# Embedded Senior Project Report

Zak Rowland

Oregon Institute of Technology – Spring 2021


Author Notes:

Submitted in partial fulfillment of the requirements of Embedded Senior Project

(CST 473)

| Revision History | | | |
|---|---|---|---|
| **Revision** | **Date** | **Reason** | **Name** |
| 1.0 | November 19th, 2020 | Published draft | Zak Rowland |
| 1.1 | December 6th, 2020 | Revised draft and published Fall term report | Zak Rowland |
| 2.0 | January 31st, 2021 | Revised Fall term report and published draft | Zak Rowland |
| 2.1 | February 1st, 2021 | Revised Winter term report and published draft | Zak Rowland |
| 2.2 | February 16th, 2021 | Revised Winter term report and published draft | Zak Rowland |
| 2.3 | March 18th, 2021 | Revised Winter term report and published draft | Zak Rowland |
| 3.0 | June 4th, 2021 | Revised Winter term report and published final draft | Zak Rowland |

# Abstract

The project is a Bluetooth diagnostic tool that interfaces with the OBD-II port in a modern vehicle. This system will consist of two major modules: the Bluetooth transceiver module that interfaces with the port, and the handheld module that includes a computer, Bluetooth transceiver, touchscreen, and rechargeable battery. Features of the system will be limited to the essentials to keep operation simple and costs low. The user will be able to read a list of diagnostic (trouble) codes and their descriptions as well as clear the codes. There will be another mode that will read sensor data and display it to the user including speed, oil pressure, and coolant temperature. A simple user interface will utilize touchscreen input for actions like selection, scrolling up and down, and changing modes or settings.

# Table of Contents

# Project Management

## Schedule

Trello is the software used to schedule and manage progress on the project. The board is divided into five sections: Fall, Winter, Spring, In Progress, and Done, and each term has its own color to differentiate tasks in the In Progress and Done lists. If a task has multiple parts, a checklist can be added underneath the main task. The Trello board can be seen in Figure 1 below.

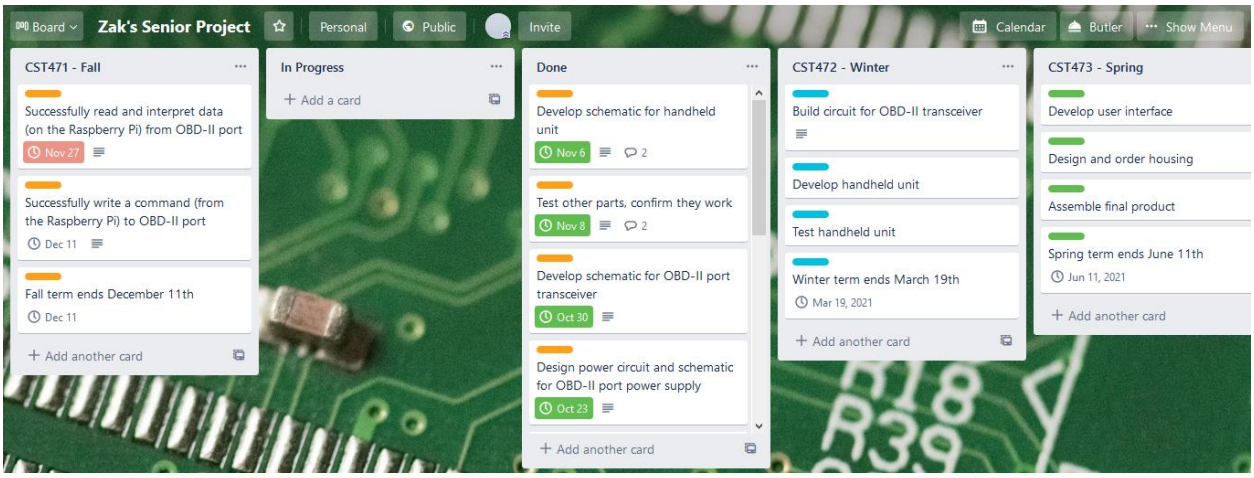



*Figure 1. A screenshot of the Trello board used to track progress on the project [1].*

Each task has a set due date to create a timeline and ensure progress is consistent. As seen in Figure 1 above, tasks marked complete turn green and tasks that are overdue turn red. Although simple, this makes it a bit more rewarding to finish a task and helps signify the urgency of completing overdue ones.

