 Encoder-Based Pose Tracking and Motor Control

The robot mouse used encoder feedback to estimate how far it traveled and how much it turned. Each N20 motor included encoder outputs that were read by the ESP32-S3. As the wheels rotated, the encoders produced counts (7 pulses per revolution) that the software converted into distance traveled. This allowed the mouse to estimate its position in the maze without needing an external tracking system. For the most accurate tracking, quadrature encoder counts were used to get a result of approximately 2800 counts per revolution of the wheel.

The software used encoder counts to support both straight-line driving and turning. While driving straight, the program tracked the average distance traveled by the left and right wheels. Since each maze cell was 180 mm long, the encoder distance estimate was used to decide when the mouse had moved into the next cell. The software also used encoder feedback during turns to estimate when the robot had rotated by the target angle.

Motor control was handled through the DRV8847RTE motor driver. The ESP32-S3 sent PWM signals to the motor driver inputs to control the speed and direction of the two N20 motors. Because the mouse used differential driving, the robot could turn by driving the two motors at different speeds or in opposite directions. For example, a left turn required the left and right wheels to move differently so the robot could rotate in place or pivot within the maze cell.

The software included separate movement routines for driving straight, turning left, turning right, and making a U-turn. The turning angles were calibrated through testing, with the left and right turns using a target angle of approximately 85 degrees and the U-turn using a target angle of approximately 170 degrees. These values were intentionally chosen because they matched the actual behavior of the physical mouse better than ideal 90-degree and 180-degree values.

While driving straight, the robot used correction logic to reduce drift. Encoder feedback helped compare the motion of both wheels, and wall sensing could be used to make small adjustments when side wall information was available. While the wall PD controller helped keep the mouse centered between the two walls, an angle PD was used to keep the mouse oriented straight. This was done by converting the difference in encoder counts to an angular difference based on the distance between both wheels. The angular difference would then be used to compensate the PWMs of each wheel to apply the necessary values to straighten the mouse. This was important because even small differences between the two motors could cause the mouse to gradually angle toward a wall. Without correction, the mouse would lose alignment over multiple cells.

One challenge with motor control was that the robot's turns depended heavily on its starting position before the turn. If the mouse began turning too far away from or too close to the front wall, it could either turn too early or too late and end up misaligned for the next cell. This issue became more noticeable because the final sensing arrangement did not provide enough reliable wall alignment feedback. As a result, even when the motor routines worked individually, the integrated system sometimes struggled to turn consistently inside the maze.

Overall, encoder-based pose tracking and motor control gave the robot the ability to estimate distance, update its maze position, and perform basic movement commands. However, the final system showed that accurate movement required both encoder feedback and reliable wall sensing. Future improvements would include stronger closed-loop motor control, better use of side sensor feedback, and more consistent positioning before turns.

3.4.3 Maze Mapping and Flood-Fill Navigation

The higher-level navigation software was built around a wall map and a flood-fill maze-solving algorithm. The wall map stored information about which sides of each maze cell contained walls. As the mouse moved through the maze, the time-of-flight sensor readings were used to update the wall map for the current cell. This allowed the robot to gradually build an internal representation of the maze instead of needing to know the full maze layout ahead of time.

The robot started at cell (0,0) facing north. At the beginning of the run, the software initialized the maze map and marked the outside boundary of the maze as walls. This prevented the robot from planning a path outside the allowed maze area. After initialization, the flood-fill algorithm assigned distance values to the cells in the maze based on the current goal. During the exploration phase, the goal was the center of the maze.

After the robot entered a new cell, the software recorded the walls detected by the sensors. It then recalculated the flood-fill values using the updated wall map. The purpose of flood fill was to give each cell a value representing how far it was from the goal based on the currently known maze. The robot could then choose the neighboring cell with the lowest flood-fill value as its next target.

