EE 40290 Senior Design Project: MicroMouse

Project Report

By: Nguyen Nguyen, Collin Noon, Zach Erling, Cole Descalzi

Senior Design Spring 2026

Department of Electrical Engineering

TABLE OF CONTENTS

1. Introduction

This report covers the design, build, and testing of an autonomous maze-solving robot — a Micromouse — for EE 40290 Senior Design. The goal was to build a robot that could navigate a 16×16 maze on its own, using onboard sensors, motor control, and a flood-fill algorithm to reach the center.

The robot runs on a custom two-layer PCB we designed in KiCad, around 70×68 mm — small enough to fit inside a single 180 mm maze cell with room for the sensors to read the walls on either side. We used an ESP32-S3 as the main microcontroller, mainly for its hardware encoder counting, PWM outputs, and built-in BLE radio. Four IR sensors cover the left, front-left, front-right, and right directions, a DRV8411 dual H-bridge drives the two DC gear motors, and quadrature encoders on each motor feed back into the ESP32's hardware counters.

The firmware runs on FreeRTOS and concurrently handles motion control, sensor readings, and BLE telemetry. Navigation uses a flood-fill algorithm on a 16×16 cell grid, and a PD controller handles wall following and turns. BLE ended up being really useful during development — we could tune all 26 motion parameters live without reflashing, and the PC-side tools gave us real-time visibility into what the robot was seeing and deciding.

The final system didn't fully navigate the maze, but it did successfully demonstrate wall detection, flood-fill path planning, and PD motion control all working together on the finished hardware.

2. Key System Requirements

One of the most fundamental requirements for the robot was size — it needed to fit within a single 180 mm maze cell, which drove the PCB down to approximately 70×68 mm and influenced basically every other form factor decision, including where the sensors could be mounted and how much clearance they'd have to read adjacent walls.

For the microcontroller, we needed a hardware encoder counting to avoid missing edges at high motor speeds and CPU interrupt overhead — the ESP32-S3's PCNT units covered this, and the built-in BLE radio meant we didn't need a separate wireless IC. PWM-capable GPIO for motor control was also a requirement, which the ESP32-S3 satisfied.

On the power side, the motor rail needed to stay consistent regardless of battery charge state, which is why we regulated it to a fixed 5 V through the BD50FC0FP LDO rather than running directly off the battery. For the sensor front-end, we chose 3.3 V over 2.8 V for the op-amp supply specifically to maximize output swing and ADC dynamic range on the ESP32's 12-bit ADC. The TLV9002 op-amp was selected for its rail-to-rail output, low offset voltage, and 3.3 V compatibility — all in support of getting the best possible sensor readings.

Hardware encoder counting on dedicated PCNT units was a requirement driven by the need to avoid missed edges at high motor speeds without burning CPU cycles on interrupts.

3. Detailed Project Description

Hardware

3.1 PCB and Form Factor

The robot runs on a custom two-layer PCB designed in KiCad, measuring approximately 70 × 68 mm. All components are surface-mount; passive components use 0603 and 0805 imperial footprints. Three fiducial markers were included to support automated optical inspection alignment. The compact form factor was driven by the maze cell size of 180 mm — the robot must fit within a single cell with clearance for the IR sensors to read adjacent walls accurately.

3.2 Microcontroller — ESP32-S3 (U1)

The ESP32-S3 module was chosen for its hardware peripheral set (PCNT units for encoder counting, multiple PWM-capable GPIO), and integrated 2.4 GHz BLE radio, eliminating the need for a separate wireless IC. A boot-assist resistor network and dedicated Reset/Boot pushbuttons (SW1/SW2) are included for reliable flashing without manual GPIO manipulation. GPIO45 is noted on the schematic as pulled low by the DRV8411, which was an important strapping-pin constraint to observe during layout.

3.3 Motor Driver — DRV8411 (U4)

A single Texas Instruments DRV8411 dual H-bridge drives both DC gear motors from a single IC. The DRV8411 supports coast mode (both inputs low, outputs Hi-Z) and brake mode (both inputs high, slow-decay), which the firmware uses to implement ramped braking before turns. The motor supply rail is held at 5 V by the BD50FC0FP LDO (U3), keeping motor behavior consistent regardless of battery state-of-charge variation. A 1N5819WS Schottky diode (D3) provides reverse-polarity protection on the battery input.

