Fully Autonomous Micromouse Final Senior Project

By:
Nicholas Bannan
Jason Grashorn
Molly Hackett
Teresa Wandor

EE 40290

University of Notre Dame
Department of Electrical and Engineering
Senior Design Final Report

Table of Contents

Introduction

This year, we were tasked with creating a small robot to autonomously navigate and solve a 16x16 maze as quickly as possible. Typically referred to as a micromouse or mouse, this challenge is a standard IEEE competition. The mouse must be able to detect walls, keep track of its location, determine the fastest route to the center of the maze, and precisely navigate to the end of the maze.

The objective of this project was to design and build a reliable and precise mouse capable of such navigation. Our team focused on integrating motors, real time sensing, and autonomous decision making to create our mouse. To do this, we utilized an ESP32-S3-WROOM-1U microcontroller, analog infrared and time of flight wall sensors, and DC motors and encoders. We designed a printed circuit board for the motors, microcontroller, and analog sensors and another board for the time of flight sensors. We also 3D printed mechanical components in order to integrate these components into a complete mouse.

A key aspect of this project was implementing the maze solving algorithm onto our mouse. We used the flood fill maze solving algorithm to explore the maze and determine the fastest route to the goal cells of the maze. We also spent significant time making motion as precise as possible as well as ensuring consistent detection of walls in each cell of the maze.

Throughout this project, we gained experience with PCB and hardware design, software development, embedded system integration, and debugging complex electrical systems.

Key System Requirements

In order to enable a fully autonomous robot to solve a maze, we created a list of key system requirements that we needed to implement.

Sensing Accuracy:

The mouse needed to know where the walls were, how far away they were, and where it was on the map itself. To do this we need accurate and precise measurements of walls.

The side sensors were used to accomplish two different goals of driving straight and detecting side walls. They needed a high fast update rate in order to keep the mouse centered in the maze while driving forward by updating its movements based on the sensor readings. This fast update rate allowed the mouse to quickly react to deviations and maintain a straight course while driving quickly through the maze. These values also needed to be accurate at close distances to walls in order to accurately and reliably detect walls.

The front sensors's primary requirement was to detect walls in front of the mouse in the maze. Ideally, the front sensors would have the ability to sense long range, allowing the mouse to detect walls in cells beyond the cell it is in. Another requirement of the front sensors was to be able to reliably square the mouse to a front wall.

Adding diagonal sensors could help in some scenarios to detect walls coming up in future cells. However, we decided they were not a strict requirement for solving the maze given our side and front sensors already.

USB-C Power Supply and Battery Switching:

The board needed to be able to switch between using our computer's 5V USB-C output power supply and our battery system's power supply. This feature enables us to upload code from our computer while simultaneously providing power to the board. It also allows us to charge the batteries while we are coding, so later we can test the board's abilities without worrying about voltage sag.

Obviously, we need a way to provide power to the board during the race, so having a switch allows for an easy transition from computer to battery operated power.

Motor Requirements:

It was required that the motors were able to simultaneously function and drive forward and backwards. This allows for the mouse to drive forward, turn left, turn right and reverse which are the necessary movements that a mouse must do while solving a maze. The motors needed to be able to drive fast enough in order to give competitive time while also giving us enough precision during movements in order to reliably control the mouse.

In order to implement the motors, a motor driver was required. This needed to be a dual H-Bridge motor driver in order to be able to power both motors simultaneously and control motor speed and direction.

It was also a requirement to know how far the motors have spun over a given time period in order to ensure consistent movement throughout the maze. To do this, the mouse required motor encoders. Motor encoder information is quantitative data used by the microcontroller to ensure consistent movement of the mouse. This information along with real world data is used to calculate how far the mouse has moved. This bridges the gap between real movement and electrical motor power. Using motor encoder information for turning also provides us which direction and how many degrees the mouse has turned. This is extremely important to giving the mouse an idea of where it physically is.

Compactability:

It was also critical to create a mouse as small as possible. A smaller mouse has an easier time turning, driving straight, and overall moving in the maze.

Detailed project description

Here we outline our hardware and software decisions.

Hardware:

Sensors:

We initially were going to use three Time of Flight (ToF) sensors: one on the front and one on either side. We planned to use only ToF because they have a very long range, return a millimeter value, and are highly accurate and highly precise. However, we learned that the update rate was insufficient to center in a corridor reliably so we switched to a hybrid model of analog infrared sensors on the sides and ToF sensors on the front. We also added a second ToF sensor to the front to be able to measure if the mouse was askew and achieve front centering capability.

