MTD BUS TRACKING DISPLAY
FINAL REPORT

By

Amber Wilt

Daniel Vlassov

Ziad Aldohaim

Final Report for ECE 445, Senior Design, Fall 2025

TA: Wesley Pang

5 December 2025

Project No. 4

Abstract

This document will detail the purpose, design, verification, and results of the Champaign-Urbana Mass Transit District live display. This system will show real-time bus data on a 3D map of the campus area of University of Illinois Urbana-Champaign intended to be used by college students. The display is intended to be shown in instructional buildings and run throughout a full school day. At the end of the Fall 2025 semester, this design was successfully implemented and demonstrated to be helpful in the daily life of on-campus commuters.

Contents

1. Introduction

Navigating the intricate bus system of Champaign, particularly for students at the University of Illinois Urbana-Champaign (UIUC), can be a daunting task. The numerous routes, transfer points, and unpredictable schedules often leave students uncertain about whether they will catch their bus, especially when faced with tight schedules between classes and long walking distances to bus stops. Current methods, such as relying on small screens at bus stops or using often unreliable apps, can exacerbate this confusion. To help address these issues, this project introduces a more intuitive solution: a 3D-printed physical display that shows real-time bus locations using color-coded RGB LED strips, providing a clear and dynamic visual representation of bus movements across the Champaign-Urbana area.

The goal of this system is to enhance student experience by simplifying route selection, reducing confusion, and ensuring more efficient planning during busy academic days. By integrating data from the Champaign-Urbana Mass Transit District (MTD) API, the display updates every 60 seconds, presenting a clear and easily understandable visualization of bus locations, routes, and directions. In addition, users can customize the system's features, such as adjusting brightness, color themes, and route highlighting to suit their preferences.

This report will review the final design of the system, covering its components, performance requirements, and any changes made to the initial design. It will also include a detailed breakdown of the subsystems and their interconnections, using a block diagram (Figure 1.1) to show the flow of data and power between them. By exploring the technical aspects of the project, we aim to demonstrate the key factors contributing to its performance and highlight the improvements made throughout the development process.

Purpose

Champaign's large and complex MTD bus system can be difficult for students to navigate due to the many routes, schedules, and transfer points spread across the University of Illinois Urbana-Champaign (UIUC) campus and surrounding city. With limited time between classes and long walking distances to get to stops, students often do not know whether they will make their bus until they arrive at the stop, where real-time information is displayed on a small screen. Furthermore, existing apps can be unreliable because of connectivity issues, unclear interfaces, or limited route comparison tools. These challenges cause a rise in confusion amongst students, leading them to often select the wrong route or miss their bus entirely.

To address these issues, we designed a 3D-printed physical display to show the real-time bus locations across the Champaign-Urbana area using color-coded individually addressable RGB LED strips. The display features a map of the campus, with each bus line represented by a distinct color and updated approximately every 60 seconds using data from the publicly accessible MTD Application Programming Interface (API). The system will also be able to periodically illuminate entire routes to help users understand bus direction and path, and include adjustable features such as brightness, themes, and route highlighting. By offering a clear and easy to understand visualization of bus movement, rather than lists or small maps, this model simplifies route selection, reduces confusion, and helps students plan their commutes more efficiently during tightly scheduled academic days.

1.1 Functionality

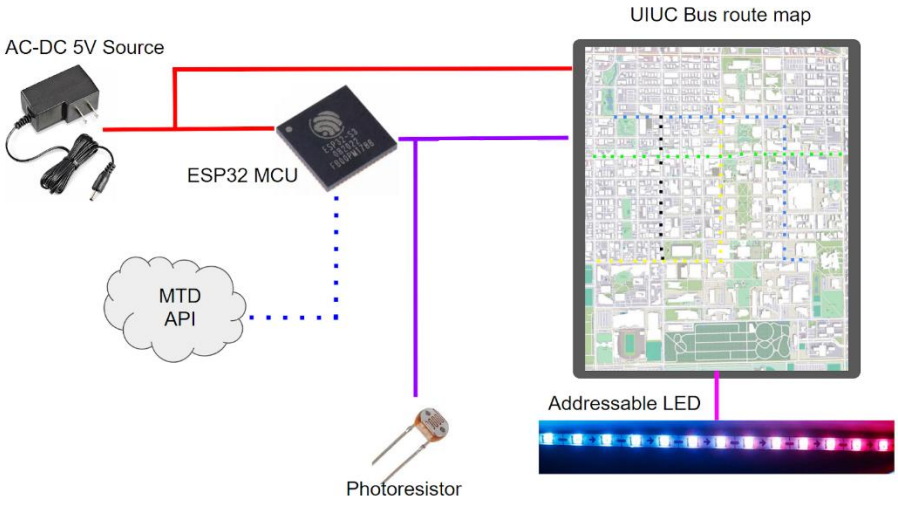


Figure 1.1 – Visual Aid of Design

1.2 Subsystems Overview

The sections below provide an overview of each subsystem within the project as well as their interconnects between other subsystems. For more information on the specific requirements for each subsystem see Appendix D. The block diagram shown in Figure 2.1 depicts the relation between the described subsystems as well as the type of connection required between them.

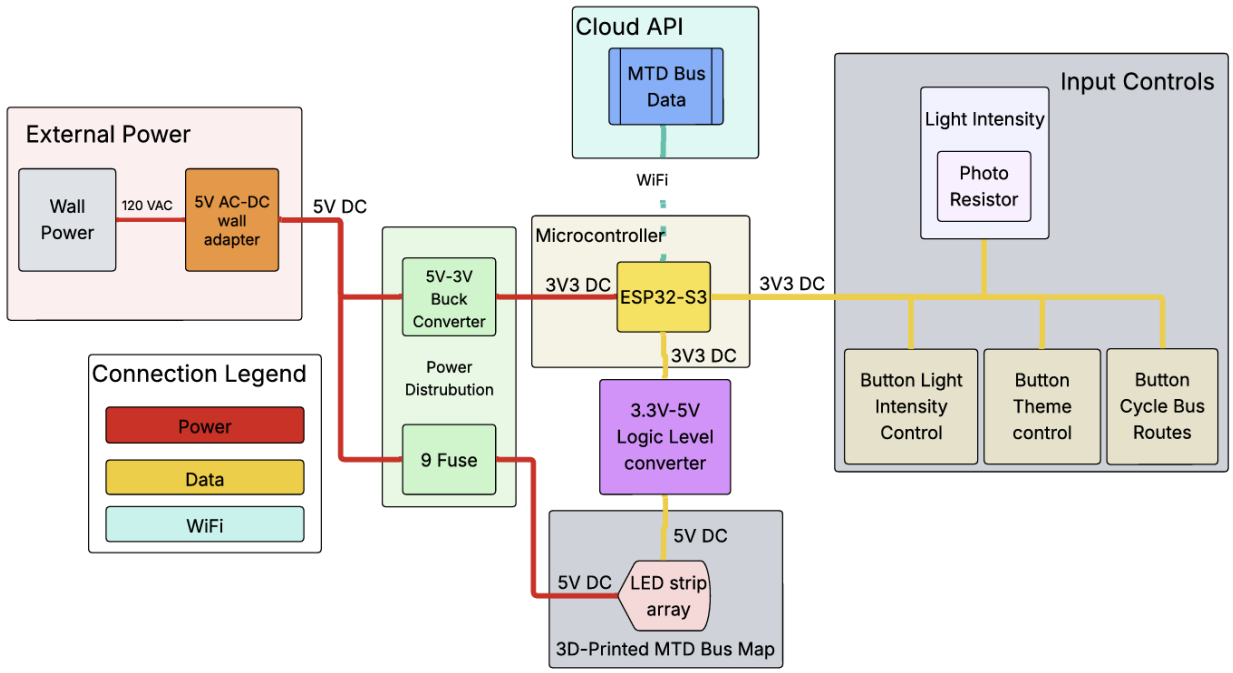


Figure 1.2 – Detailed Block Diagram of Design

1.2.1 External Power

The External Power Subsystem is responsible for providing power to both the microcontroller and the LED strip array via the Power Distribution Subsystem. This external power will come from a 120V AC wall plug and will be stepped down by a barrel jack wall adapter (rated for 5V and 10A) to a 5V DC signal. Step-downs and protection circuitry will be implemented by the Power Distribution Subsystem to make external power usable by the project.

1.2.2 Power Distribution

The Power Distribution Subsystem is responsible for transferring power from the External Power Subsystem and making it usable for the ESP32-s3 and the LED strip. This system uses a 5V to 3V3 buck converter to step down the external power and supply steady 3V3 power to the microcontroller. The rest of the Power Distribution goes into resettable fuses that will trip if the input current goes above 10A, which powers the LED strip that is housed in the 3D printed map. The fuses are implemented to provide redundant protection for the LED Subsystem to limit the risk of blowing or damaging any LEDs. For more details about the design of the power distribution, refer to Sections 2.2.1 and 2.2.2.

1.2.3 Microcontroller

The microcontroller we will be using is the ESP32-S3-WROOM-1. This chip has many capabilities, including an onboard RTC, BLE, and Wi-Fi modules. The microcontroller will be the brain of our project, having an FSM that will constantly probe the MTD Cloud API Subsystem for bus data, then will parse, process, and analyze the data to determine information such as bus locations. This information will then be mapped to the LED map, and an LED control signal (GPIO output of the microcontroller) will be sent to the LED Map Subsystem, where bus locations and ID will be shown on the display. The Microcontroller Subsystem will also take in 4 inputs from the Input Subsystem that will be used to control bus ID colors, LED intensity, bus lines, and themes that will be communicated to the LED Map Subsystem via the Microcontroller Subsystem.