3.4 Power Architecture

The board runs from 2, 3.4V LiPo batteries in series to create a 7.4V supply. Three voltage rails are derived from it:

- +BATT — raw battery rail, feeds the DRV8411 motor supply through the BD50FC0FP 5 V LDO
- +5V — regulated 5 V for the motor driver logic and IR emitter drive
- +3.3V — regulated 3.3 V from the AP2112 LDO (U2), powering the ESP32-S3 and all sensor analog circuitry. The schematic notes "Using 3.3V instead of 2.8V" for the IR sensor op-amp supply, reflecting a deliberate decision to use the higher rail to maximize op-amp output swing and sensor sensitivity.

AO3401A P-channel MOSFETs (Q1) handle power-path switching. A USB-C connector (USB4105, J7) with 5.1 kΩ CC resistors (R7, R8) provides USB power and programming access. Two indicator LEDs (green, blue) give power-on and status feedback.

3.5 IR Sensor Front-End

Each of the four IR channels (Left, Front-Left, Front-Right, Right) consists of:

1. An NMOS transistor (Q2/Q4/Q6/Q8/Q12) that switches the IR emitter LED on and off under GPIO control, allowing ambient-light subtraction in software.
2. An SFH_313_FA phototransistor (Q3/Q5/Q7/Q9/Q11) as the receiver, which produces a current proportional to reflected IR intensity.
3. A TLV9002IDR dual op-amp (U5–U7) stage that converts the phototransistor current to a voltage and buffers it to the ESP32-S3 ADC input.

The TLV9002 was chosen for its rail-to-rail output, low offset voltage, and 3.3 V supply compatibility, important for maximizing ADC dynamic range on the ESP32's 12-bit ADC.

Resistor networks in the 2.2 kΩ–47 kΩ range set the transimpedance gain and bias. The schematic note "Front facing needs opposite pinout" reflects the fact that the front-left and front-right sensors are mounted facing forward at an angle, requiring their emitter/receiver pairs to be mirrored on the PCB relative to the side sensors.

3.6 Encoders

Quadrature encoder signals from both motors are routed to dedicated ESP32-S3 PCNT hardware counter units (Unit 0 = left, Unit 1 = right) through 10 kΩ pull-up resistors on the A/B channels. Hardware counting eliminates CPU interrupt overhead and prevents missed edges at high motor speeds.

3.7 Board Errors and Corrections

In the initial board design, the dimensions of the board made it difficult for the mouse to rotate inside a maze cell. In addition, we had flipped one of the power transistors' footprints, which misrouted the power delivery to the rest of the circuit.

In the second iteration of the mouse, we experienced issues with the motor driver and power circuitry.

3.7.1 Motor Driver

With regards to the motor driver, we designed the PCB with the wrong footprint. The footprint on the PCB was for a 4x4mm motor driver, where our own driver was 3x3mm for this reason, the pitches of the pins of the motor driver would never line up with the pads fully. This meant that only one of the motors would work at once as the driver would shift towards one side of the set of pads.

To amend the footprint mismatch, we utilized an external H-Bridge from an Arduino Dev Kit. It was a through-hole package we put on a breadboard. We then used jumper wires and soldered thin wires onto the pads of the motors and power rails.

3.7.2 Power Circuitry.

With regards to the power circuitry, we realized we were supplying our 3.3V regulator(rated for 6V) with the full battery voltage(7.4V). Eventually, our regulator would be burnt out under this overload. To amend this, we used a jumper wire to connect the 5V output of the 5V LDO to the input of the 3.3V regulator. We also made sure that there was no way for the 7.4V from the batteries to reach the 3.3V regulator by removing the power switching circuitry(PMOS and Schottky diode)

After this fix, we experienced severe overheating problems with the 3.3V regulator. Something on the 3.3V rail was sinking an unexpected amount of current. The only peripherals connected to this rail were the sensors and the ESP32. We have good reason to believe that the ESP32 power circuitry could be damaged, leading it to sink more current.

In the future, we would design the power circuitry to supply the 3.3V regulator with the 5V from the LDO instead of the batteries. In addition, we would get our footprints triple-checked to make sure the motor driver gets placed correctly.

Software

3.8 Overall Structure

The firmware uses FreeRTOS with three concurrent tasks: motionTask (2 ms period, highest priority), bluetoothTask (50 ms period, telemetry streaming), and loggingTask (2 s period, serial debug). All shared state (sensor readings, encoder counts, parameters) is accessed from motionTask and read-only from the other tasks, avoiding the need for explicit locking in the common case.