We decided to use this hybrid model of two ToF and two analog IR photodiode and phototransistor pairs because each type of sensor offers specific advantages. The analog IR pair is advantageous for side sensing because it can update in the kilohertz range, enabling highly accurate side wall centering while moving. We use an emitter delay of 30 µs, and we sample analogRead twice which takes about 10 µs for each reading. Putting this together, it flows like this:

- 30 µs (on delay)
- 2 x 30 µs (ADC reads)
- 30 µs (off delay)
- 2 × 30 µs (ADC reads)
- Total per sensor: 180µs

This corresponds to 360µs for both sensors or an update rate around 2.78 kHz. ToF sensors typically have a max update frequency around 100Hz and we used two VL53L4CD ToF chips for the front. We selected this sensor for the front because we envisioned the front topography changing less frequently than the sides while providing precise and long range measurements. The analog IR pair can only sense reliably from 5-200 mm while this chip can measure reliably from 0-1300mm.

Motor Driver:

We selected the DRV8411A motor driver because it is a robust dual H-bridge motor driver with internal current regulation. It allowed us to reliably control the speed and direction of each motor separately.

Motors:

We decided to choose Pololu 5161 motors because they were the recommended size, compatible with our motor driver, and we wanted to reduce the size of the mouse as much as

possible. The Pololu 5161 has a side connector which reduces the required PCB size. 5V motor operating voltage was chosen because it is a compatible voltage on our motor driver (1.65 to 11V), USB-C supplies 5V, and our two 3.7V LiPo batteries together supply 5V after passing through a 5V LDO (LDL1117S50R). Thus, we could test the motors with USB and battery power. We chose a 50:1 gear ratio because we had an assignment during the first semester that allowed us to get a grasp of the PWM values required to turn on various motors and the speed/torque that produced. We tested gear ratios from 25:1 to 200:1 in that assignment and determined that 50:1 gave us high speed resolution (a wide range of PWM values).

Software Structure:

Originally, our software structure had all functionality in one file. This caused difficulty with debugging and working as a team. If we changed one aspect, it would impact the whole program and make it difficult to understand the work one teammate did and how to continue improving the mouse movements. Thus, we switched to modular code. This way, we could write the program into smaller, independent pieces that handled each task. This significantly increased the maintainability and readability of our program. To break into modular code, we sectioned out our functionality into hardware configuration, basic movements, OLED display, and maze search algorithm.

Algorithm:

The most commonly used algorithm for the micromouse competition across the world is Flood Fill. It essentially treats the ending of the maze like a drain, and the start as the highest point. Initially, the mouse assigns a potential value, or distance, to every cell, starting with zero at the center goal and increasing as it moves toward the perimeter. As the mouse moves and detects physical walls that were not in its initial memory, it refloods the maze by updating the distance values of all affected cells to ensure the gradient always flows toward the goal. By consistently moving to the neighboring cell with the lowest numerical value, the mouse effectively flows through the most efficient path, eventually perfecting its route as more wall data is gathered.

Overall Software Design:

Our system implements an autonomous maze-solving architecture designed for Flood Fill. We utilize a three-phase operational strategy: orientation, exploration, and optimization, supported by a real-time multitasking backbone to ensure precise navigation and accurate environmental awareness.

High Level Software Design:

We have engineered the software to handle the uncertainty of the starting position through a robust orientation phase. Upon boot, the system executes a start nudge and begins a

local search to resolve its global coordinates and heading. Practically, this looks like the mouse traveling a few cells, making a few turns, and heading back to the starting point. Once the orientation is locked, we transition into the exploration phase. In this mode, we utilize a flood fill algorithm to update a virtual map of the maze based on live sensor data. We prioritize the discovery of the center goal cells while recording a visited subgraph to ensure a safe return to the starting point.

To maintain system stability, we offload intensive hardware polling to dedicated background tasks. By implementing a time-of-flight sampler, we ensure that distance measurements are processed asynchronously, providing the main logic with fresh snapshots via a semaphore controlled interface. We manage the shared I2C bus, which hosts both the VL53L4CD distance sensors and the OLED display, using a mutual exclusion lock to prevent data collisions during high speed transit.

Upon reaching the goal, we perform a final map reconciliation, assuming standard goal perimeter walls to refine our flood fill values. We then execute a trace back routine to calculate the mathematically optimal path from the origin to the goal. Our final operational stage is the fast run, where we suppress the 1,000ms delays at each cell used during exploration, allowing the mouse to traverse the discovered optimal route at peak velocity using strictly downhill flood values.