1.2.4 Inputs

The Input Subsystem will control the 4 previously discussed inputs that will be communicated to the LED Map via the Microcontroller Subsystem. These inputs consist of three buttons for user input and a photo resistor for environmental input. The user will be able to change the color theme of the LED strip corresponding to the time of year (holidays), change the LED lights' base intensity, and cycle through the different bus routes. The tunable photoresistor circuit will then affect the base intensity of the LED by measuring the ambient light and dim or brighten the circuit based on the light levels around it, dimming in the dark and getting brighter during the day.

1.2.5 LED Map

The LED map will be made up of a few meters of LED strip going both ways, indicating two lanes of traffic. There will be 160 LEDs per meter of strip, but the overall streets will be smaller in length, making sure to save as much power and cost of material as possible by only wiring the roads that have bus routes. The LED map will sit in lots on a custom 3D printed Map of the Champaign campus. 25 tiles of 850x830mm in area. The LED Map will be powered by the External Power Subsystem, which includes a 5V supply and some fuse protection via the Power Distribution Subsystem. The LED Map will also receive a 5V logic control signal from the Microcontroller Subsystem that will indicate which LEDs within the map should be lit up and at what color or intensity.

1.2.6 Cloud API

The Cloud API Subsystem describes the API provided by the MTD Service that we will be utilizing to acquire the necessary data for our project solution. We will request different data for each bus every 60 seconds to obtain the longitude and latitude of each bus. All the requested data will be sent to the Microcontroller Subsystem, where it will be parsed and processed to be mapped to the LED Map Subsystem. For more information about the process of connecting to the API program and parsing the data, refer to Section 2.3.

2 Design

2.1 Design Procedure

The design of this project was organized around the 6 major subsystems described in Section 1.3 with the initial design revolving around the voltage and current specifications of the microcontroller and LED matrix. Each of the subsystems requires comparing several alternative design approaches before selecting the most suitable solution.

We began by choosing the ideal microcontroller for the Microcontroller Subsystem and made further design decisions based on the specifications chosen. For the given project outline, the microcontroller would need to meet computational capability, input/output needs, and communication interfaces for our desired final product. We compared several platforms, including ESP8266, Raspberry Pi, and more. The ESP32-S3-WROOM-1 was ultimately selected for its built in Wi-Fi capabilities, adequate GPIO count, low power draw, and ability to handle the finite state machine (FSM) logic to parse API data. Since the microcontroller outputs the serial control line to the LED strip, the control output would now need to be stepped up from 3.3V to 5V given the specification of the chosen chip. Because the LED strip would require a clean 5V logic high, the generic MOSFET-based level shifter shown in Figure 2.1 was chosen for its simplicity, low cost, and reliability. Other alternatives included a voltage divider or a transistor-based buffer, but the MOSFET implementation was more robust and able to dissipate power which met the subsystem requirements.

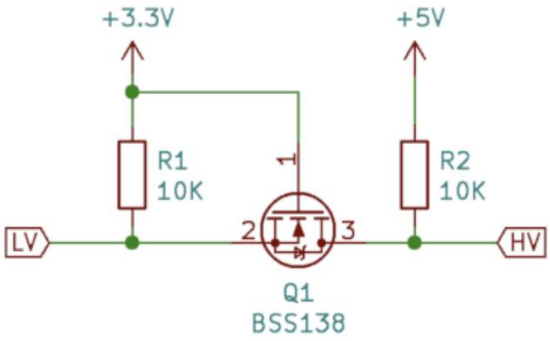


Figure 2.1 - General Circuit for MOSFET Logic Level Converter [1]

For the External Power Subsystem, there was a choice between directly powering microcontroller (3V) or the LED strips (5V). Since it is more efficient and easier to implement a voltage step down, we decided to provide a 5V signal to the system via the External Power Subsystem. Now, the power delivered by the subsystem would have to be

$$P_{max} = N_{LED} \times 30mA \times 5V \quad (1)$$

where N_{LED} is the number of LEDs on and 30mA is the maximum current draw of an LED. We evaluated multiple power-delivery options such as USB-C, computer power, etc. and ultimately decided to use a barrel-jack output due to the high current/power capability, low cost, and ease of implementation.

After choosing to supply a regulated 5V rail from the External Power Subsystem, the Power Distribution Subsystem would need to reliably provide the 3.3V required by the Microcontroller Subsystem while still safely delivering the full 5V to the LED strip. Several options exist for stepping down voltage, such as voltage dividers, linear regulators, and transformers, but a buck convert fit this design the best. Voltage dividers can struggle to provide specific currents and can burn out under load, while linear regulators dissipate significant power to heat, which becomes problematic given the continuous current demand. However, the buck converter provides high efficiency, maintains a regulated voltage under varying load, and minimizes thermal dissipation, making it ideal for a system with an upper current limit of 10A. This ensures the ESP32 receives a stable 3.3V supply without a loss of efficiency, heat, or risk of burnout. In this design, the buck converter would use closed loop control as shown in Figure 2.2 to ensure a steady voltage is provided regardless of the load draw.

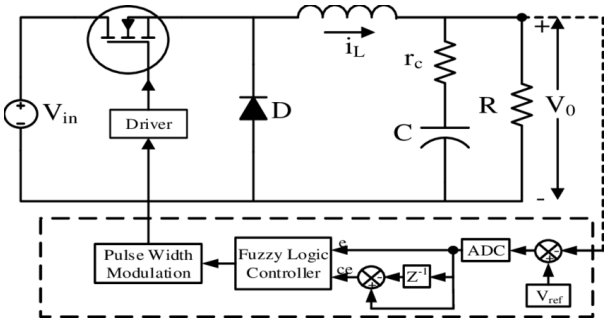


Figure 2.2 - Generic Circuit for Closed Loop Buck Converter [2]

The Input Subsystem required selecting between analog knows, switches, or push buttons for user input to determine theme and route indication. Momentary push buttons were chosen because they provided easy and robust user control and would be simple to digitally debounce. For ambient light sensing, a photoresistor-based voltage divider was selected instead of digital light sensors or photodiodes, as it was low-cost and was sufficiently accurate solution to integrate with the ESP32. The design equations for the photoresistor involve computing the voltage divider output

$$V_{out} = V_{in} x \frac{R_{LDR}}{R_{LDR} + R_{fixed}} \quad (2)$$

where V_{out} is inputted to the ESP32, R_{LDR} is the resistance of the photoresistor, and R_{fixed} is the resistance used to control sensitivity to ambient light.

2.2 Hardware Detailed Design

The following sections will detail the implemented design of each of the subsystems described in Section 1.3 and detail the design choices made via the design procedure described in Section 2.1. To view the entire system schematic, refer to Figure A.1 in Appendix A.

2.2.1 External Power Subsystem

Using equation (1) shown in Section 2.1, the maximum power needed to be delivered by the External Power Subsystem can be calculated to find the specification for the barrel jack connection. If we assume that the maximum number of LEDs occurs when an entire route is lit up and that the longest route is approximately 250 LEDs, then P_{max} = 250 x 0.03A x 5V = 37.5W. To ensure the safety of the system, we overestimated this to be approximately 50W with a current draw of 7.5A. This led to choosing a barrel jack adapter rated as a 5V output with a maximum 10A current draw. As shown in Figure 2.3, the output of this barrel jack was then connected to a resettable fuse to offer a redundant safety feature to protect the connecting subsystems.

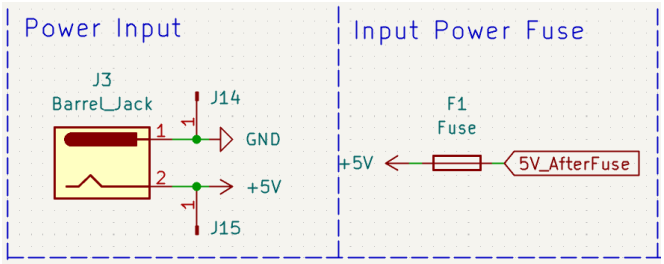


Figure 2.3 – Schematic of External Power Subsystem

2.2.2 Power Distribution Subsystem

As described in Section 2.1, the design choice was used to step the 5V External Power Subsystem voltage to a 3.3V input voltage to the Microcontroller Subsystem using a closed-loop controlled buck converter. The AP62250 synchronous buck converter was chosen due to its low voltage ripple, constant current output, and resistance to input or output noise and load conditions. According to the AP62250 datasheet [3], the buck converter provides a constant 2.5A current and variable output voltage in an input voltage range of 4.2-18V. According to the ESP32 datasheet [4], the microcontroller will take a maximum current draw of 250mA with every GPIO and API pin in use. This means that the buck converter will provide more than sufficient current and voltage to safely power the microcontroller and avoid burnout. To ensure the output voltage of the Power Distribution Subsystem meets the ripple voltage requirement in Appendix D Table D.2, values of the output capacitors need to be large enough such that the equation $V_{OUT,ripple} = \Delta I_L(ESR + \frac{1}{8*f_{sw}*C_{out}})$ is satisfied. Two 22µF ceramic capacitors were selected as the output capacitors (as shown in Figure 2.4) for their low Equivalent Series Resistance (ESR) and low output voltage ripple. Furthermore Figure B.2 in Appendix B, shows a 20mV output voltage and 1A current ripple under a 2.5A load and 3.3V output in simulation with this output capacitance to demonstrate meeting these requirements. The values of R17 and R18 were calculated using the equation $R_{17} = R_{18} * (\frac{V_{out}}{0.8V} - 1)$ and selecting values that satisfied this equation while maintaining a balance between overall efficiency and accuracy of the output voltage. Figure B.1 in Appendix B shows the Constant On-Time (COT) control used in the AP62250 that allows for quick transient response, stable loop performance, and low V_{out} ripple regardless of the load conditions which meets our desired design described in Section 2.1.

