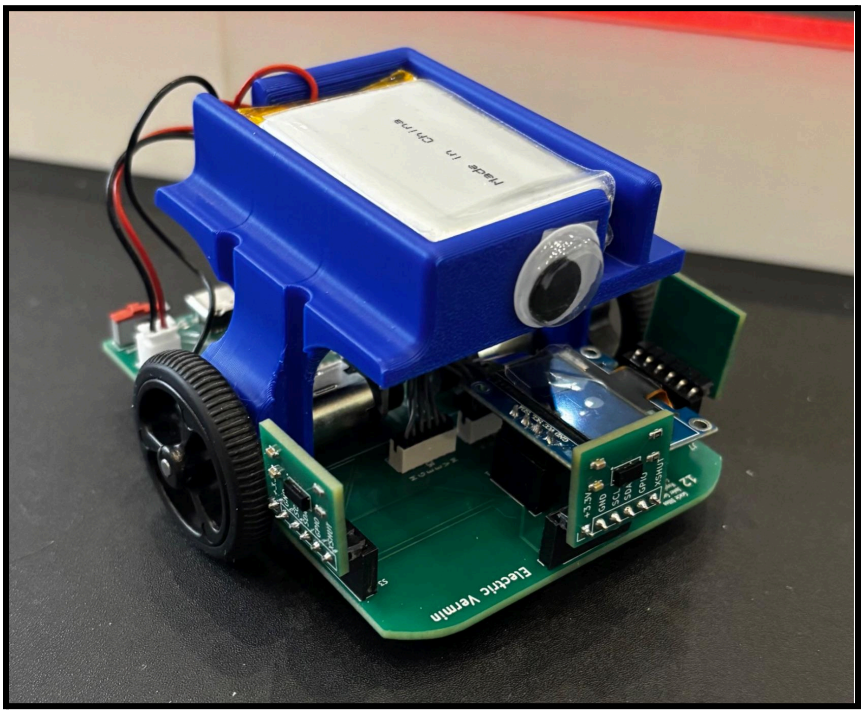

Electric Vermin
Autonomous Maze-Solving Robot
FINAL REPORT

Gracie Dillon, Javier Garcia-Renteria, Maggie Griffiths, Gabe Hinrichs
University of Notre Dame
Electrical Engineering Senior Design
6 May 2026

Table of Contents

1. Introduction

For this year's Electrical Engineering Senior Design course, students were placed in groups of four (or five) and were tasked with developing a micromouse robot. This micromouse robot had to be able to traverse a 16 by 16 cell maze and navigate itself to the end (the center of the maze) within ten minutes. Following guidelines provided, particularly those included in the IEEE Micromouse Competition Rules, each group in Senior Design raced their micromouse against other micromice on Race Day (May 1, 2026). The goal on Race Day was to have your micromouse solve the maze in the shortest amount of time. The micromouse with the fastest time was crowned the winner.

In developing our micromouse, our team began by determining which hardware components we needed for our micromouse to move and detect walls. Once those hardware decisions were made after thorough research and testing, we proceeded by developing software for our micromouse, starting with simple code focused on solely on moving the mouse forward in a straight line and ending with the fully fledged code we used on Race Day, which included maze traversal capabilities and a maze-solving algorithm. While the maze used on Race Day was not revealed until the morning of May 1st, we tested our hardware and software by utilizing practice maze segments and the full-size 16-by-16 cell practice maze.

We succeeded in developing our micromouse by meeting regularly as a team and dividing up work effectively.

2. Key System Requirements

2.1 Competition Requirements

The primary competition requirement was for the micromouse to autonomously navigate a maze, identify a path to the end goal of the maze, and complete a faster final run time using the information gathered during maze exploration. Meeting these requirements meant the robot had to reliably detect walls, track its position within the maze, move accurately from cell to cell, execute consistent turns, store the maze data, and compute the shortest known path to the goal. These objectives drove the lower-level design decisions made throughout the entirety of the project, which spanned sensor selection, motion control, motor and encoder choices, and the maze-solving algorithm.

2.2 Supporting System Requirements

To meet the competition requirements described above, the robot needed to satisfy several system requirements. These requirements defined the hardware and software performance necessary for the mouse to explore, map, solve, and ultimately speedrun the maze at the end. The

most critical requirements involved wall sensing, motion control, encoder feedback, motor selection, PCB compatibility, and maze-solving software.