3.9 Sensor Reading

Four analog IR sensors (Left, Front-Left, Front-Right, Right) measure wall distances. Each sensor is driven by a dedicated emitter LED controlled by a GPIO pin. Readings use ambient-light subtraction: the ADC is sampled with the emitter off, then again with it on, and the difference is used. This approach substantially reduces sensitivity to room ambient illumination. Raw ADC counts are converted to millimeters in software using a per-sensor logarithmic model of the form:

$$d = K \log_{10}\left(\frac{A}{ADC - C}\right)$$

with constants A, K, and C calibrated empirically for each sensor by measuring known distances and fitting the curve.

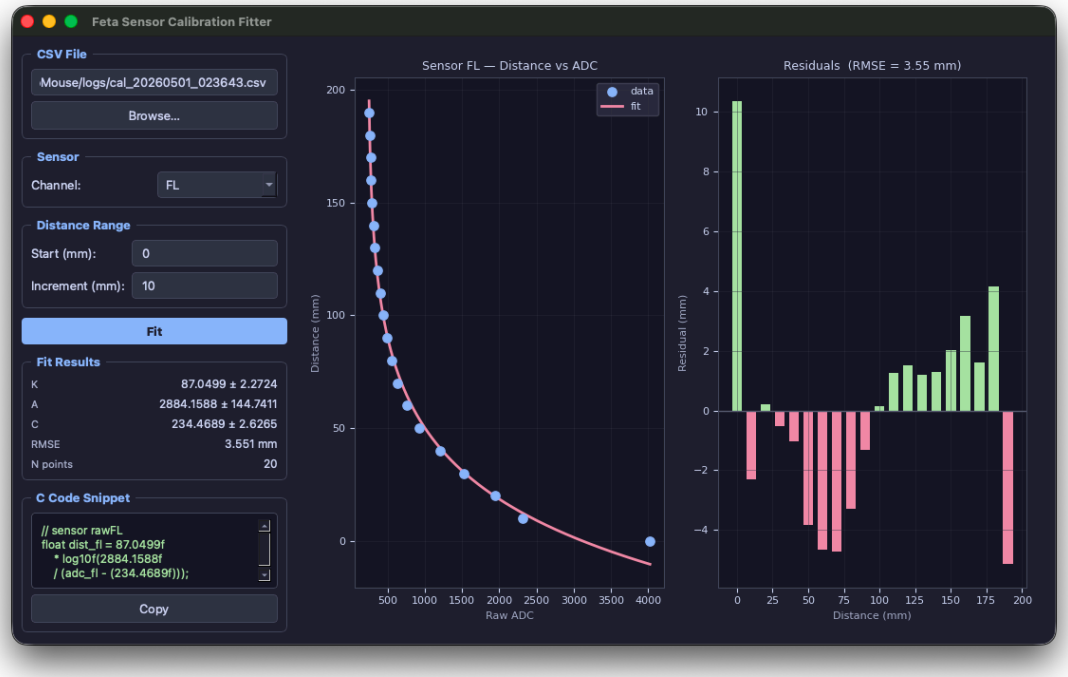
3.9.1 Calibration

For calibration, the robot was placed against a maze wall and moved away in fixed increments (typically 10 mm per step) using a ruler. At each position, the BLE telemetry stream was recorded using ble_monitor.py with CSV logging active (toggled via the C key). The resulting CSV contains one row per position with columns rawL, rawFL, rawFR, and rawR, which are the raw ADC difference values (emitter-on minus emitter-off) for all four sensors simultaneously. A session of 20 positions covered 0–190 mm, spanning the full useful sensing range.

To fit the curve, the sensor_fit.py script takes a calibration CSV, a sensor channel (L, FL, FR, or R), a starting distance, and a distance increment. It generates the corresponding ground-truth distance array, then calls scipy.optimize.curve_fit with the logarithmic model to find K, A, and C. Bounds are enforced to keep the denominator (ADC − C) positive across all data points. The tool reports the fitted parameters with their standard-error uncertainties, RMSE in mm, and prints a ready-to-paste C code snippet:

```
K =   82.0400  ± 1.2310
A = 1183.1000  ± 14.820
C =  209.7200  ± 3.410
RMSE = 2.341 mm
C code snippet:
 // sensor rawFR
 float dist_fr = 82.0400f * log10f(1183.1000f / (adc_fr -
(209.7200f)));
```

We also have this tool in a GUI version (sensor_fit_gui.py). A PyQt6 graphical interface wraps the same fitting logic for faster iteration. The user browses to the CSV, selects the sensor channel, enters the distance step, and clicks Fit. The tool displays a live two-panel plot (the fitted curve overlaid on the data points on the left, and a residuals bar chart on the right) along with the K/A/C values and RMSE in a results panel. A Copy button puts the C snippet directly on the clipboard for pasting into analog.cpp.