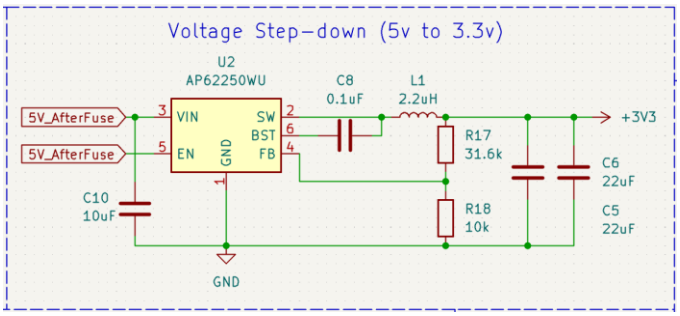
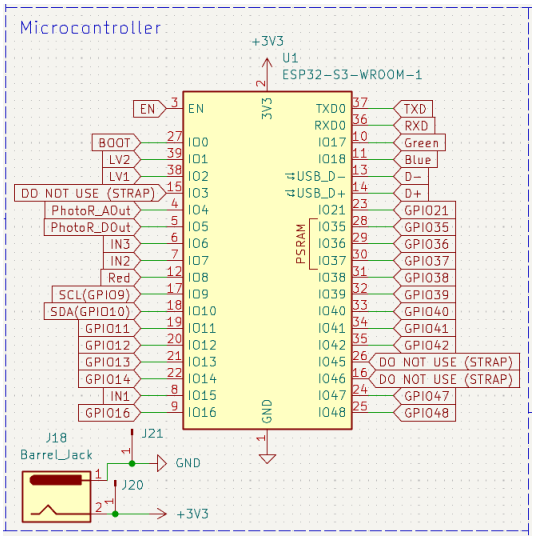


Figure 2.4 – Power Distribution Subsystem Buck Converter

2.2.3 Microcontroller Subsystem



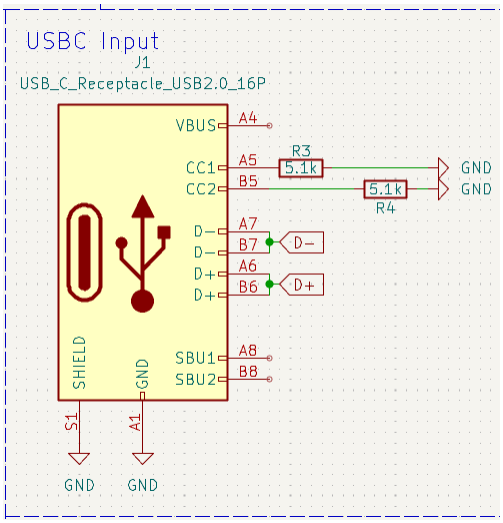


Figure 2.5 – Microcontroller Subsystems

Figure 2.6 – USBC Input Subsystem

2.2.4 Input Subsystem

As described in Section 1.3.4, the input subsystem consists of physical pushbuttons that will be used for user input as well as an ambient light control that will adjust the brightness of the LED Map Subsystem depending on the lighting conditions of the display. The photoresistor input schematic, shown in Figure 2.7, was modeled after the ST0250 Photoresistor Module [5] designed for use with an Arduino. In this design, the photoresistor responds to the ambient light level and alters the (+) input to the LM393 comparator via voltage divider while the (-) input is set via a potentiometer. The circuit then outputs an analog and digital value that is inputted to the ESP32 (via AOut and DOut in Figure 2.7) to be used by the Microcontroller Subsystem to control the LED intensity. The analog value allows the LEDs to gradually change with respect to the current conditions while the digital value allows the microcontroller to control if the LEDs turn on or off depending on the threshold set by the potentiometer. In our design, the ESP32 will mainly use the analog value to increase light intensity when the room is bright and decrease when it is dark. Furthermore, there are three large push buttons located on the display that are used to control the theme of the board, cycle through available routes, and adjust the brightness of the LEDs. These inputs are shown in Figure A.1 in Appendix A and are implemented via a push-button

and pull-down resistor to avoid any floating voltages. Furthermore, the inputs are digitally debounced within the Microcontroller Subsystem to improve the user experience and avoid any unexpected behavior.

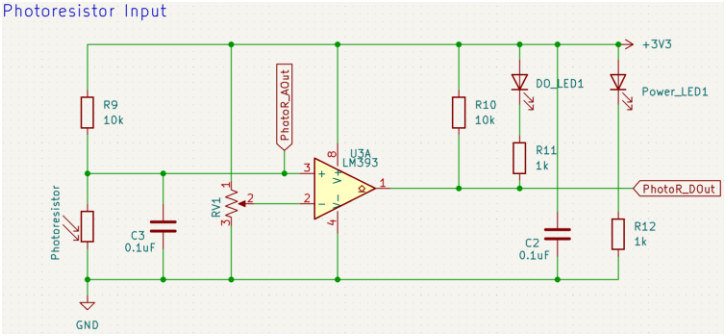


Figure 2.7 – Ambient Light Sensitivity Circuit

2.2.5 LED Map Subsystem

The LED map subsystem is a single LED strip that follows all of the designated roads for the bus routes. With how the LEDs are made the data line must be wired in series for each and every LED. We implemented the system was wiring every road individually, each road has between one to three points where 5V and ground are connected to the strip for redundancy. Then very small wires are jumped from the data out of one road to the data in of the next. Some special cases where there are multiple intersections or the end of a road, we wire the data line through the back side of the board with small drilled holes to the next section of the map. Shown in figure 2.8 the led strips are wired with some LEDs turned on to represent the buses currently on campus.

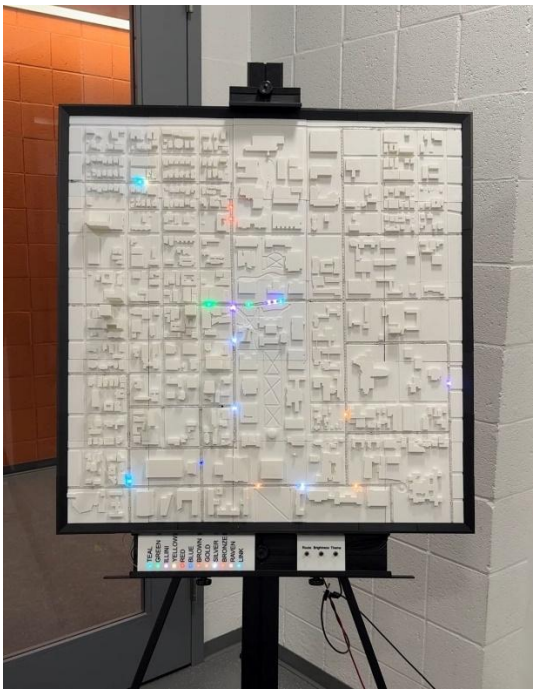


Figure 2.8 – 3D printed map with led strips throughout

2.3 Software Detailed Design

The software architecture of the MTD Bus Tracker is made up of four major components: WIFI connectivity, API fetching, GPS-to-LED coordinate mapping, and user interface management. The system operates on a finite state machine model that continuously loops through data fetching, processing, and displaying updates while responding to user inputs.

2.3.1 System Overview

The system runs using the Arduino framework and operates in three main modes: Bus Data Mode, Menu Mode, and Route Display Mode. Appendix C shows the complete logic flow; the main loop executes every 10ms and coordinates between subsystems while the API fetch cycle operates on a 10 second interval (can be changed, depending on user need).

2.3.2 Wi-Fi Connectivity and API Integration

The system establishes a Wi-Fi connection during initialization using the ESP32's built-in Wi-Fi module, with connection attempts limited to 50 iterations (25 seconds maximum) before timeout. Once connected, the system maintains an HTTPS connection to the MTD API endpoint using the WiFiClientSecure and HTTPClient libraries.

The fetchBusData() function executes every 60 seconds, initiating an HTTPS GET request to https://developer.mtd.org/api/v2.2/json/GetVehicles?key=[API_KEY]. The API returns a JSON payload (typically 15-25KB) containing real-time location data for all active buses. This response is parsed using the ArduinoJson library with 64KB dynamic memory allocation. To extract the necessary data, the system iterates through each vehicle object in the JSON array:

```
for (JsonObject vehicle : doc["vehicles"].as<JsonArray>()) {
  const char* route_id = vehicle["trip"]["route_id"];
  double lat = vehicle["location"]["lat"];
  double lon = vehicle["location"]["lon"];
}
```

Figure 2.9 - JSON parsing structure for extracting route IDs and GPS coordinates from MTD API response.