## Estimated Non-Recurring Engineering Costs

| **Description** | **Cost** | **Total** |
|---|---|---|
| Entry level computer engineer salary | $70,000 | 9-months for $52,500 |
| (2) Breadboards | $2.50 each | $5.00 |
| 5V/3A+ USB-C power supply | $10.00 | $10.00 |
| Multimeter (Fluke 117) | $180.00 | $180.00 |
| Oscilloscope (Rigol DS1054Z) | $349.00 | $349.00 |
| Micro HDMI to HDMI cable | $9.00 | $9.00 |
| VSSOP-8 to DIP-8 adapter | $3.19 | $3.19 |
| SOT-223 to DIP-6 adapter | $2.69 | $2.69 |
| 15V/1.5A wall power supply | $1.99 | $1.99 |
| | | **$53,060.87** |

*Table 1. Estimated non-recurring engineering costs.*

## Estimated Production Costs

| **Description** | **Cost** | **Total** |
|---|---|---|
| Raspberry Pi 4 Model B 2GB | $35.00 | $35.00 |
| ELM Electronics ELM327 IC | $16.00 | $16.00 |
| OSOYOO 5” DSI display | $43.88 | $43.88 |
| Samsung 32GB MicroSD card | $7.99 | $7.99 |
| ESP32-DevKitC-32D microcontroller | $9.99 | $9.99 |
| MCP2551 CAN transceiver | $1.09 | $1.09 |
| 4MHz crystal oscillator | $0.50 | $0.50 |
| L7805 regulator | $0.50 | $0.50 |
| 317L regulator | $1.00 | $1.00 |
| TXB0102 logic level converter | $0.71 | $0.71 |
| Male J1962 Type A connector | $2.00 | $2.00 |
| MakerFocus Battery Pack/UPS for Pi | $23.99 | $23.99 |
| Momentary switch with LED | $1.95 | $1.95 |
| MCP3002 ADC | $1.79 | $1.79 |
| Assorted resistors | $3.00 | $3.00 |
| Assorted capacitors | $3.00 | $3.00 |
| Assorted diodes | $2.00 | $2.00 |
| Assorted transistors | $4.00 | $4.00 |
| | | **$158.39** |

*Table 2. Estimated production costs.*

### Change Management Procedures

If a change must happen in the requirements or design of the system, engineering change requests will be submitted to the project supervisor as well as a memo including information on why the change is needed, what the change is, and what the alternatives are.

### Status Reports

The status of the project will be updated to management or the customer with memos, demonstrations, or meetings.

### Skills and Knowledge Areas

This project will require knowledge of many hardware components and protocols. Information about the OBD-II port and the CAN Bus protocol is necessary for communicating with the vehicle. Bluetooth will be used to send and receive data to and from the OBD-II port, so knowledge of the protocol is needed. The handheld module utilizes a touchscreen user interface which means GUI development is required as well. Finally, schematic capture experience is needed to develop schematics for the two modules, including power circuitry. PCB layout and 3D printing experience may also be required.

## Conceptual Overview

### Introduction

Cars and trucks have become so common and inexpensive that almost every licensed driver owns one. Many of these people prefer repairing, maintaining, or

upgrading vehicles on their own, so this project is an open source, handheld Bluetooth OBD-II diagnostic tool for diagnosing and repairing vehicles. The system consists of two primary modules: the OBD-II port module and the handheld touchscreen module, both of which contain Bluetooth transceivers for wireless communication. Costs will be kept low to make the project accessible to as many people as possible. A tool like this is great for enthusiasts, hobbyists, home mechanics, and even professional technicians.

## Problem Statement

There is no standalone, Bluetooth, and handheld system with the required basic features commercially available at a reasonable cost (<$250.) The existing projects that are similar do not utilize a handheld touchscreen device or a custom Bluetooth OBD-II transceiver circuit.

## Intended Audience

This project is targeted primarily towards home mechanics who wish to perform their own automotive repairs. However, paid technicians at a professional shop, or even electronics hobbyists and engineers, could benefit from a low cost and open source tool to interface with a vehicle wirelessly.