Once the best neighboring cell was chosen, the software converted that decision into a movement action. If the target cell was straight ahead, the robot continued forward. If the target cell was to the left or right, the robot prepared to turn in that direction. If the best available cell was behind the robot, the software commanded a U-turn. This allowed the robot to translate the maze-solving decision into actual motor commands.

The intended navigation sequence had three phases: exploring to the center, returning to the start, and performing a speed run back to the center. In the first phase, the flood-fill goal was set to the center cells of the maze. After reaching the center, the goal changed to the start cell so the robot could return home. After returning to the start, the goal changed back to the center for a final faster run using the wall information collected during exploration.

This approach was useful because it separated mapping from movement. The wall map stored what the robot knew about the maze, the flood-fill algorithm decided which cell was most useful to visit next, and the motor control routines handled the actual movement. Even though the final robot did not complete the maze, this software structure provided the main framework needed for autonomous maze solving.

3.4.4 Software Debugging and Visualization

Debugging was an important part of the software design because the robot's behavior depended on several systems working at the same time. When the mouse made a mistake, the cause was not always obvious from watching the robot move. A problem could come from the sensors, encoder readings, motor control, wall map, flood-fill values, or the action decision logic. Because of this, the software included debugging tools to make the internal state of the robot easier to understand during testing.

Serial output was used throughout development to print important values from the robot onto our OLED screen. These values included sensor distances, encoder counts, current cell position, and heading direction. This helped verify that the software was correctly reading the hardware and making reasonable navigation decisions. For example, if the mouse turned the wrong direction, the OLED screen could show whether the issue came from an incorrect wall reading, an incorrect position estimate, or a wrong flood-fill decision.

A maze visualization tool (Figure 12 below) was also developed to help debug the navigation algorithm. This visualization displayed the maze grid, known walls, flood-fill values, robot

position, heading, and action log. This was useful because it allowed the team to see how the robot interpreted the maze internally. Instead of only watching the physical mouse, the team could compare the robot's internal map with the real maze layout and identify where the robot's understanding became incorrect.

The visualization was especially helpful when testing the flood-fill algorithm. Since flood fill assigns values to maze cells based on the current goal, displaying those values made it easier to confirm that the robot was planning toward the correct target. It also helped verify whether newly detected walls were being added to the map correctly and whether the best next cell matched the expected path.

The map visualization was made possible via connecting the mouse to a local WiFi network and having it support its own html dashboard at a provided IP address. We had our mouse provide the IP address on our OLED screen so we could easily type it into our laptop browser. This webpage also allowed us to remove the OLED from the mouse for testing purposes as it would often hit walls during turns.

In addition to serial output and visualization, the PCB included LEDs and buttons for basic debugging. LEDs helped confirm that the board was powered and that the ESP32-S3 was running code. The button allowed the team to start or control test behavior without needing to immediately reprogram the board. Our blue blink LED was set to flash when our mouse entered the STUCK state, allowing us to know when things shut down. These simple hardware debugging tools were helpful during repeated testing because they gave quick feedback about whether the system was alive and responding.

Overall, debugging and visualization were necessary because the robot mouse was an integrated hardware and software system. The final behavior depended not only on whether individual components worked, but also on whether the robot interpreted its environment correctly. The debugging tools helped the team identify issues in sensor readings, movement control, and maze navigation logic during development.