Low Level Software Design:

We separated our tasks into nine different .cpp files, each responsible for a distinct subsystem of the robot. This modular design improved readability, debugging, and maintainability by isolating hardware control, navigation logic, and decision making into clear components.

main.cpp:

The main control loop implements a two phase micromouse navigation system built around real time sensing, incremental mapping, and flood fill path planning. On boot, the system initializes hardware, ToF sensing, display, and an orientation phase that resolves the robot's initial position and heading using local wall observations. During this phase, the robot explores in a constrained loop until a consistent global frame is established, after which it transitions into normal operation and enables maze persistence.

In the main exploration loop, the robot first centers itself against the front wall using ToF distance control, then performs a synchronized sensor read combining infrared side sensors and a freshly updated ToF snapshot to detect walls in the current cell. These observations are fused into a global maze representation, which tracks both known edges and confirmed walls. The maze is continuously updated and immediately used by a flood fill algorithm rooted at the goal region to compute distance-to-goal values for every cell.

Each iteration selects the next movement direction using the flood gradient while avoiding the previous cell when possible, then executes a corresponding motion primitive

(straight, left turn, right turn, or U-turn with forward motion). After each move, the system updates its internal pose estimate and logs the visited cell for later path reconstruction. The OLED display is updated every cycle with live position, heading, sensor readings, wall state, and chosen action, while Serial output provides detailed step level diagnostics.

When the goal region is reached, the system switches into a return and optimization phase. It first finalizes the wall map by assuming the known goal geometry, then executes a controlled return-to-start routine while continuing to refine the maze. Once back at the origin, a best path trace is generated using the fully explored flood fill map and visited cell constraints. This optimal route is visualized and saved, after which the robot executes both a slow demonstrative run and a final fast run along the computed optimal path.

display.cpp:

The display module drives an SSD1306 OLED over I2C and is used mainly for debugging and runtime telemetry. It shows the robot's current cell position, heading, motor sensor values, and front/side sensor readings. It also draws a simple local wall sketch of the current cell using the most recent sensor based wall detections. While doing this, it updates a global maze knowledge buffer (gKnown and gWalls) based on what the robot observes in each cell.

It also handles persistence of exploration data. It periodically stores the accumulated maze wall map into non-volatile storage (NVS), and it can retrieve and print saved mazes on boot. It also provides diagnostic tools like ASCII maze dumps, path overlays, and saved run replay (best path, return path, etc.) via Serial.

drive_best_path.cpp:

This module implements the robot's high speed navigation phase after exploration, where a deterministic shortest path is computed and executed using the previously built maze map. Path planning is based on a flood fill distance field generated from either the goal or start cell, but restricted to the subset of cells stored in the visited mask. This ensures the robot only plans through areas it has physically confirmed during earlier phases.

The next move is selected using a greedy control policy that always chooses a neighboring cell that is closer to the target. If multiple valid moves exist, a deterministic tie-breaking rule is applied, prioritizing straight motion first, then right turns, left turns, and finally 180^{o} reversals. Movement is only considered valid if there is no wall between cells according to the stored maze map (mazeHasWall).

During execution, the robot performs periodic alignment using centerFrontDist() to correct lateral drift before each decision step. The selected heading from mazeNextMoveHeadingVisited() is converted into a fundamental movement using headingToRelativeMove(), then executed via executeMoveLocal(), which maps movement commands to motor control sequences. After each action, the robot updates its internal position and heading state and increments a step counter tracking execution progress.

A counter is used to prevent infinite loops in the event of an invalid or disconnected visited graph, ensuring safe termination if no valid path exists.

hardware.cpp:

This module implements the robot's motor control, encoder feedback, and IR based wall following control system. It is responsible for converting sensor inputs into real time motion corrections and ensuring stable navigation under both dual wall and single wall conditions.

Motor control is implemented using PWM motor driver functions (setMotorL, setMotorR), which translate signed speed commands into H-bridge signals using ESP32 LEDC channels. A hard braking function (brakeForwardDriveHard) briefly pulses motors in the reverse direction to quickly stop forward momentum after movement transitions.

Wheel position is tracked using quadrature encoder interrupts, where ISR callbacks update encLeft and encRight. These encoders provide a blind gauge of distance and are the basis of movement routines.

The IR subsystem uses modulated emitter and receiver sampling (readIR) to reduce ambient noise. Each sensor reading is computed as the difference between IR on and IR off measurements, providing a normalized proximity estimate for wall detection.