## Proposed Project

The project is a Bluetooth OBD-II diagnostic tool for diagnosing and repairing vehicles. The system consists of two primary modules: a Bluetooth transceiver that interfaces with the OBD-II port, and a handheld rechargeable unit that displays information on a touchscreen and receives input from the user. The

handheld unit's default mode will be scanning for and clearing diagnostic (trouble) codes. However, there will be an option to switch modes and read data such as speed, coolant, temperature, and oil pressure.

## Relation of Proposed System to Existing Systems

While commercial products similar to this do already exist, there is no reasonably priced solution (<$250) that provides the required features, such as having a rechargeable handheld unit. The solutions that do exist, like the one pictured in Figure 2 below, can easily cost over $500, and are bloated with unnecessary features.


*Figure 2. An example of an existing solution that is high cost from Amazon [2].*

These products are targeted more towards professional mechanics and technicians who work on multiple vehicles a day, making the investment justified. The low-cost options, like the one pictured in Figure 3 below, connect to a proprietary phone app or Windows computer which does not meet the requirements of having a standalone handheld device.


*Figure 3. An example of an existing solution that is low cost from Amazon [3].*

Many people do not like installing apps on their phone from small, unknown companies. Phones may also be needed for other things while working on a vehicle such as a making phone calls or texts, playing music, looking up information and manuals, or even using the flash for a flashlight. Another problem with these cheap products is the high probability that the OBD-II interpreter integrated circuit used is a clone of the real one. The cloned chips are prone to malfunctions and sometimes have a limited command set compared to the genuine chips. This project is different from these existing commercial solutions because it will have a dedicated touchscreen handheld device that is rechargeable, a genuine OBD-II interpreter integrated circuit from the manufacturer will be used, all documentation and code will be open source, and costs will be kept as low as possible while retaining the necessary functionality.

There are projects completed by other students and engineers that are similar to this one. One example, called “OBD-Pi,” was done in 2014. However, this

solution used a prebuilt OBD-II Bluetooth reader and an aftermarket stereo head unit to display the data. [4] The proposed project will include a custom OBD-II interfacing circuit and a handheld touchscreen device. Another example is a project made for a 1997 BMW. Like this first example, this solution uses a prebuilt OBD-II reader that also utilizes a different interpreter chip (STN1110) [5]. A Raspberry Pi is also used in this project, but the touchscreen interface and the Pi itself are embedded permanently into the dash of the car.

The proposed solution differs from the existing projects because the OBD-II circuit will be designed from scratch using a genuine ELM327 IC, the touchscreen interface is portable and rechargeable, and all documentation and code will be open source. A comparison of the different solutions can be seen in Table 3 below.

| **Name** | Autel MaxiCOM MK808BT [2] | Friencity Bluetooth Car OBD ii 2 OBD2 Scanner Adapter [3] | M3 Pi [5] | This project |
|---|---|---|---|---|
| **Cost** | $560.00 | $9.99 | $130.00 | $160.00 |
| **Bluetooth** | Yes, Wi-Fi for setup | Yes | Yes | Yes |
| **Touchscreen Handheld** | Yes | No | No | Yes |
| **Features** | • Over 25 functions and services<br>• Works with all OBD-II protocols<br>• Includes a USB cable for optional wired mode | • Reads and clears diagnostic codes<br>• Displays real-time data including engine RPM, coolant temperature, etc.<br>• Inexpensive | • Displays data including airflow sensors, load percentage, throttle percentage, current gear, speed, and temperatures | • Reads and clears diagnostic trouble codes<br>• Displays data including speed, temperatures, and pressures |
| **Cons** | • Expensive<br>• Wi-Fi setup<br>• Bloated with features | • Functionality is hit or miss<br>• Likely uses a cheap clone of a real OBD-II interpreter<br>• Connects to phone or PC application | • Does not read or clear diagnostic codes<br>• Specially designed for one vehicle<br>• Pi and screen mounted in dash | • Supports only one OBD-II protocol (CAN-BUS) |

*Table 3. A comparison table for some of the existing solutions.*

## Deliverables

All source code and documentation will be delivered to program director Kevin Pintong as well as made open source. The components used to build the project will remain in Zak Rowland's possession.