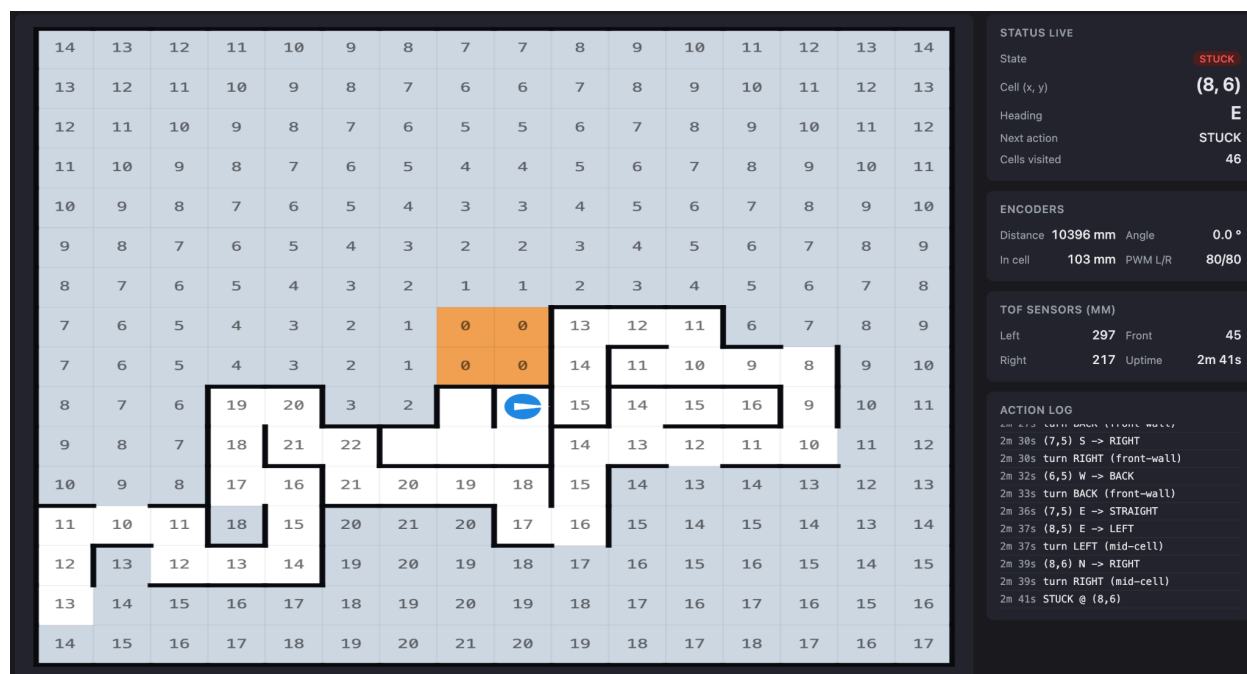


Figure 12. Mouse WiFi HTML debugging dashboard

4 System Testing

The design was tested and validated progressively throughout the semester. Instead of waiting until the entire robot was assembled, each subsystem was tested individually first and then integrated into the full mouse. This helped the team identify whether problems were coming from the motors, sensors, PCB, power system, or software logic.

Early testing focused on basic movement. The initial 500rpm motors proved to have too low of a gear ratio to move the mouse in a controlled manner. After working on getting our practice mouse to drive straight we quickly realized we were going to require a higher gear ratio and thus we purchased the 150rpm motors. The encoders were also tested to make sure wheel rotation could be measured and converted into distance traveled. Through testing we found a rotation of our wheel to be equivalent to about 2800 motor encoder counts.

The time-of-flight sensors were tested separately by placing them at known distances from maze walls and checking whether the readings were consistent. The initial VL53L0X sensor breakout boards we purchased produced somewhat reliable readings from approximately 40 mm to 400 mm. Upon continued use, their readings became inconsistent and we realized we valued accurate readings under 40 mm, and that led us to populate our own custom pcb sensor breakout boards with VL53L4CD sensors which were accurate down to 20 mm.

After basic motor and sensor testing, the team tested movement routines individually. Straight-line driving, left turns, right turns, and U-turns were tested before combining them with the full maze-solving logic. The turning angles were calibrated through trial and error, with about 85 degrees used for left and right turns and about 170 degrees used for U-turns. These values were chosen because they matched the actual behavior of the robot better than ideal 90-degree and 180-degree values.