Table 1. System Requirements and Design Approach

System Requirement	Design Approach
Detect front, left, and right walls	Wall detection was necessary to avoid collisions, identify open paths, and map out the maze. Time-of-flight sensors were mounted on the front, left, and right sides of the robot.
Support real-time corridor centering	The mouse needed to stay centered while moving, not only when stationary. Side time-of-flight readings were used as feedback for continuous wall-centering correction.
Calibrate sensor readings to the robot	The left and right sensors did not report identical distances when the mouse was physically centered, so target values were manually adjusted in the software.
Move fast enough for the competition	The robot needed to complete both exploration and the speedrun within the competition time limit of 10 minutes. Switched to a lower gear ratio to aid this.
Measure cell-to-cell movement	The software needed to determine when the robot had moved from one maze cell to the next. Encoder counts were used to estimate distance traveled and update the robots position in the maze.
Execute repeatable turns	Reliable left turns, right turns, and U-turns were essential for maze navigation. Our team used encoder-count based turns with separately turned values for each turn type, including the speedrun turns.
Correct left/right motor mismatch	The motors did not respond identically to the same PWM, causing the robot to initially drift. PID-style correction using encoder feedback and side-wall readings was used to correct that.
Support selected hardware on the PCB	The custom board needed to support the specific hardware chosen.
Separate major robot behaviors in the software	Driving, turning, idle, returning, and speedrunning stages each needed their own logic. A finite state machine was implemented with certain states the robot would be in during maze navigation.
Store maze information	The robot needed to retain wall data, visited cells, and confirmed open paths across the full exploration run. This information was stored in arrays representing the maze grid
Compute a path to the end	The mouse needed a method for choosing directions and solving the maze. Flood-fill logic was used to assign distance values to each cell and guide navigation to the end goal of the maze.
Execute a speedrun of the maze	The speedrun needed to use previously collected maze data rather than continuing to explore. The final implementation built a speedrun path from confirmed open edges using pessimistic flood-fill logic.

The supporting requirements shaped the final design. The need for navigating the maze fast enough drove to getting a lower gear ratio. The need for repeatable movement led to encoder-based distance tracking and turning controls. Reliable corridor navigation required ToF-based wall sensing combined with PID-style correction. Finally, the need for an autonomous speedrun tied together the maze memory, flood-fill path planning, and a dedicated speedrun mode. Collectively, these requirements connected every hardware and software decision back to the overall goal of the competition.

3. Detailed Project Description

3.1. Hardware Design

Each category in the hardware section will include a description of the component, the reason for selecting those components, and how they fit into the overall design of the final board. Not included will be resistors, capacitors, or LEDs, as they were mainly chosen from the datasheets of each component, and because they were stocked by the EIH.

Microcontroller

ESP32-S3-WROOM Microcontroller: This family of microcontrollers was chosen because it is the same family that was used on the practice board given to us during the first half of the semester. Our team used the N4 model instead of the N16R8 model used on the practice mouse primarily because the N16R8 model was sold out, and we never used any of the additional PSRAM. The N4 had enough GPIO for our sensors, encoders, and motor driver inputs, enough flash to hold our firmware, and let us port code from the practice board with only minor changes.

Batteries and Voltage Regulators

3.7 V LiPo Batteries: Our team used two 3.7 V LiPo batteries connected in series to supply 7.4 V to the board. With this configuration, we were able to have a long battery life to test our mouse for long periods without needing a constant recharge. They were connected via two JST-PH connectors. Both regulators below have approximately 1.1 V dropout, which our battery pack comfortably exceeds across its full range.

LD1117S50TR Motor Regulator: This motor voltage regulator was chosen because it is able to regulate our 7.4 V battery voltage to the desired 5 V for the motors. These were also widely available and cheap. Our motors ran at 6 V (nominal), which meant this regulator was more than enough.

AZ1117 Voltage Regulator: This regulator was a stocked 3.3 V regulator in the EIH in the SOT-223 package. It was used to supply the microcontroller, the OLED display, the three time-of-flight sensors, and the DRV8411 motor drive reference input.

Slide Switch SPDT: Selects between battery and USB-Supplied VIN, allowing the board to be powered from a bench supply during testing if need be.