A complete sample API response structure and additional MTD API methods used in development are documented in Appendices E.3 and E.2.

2.3.3 GPS-to-LED Coordinate Mapping Algorithm

The main function of this system is the GPS-to-LED coordinate transformation, which converts real-world geographic coordinates into discrete LED positions on the physical display. This is accomplished through a two-step process: segment identification and linear interpolation.

The system maintains a vector of 16 GPStoLED structures, each defining a street segment with GPS bounding boxes:

```
struct GPStoLED {
const char* segment_name;
std::vector<LEDRange> ranges; // LED positions
double minLon, maxLon; // GPS boundaries
double minLat, maxLat;
bool reverse; // Direction flag
};
```

Figure 2.10 - GPStoLED struct defining street segment GPS boundaries and corresponding LED ranges

When a bus's coordinates are received from the API, the calculateLEDPosition() function (shown in Appendix F) iterates through all segments to find a GPS bounding box that contains the bus location by verifying that both latitude and longitude fall within their respective min/max bounds.

Once the correct segment is identified, the bus's position within that segment is normalized to a 0.0-1.0 range using linear interpolation. The algorithm determines whether the street runs predominantly East-West or North-South by comparing GPS bounding box dimensions. For East-West streets, longitude is used as the primary coordinate:

$$t_{normalized} = \frac{lon - lon_{\min}}{lon_{max} - lon_{min}} \tag{3}$$

For North-South streets, latitude is used instead. The normalized value is then mapped to the actual LED position:

$$LED_{position} = LED_{start} + (t * total\_LEDs_{in\_segment}) \tag{4}$$

For segments with multiple discontinuous LED ranges (such as intersections where the LED strip jumps to another part of the physical map), the function distributes the normalized position proportionally across all ranges. The reverse flag allows for LED ordering opposite to GPS progression when the physical LED strip runs counter to the geographic direction of travel.

2.3.4 User Interface and Input Handling

The software implements a debounced button handling system with both quick-press and long-press detection. Button states are sampled every 10ms with a 50ms debounce delay to filter mechanical noise. A 2-second hold threshold triggers menu mode entry or selection confirmation.

The photoresistor input is processed through a 10-sample moving average filter to smooth ambient light readings. The analog value (0-4095 on ESP32's 12-bit ADC) is mapped to LED brightness (1-255) using a piecewise linear function optimized for human perception of brightness changes

3. Design Verification

Each component was first tested independently to make sure each requirement was hit then when ready we combined different systems and made sure that all components functioned correctly together.

3.1 Power System

As discussed in the R&V table in appendix D the power system of out project has two main parts: the five-volt input being regulated from a wall power supply, and a 5V to 3.3 V buck converter to power the microcontroller system and its peripherals.

3.1.1 Five Volt input

While powered on we measured the 5V input voltage at no load and at load and for both cases our requirements for voltage ripple were satisfied. See Figure 3.1 for voltage input. At our worst case of 4 amps of current draw from the lab bench supply our input voltage 5V did not deviate or drop.

Figure 3.1 – 5V Output from Wall Supply

3.1.2 5V-3V3 buck converter

Once we verified that the 5V input met our requirements, we probe the 3V3 terminal and measure the mean voltage as shown in Figure 3.2. At our worst case of 4 amps of current draw from the lab bench supply our buck converter voltage of 3V3 did not deviate or drop allowing sustained power for all the esp32 modules.

Figure 3.2 – 3V3 output of the buck converter

3.2 ESP32-S3

As discussed in the R&V table in Appendix D the ESP32-S3 must be able to be programmed and connected to the MTD bus API and a hotspot while also allowing inputs and outputs through buttons, light sensors, and the led strip data pins.

3.2.1 Programming

As shown in Figure 3.3 the ESP32S3 connects to a COM port through the Arduino IDE and is able to be flashed and programmed.

```
Writing at 0x000c6cd9... (88 %)
Writing at 0x000cf36e... (91 %)
Writing at 0x000d7c1d... (94 %)
Writing at 0x000dcf72... (97 %)
Writing at 0x000e260d... (100 %)
Wrote 877856 bytes (567629 compressed) at 0x00010000 in 6.5 seconds (effective 1076.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Figure 3.3 – ESP32-S3 successfully programmed

3.2.2 Input/Output control

Once we were able to program the ESP32-S3 we wrote some quick code tests to verify that pins could read and write digital values. As shown in Figure 3.4 the current code on the board reads the different button inputs.

```
theme button pressed
3
Switched to theme: 3
buzzing
brightness button pressed
Brightness mode: MEDIUM-LOW
buzzing
Short Press
```

Figure 3.4 – Button press, serial monitor

3.3 Inputs

As discussed in section 3.2.2 the ESP32 has a few inputs and outputs that are considered when controlling our map. There are three button inputs, theme, route, and brightness, and an environmental input being our light sensor.

3.3.1 Buttons

As discussed in section 3.2.2 each button press gets output to the serial monitor for debugging and as shown in Figure 3.4 each button press gets read.

3.3.2 Photo Resistor

As discussed in section 3.2.2 there is one environmental input being our photo sensor which we read as an analog input to control the LED strip brightness. As shown in Figure 3.5 we are able to read the analog input of our photo resistor and use this varying signal to map the brightness to a value between 10 and 255.

```
20:56:13.636 -> Photo ADC: 1468 -> Brightness: 255
20:56:13.675 -> Photo ADC: 1481 -> Brightness: 255
20:56:13.710 -> Photo ADC: 1585 -> Brightness: 242
20:56:13.746 -> Photo ADC: 1727 -> Brightness: 221
20:56:13.781 -> Photo ADC: 1881 -> Brightness: 198
20:56:13.821 -> Photo ADC: 2058 -> Brightness: 172
20:56:13.857 -> Photo ADC: 2266 -> Brightness: 142
20:56:13.895 -> Photo ADC: 2483 -> Brightness: 117
20:56:13.931 -> Photo ADC: 2705 -> Brightness: 91
20:56:13.965 -> Photo ADC: 2929 -> Brightness: 69
20:56:14.006 -> Photo ADC: 3155 -> Brightness: 51
20:56:14.041 -> Photo ADC: 3369 -> Brightness: 33
20:56:14.080 -> Photo ADC: 3495 -> Brightness: 26
20:56:14.117 -> Photo ADC: 3584 -> Brightness: 22
20:56:14.154 -> Photo ADC: 3662 -> Brightness: 19
20:56:14.192 -> Photo ADC: 3718 -> Brightness: 17
20:56:14.226 -> Photo ADC: 3743 -> Brightness: 16
20:56:14.264 -> Photo ADC: 3760 -> Brightness: 15
```

Figure 3.5 – Analog read of the photo sensor and mapping to 0-255 variable for brightness

3.4 Outputs

The system only has two outputs, and they are both for LED strips. The sections below will describe the verification and testing of the signals controlling the output.

3.4.1 LED strip data

The LED outputs are from the ESP32S-3, the outputs of the ESP32 data pins are driven at 3V3 volts and our LED strip are rated for 5v meaning we need to step up our data signal to this new voltage. To do this we use a simple Mosfet with the esp32 data pin as our gate driver and 5v as the switching signal. As shown in Figure 3.6 the WS2812B data protocol is shown with the correct voltage for the LED strips.

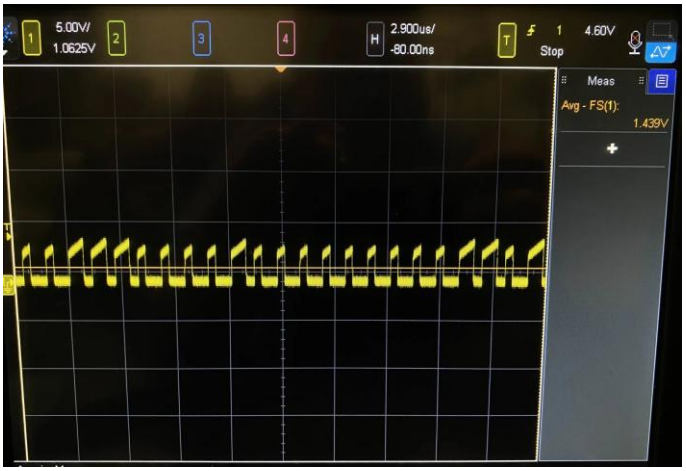


Figure 3.6 – WS2812B data protocol at 5v

3.5 Cloud API

3.5.1 Wi-Fi Connection

The ESP32-S3 wireless module was tested for its ability to establish a reliable WIFI connection during system initialization. The ESP32 attempts to connect to the configured network with a timeout threshold of 50 attempts at 500ms intervals, allowing a maximum of 25 seconds before failure. As shown in Figure 3.7, the system successfully establishes WIFI connectivity and is assigned IP address 192.168.137.252.

```
=== MTD Bus Tracker Initializing ===
GPS Mappings: 16
Total LEDs: 874
..
WiFi Connected!
IP: 192.168.137.252
```

Figure 3.7 - Serial Output for WIFI Connection Verification

3.5.2 MTD API Connection

Once Wi-Fi connectivity is established, the system initiates an HTTPS connection to the MTD API endpoint at *https://developer.mtd.org/api/v2.2/json/GetVehicles.* As shown in Figure 3.8, the HTTP status code 200 confirms that the MTD server successfully accepted our API request and returned valid data. The 200-status code is the standard HTTP success response, indicating proper authentication with our API key and successful data retrieval. The API returned a 15,029-byte JSON payload containing real-time location data for 33 active buses across the Champaign-Urbana transit network.