The custom PCB was also validated during the semester. The team tested the USB-C programming connection, voltage rails, sensor headers, motor driver outputs, buttons, and LEDs. This testing revealed two major hardware issues. First, the ISEN12 and ISEN34 pins on the DRV8847RTE motor driver had been left floating, so they had to be manually soldered. Second, ESP32-S3 pin 45 was pulled high during startup, which interfered with uploading code and running the robot. Because of this, the right motor had to be unplugged when uploading code or starting the mouse.

Once the main subsystems were working, the robot was tested in the physical maze provided by the professor. The team moved the maze walls into different arrangements to create different routes and test cases. This allowed the robot to be tested against multiple maze layouts instead of only memorizing or repeating one path. These tests validated whether the mouse could sense walls, update its internal wall map, run flood fill, choose a movement, and physically execute that movement inside the maze. In addition, prior to writing the maze solving algorithm, to ensure that all our movement functions (i.e. `straight`, `left_turn`, `right_turn`, `U_turn`) worked effectively together, had a random left or right turn selector so the mouse would move on its own without always taking the same path.

Throughout testing, the OLED screen and WiFi visualization dashboard were used to debug the robot's behavior. The team monitored sensor readings, encoder counts, current cell position, heading, wall map updates, selected actions, and flood-fill decisions. This made it easier to understand whether a failure came from sensing, movement, position tracking, or the navigation algorithm.

Overall, the design was validated by moving from isolated subsystem testing to integrated maze testing. This process showed that the final robot could operate as a complete autonomous system and make it approximately halfway through the maze. It also revealed the main limitations of the design, especially inconsistent alignment before turns, limited wall-following feedback, the floating motor driver ISENSE pins, and the ESP32-S3 pin 45 startup issue.

5 Conclusion

The goal of this project was to design and build an autonomous robot mouse capable of navigating a maze, learning the wall layout, and attempting a faster final run. Although the final robot did not complete the maze on race day, the project was still successful in producing a working integrated prototype. The final system brought together the ESP32-S3 microcontroller, custom PCB, DRV8847RTE motor driver, N20 DC motors, motor encoders, VL53L4CD time-of-flight sensors, and flood-fill navigation software.

One of the main accomplishments of the project was getting the major hardware and software subsystems to work together. The robot was able to move under its own power, read sensor values, track encoder counts, update its internal maze position, and make navigation decisions. The mouse was also able to make it approximately halfway through the maze, showing that the system was functional even though it was not fully reliable.

Several lessons were learned throughout the design process. One major lesson was the importance of urgency. Earlier PCB ordering and assembly would have given the team more time for code integration and full-system testing. Since many issues only appeared after the board, motors, sensors, and software were running together, having the completed PCB earlier would have made debugging less rushed.

Another important lesson was the need for greater thoroughness during schematic and component review. Datasheets, pinouts, and support pins should be checked carefully by multiple team members before fabrication. The floating ISENSE pins and the pin 45 issue showed that small electrical details can have a major effect on the final robot. More careful review before ordering the PCB could have prevented some of these problems.

The team also learned the importance of careful board assembly. During PCB assembly, too much solder paste was used in some areas, which required rework and resoldering of surface-mount components. More practice with the assembly process and more careful solder paste application would have reduced rework and made the board bring-up process smoother.

If the project were continued, the most important improvements would be a revised PCB, improved sensor placement, and stronger closed-loop control. A new PCB should correctly connect the motor driver ISENSE pins and avoid using ESP32-S3 boot-sensitive pins for motor control. The sensing system should include reliable front, left, and right wall detection so the mouse can center itself more consistently while driving. The motor control software could also be improved by using stronger feedback from both encoders and wall sensors to correct drift before turns.

Overall, this project showed how difficult it is to build a reliable autonomous maze-solving robot. The individual components each had to work, but the larger challenge was making all of

them work together in real time. Even though the robot did not complete the maze, the final prototype demonstrated meaningful progress in embedded systems, PCB design, motor control, sensing, and autonomous navigation.