Sensors

VL53L0X Time-of-Flight: We used three ToF sensors, one on the left, front, and right sides of the board. These sensors were powered by 3.3 V power. These sensors faced outward to detect walls and for centering purposes. During the testing phases of the project, we settled on this specific model, but as a breakout board. We did notice that the update rate was relatively slow compared to other types of sensors, but for the motors we were using, we were satisfied with the speed. The I^2C bus carries all three sensors, keeping GPIO usage on the ESP32-S3 low. The sensor's 30 mm minimum measurement range is below our 60 mm wall-centering setpoint, so we never operate in a dead zone. The front sensor was used to detect a front wall. When the front sensor reads below 80 mm, the mouse stops and makes a turn decision using the side sensors. The left and right sensors had two primary goals: keep the mouse centered at all times, and detect openings or dead ends. Altogether, these three sensors allowed the mouse to be centered in the maze at all times, as well as help make turn decisions.

Motors

Adafruit N20 DC Motor 50:1 Gear Ratio (w/ Magnetic Encoder): We selected this motor primarily because it came with an encoder, and we were familiar with the family of motors, as we had tested with a 298:1 previously. It had the appropriate torque and speed needed for the maze and, more importantly, our ToF sensors. The motors were also small and the perfect size for our mouse. They are listed as 6 V, but we ran them at 5 V due to our voltage regulator design. The motors operated at 94% PWM (basePWM 240/255) during phase one of the maze navigation, and 100% during the speedrun phase of the algorithm. The encoders are the second type of sensor on the board. They serve three purposes:

1) Terminating turns when the count delta reaches the target
2) Equal left and right wheel speeds via encoder-balance PID
3) Detect cell-boundary crossings during forward driving

DRV8411A Motor Driver: This dual H-Bridge motor driver was chosen because it is capable of independently controlling both drive motors. It has forward, reverse, and brake support. It can read PWM input logic from the ESP32-S3. It provides 4 A peak, comfortably above our motors of around a 1.5 A stall current. This motor driver also allows more than enough voltage to the motor. It used two channels in one package, which meant we only needed one motor-driver IC on the board. This motor driver is in a small QFN package that requires a manual pick and place machine, which caused a soldering issue on the first board assembly that we resolved on the second board.

Additional Components and Features

USB-C Port: We used the EIH-stocked USB-C. We had used this for our practice mouse as well. This allowed us to upload code to the microcontroller. We were also able to power the board with this function, allowing simple tests of each component individually if needed.

OLED Screen

SSD1306 OLED Screen: Used for debugging purposes during testing. On race day, the display showed the current heading and FSM state for spectator visibility.

Battery and Motor Holder

3-D Printed Bracket: Our design is very similar to the design used on the practice mouse. Since our mouse had a smaller width than the practice mouse, we redesigned the holder to accommodate this change. To allow us to mount this piece, our board had four holes. One on each side of the motors. Our battery holder was above the board with an open box format and two little slits on either side to allow our battery wires easy access to our board. The design is essentially what was provided by Professor Schafer.

3.2. Software Design

Disclaimer: While our team developed the general overall structure and functionality of the software used by our micromouse robot, AI tools, particularly Claude and ChatGPT, were used to assist in the debugging process. We also found AI input helpful when learning how to go about implementing maze solving algorithms.

Overall Software Structure

The code is organized as a finite state machine with four states. It also runs inside a loop. IDLE, DRIVE, ROTATE, and SPEEDRUN. In the IDLE state, the motors stop and wait for the start button after maze navigation. In the DRIVE state, the mouse drives forward through a cell under the flood-fill guidance. In the ROTATE state, the mouse either executes a 90° or 180° turn in place. Finally, the SPEEDRUN state happens after the IDLE state when the button is pressed and uses the computed shortest path with continuous motion. The state transitions happen during defined moments: cell-boundary crossings, front-wall stops, turn completions, and arrivals at the goal or start. Each state resets the encoder counters and PID integrators so the next state begins cleanly.

Within the loop, we run three time bases at the same time. The code used a motor control loop that runs every 1 ms. This loop runs the two PID controllers, applies PWM values, and checks for cell-boundary crossings. The second time base is a sensor and decision loop that runs every 30 ms. This checks all three ToF sensors, updates the average, evaluates the wall-centering error

variable, and handles front-wall detection. The last of these is the display update every 100 or 250 ms, depending on the state.