```
=== Fetching Bus Data ===
Free heap: 272840 bytes
HTTP Response Code: 200
API Response received, length: 15029 bytes
JSON parsed successfully!
Route: ILLINI | LED: 826
Route: TEAL | LED: 74
Route: YELLOW | LED: 21
Route: ILLINI | LED: 366
Route: SILVER | LED: 807
Route: SILVER | LED: 827
Route: ILLINI | LED: 259
Route: ILLINI | LED: 366

=== Summary ===
Total buses: 33
On tracked routes: 8
ESP32-S3 MAC Address: 70:04:1D:AF:89:A0
```

Figure 3.8 - Serial output for MTD API verification

In this example, the available heap memory of 272,840 bytes shows that the ESP32 has enough resources to handle the large JSON payload without memory allocation of failures or system instability. The system successfully parsed this JSON structure using the ArduinoJson library with a 65,536-byte (~64KB) dynamic memory allocation, extracting route IDs and GPS coordinates for each vehicle. Of the 33 total buses in response, 8 were identified as traveling on our mapped street segments. Additional verification methods using Insomnia for API testing are documented in Appendix E.2 demonstrating that the MTD API endpoint responds consistently with the expected JSON structure and data format.

4. Costs

4.1 Parts

Table 4.1 Parts Costs

Part	Manufacturer	Quantity	Retail Cost ($)	Total Cost ($)
Barrel Connector Jack	Same Sky	4	$1.94	$8.54
10k Potentiometer	Vishay Sfernice	4	$1.42	$5.68
A113321	TE Connectivity AMP	2	$1.05	$4.20
BSS138CT	ONSEMI	4	$0.26	$2.70
AMS1117-3.3CT	Ever Semiconductor Co	9	$0.24	$2.16
LM393NNS	Texas Instruments	2	$0.74	$2.96
ESP32-S3-WROOM-1	LE Plus Plus	4	$4.95	$21.78
USB4216-03-ACT	GCT EMEA LTD	4	$0.56	$2.64
0ZCF00500FF2ACT	Bel Fuse/Bel Power	8	$1.44	$11.52
16.4FT WS2812B LED	Xnbada	1	$42.99	$42.99
PLA Plus Filament	ELEGOO	1	$28.69	$28.69
5V 10A Power Supply	ALITOVE	1	$20.63	$20.63
Total				**$154.49**

4.2 Labor

Table 4.2 Labor Costs

Team Member	$/hr	Hours/week	$/week	Weeks Worked	Total Cost
Daniel Vlassov	$42	6	$252	14	$3528
Amber Wilt	$42	6	$252	14	$3528
Ziad AlDohaim	$42	6	$252	14	$3528
Machine Shop	$50	2	$100	1	$100
Total Labor Costs:					**$10,684**

4.3 Schedule

Table 4.3 Semester Schedule

Week	Task	Person
Sep 29 - Oct 5	Order Parts for Breadboard Demo	Everyone
	Start PCB Design 1	Amber
	Get API calls and test	Ziad

	Start CAD Modeling	Daniel
Oct 06 - Oct 12	Finish PCB Design 2	Amber
	Finish Breadboard Demo Set Up Components on Breadboard	Everyone
	Code and flash ESP32	Daniel & Ziad
	Work on CAD Modeling	Daniel
Oct 13 - Oct 19	Finish ECE445 Design Document	Everyone
	Revision for Machine Shop (For Stand)	Everyone
Oct 20 - Oct 26	Finish Breadboard Demo 2	Everyone
	Solder and Test PCB (when it arrives)	Amber & Daniel
Oct 27- Nov 2	Modify PCB for 3rd round	Amber & Daniel
	Finish printing 3D Model	Daniel
	Get Machine Shop Stand	Ziad
Nov 3 - Nov 9	PCB Round 3	Amber & Daniel
	Finalize Code	Ziad
	Assemble Display & LEDs	Everyone
Nov 10 - Nov 23	Mock Demos	Everyone
Dec 1 - Dec 11	Final Demos & Presentation	Everyone

5. Conclusion

5.1 Accomplishments

Throughout the semester, our team was able to successfully develop a solution to the previously described problems and meet the high-level requirements detailed at the start of the project. Furthermore, we were able to evaluate and verify all the requirements described in Appendix D along with a fully functional PCB, shown in Appendix G. In the end, the project won an award and, in all, did not leave out or do not include any goals that had been set. Additionally, we were able to decrease the MTD API polling time to 10 seconds, allowing us to achieve even greater accuracy (within 10 seconds) of bus locations for use. In all, the project accurately mapped real-time bus locations to a map of the campus area around UIUC that was easily accessible and understandable for the average viewer.

5.2 Uncertainties

The only unsatisfactory component of this project was the low-quality LED strips used in the display. Due to budget restrictions, we were limited in the amount and quality of LEDs we could order. This meant that we could only display the buses going one direction in the streets and there were a handful of LEDs that would enter a "stuck-on" mode where they would not turn on and became uncontrollable. The solution to these problems would be to add additional costs to purchase higher quality and number of LEDs strips to mitigate both of these issues and expand upon the design. An additional uncertainty about this project would be how to continue to improve upon and expand the design that was created. Ideally, if this was to be implemented throughout campus, there would need to be a larger amount of API pulls that would be coordinated with MTD and the maps would have to be much larger. Both of these would require man hours and significantly more costs which would bring rise to the tradeoffs of continuing to build upon and implement a more widespread solution to this issue.

5.3 Ethical considerations

5.3.1 Safety Concerns

5.3.1.1 Electrical Safety

The LED matrix and ESP32 microcontroller system require a power supply capable of handling significant current. Risks include overheating, short circuits, or electric shock. Compliance with Underwriters Laboratories and IEC electrical safety standards is required. Proper fuses, insulated wiring, and thermal management is integrated properly.

5.3.1.2 Fire Hazard

High current draw in LED matrices can generate heat. Overloaded circuits could present a fire hazard. Following NFPA (National Fire Protection Association) guidelines on electronic equipment and ensuring circuit breakers and buck converters are properly rated will mitigate this.

5.3.2 Ethical Concerns

5.3.2.1 Fairness and Accessibility

The ACM Code of Ethics [2] emphasizes avoiding harm and ensuring equal access to technology. To address this, the design should adhere to ADA-compliant visibility and brightness standards brightness, and light patterns are managed properly to reduce the risk of triggering photosensitive epilepsy.

5.3.2.2 Data Privacy and Security

We will be using public API data; however, there are some constraints on its usage and how often the API should be called. To avoid misuse, we will never collect or store user-specific data and will abide by the rules set by the MTD Bus API.

5.4 Mitigation of Ethical and Safety Risks