# System Description

## System Block Diagram

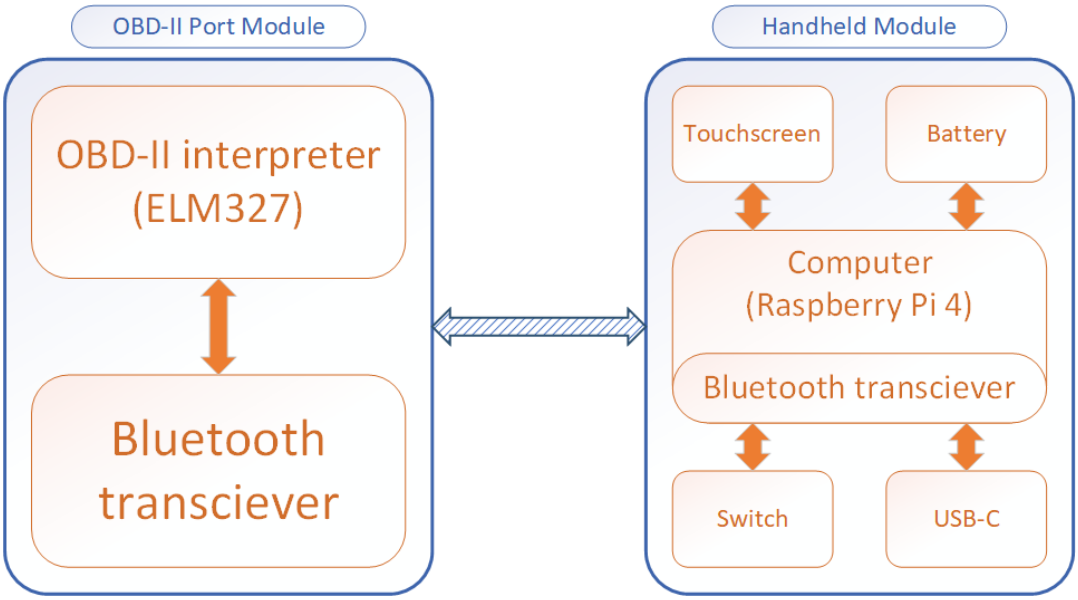



*Figure 4. A block diagram of the system.*

## System General Description

Upon power up, the system's default mode is to scan diagnostic codes. To accomplish this, the circuit connected to the OBD-II must read all the trouble codes and send that information to the handheld unit through a Bluetooth connection. The computer inside the handheld device must process this information and drive a display for the user to see. Input from the user will occur through the touchscreen. A list of codes, or a message indicating the lack thereof, will be displayed to the user and they will be able to tap on a code to view the description and/or the possible cause. The user will have the option to clear all the diagnostic codes which involves sending a command over the Bluetooth connection so the OBD-II can respond accordingly. There will also be an alternate mode for viewing sensor information

live (at least 30 times per second,) and a settings menu to adjust various options like brightness or units.

## Major Subsystems

The major subsystems of the project are the power circuitry for the OBD-II module, the Bluetooth circuitry for the OBD-II module, the rechargeable battery circuitry for the handheld module, and the user interface for the handheld module.

## Hardware Platform Description

A Raspberry Pi 4 will be used as the computer inside the handheld unit. The Pi is a good choice for this project for several reasons. It can run popular operating systems which makes user interface development much easier. There is plenty of memory, storage, I/O, and even documentation, and the Pi 4 has a DSI connection that makes adding a touchscreen trivial. An ESP32 microcontroller will act as the Bluetooth transceiver for the OBD-II module. This microcontroller supports Bluetooth version 4.2, has several GPIO pins, and will make debugging the OBD-II module much easier.

## Software Platform Description

For the Raspberry Pi, scripts will be written in Python and any other software, if necessary, will be written in C. The code for the ESP32 microcontroller will be written in C. Python scripts will be written in an IDE or text editor such as Spyder or Notepad++, and C code will be written in an IDE such as Visual Studio or Atmel Studio.