The core of this module is the wall aware steering controller (moveBalancedFast), which implements a closed-loop control system. Left and right IR values are filtered and compared against dynamically learned baselines (wallBaseL, wallBaseR). Wall state estimation is computed by classifying each side as present or lost using thresholds and sample debouncing. It distinguishes dual wall mode (centering between walls) and single wall mode (one wall follow). This function also uses adaptive control mode switching by utilizing gains, deadbands, and target offsets that change depending on whether one or both walls are present. It computes proportional error as the difference between the normalized left and right wall distances. The system applies proportional and derivative terms (KP, KD) with filtering (FILTER_ALPHA) to smooth noise and improve stability. Final correction is applied by adjusting left/right motor speeds around a constant forward velocity (BASE_FWD).

To maintain stability during sudden wall loss events, the system includes debounced wall loss detection (different thresholds for single vs dual wall scenarios), immediate counter steering pulses when a wall is lost to prevent drifting into open space, baseline learning only when both walls are stable, and single wall switching logic, which allows transition between left and right wall following.

The module also provides initialization routines (setupHardware) which configure PWM motor channels, IR emitter pins, and encoder interrupts, ensuring all hardware components are ready before motion begins.

maze.cpp:

This module implements the robot's maze representation and path planning system. It stores the maze as a grid based map (4-bit directional encoding per cell) and a flood fill distance map used for shortest path navigation.

Wall data is updated using sensor inputs through mazeUpdateFromSensors(), with consistency enforced between adjacent cells. The maze is initialized with boundary walls and a fixed 2x2 goal region, with mazeAssumeGoalPerimeterWalls() ensuring correct goal definition for navigation termination.

Shortest path computation is performed using a flood fill algorithm (runFloodBFS), which assigns each cell a distance value relative to the goal or a chosen start cell. Navigation decisions are based on this flood map, where nextMoveHeading() selects the direction that minimizes flood value, using a fixed priority order (straight, right, left, back) and handling ties and plateau cases to avoid unstable movement.

For replay phases, the module supports visited-only planning, where only previously explored cells are considered valid. Functions like mazeFloodFromGoalVisited() and mazeNextMoveHeadingVisited() restrict flood fill and movement decisions to this known subgraph, ensuring safe execution without exploring unknown areas.

Additional utilities generate full paths (traceBestRoute and variants) by simulating flood fill based decisions, producing either movement commands or coordinate sequences for execution and debugging.

movement.cpp:

This module implements the robot's fundamental movements and cell based navigation control. It combines encoder odometry, IR wall steering, and ToF stopping logic.

Forward motion is primarily controlled using wheel encoder feedback (encLeft, encRight), where movement functions run until a target tick distance is reached (encAbsAvgDelta). During forward motion, wall correction is handled by moveBalancedFast(), which continuously adjusts motor PWM to maintain alignment.

Several forward behaviors are defined depending on navigation context to drive under several conditions. There are functions to move one full maze cell using encoder distance, perform a short initial centering movement at startup, drive blindly using fixed PWM when no wall reference exists, use ToF distance sensors to stop when a front wall is detected, with an encoder based safety fallback, and drive an adjustable distance for partial cell correction.

Turning is implemented as zero point pivots using encoder targets, with separate calibration constants for left, right, and 180° rotations. All turns drive both motors in opposite directions until encoder thresholds are reached. A two phase speed profile is used: fast rotation followed by a slow zone for improved angular accuracy. A short stabilization delay (TURN_SETTLE_MS) is applied after each turn. After every turn, the system resamples sensors.

High level movement decisions are handled by helper functions. executeStraightOneCell() selects between blind drive, ToF stop, or encoder based motion depending on wall configuration.

executeTurn*AndForward() performs a turn, then chooses the forward strategy using local wall detection.

orient.cpp:

At startup it resolves the robot's true position and heading by testing 8 possible corner-and-orientation hypotheses. Each hypothesis maps the robot's local coordinate system into the global maze frame.

As the robot senses front/left/right walls, those observations are recorded in a local map and projected into each hypothesis. Any hypothesis that conflicts with boundaries or sensed open edges is eliminated.

The system continuously prunes candidates until only one remains, or forces a decision based on step/drift limits. That remaining hypothesis is taken as the correct global pose.

After locking, it rebuilds the global maze by replaying all observed walls through the chosen mapping, then converts the robot's final local position and heading into world coordinates.

return_to_start.cpp:

Implements the robot's "go back to (0,0)" phase using flood fill path planning combined with an exploration bias. It has a visited grid to prefer unvisited cells during the return trip. Each step uses a weighted move picker that prioritizes: non-uphill moves toward lower flood values (distance to origin), unvisited cells, and avoidance of immediate backtracking, with a straight/right/left/around tie break.