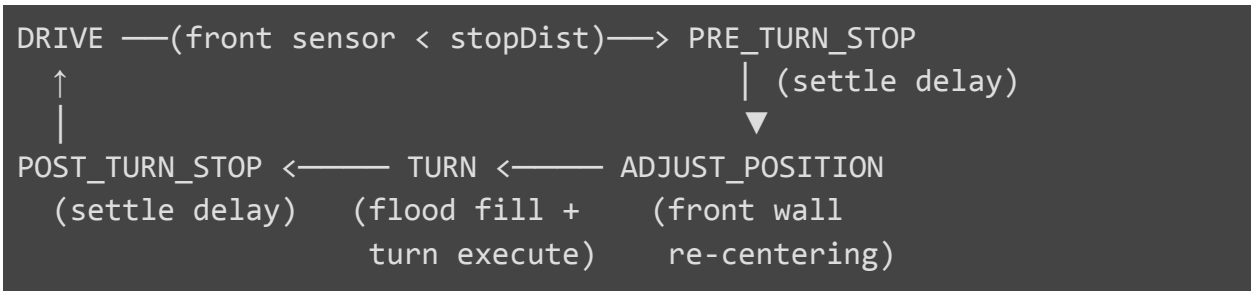


3.10 Sensor Control Algorithm

During straight-line driving, driveStraightCenterStep() implements a proportional-derivative (PD) controller that centers the robot between walls. When both side walls are visible, the error is the difference between left and right sensor readings relative to their respective target

distances. When only one wall is visible, separate single-wall gain parameters (singleKpL/singleKdL or singleKpR/singleKdR) are used to hug that wall at a fixed offset. When neither wall is visible, the encoder balancing fallback activates, using the difference in left and right encoder counts to maintain straight travel by proportional correction on motor PWM.

Motion state machine: The motionTask implements a five-state machine:



At each cell boundary, wall readings are recorded, the flood-fill distances are recomputed, and the next direction is selected as the accessible neighbor with the lowest flood value.

3.11 Maze Search Algorithm

The maze is represented as a 16×16 grid of cells, each storing a wall bitmask (known/unknown separately), a visited flag, and a flood-fill distance. Boundary walls are pre-initialized. The flood-fill runs as a BFS from the target cell(s), assigning distances to all reachable cells.

Unknown walls are treated as open passages during fill, so the robot proactively explores rather than assuming blocked paths. The solve proceeds in three phases managed by SolvePhase: TO_CENTER (flood to the four center cells), EXPLORE_GOAL (visit all four goal cells in sequence), and TO_START (flood back to origin).

3.12 Bluetooth Communication and Tooling

Bluetooth Low Energy was not just a debugging convenience — it formed the primary control interface for all iterative testing and tuning throughout the project. The entire parameter space of the robot (**26 tunable values**) could be modified live, without reflashing, while the robot was running. This section describes the firmware-side protocol and the PC-side tooling built around it.

3.12.1 Firmware BLE Implementation

The firmware implements the Nordic UART Service (NUS), a widely adopted BLE profile that exposes two GATT characteristics: a TX (notify) characteristic for data from mouse to PC, and an RX (write) characteristic for commands from PC to mouse. The ESP32-S3's BLE radio is initialized with an MTU of 512 bytes to reduce fragmentation of longer telemetry packets. Three volatile flags are the primary interface between the BLE callback and the motion task:

- motionEnabled — set by M1/M0 commands; gates whether motionTask drives the motors
- streamEnabled — set by R1/R0; gates whether bluetoothTask sends sensor data
- posResetRequested — set by ZP; triggers a full position and maze reset at the top of the motion loop

A safety invariant is enforced on disconnect: motionEnabled and streamEnabled are both cleared when the PC drops off, so the robot always stops if the BLE link is lost.