Low-Level Software Decisions

ToF Sensor Interface: All three VL53L0X sensors share one I^2C bus along the OLED. The sensors are initialized one at a time at boot using XSHUT pins to avoid address overwrites since they are all initially set at an address of 0x29. The sequence: hold all three XSHUT pins low, raise the front sensor, initialize the sensor, and write it to the new address of 0x30. This repeats for the left sensor, setting it at an address 0x31, and the right sensor, setting it at an address of 0x32. Each sensor reading passes through a three-tap moving-average filter before being used by the control logic. Wall presence is judged against a 120 mm threshold, meaning that if either side reads anything below 120 mm, the centering will engage. The wall-centering loop targets a 60 mm setpoint, holding the mouse roughly halfway between two walls of a standard cell. Anything above that means that there is an opening on that side. The front wall uses a threshold of 80 mm to trigger a stop.

Encoder Interface: Each motor's encoder is read by dedicated pin-change interrupt service routines. This forces the ISR code to live in RAM instead of flash. We have the rotation direction from the A and B channels at the moment the A-channel transitions. The two ISRs use opposite quadrature comparison on the right because the motors are mounted as mirror images. With this, one counter would tick down during forward driving, and the encoder-balance PID would interpret the difference as an error, causing the right to stall.

Motor Control: Each motor has two PWM pins (forward and reverse). There are four motion primitives: driveForward, spinRight, spinLeft, and stopMotors. Higher-level driving code calls these instead of analogWrite.

Dual PID Control Loops: Straight-line driving is governed by two PID controllers running simultaneously. The corrections are summed before being applied to the motors.
Encoder-balance PID keeps the left and right encoder counts equal, so the mouse drives in a straight line. *Wall-centering PID* uses the side ToF readings to keep the mouse in the middle of the corridor. The two corrections sum to a single term, which is added to one motor's PWM and subtracted from the other. The base PWM of 240 is intentionally below the max, so the PID has room to push a motor faster when needed, not just slower.

Open-Corridor Encoder Boost: When both side walls disappear, the wall-centering PID has no signal to lock onto, and corrections head toward zero. Without compensation, the mouse would drift to a side in an open region. To handle this, we boost the encoder-balance PID proportional gain by 4x whenever we encounter an open corridor. We also re-zero the encoder differential at the wall-to-open transition. This prevents the boosted gain from acting on whatever differential

error had built up while the wall PID was masking it, which would have caused a swerve at the point the robot crossed into a cell with no walls on either side.

Higher-Level Software Design

Cell Counting: We measure cell crossings by encoder counts rather than by time or wall transitions. These values were tested repeatedly until the cell-boundary detection was reliable. During exploration, the counts per cell equal 1185. When this encoder count is reached, we track a new cell, the mouse stops, and moves forward. During the speedrun portion, the counts per cell equal 1235. Here, we noticed that higher PWM throughout the cell increases wheel slip and miscalculations. Counts per cell during the speed run on cells with turns equal 1195. When the next cell ends in a turn, this encoder count is active. Three different constants are used because the slipping depends on the PWM. Using a single variable would cause the mouse to turn in too early or turn in too late, and cause crashes.

Cell-Boundary Handling (Phase One): When the cell-progress counter reaches the desired length, we stop the motors briefly, update the positions, mark the new cell as visited, sample all three sensors to record for walls, run the maze-search algorithm to pick the next move, and finally reset the encoder counters and continue forward or transition to ROTATE. If the front sensor reads a distance below 80 mm before the cell-progress counter reaches its target, the mouse treats the encounter as a cell crossing in place, logs the front wall, and falls through the same decision logic.

Turn Execution: Turns are open-loop, encoder-counted spins in place. We hold one motor forward and the other in reverse at fixed PWM until the encoder count reaches the target. The right and left targets are not the same due to the difference in the weight of the mouse, primarily because the chassis weight is not perfectly symmetrical. The speedrun phase uses its own turn-count constants, about 5% larger than the exploration values. We also have a filter that is called to reload moving-average buffers from the new orientation to prevent stale readings from the previous heading.