To address the ethical and safety risks described above and ensure the project meets the regulatory standards, we implemented a few design decisions to mitigate these concerns. We implemented redundant checks for API accuracy and display error messages on serial if the data has been deemed unreliable. This will allow us to ensure the reliability of the information being provided to the user. We have used brightness control and energy-efficient components to reduce the environmental impact and power consumption of our devices. We will follow UL/IEC standards [3] for wiring, insulation, and overcurrent protection to ensure the safety of the display. We will prevent data misuse by never storing or transmitting personal location information and limiting the data we access from the API.

5.5 Future work

There are several improvements that could improve our project. The current implementation could be extended to cover a bigger portion of the map, as well as handle additional buses and specific weekend routes (e.g. Illini Express, night/weekend routes). This would allow students to more effectively plan their trips dependent on campus events of time of day. Another interesting addition would be real-time bus capacity information to show how crowded each bus is, this could be indicated by the intensity of the LED, or color gradients as another “brightness” mode. Finally, adding an OLED display to show the estimated arrival times for a few buses would be very useful to aid students in easily viewing what buses are coming to the location they are at.

References

[1] “MOSFET Level Shift.” PenguinTutor. Accessed: Dec. 10, 2025. [Online]. Available: https://www.penguintutor.com/electronics/mosfet-levelshift

[2] “FLC-based closed-loop control of buck converter.” ResearchGate. Accessed: Dec. 10, 2025. [Online]. Available: https://www.researchgate.net/figure/FLC-based-closed-loop-control-of-buck-converter_fig2_324532004

[3] Diodes Inc., “AP62250 datasheet.” Accessed: Dec. 10, 2025. [Online]. Available: https://www.diodes.com/datasheet/download/AP62250.pdfq

[4] Espressif Systems, “ESP32-S3-WROOM-1 & ESP32-S3-WROOM-1U Datasheet,” ver. 1.6, Oct. 29, 2021. Accessed: Dec. 10, 2025. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf

[5] STMicroelectronics, “ST0250 datasheet,” Accessed: Dec. 10, 2025. [Online]. Available: https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/6649/ST0250%20datasheet.pdf

[6] [1] IEEE Policy 7.8. *IEEE Code of Ethics*. IEEE. https://www.ieee.org/about/corporate/governance/p7-8.

[7] ACM. *ACM Code of Ethics and Professional Conduct*. Association for Computing Machinery, 2018. https://www.acm.org/binaries/content/assets/about/acm-code-of-ethics-and-professional-conduct.pdf.

[8] “Developer Resources.” Champaign–Urbana Mass Transit District (CUMTD). Accessed: Dec. 10, 2025. [Online]. Available: https://developer.cumtd.com/

Appendix A Complete Project Schematic

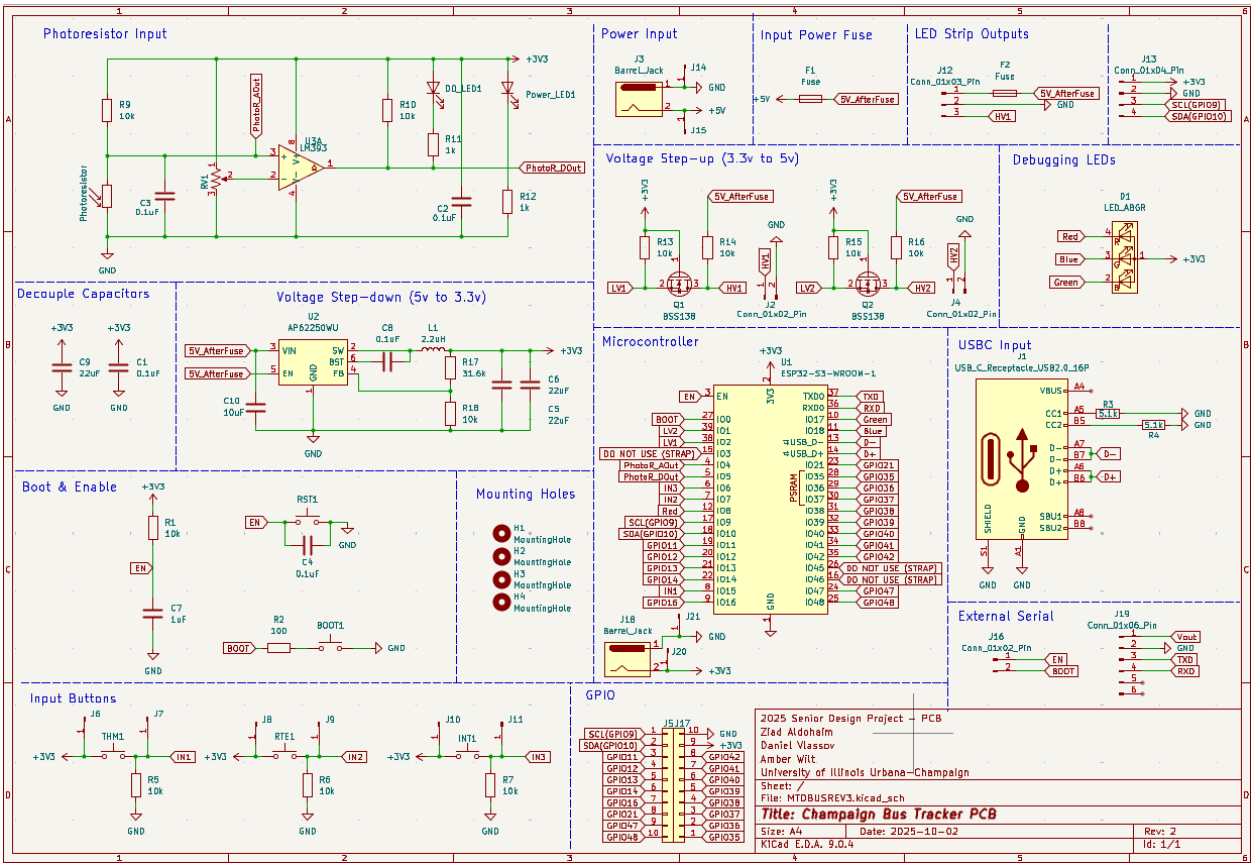


Figure A.1 – Comprehensive Project Schematic

Appendix B Datasheet Figures

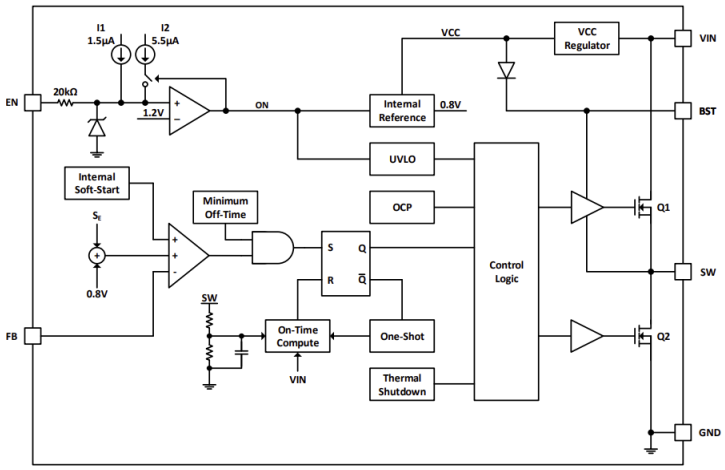


Figure B.1 – AP2550 Buck Converter Schematic [3]

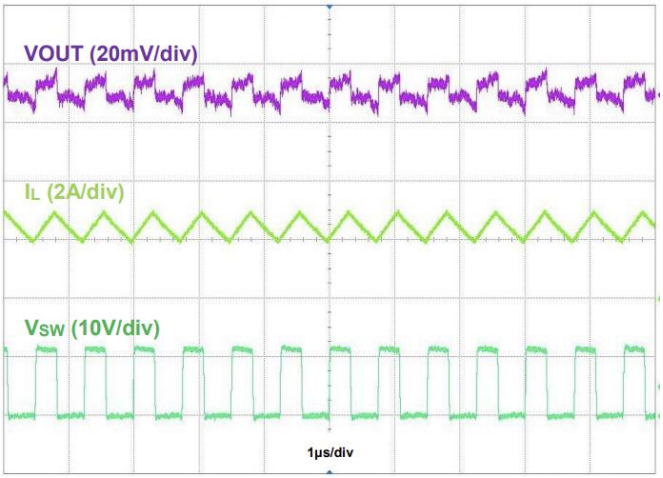


Figure B.2 - AP2550 V_{OUT}, I_L, and V_{SW} Simulation at 3.3V Output [3]

Appendix C Software Flowcharts

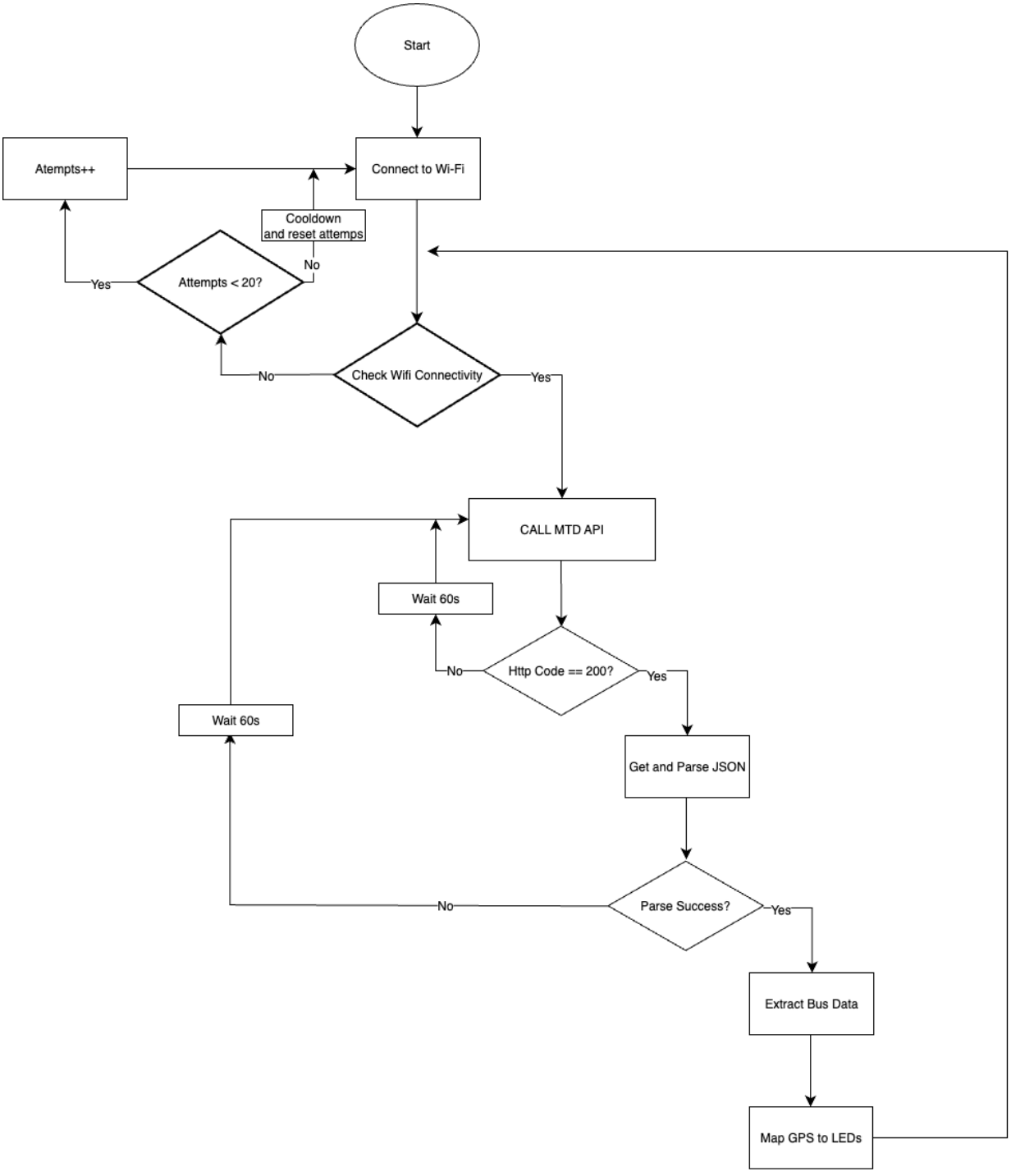


Figure C.1 – Flowchart of Logic for Connecting ESP32 to WIFI, Extracting API Data, and Mapping to LEDs

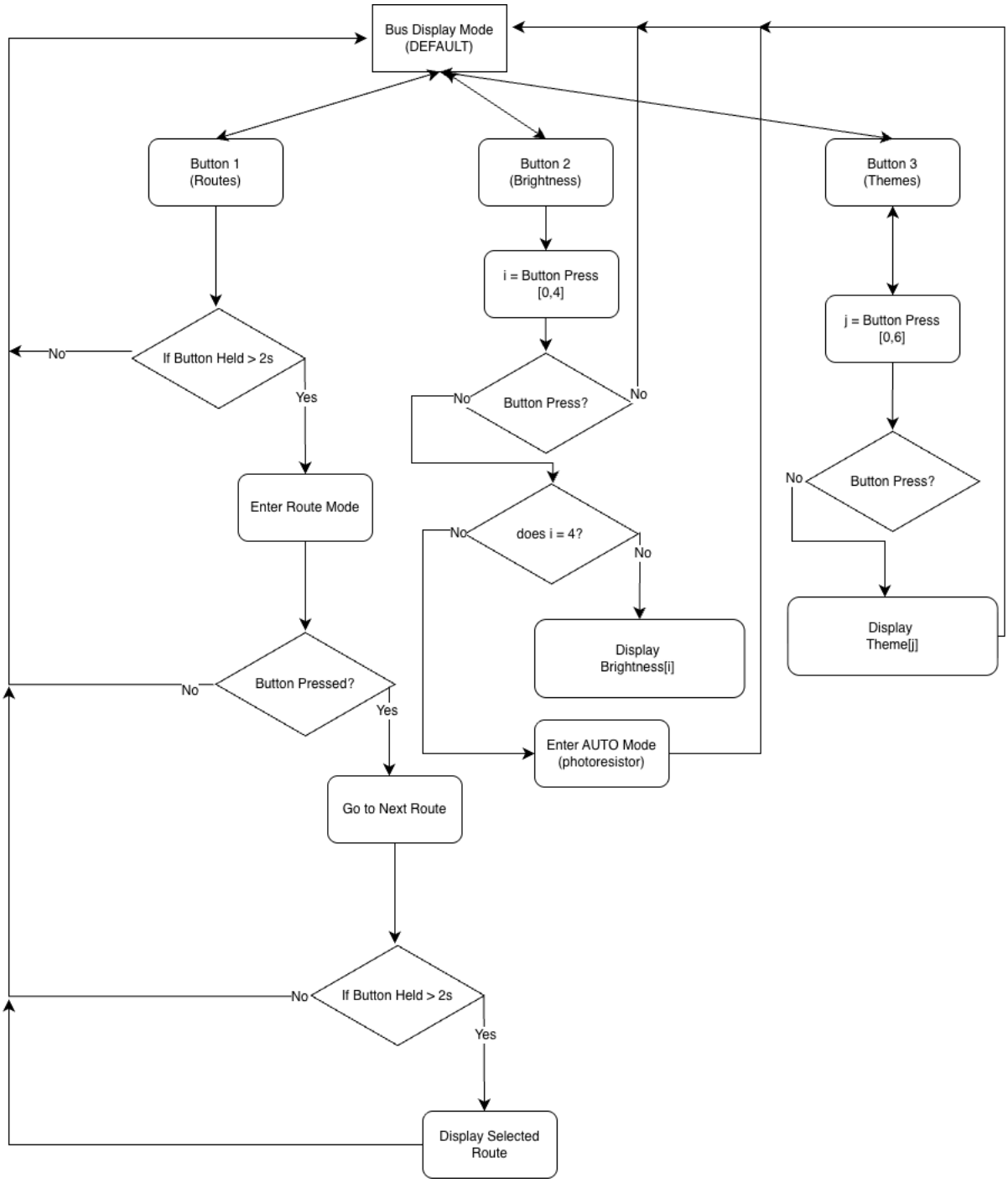


Figure C.2 – Flowchart of Logic for different modes

Appendix D Requirement and Verification Table

Table D.1 External Power Subsystem

Requirements	Verification	Verification Status (Y/N)
1.1 The external power supply must output a voltage between 4.75V and 5.25V DC under a load of up to 2A.	• **Test**: Connect the power supply to the DMM and set the programmable load to draw 2A. Measure the voltage at the output terminals. • **Demonstration**: Table with current vs. measured voltage values.	**Y**
1.2 The power supply must be capable of supplying at least 2.5A of current continuously without thermal shutdown or voltage drop below 4.75V.	• **Test**: Connect a 2.5A load and operate for 10 minutes. Monitor output voltage and temperature. • **Demonstration**: Voltage and temperature log vs. time graph.	**Y**
1.3 The power distribution module must attenuate high-frequency noise such that the output ripple is less than 50mV peak-to-peak.	• **Test**: Measure voltage ripples at the output of the power distribution module under normal load. • **Demonstration**: Oscilloscope screenshot showing ripple waveform.	**Y**
1.4 The system must contain fuses or equivalent protection such that a short in either the microcontroller or LED circuit will not damage the other systems.	• **Test:** Verify functionality and placement of fuses within necessary systems • **Demonstration:** Can short components without connection to rest of circuit and verify protection	**Y**

Table D.2 Power Distribution Subsystem

Requirements	Verification	Verification Status (Y/N)
2.1 The buck converter must step down 5V to 3.3V with a maximum	• **Test**: Connect the output of the DC-DC regulator to the oscilloscope to	**Y**

	output ripple of 100mV peak-to-peak and a load current capacity of at least 1.5A for the ESP32-S3.	measure ripple. Simulate load by drawing up to 1.5A from the output using an electronic load. Measure voltage stability and ripple. • **Demonstration**: Oscilloscope trace showing ripple and voltage level under load.	
2.2	The buck converter should maintain a 3.3V output ± 5% (i.e., between 3.135V and 3.465V) under full load conditions.	• **Test**: Apply a full load (1.5A) to the output of the converter and measure the voltage with the DMM. • **Demonstration**: Table or graph showing the measured voltage at different loads (0A, 0.5A, 1.5A).	**Y**
2.3	The system must provide steady power to the LED strip such that the voltage does not drop below 4.75V even under full load.	• **Test**: Apply a full load to the LED strip (equivalent to 9A) and measure the voltage at the power input terminals. • **Demonstration**: Graph showing voltage drop across the system under different load conditions (0A to 9A).	**Y**
2.4	The total power distribution system (from 5V to 3.3V) should have a total efficiency of at least 85% at full load.	• **Test**: Measure input power and output power, calculate efficiency, and compare to the required efficiency. • **Demonstration**: Table of efficiency calculations at different load levels.	**Y**