3.12.2 Telemetry Packet Types

Four packet types are emitted over the TX characteristic, each prefixed by a single-letter tag:

Tag	Rate	Contents
H	once per stream session	CSV column header
S	4 Hz	Live sensor readings — timestamp, L/FL/FR/R distances in mm, plus raw ADC values
T	each cell boundary (turn event)	Decision frame: timestamp, solve phase, robot position (x,y), previous and next heading, front/left/right wall booleans, all four sensor distances
X	on events	Out-of-band events: HEADING (initial direction resolved), DONE (maze completed with timestamp), RESET (position cleared)

The T frame is particularly important for post-run analysis: it captures a complete snapshot of everything the robot knew at each navigation decision, allowing the path taken and any misclassified walls to be reconstructed offline from the CSV log.

3.12.3 Live Parameter Updates

The RX characteristic accepts KEY=VALUE strings that update the Params struct in-place and immediately persist the new value to ESP32 NVS flash (so it survives power cycles). The full set of 26 tunable parameters covers:

- Wall-following PD gains: Kp/Kd (dual-wall), SKL/SDL (left-open), SKR/SDR (right-open)
- Distance thresholds: SD (front stop), WL/WR (wall hug targets), OT (open-wall threshold), AT/AD (position-adjust trigger and target)
- Motor parameters: MX/MN (max/min speed PWM), LB (left motor bias), BD (brake delay)
- Turn geometry: TL/TR/TA (encoder count targets for left/right/U-turns), LRP/RRP/ARP (right motor PWM during each turn type)

- Timing: TS (pre/post turn dwell delay), ME (max encoder counts per cell), SCT (sensor freeze ticks on dual→single-wall transition)

A GET command causes the firmware to send back all 26 current values in a single I,... packet, keeping the PC tool in sync after a reconnect.

3.12.4 PC Monitor (ble_monitor.py)

ble_monitor.py is an asyncio application built on bleak (cross-platform BLE) and rich (terminal UI). On launch, it auto-scans for the device named "Feta", connects, sends R1 to start the stream, and GET to read back current parameters, then enters a live-updating terminal UI refreshed at 10 Hz.

The monitor has two display modes: Normal mode shows a scrolling cell-traversal table (the last 8 T frames with position, heading, wall booleans, and sensor distances colour-coded by proximity), a live sensor panel (current distances and raw ADC values), current parameter values, and status indicators for recording and motion.

Nav mode (N key) replaces the traversal table with a full 16×16 ASCII maze rendered from the client-side wall reconstruction — visited cells, known walls, unknown passages (shown as dashed), goal cells, and the robot's current position and heading as a direction character. A decision log shows the last 16 turn events with phase annotation.

Calibration mode (C key) adds a third column to the display. Each press of the spacebar captures the current raw ADC values for all four sensors into a timestamped CSV file (cal_YYYYMMDD_HHMMSS.csv). The captured readings are shown live in the panel as they accumulate, and the file is flushed after each reading so data is safe even if the session is interrupted.