Maze Search Algorithm: The mouse uses flood-fill search for both exploration and speed-run path planning. Flood fill is a breadth-first search variant. It computes the shortest distance from a target cell to every other cell, propagating outward through open passages. The map lives in four arrays: walls, openEdges, visited, and floodDist. The walls array is a bitmask of the cardinal directions of walls recorded during exploration. A wall in this array means the ToF sensed a wall there. The openEdges tracks edges we physically drove through, combining to form confirmed passages. The visited array is a boolean marking if the mouse has been in a cell. The floodDist array is the flood-fill distance from the current target, recomputed at every cell. The difference between walls and openEdges is important. The absence of a wall in the walls array means we never sensed one, not that we know the passage is open. The openEdges array is a stronger

guarantee that the mouse actually drove through it. This makes more sense in the two flood-fill variants implemented:

1) Optimistic flood: This is used during the non-speed run portion when heading to the target. This propagates through any edge not blocked by a recorded wall. Unknown cells default to “open”. If the mouse guesses wrong, it runs into a wall, logs it, and the next flood routes around the obstruction.
2) Pessimistic flood: This is used during the return trip in the non-speed run phase and the speed run phase. This propagates only through edges in the openEdges array and only into cells in the visited array. This way, we guarantee that any path we follow is one already physically driven.

Both variants are seeded from the 2x2 goal region at the center. We initialize all four goal cells (7-8,7-8) with distance 0. This multi-seed approach lets the flood propagate outward as if the entire goal region is a single cell, so the mouse heads to whichever goal cell is closest to its current cell, rather than committing to just (7,7), for example.

Per-Cell Decision: There are four key decisions the mouse performs:

1) Recompute the flood from the current target.
2) For each of the three neighboring cells (front, left, right), look up its flood distance.
3) Pick the neighbor with the smallest distance. (The front always has priority on ties, followed by left, then right. This way, we have a zero turn cost.)
4) If all three neighbors are blocked or unreachable, make a U-turn.

Goal Handling and Return: When the mouse first enters any of the first goal cells, it pauses for two seconds and U-turns. It then re-runs the same decision loop with the flood seeded from (0,0) instead of from the goal region, and follows the pessimistic flood-fill back. Once the mouse returns to start, it faces north again and determines whether to run another exploration pass-through. This is via an iterative exploration. A single exploration run rarely finds the optimal path in a maze. We re-run the exploration as many times as we want (currently 2) or if the speedrun path computed at the end of a run is identical to the previously calculated path at the end of the previous run.

Speedrun Path Planner: After exploration completes, we build a contiguous path from (0,0) to the nearest goal cell by following the saved cardinal directions on the pessimistic flood gradient. From the maze search algorithm during phase 1, we have a sequence of headings saved (N, E, S, W) stored in a speedrun path. Since the flood is pessimistic, every step in the resulting path is guaranteed to go through the openEdge array that the mouse has physically driven. The path cannot lead into an unverified passage.

Speedrun Motion: The speedrun is different from the exploration in three ways:

1) The mouse does not stop at cell boundaries. We make it drive continuously through the entire path, only stopping to turn.
2) The motors use three PWM values. We start each cell at 255. When the robot approaches a cell, we lower it to 220. When the next cell is a turn, we slow it down to 200.
3) We also disable the front sensor during the speed run. The path is precomputed and verified, so a front reading only produced false aborts for our maze. The side sensors continue to drive the wall-centering PID for in-corridor positioning at a reduced strength.

4. System Testing

4.1 Overall Testing Approach

The testing approach throughout the semester was to build and validate the robot in layers. Rather than waiting until the custom board was fully assembled, our team used the practice mouse to test basic motion, encoder behavior, PID control, ToF sensing, wall-centering, and maze navigation logic. This allowed the software to develop alongside the hardware and helped identity control and sensing problems before fully integrating hardware with our custom mouse board.

The testing philosophy was to test early and often, making small improvements as issues arose. Each subsystem was first tested independently before being combined with other parts of the robot. Basic driving was tested before PID control, PID control was tested before wall-centering, wall-centering was tested before maze navigation, and maze navigation was tested before the speedrun logic. This layered approach made it easier to determine whether problems came from the motors, sensors, encoders, PCB, or software logic, making debugging much more manageable.