At each step it recenters on the path, senses IR + ToF walls, updates the maze map, recomputes flood fill toward (0,0), selects the next move using the biased scoring system, executes the move and updates its position. The loop runs until (0,0) is reached or a safety step limit triggers.

tof.cpp:

This module runs dual VL53L4CD ToF sensors in a FreeRTOS task, continuously sampling both front distance sensors and publishing only valid readings. Invalid or missing frames are ignored so the system always holds the last good measurement.

It gets fresh data (getLatestToF, getFreshToF) with age tracking and utilizes synchronization helpers like waitForFreshToF() and counters for freshness validation.

Its core logic includes wall detection and front centering. Wall detection is accomplished using wallInNextCell() and wallInFrontFromToF() which uses thresholds to decide if a wall is directly ahead or in the next cell, avoiding stale reads. Centering control uses centerFrontDist() to actively align and stabilize the robot in front of a wall using yaw correction (left/right balance) and distance correction (maintain nominal gap), with settling logic.

System Testing

System testing was performed incrementally throughout the project rather than only at the end. Each subsystem was validated independently before integration into the complete mouse. This allowed electrical, mechanical, and software issues to be isolated and corrected before full maze solving tests were attempted.

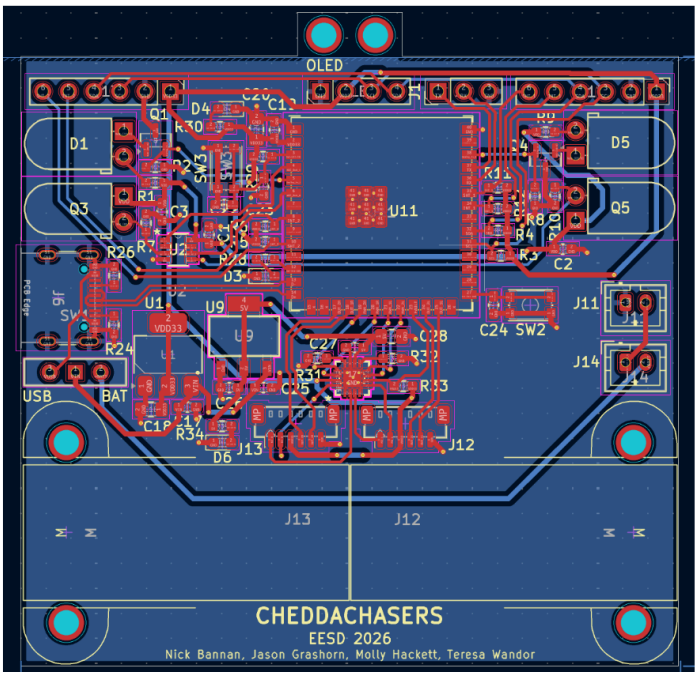
PCB:

During the first semester, Nick and Molly worked together to design the PCB schematic that we would build the board around. Molly proceeded to design the PCB in KiCad with periodic reviews by members of the team and Professor Schafer. We had meetings with Professor Schafter every other week and the members of the team all reviewed Molly's work and made suggestions for changes. Some changes that were requested were functionality, such as for two front sensors at opposite sides of the front to achieve front centering and a protrusion on the front to secure a device to slide along the ground. The PCB design choices were largely informed by the parts we were familiar with on the practice mouse, data sheets, and team functionality requests.

When the main PCB board arrived, along with 4 copies, we used the Engineering Innovation Hub's (EIH) solder paste and pick & place machine to put the components together on the board. Then, we processed it through the reflow oven, allowing the solder to solidify onto the board. After testing the board using our 5V USB-C connection via our laptops, the board did not fully function. To understand why, we used a digital multimeter to test for shorts, grounds, or loose connections. We discovered that the 3.3V line was shorted to the GND line. This caused almost none of our components to function properly. After some troubleshooting, it was determined that the H-Bridge motor driver shorted these pads, which affected the microcontroller itself. Although the microcontroller had great solder connections, the ground in the 3.3V line ruined the controller's ability to function correctly.

This conclusion led us to order a new set of components and use one of our copy boards to start over again. This time, we made sure to be careful with all the pins' connections, especially regarding the H-Bridge motor driver and the ESP32 microcontroller. After going through the same fabrication process as before, we tested it with proper power and all of our components got power. We did have a few small expectations that we outline later in this section, but nothing major.