## Component Comparison and Selection

This project was initially intended to be as low cost as possible. However, the COVID-19 pandemic has caused limited access to lab equipment. To keep the project achievable during such uncertain times, some parts may be chosen primarily due to their popularity and documentation. Table 4 below compares different single board computer options for the project. The Raspberry Pi 4 Model B was chosen due to its popularity, ease of use, and built in DSI connection for a display. However, the Raspberry Pi Zero W would be the ideal computer for the project due to its smaller size and lower power consumption.

### Computer for Handheld Unit

Table 4 below compares different single board computer options for the project. The Raspberry Pi 4 Model B was chosen due to its popularity, ease of use, and built in DSI connection for a display. However, the Raspberry Pi Zero W would be the ideal computer for the project due to its smaller size and lower power consumption.

| **Name** | Raspberry Pi 4 Model B | Raspberry Pi Zero W | BananaPi-M3 |
|---|---|---|---|
| **Price** | $35 | $10 | $43 |
| **Processor** | Broadcom BCM2711 (Quad core Cortex-A72 ARM v8), 64-bit, 1.5GHz | Broadcom BCM2835 (Single core ARM1176JZFS), 1GHz | Realtek RTD1395 (Quad core Cortex-A53), 64-bit |
| **Storage** | MicroSD | MicroSD | 8GB eMMC + MicroSD |
| **Memory** | 2GB LPDDR4-3200 SDRAM | 512MB | 1GB DDR4 |
| **Wireless** | Wi-Fi and Bluetooth 5.0 | Wi-Fi and Bluetooth 4.1 | Wi-Fi and Bluetooth 4.2 |
| **I/O** | (2) USB 3.0, (2) USB 2.0, (2) Micro HDMI, DSI, CSI, 3.5mm A/V, Ethernet | Mini HDMI, USB OTG, CSI, Composite video | (4) USB 2.0, M.2, 3.5mm, HDMI, Ethernet |
| **GPIO** | 40-pin | 40-pin | 28-pin |
| **USB/Power** | USB-C, 5V/3A+ | Micro-USB, 5V/2A+ | USB-C, 5V/2A+ |
| **Other** | OpenGL ES 3.0 graphics | | OpenGL ES 1.1/2.0 graphics |

*Table 4. A comparison table for single board computers.*

Table 5 below compares two different OBD-II interpreter integrated circuits, the ELM327 and the STN1110. The ELM327 was chosen for the project. This chip is uncontestably the most popular when making OBD-II readers. The documentation is superb and there are many usage examples to reference, and it was chosen primarily for this reason. The STN1110 is almost an exact replica of the ELM327, it is even completely compatible with all commands used for the ELM chip. However, the STN1110 is half the price and includes an extended command set for even more

available features. Since the STN1110 is not documented as well as the ELM327 and access to lab equipment is restricted, it is more sensible to use the ELM chip.

## OBD-II to UART Interpreter

| **Name** | ELM327 | STN1110 |
|---|---|---|
| **Price** | $21 | $10 |
| **Operating voltage** | 4.5 – 5.5VDC | 3 – 3.6VDC |
| **Operating current** | 12mA | 63mA |
| **Power saver mode** | 0.15mA | <2mA |
| **Operating power** | 60mW at 5V | 207.9mW at 3.3V |
| **Power saving power** | 0.75mW at 5V | 6.6mW at 3.3V |
| **Features** | • Power control with standby mode<br>• Serial interface<br>• Automatically searches for correct protocol<br>• Fully configurable with AT commands<br>• Low power CMOS design<br>• Very popular and well documented | • Fully compatible with the ELM327 (AT) command set<br>• Extended (ST) command set<br>• UART interface<br>• Automatically searches for correct protocol<br>• Large memory buffer (more RAM)<br>• Voltage input for battery monitoring |
| **Cons** | • Expensive<br>• Many buggy fakes/clones if not purchased from manufacturer | • Less documentation and examples<br>• Uses more power |