Table D.3 Cloud API Subsystem

Requirements		**Verification**	**Verification Status (Y/N)**
3.1	The programmed system must fetch bus data every 60 seconds (±5 seconds tolerance).	• **Test:** Record timestamps of 5 consecutive API fetches from the Serial output. • **Demonstration:** Table of timestamps showing Δt between 55-65 seconds for each interval.	**Y**
3.2	The program must correctly extract vehicle_id, route_id, latitude, and longitude from the JSON response.	• **Test**: Compare Serial output values for 3 vehicles against raw JSON data from the same timestamp. • **Demonstration**: Table comparing extracted values vs. actual JSON values (must match 100%).	**Y**

3.3	The program must be able to differentiate between and track all routes within the scope of the project.	• **Test**: Trigger API fetch when multiple routes are active. Count total vehicles in response vs. tracked buses in the Serial output. • **Demonstration**: Serial log showing "Total vehicles: N" and "Found X target buses" where X ≤ N.	**Y**
3.4	GPS coordinates must map to LED positions with an accuracy of ±2 LEDs within bounds	• **Test:** Record bus latitude from Serial. Manually calculate expected LED position: LED = ((lat - 40.09) / 0.04) × 874 (number of LEDs). Compared to the reported position. • **Demonstration:** Table with 5 test cases showing lat, calculated LED, reported LED, and error (must be ≤2).	**Y**
3.5	The system must detect a WiFi disconnect and retry the connection automatically without crashing.	• **Test:** Disable WiFi while the system is running. Monitor Serial for "WiFi not connected!" Re-enable WiFi and verify reconnection within 20 attempts (10s). • **Demonstration:** Serial log showing disconnection detection, retry attempts, and successful reconnection.	**Y**

Table D.4 Microcontroller Subsystem

Requirements		**Verification**	**Verification Status (Y/N)**
4.1	The ESP32-S3 program must successfully connect to a WiFi network within 5 seconds after boot, with a connection success rate of 95% or higher over 100 trials.	• **Test:** Reset the ESP32 and log connection time over 100 trials. Count the number of successful connections and the average connection time. • **Demonstration:** Table and graph showing success rate and timing distribution.	**Y**
4.2	The ESP32 program must control the LED map to reflect bus location data with a latency of less than 500ms from API response to LED update.	• **Test:** Trigger known API response and measure time to the corresponding LED output change. • **Demonstration:** Timestamped log showing API response and LED update event.	**Y**

Table D.5 Input Subsystem

Requirements	Verification	Verification Status (Y/N)
5.1 The photoresistor circuit must detect ambient light changes and adjust LED brightness in 4 discrete levels within a range of 0–100% PWM duty cycle.	• **Test:** Expose the photoresistor to varying light levels. Observe analog reading and verify corresponding PWM output to LED control pins. • **Demonstration:** Table of light level to PWM level mappings, graph showing correlation.	**Y**
5.2 Each button must register a press within 50ms, including software debounce, and trigger the associated function (theme, brightness, or route cycling).	• **Test:** Press each button and measure the time between physical press (voltage change on GPIO) and software acknowledgment (GPIO toggle or serial print). Repeat 10 times per button. • **Demonstration:** Table showing latency per button press, averaged and max.	**Y**
5.3 Button presses must not trigger multiple unintended events (i.e., no false multiple triggers) with debounce time <50ms.	• **Test:** Hold each button for 1s. Log number of detected press events. Should only count one per press. • **Demonstration:** Table showing number of registered presses per test trial.	**Y**
5.4 The route button must allow the user to cycle through the bus routes, and this must update the LED map accordingly.	• **Test:** Press the route button and verify that the route selection variable changes and the corresponding LEDs update correctly. • **Demonstration:** Display showing route selected vs. visible LED map pattern.	**Y**

Table D.6 LED Map Subsystem

Requirements	Verification	Verification Status (Y/N)
6.1 The LED map should be capable of being controlled to represent real-time bus location, with the bus position updated every 60 seconds or less (as per the FSM).	• **Test:** Simulate bus route data and verify that the LEDs update in real time based on bus location every 10 seconds. Ensure the LED representation matches the bus position on the map. • **Demonstration:** Screenshot/log of bus location vs. LED behavior, showing bus position update.	**Y**
6.2 The brightness adjustment from the photoresistor must update LED intensity with a latency of less than 1 second after an ambient light change.	• **Test:** Change the light level suddenly and measure the time until the PWM signal or LED brightness changes. • **Demonstration:** Table showing latency per light level change.	**Y**

Appendix E API Information

Table E.1 API Modules used for Implementation

API Endpoint	Description	Response	Usage in Project
GetVehicles	Returns the real-time GPS location of all active buses	JSON with vehicle_id, route_id, **lat/lon,** timestamps	**Main data source** - Fetched every 20s to update bus positions on the LED strip
GetRoutes	Returns a list of all bus routes with colors and names	JSON with route_id, route_color, route_name	Used to get official route colors for LED themes and directions
GetShape	Returns GPS path points that define a route's shape	JSON with lat/lon points along the route	Could be used for more accurate LED position mapping

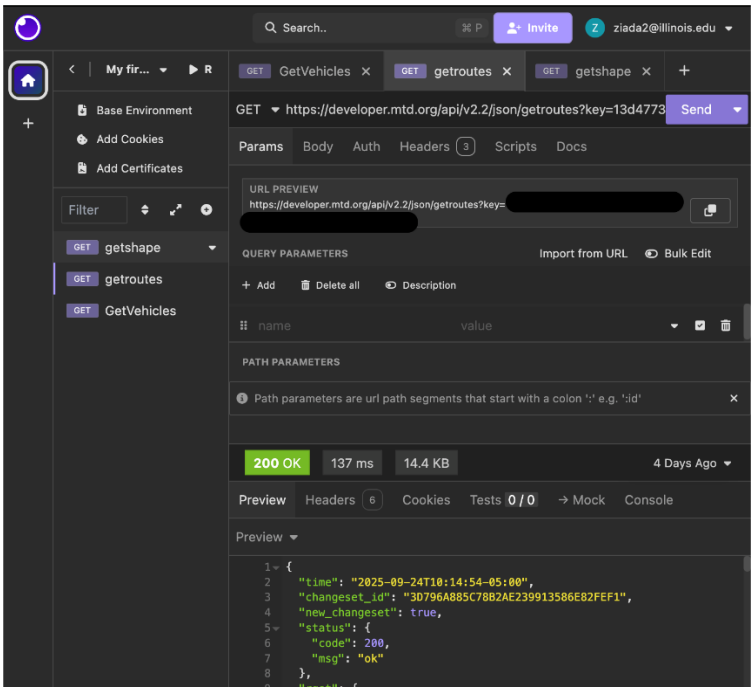


Figure E.2 Verification of Functionality

Response

`vehicle_id`	vehicle number associated with vehicle
`trip`	trip information for the departure
`location`	the last known latitude and longitude of the vehicle
`previous_stop_id`	The last stop that the vehicle served
`next_stop_id`	The next stop that the vehicle will serve
`origin_stop_id`	The stop where the vehicle began it's trip
`destination_stop_id`	The stop where the vehicle will end its trip
`last_updated`	The last time the vehicle sent a real-time location update to our system.

Figure E.3 Response format of GetVehicles() [8]

Appendix F Software Implementation