4.2 Basic Motion and Encoder Testing

Early testing focused on basic motion. During the first design review, the mouse was tested on simple movement requirements: driving straight for a fixed distance, stopping, turning 180 degrees, and repeating. This showed that our original motors were too slow for practical maze navigation, which led the team to switch from 1:298 gear ratio motors to faster 1:50 gear ratio motors.

This stage of testing also showed that the left and right motors did not behave identical to one another under the same PWM command, making straight-line driving unreliable without feedback control. To better understand the robot's motion, our team measured wheel diameter, calculated wheel circumference, and used encoder counts to estimate both straight-line travel and turn distances. This gave us a starting point for determining how many encoder counts corresponded to one maze cell, a 90° turn, and a 180° turn.

4.3 PID and Wall-Centering Testing

After identifying the motor mismatch problem, our testing shifted to PID control. The team tested methods for synchronizing the two motors using encoder feedback, with early work focused on making the robot drive straight on open ground by minimizing the difference between left and right encoder counts.

Testing showed, however, that solely driving straight on the floor was not sufficient for maze navigation. In the maze, a critical requirement was for the mouse to stay centered between the walls. This shifted the control strategy from only matching motor speeds to also using ToF sensor readings for wall-centering correction. Encoder feedback remained important for measuring distance and turn progress, but side sensor feedback became needed for keeping the robot aligned within the maze corridors.

4.4 ToF Sensor and S-Maze Testing

A major testing milestone was navigating a small section of the maze shaped as an S. For this test, the robot used ToF sensors, PID correction, and a finite state machine to navigate an S-shaped path through the practice maze. The software separated the robot's behavior into a drive state and a rotate state. In the drive state, the robot used calibrated side-wall distances to stay centered and stopped when the front sensor detected a wall. It then entered the rotate state, turned using encoder counts, and returned to the drive state.

This test validated several parts of the system we ultimately incorporated into our custom mouse PCB board. It confirmed the ToF sensors could be used for wall detection, side sensor feedback could keep the robot centered, and encoder-count based turns were dependable enough for basic maze movement. It also confirmed that the finite state machine improved reliability by preventing the robot from attempting to drive and turn at the same time.

The S-maze testing also revealed limitations. The left and right ToF sensors did not report the same distance when the robot was in the center of a corridor, even after replacing and using another sensor. To compensate, our team manually calibrated the center distances in the software rather than assuming equal left and right target values. Testing also showed that at higher PWM values the robot overshot the intended front-wall stopping point, making sensor update timing, stopping distance, and speed turning important design considerations going forward from this point.

4.5 PCB and Hardware Testing

While doing tests with the software on the practice mouse, our team tested and revised our custom PCB design. Before ordering the board, we reviewed important layout details such as USB differential pair routing, ToF connector orientation, motor connector spacing, motor trace width, and motor driver grounding and thermal behavior. These reviews were necessary because

the board needed to support the ESP32-S3, DRV8411A motor driver, ToF sensors, motor connectors, encoders, USB programming, and power routing.

After the board was assembled, our team tested basic functionality, including whether the board could be programmed, whether USB worked, whether the reset and boot buttons were functional, whether the sensors received power, and whether the motor connectors received proper voltage. We also debugged voltage levels across board traces using a power supply and multimeter.

Hardware testing revealed several issues. Solder joints around the USB and motor driver needed to be cleaner, and the motor connectors had been attached backwards. Because the connectors could not simply be removed and reoriented, the team rebuilt the main board. This highlighted the importance of verifying both our PCB design as well as our physical board's construction. Once this was done, our custom board was able to integrate the software and work as intended.

4.6 Maze Mapping and Solving Tests

Once basic movement, sensing, and hardware behaviors were working, testing shifted toward maze mapping and solving. The robot needed to move beyond only reacting to walls toward storing information about the maze, which required testing its ability to track position, record walls, mark visited cells, and remember confirmed open paths inside the maze.

Maze-solving logic was tested by adding memory and flood-fill behavior. The final software represented the maze as a grid and stored wall data, visited cells, flood-fill distances, and confirmed open edges. This distinction was important because a missing wall reading did not necessarily mean the robot had physically traveled through that path. Confirmed open edges represented paths the robot had actually driven through, making them more reliable for the return to the start behavior and the speedrun planning.