Figure 1: KiCad Board Design PCB



ToF:

Before we implemented ToF sensing on our final board, we validated it on our practice mouse. We bought 4 x VL53L0X break out boards and used them for front sensing. We used these breakout boards to navigate different benchmarks throughout the semester such as straight driving, left and right turning, 180 degree movements, and solving an s-shaped maze.

Once we got our ToF board delivered, we began to fabricate the ToF sensors and integrated them into our final board. To test the ToF sensors, we wrote a test script to initialize the I2C bus and then change the I2C address of one sensor so they could both communicate. We then output ToF values to the serial monitor to evaluate the range, precision, and accuracy of the sensor. We built the board with the intention of using the fastest sensing mode (100Hz), but in the software, we used 30Hz. The specified precision was ± 7 mm from 1-100mm, ± 8 mm from 101-200mm and ± 3% if above 200mm indoors when sensing on a white target. In testing, the precision was about ± 3mm from a distance of 100mm. The accuracy was within 3 mm of the measured value reliably. The upper range of the VL53L4CD was approximately 1350 mm before reading incorrect values. When centering on a front wall by pulsing the motors in opposite directions until the two sensors agreed with a target distance of 65mm, we used an acceptance

band of ± 3 mm to get front centering to stop. This suggests that ± 7mm at 1-100mm is a well founded statement. In testing, the lower limit before reading 0mm was about 10mm.

Figure 2: ToF KiCad Board PCB

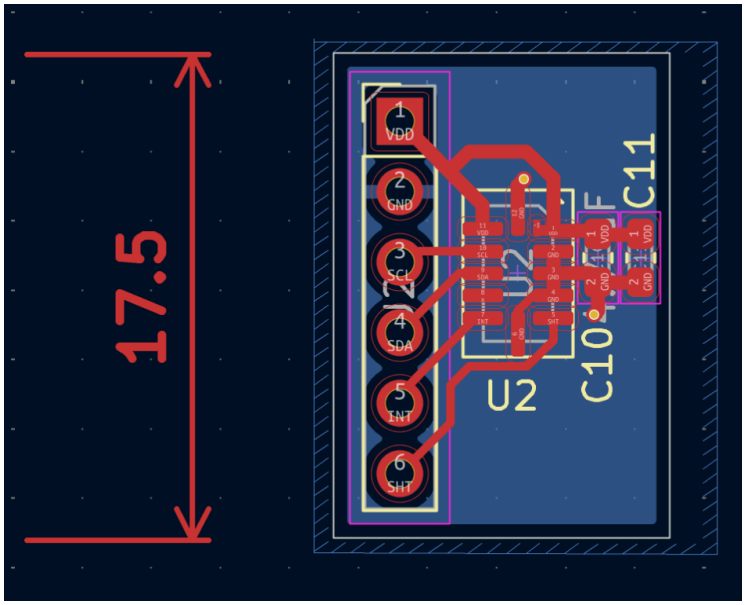
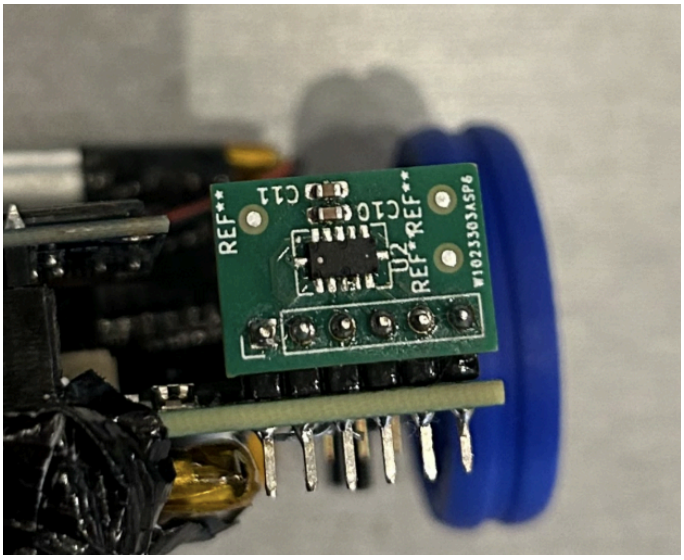


Figure 3: Assembled ToF sensor on Mouse



Analog IR:

During the semester, we switched from ToF to analog sensors on the sides. We did this because ToF sensors proved too slow to accurately center on side walls. We implemented these sensors and they substantially improved centering performance in benchmarks like the s-shaped maze.