*Table 5. A comparison table for OBD-II interpreters.*

## Bluetooth Transceiver

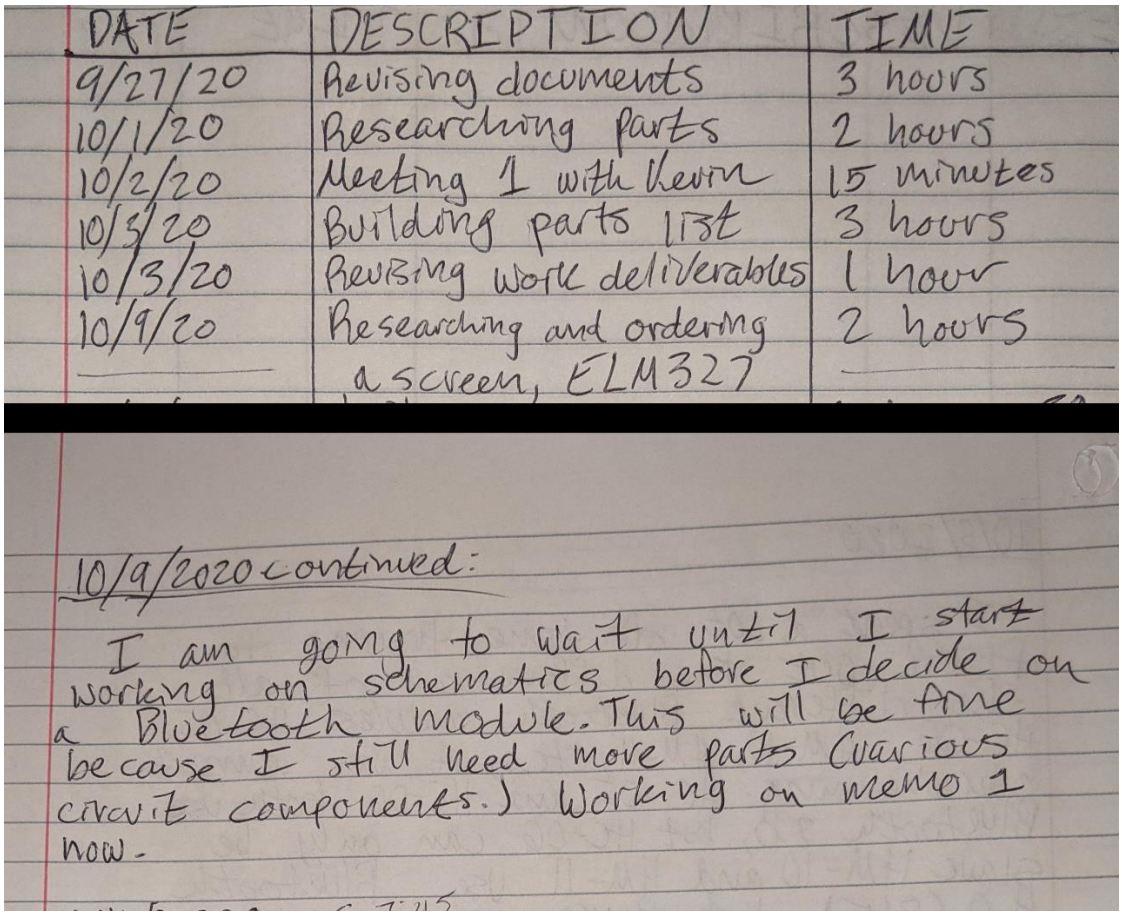
Table 6 below compares different Bluetooth modules for potential use in the project. The HM-19 or HM-10 module was going to be used originally, however the ESP32 microcontroller will be used instead. The ESP32-DevKitC-32D uses Bluetooth version 4.2, costs about the same as the modules below, and provides more functionality for debugging.

| **Name** | HC-05 | HC-06 | HM-19 |
|---|---|---|---|
| **Price** | $7.99 | $7.39 | $9.99 |
| **Bluetooth version** | 2.0 | 2.0 | 5.0 |
| **Operating voltage** | 3.6 – 6VDC | 3.3VDC | 3.6 – 6VDC |
| **Logic voltage** | 3.3V | 3.3V | 3.3V |
| **Modes** | Master/slave | Slave only | Master/slave |

*Table 6. A comparison table for Bluetooth transceiver modules.*

# Fall Term Progress, Problems, and Discussion

Work completed so far has primarily been research and documentation. Notes, calculations, drawings, and time spent are all tracked in a spiral notebook as seen in Figure 5 below.