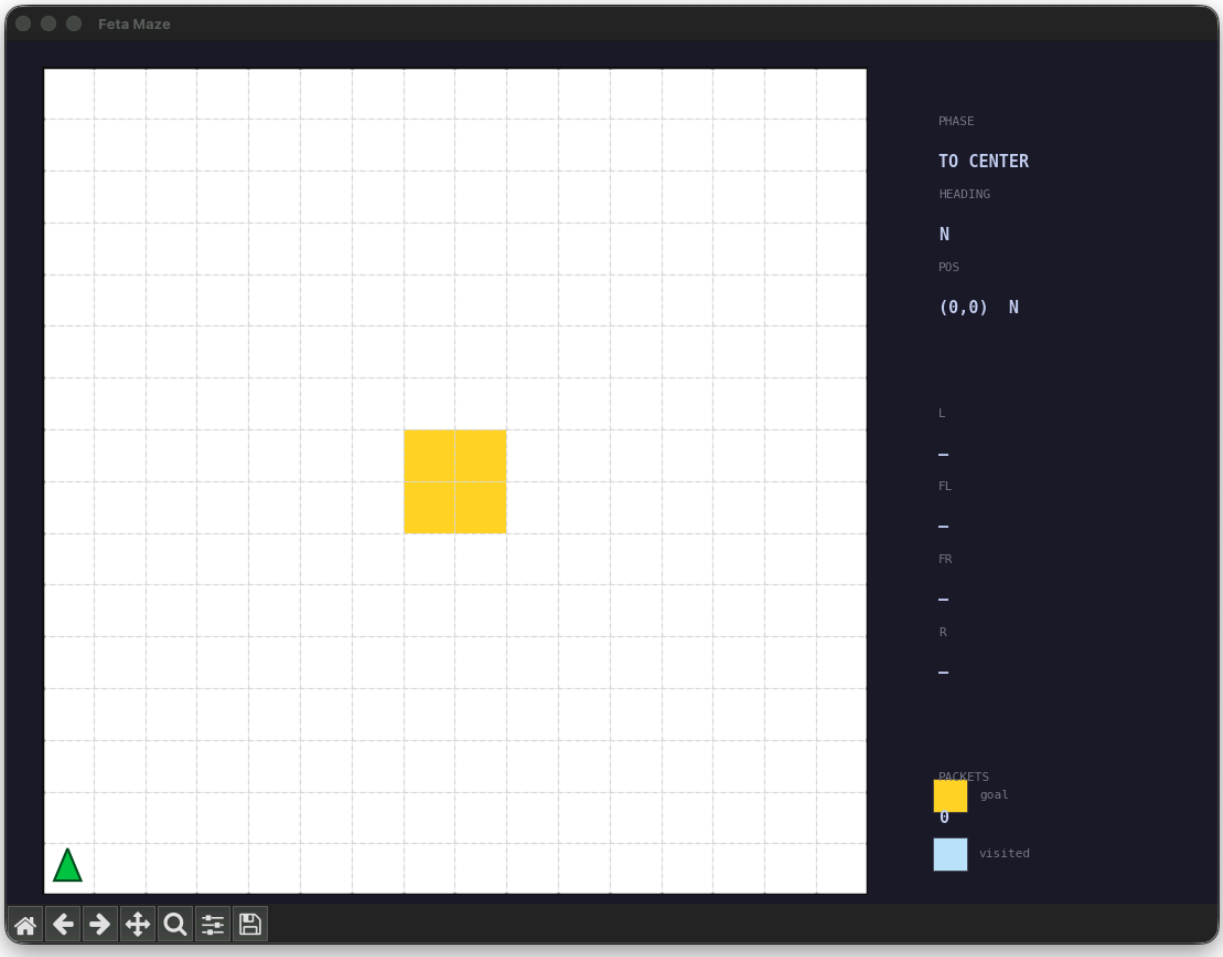
Parameter update (P key) pauses the live display, prints a numbered menu of all 26 parameters with their current and default values (parameters differing from default are marked with *), and prompts for a selection and new value. The KEY=VALUE command is sent over BLE, and the firmware's acknowledgment I,KEY=value is shown in the status bar when received.
Additional controls: M toggles motor enable (M1/M0), R starts/stops CSV recording of all T frames with timestamps, Z sends a position reset (ZP) and clears the client-side maze state, and Q safely disables motion and streaming before disconnecting.

3.12.5 Graphical Maze Visualizer (maze_gui.py)

When not run with --no-gui, ble_monitor.py launches a second thread that runs a matplotlib-based graphical maze window alongside the terminal UI. The visualizer draws the 16×16 maze at 10 fps using matplotlib patches and lines: cell backgrounds are coloured by state (white = unvisited, light blue = visited, gold = goal region), known walls are drawn as thick

black lines, unknown passages as thin dashed grey lines, and the robot as a filled triangle pointing in its current heading direction. When the initial heading is ambiguous (before the first open side sensor is seen), the robot is drawn as two overlapping orange ghost triangles with a ? label.

A sidebar shows phase, heading, position, live sensor readings, packet count, and — when the run completes — elapsed solve time. The matplotlib window and the terminal UI share the same AppState object (protected by Python's GIL), so both views update in real time from the same BLE notification stream.



4. System Testing

Throughout the semester, two critical issues came up during hardware bring-up. First, the 3.3 V regulator was being supplied directly from the 7.4 V battery rail — exceeding its 6 V rating — which ended up burning it out. We fixed this by routing the 5 V LDO output into the 3.3 V regulator input and removing the power-switching circuitry. Even after that fix, though, the

regulator kept overheating severely, which we think was due to damage to the ESP32 power circuitry, causing it to sink more current than expected on the rail. On top of that, the DRV8411 footprint mismatch — a 4×4 mm PCB footprint for a 3×3 mm package — caused only one motor to work at a time because the pads never lined up correctly. We worked around this by wiring in an external through-hole H-bridge on a breadboard using jumper wires connected to the motor and power pads.