In testing the analog IR sensors on our final board, we first discovered that the footprint of the NMOS we designed did not match the footprint of the EIH stock part. The NMOS biased a current to the IR diode and was required to get the sensors working. We resolved this by rotating the NMOS approximately 135°, allowing pad alignment. Our first attempt resulted in the removal of one of the side's copper pads. To fix this, we used jumpers to connect the NMOS as intended. Once we got the board in working order, we installed the IR diode and phototransistor as specified in the spec sheets and our PCB design. We wrote test scripts to initialize the sensors and calibrate the sensors. We found through testing that all of the left sensor's values returned about 2.4 times lower than the right. In open space, the left sensor would return values of as low as 65 while the right sensor would return values as low as 160. Perfectly centered in a corridor, the left sensor would read about 270 and the right sensor would read 650. Through more testing, we decided that the threshold to decide if a left wall is present should be 150 and the threshold for a right wall present should be 400.

Motor Driver:

When we soldered this onto our first board attempt, it had several shorts. We energized the board before fully testing for shorts and ended up damaging the MCU. Because of this error, we needed to place all new components on our second board attempt.

On our second board, we were much more liberal with the application of flux and much more careful with the application of solder paste. We carefully placed all of the fine-pitch parts and even removed some solder paste from the H-bridge pads using a tooth pick to ensure there wasn't too much solder before reflow. After reflow, we rigorously tested for shorts and opens before energizing. We found that three of the four motor driver channels were working. We tried reflow on motor driver pins 5 and 7 which correspond to the channel that was not working. We ordered 3 new motor drivers through DigiKey because we thought the motor driver had an internal transistor stuck in a floating state or the path was open. We were leaning towards thinking the internal circuitry was broken because we reflowed and added solder to the problem pins several times and the issue persisted. We communicated with Professor Schafter to source a new motor driver and he provided us with a new one, but before we attempted to put it on, he assisted us in getting a good motor driver connection. He reflowed using gel flux and the connection was now solid, restoring function to all motor driver channels.

During system operation, testing, and handling for hours on end, the motor driver, 3.3V regulator, and the 5V regulator did not get noticeably hot.

Motors:

We used Pololu 5161 motors for every test using the practice mouse and so we had a good grasp of their functionality once we got the final board.

Once we received our final board motors from Pololu, we wrote a test script to figure out the starting PWM going forwards and backwards. We determined that the starting PWM for forwards was 110 on the right and 150 on the left, while for backwards the right needed 15 and the left needed 20. With this in mind, we flipped the motors to access more speed resolution. With the motors and encoders working, we worked through each turn (left, right, u-turn) and straight path navigation. After testing, we determined that we needed to set the right motor to negative and the left positive to go forward. After learning this, we applied this logic throughout the code. This makes sense because if both motors were facing the same direction, a positive value would make both the same direction. If you rotate one 180 degrees, that motor's sign must be flipped to be "forward". We determined our "working PWM" range which is 150 to 255 PWM (negative for right) and found that to go straight in a situation with no walls to sense off of, we would use PWM values of 171 (left) and 169 (right).

We introduced "pulsing" our motors to correct for alignment issues that arose after completing motions. For instance, if we used a straight -180 PWM active brake after each cell movement, the mouse would drift left because the left motor physically stops moving before the right due to a higher coefficient of static friction. We overcame this by pulsing the right and left motors for different amounts upon stopping.

Software:

Testing was performed in order of ascending complexity: first fundamental motion testing, then subsystem integration testing, and finally full maze validation.

We began testing our software by validating fundamental movements in isolation. This included calibrating encoder tick counts for fixed distance forward motion, 90^{o} and 180^{o} turns, and consistent cell based movement. Each movement was tuned to ensure repeatability across multiple runs and to reduce drift caused by mechanical and frictional variation between motors. Once we began utilizing active braking, we had to retune every one of these movements, but the resulting movements became more precise as the motors stopped drifting. This reduced leftward drift caused by asymmetric static friction between the motors

Once these base movements were reliable, we integrated them into higher level navigation behaviors. This included front centering using ToF feedback, where the robot adjusted its position until both front sensors converged within a tight tolerance band, and wall centered driving using the moveBalancedFast function, which continuously corrected lateral error using IR sensor feedback.