```
int calculateLEDPosition(double lat, double lon) {
  for (auto& road : gps_mappings) {
    bool inLat = (lat >= road.minLat && lat <= road.maxLat);
    bool inLon = (lon >= road.minLon && lon <= road.maxLon);
    if (!inLat || !inLon)
      continue;

    float tLat = (road.maxLat != road.minLat)
                 ? (lat - road.minLat) / (road.maxLat - road.minLat)
                 : 0;

    float tLon = (road.maxLon != road.minLon)
                 ? (lon - road.minLon) / (road.maxLon - road.minLon)
                 : 0;

    bool horizontal = fabs(road.maxLon - road.minLon) > fabs(road.maxLat - road.minLat);

    float t = horizontal ? tLon : tLat;

    if (t < 0) t = 0;
    if (t > 1) t = 1;

    return ledFromNormalized(road, t);
  }

  return -1;
}
```

Figure F.1 Implementation of LED Position Calculation

Appendix G Image of Final PCB

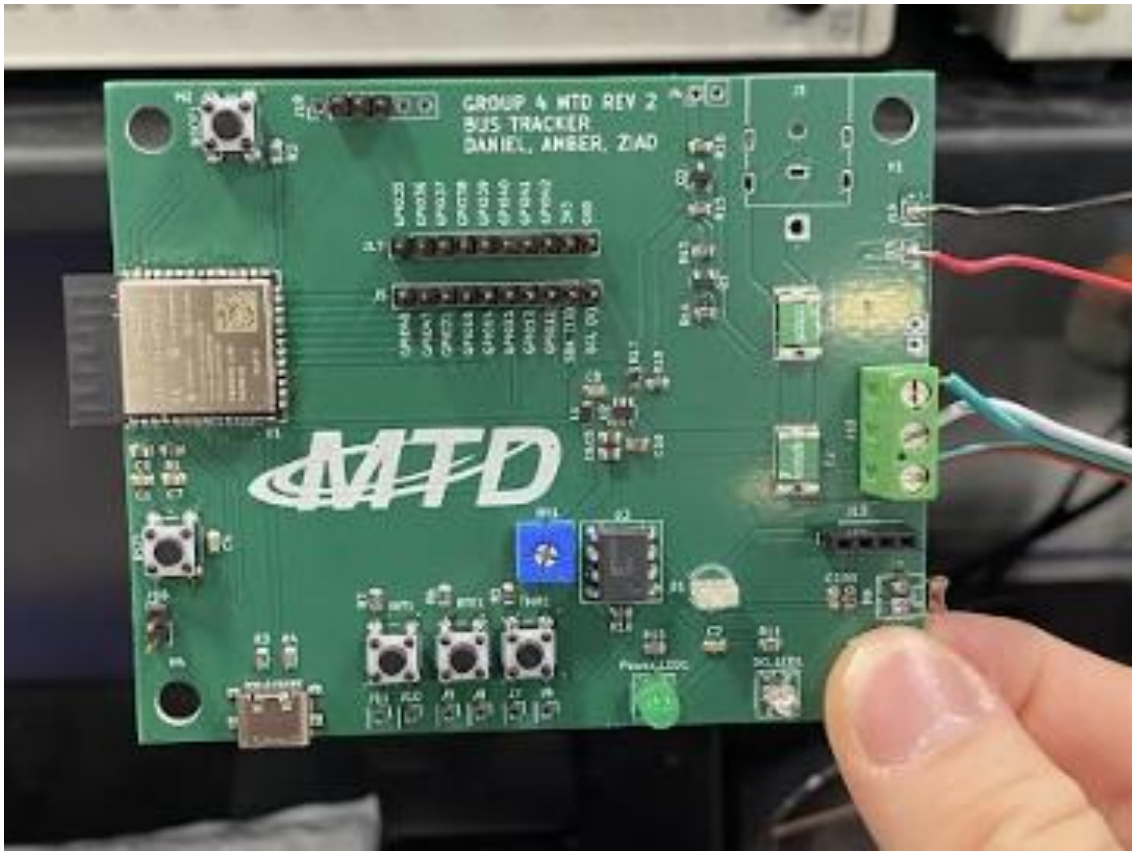


Figure E.1 Final PCB Implementation