For IR sensor calibration, we placed the robot against a maze wall and stepped it away in fixed increments, recording ambient-subtracted ADC values at each position through the BLE telemetry stream using ble_monitor.py with CSV logging active. A full calibration session covered 20 positions from 0–190 mm. From there, sensor_fit.py fits a logarithmic model to the data using scipy.optimize.curve_fit, giving us per-sensor K, A, and C constants with uncertainties and RMSE in millimeters, plus a ready-to-paste C snippet for analog.cpp. The graphical sensor_fit_gui.py version showed a live two-panel plot of the fitted curve against the data and a residuals bar chart, which made iterating a lot faster.

Motion control parameters were tuned live over BLE, which let us modify all 26 tunable values on the fly without reflashing — and changes persisted to NVS flash across power cycles. The ble_monitor.py terminal UI showed live sensor readings, and Nav mode displayed a full 16×16 ASCII maze built from the client-side wall reconstruction, including visited cells, known walls, unknown passages, goal cells, and the robot's current position and heading. The maze_gui.py visualizer showed the same thing graphically at 10 fps, with a sidebar reporting phase, heading, position, live sensor readings, and elapsed solve time at the end of a run. The T-frame telemetry log captured a full snapshot of every navigation decision to CSV — position, heading, wall detections, and sensor distances — so we could reconstruct exactly what the robot saw and decided at each cell after the fact.

5. Conclusions

The Micromouse project demonstrated a complete autonomous maze-solving robot, integrating PCB hardware, embedded firmware, and a supporting suite of PC-side tooling. Although the final system did not fully navigate the 16x16 maze, it did properly demonstrate the IR-based wall detection, flood-fill search algorithm, and a PD-motion control, all coordinated by an ESP32-S3 running a FreeRTOS multitasking firmware.

The project’s most significant lessons came from hardware bring-up. Two design errors - supplying the 3.3 V regulator directly from the 7.4 V battery rail and a mismatched DRV8411 footprint - forced mid-project workarounds that consumed substantial debugging time. Both errors were ultimately traced to insufficient design review before fabrication: the voltage rating violation could have been caught with a cross-check of the datasheet, and the footprint mismatch

would have been identified by verifying the package dimensions against the PCB land pattern. Going forward, there should be a more formal and detailed hardware review checklist covering voltage rail budgets, footprint-to-package verification, and strapping-pin constraints before any PCB is sent to fabrication.

On the software side, the BLE-based live tuning infrastructure proved to be one of the project's most effective design decisions. The ability to modify all 26 motion parameters on the fly - without reflashing - dramatically accelerated the iteration cycle for sensor calibration and PD gain tuning. The ble_monitor.py terminal UI and maze_gui.py graphical visualization provided real-time observability into the robot's navigation state, and the T-frame telemetry log enabled post-run reconstruction of every decision the robot made. This level of instrumentation was invaluable for diagnosing subtle issues in wall detection and turn geometry that would have been nearly impossible to debug through serial output alone.

The sensor calibration pipeline - using ambient-light-subtracted ADC readings, scipy curve fitting, and the sensor_fit_gui.py tool - produced per-sensor logarithmic models with RMSE values in the low single-digit millimeter range, providing sufficient accuracy for reliable wall detection throughout the 0-190 mm sensing range. The flood-fill maze solver, with its three-phase exploration strategy (to center, explore goal, to start) and treatment of unknown walls as open passages, enabled proactive exploration rather than conservative wall-hugging.

Several areas remain as clear targets for a future design revision. The power architecture should be redesigned to feed the 3.3 V regulator from the 5 V LDO output rather than the battery rail, and the power-switching circuit should be retained with correct component ratings. The motor driver footprint must be verified against the physical package dimensions, and all footprints should be triple-checked against datasheets before layout sign-off. Additionally, if the ESP32-S3 sustained damage to its internal power circuitry - as the overheating suggested - replacing it with a known good unit and adding current monitoring on the 3.3 V rail would help confirm rail health early in bring-up.

Despite these hardware setbacks, this team gained hands-on experience with the full embedded systems development cycle: schematic capture, PCB layout, firmware architecture, sensor modeling, closed-loop control, and wireless instrumentation. The MicroMouse platform provides for future iterations, and the lessons learned from this design cycle will directly inform more robust hardware choices and rigorous design verification in the future.