After validating these subsystems independently, we tested full navigation behaviors inside structured mazes. This included straight corridor traversal, S-shaped maze benchmarks, wall loss conditions, no wall conditions, wall recovery conditions, and full flood fill exploration runs. At this stage, we focused on verifying that movements, sensor updates, maze updates, and movement decisions remained stable under continuous operation. We had a lot of trouble getting

fundamental movements correct as motor characteristics and IR sensor sensitivity varied over time, likely due to ambient conditions and mechanical wear. We made final tuning edits to our functions, but the movement was not perfect for race day. Final validation was performed by running complete solve cycles, where the robot successfully transitioned from exploration to goal detection and return path execution using the flood fill algorithm.

Conclusion

Our Senior Design project successfully culminated in the development of a fully autonomous Micromouse capable of navigating and solving a complex 16x16 IEEE standard maze. By integrating an ESP32-S3-WROOM-1U microcontroller with a hybrid sensing array of Time-of-Flight and analog infrared sensors, we engineered a platform that balances long range environmental awareness with the high speed feedback necessary for precise corridor centering. The transition from a single file software architecture to a modular, RTOS supported framework proved essential, allowing us to offload intensive hardware polling to background tasks and maintain system stability during high velocity maneuvers.

The implementation of the Flood Fill algorithm, complemented by a robust three phase operational strategy, orientation, exploration, and optimization, allowed the mouse to move beyond simple reactive navigation to intelligent path planning. Through rigorous system testing, we overcame significant hardware hurdles, including PCB short circuits and motor driver soldering complexities. We also addressed mechanical variances, such as asymmetric motor friction, by implementing sophisticated software "pulsing" and active braking techniques. These refinements ensured that the mouse could not only find the goal but also calculate and execute a mathematically optimal "fast run" by suppressing exploration delays.

Ultimately, this project served as a comprehensive exercise in multidisciplinary engineering. We successfully bridged the gap between theoretical control logic and physical execution, gaining invaluable expertise in PCB fabrication, embedded systems integration, and real time debugging. The final iterations of our hardware and software stand as a reliable, precise, and compact solution to the Micromouse challenge, meeting all key system requirements and demonstrating the efficacy of our hybrid sensing and modular design approach.

Figure 4: Back of Completed Mouse

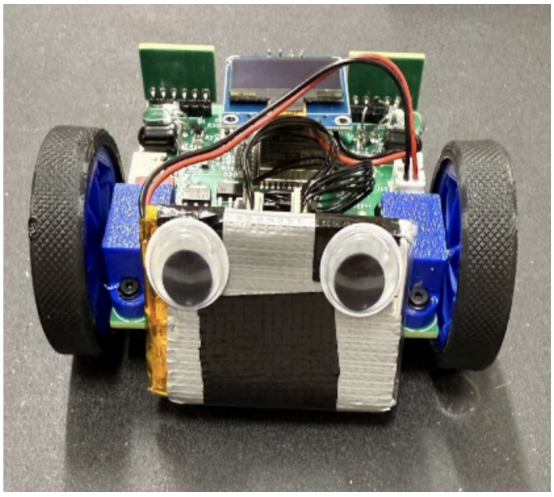
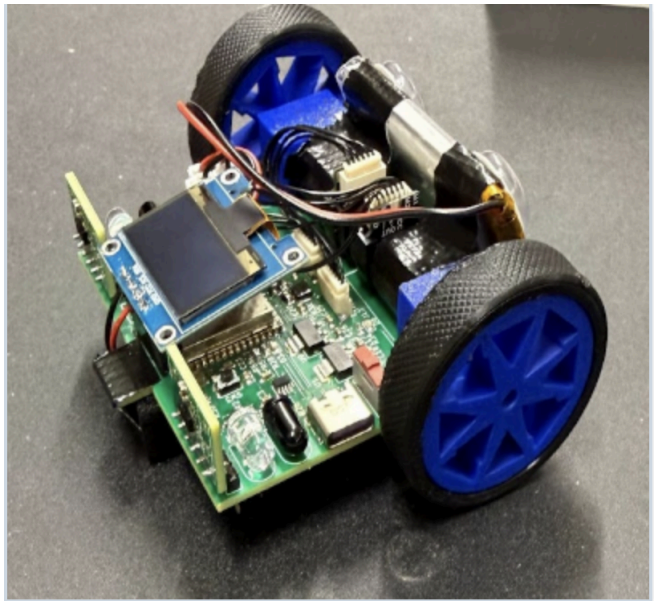


Figure 5: Isometric View of Completed Mouse



Race Results:

Our mouse successfully oriented itself and got through half of the maze solving. However, it got stuck at the wall, most likely due to the imbalance of PWM on motors accumulating overtime. This can be adjusted through the programming, but we did not have enough time to fine tune it for larger scale mazes.

Figure 6: Micromouse Solving Maze