4.7 Full-System and Speedrun Testing

Once our custom board was complete, testing focused on integrating all the robot's systems together. The robot needed to combine all of its subsystems: ToF sensing, encoder feedback, PID correction, finite state machine logic, maze memory, flood-fill path planning, and the speedrun execution. This was the most difficult stage because problems that did not surface during isolated tests showed up when all the parts of the system had to work together.

Speedrun testing required additional tuning because the robot behaved differently at the higher PWM and speed. The final code used separated constants for speedrun movement, including different encoder count targets, turn counts, and PWM values. The robot also slowed near the end of each cell, particularly when the next action was a turn, to reduce overshoot and improve the turning reliability.

During the speedrun testing, our team also reduced the role of the front sensor. Even though it was useful during the exploration, the front sensor kept causing incorrect behavior and movements during the speedrun. For this reason, the final speedrun logic relied more heavily on the encoder counts and side-wall correction while following along the precomputed speedrun path discovered by the flood-fill algorithm.

4.8 Summary of Testing Results

Overall, the testing process showed that the most significant design problems appeared while we integrated all of the parts together. Individual subsystems could function correctly by themselves, but the full system required the motors, encoders, ToF sensors, PCB layout, PID control, and maze-solving software to all operate together reliably so that our micromouse could solve the maze regardless of the maze's layout. Each stage of testing showed different issues: motor mismatch during early driving tests, sensor calibration problems during the wall-centering, overshooting at higher speeds, board construction issues during the hardware validation, and the maze-tracking challenges when the full system was all put together.

These results directly shaped the final design. Motor testing led to the gear ratio change. Encoder testing led to encoder-count based movement and turning. ToF testing led to calibrated and fixed wall-centering values. The S-maze demo validated the finite state machine structure. PCB testing led to board repairs and having to reconstruct it. Full-system testing led to a flood-fill path algorithm approach, confirmed open edges, and a dedicated speedrun mode at the end. Through this process, the team continuously improved the robot based on the performance rather than just relying on the original design assumptions, leading to our micromouse being able to complete the requirements of the competition.

5. Conclusions

On Race Day, Electric Vermin placed fifth out of the twelve senior design teams. (Fifth out of the five teams who each managed to get their micromouse to the center of the maze.)

During our ten minutes on Race Day, our micromouse succeeded in solving the maze. Our micromouse worked by starting at the beginning of the maze, exploring the maze while pausing briefly at each cell until reaching the center, and then returning to the starting point. To give our micromouse a better chance of finding a faster route to the center, the micromouse would then repeat these steps, taking a different path to the center and returning again to the start. Upon its second return to the starting point, we would press a button located on our PCB to tell our micromouse to begin the "speedrun" stage, where after determining the fastest route to center based on what is has seen, the micromouse would take the fastest route to the center of the maze it knows without pausing in each cell. Due to the complexity of the Race Day maze and the length of its solving route, we quickly realized it would take slightly over ten minutes for our two

exploration stages and one speedrun to finish. Regardless, our micromouse successfully arrived at the maze's center and returned to the start during the first exploration stage. However, although our micromouse did arrive again at the maze's center during the second exploration stage, it bumped into a wall only a few turns away from reaching the starting point again, therefore ending our official runthrough of the maze. Despite not being able to proceed to the speedrun stage, we still managed to earn a time of 3:09. Although our 3:09 time made our micromouse the slowest of the micromice that solved the maze, we were pleased with our results.

Later, we ran our micromouse again through the Race Day maze, and it successfully made it to the speedrun stage without any crashes. With the speedrun during our unofficial second run through, our micromouse managed to shave down an entire minute off our official race time of 3:09 to a time of 2:09. Although the crash in our first run through was unfortunate, it was rewarding to witness a faster time with the speedrun in our second run through. The shorter solving time demonstrated how our micromouse could determine the fastest route to the center of the maze it knew and how other elements of the speedrun stage made our mouse move faster, like running at a higher PWM and not pausing in each cell. (Also, our 2:09 time put us not too far off from our closest competitors, Team Tango and Anyone Can Cook.)

Despite not achieving the fastest time, we were happy to succeed in placing on the leaderboard. If we could have gone back in time and made different design choices, we would consider using faster motors and a potentially mix of IR and ToF sensors.