| DATE | DESCRIPTION | TIME |
|---|---|---|
| 9/27/20 | Revising documents | 3 hours |
| 10/1/20 | Researching parts | 2 hours |
| 10/2/20 | Meeting 1 with Kevin | 15 minutes |
| 10/5/20 | Building parts list | 3 hours |
| 10/3/20 | Revising work deliverables | 1 hour |
| 10/9/20 | Researching and ordering a screen, ELM327 | 2 hours |

10/9/2020 continued:

I am going to wait until I start working on schematics before I decide on a Bluetooth module. This will be fine because I still need more parts (various circuit components.) Working on memo 1 now.

*Figure 5. An example of the notes and time logs from the notebook.*

The first difficult decision was choosing a Bluetooth module. The class was recommended to use the HC-05 module. However, that module uses Bluetooth 2.0, and the project requirements require at least 4.0. While deciding on a Bluetooth module, parts that were already chosen including the ELM327 chip, a 32GB MicroSD card, and a touchscreen were purchased.

Still undecided on Bluetooth parts, the Raspberry Pi was loaded with an operating system, Raspberry Pi OS, to test it. This operating system was chosen because it is the one recommended by the manufacturer, but it would be simple to change to a different one if necessary. An undervoltage warning appeared

intermittently while the Pi was in use, and it was discovered that the Pi prefers closer to 5.1V/3A instead of 5V/3A. The undervoltage will not be an issue because the rechargeable battery circuit will use a regulator that can supply 5.1V. After testing the various applications included with the operating system, a very simple Python GUI tutorial was completed. The tutorial was completed to get a feel for the process, and the result can be seen in Figure 6 below.

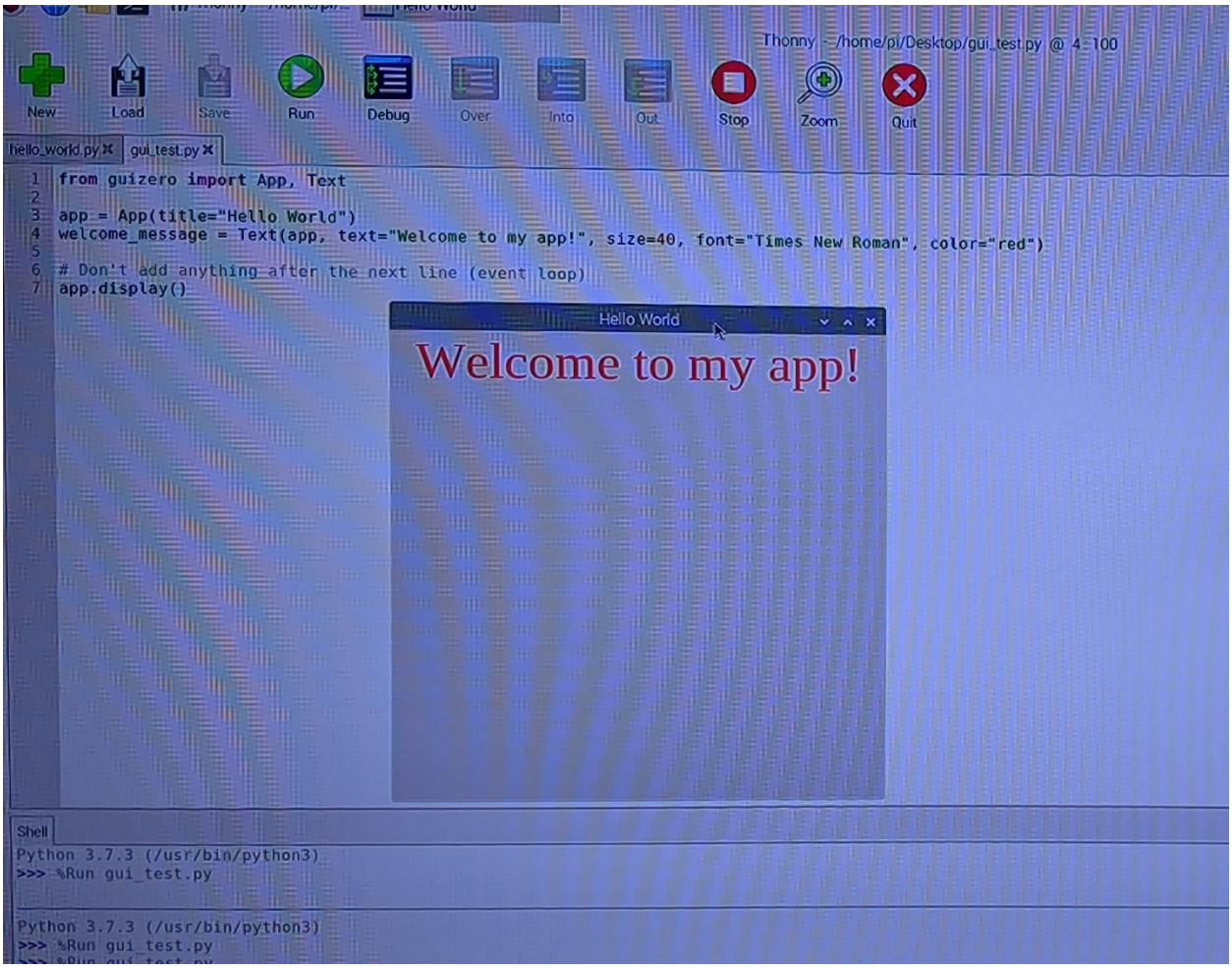



*Figure 6. A picture of the basic GUI created with Python script.*

While working on the documentation for the preliminary design review, more research was done on the components required for the OBD-II port module. Some

pins on the ELM327 are not required depending on the protocol used in the vehicle, so the pinout of the OBD-II port in a 2008 Nissan Altima had to be verified by inspecting the port to see which holes had metal contacts for pins. This pinout can be seen in Figure 7 below.

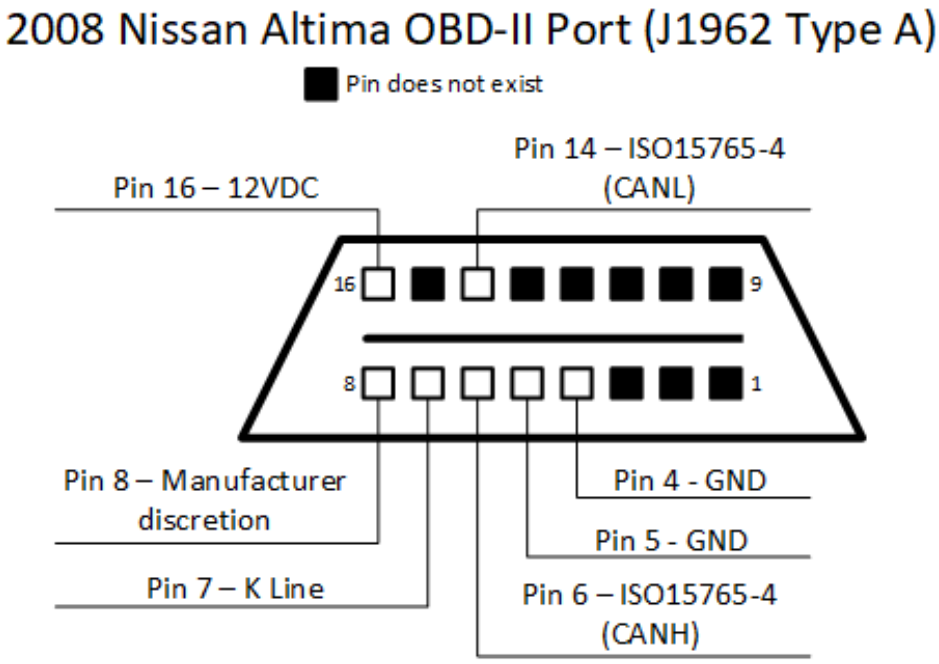



*Figure 7. The pinout for the OBD-II port in a 2008 Nissan Altima.*

Schematic capture with the KiCAD software began by creating a symbol library for the project, and the first symbol created was the ELM327 chip. An image of the symbol can be seen in Figure 8 below.

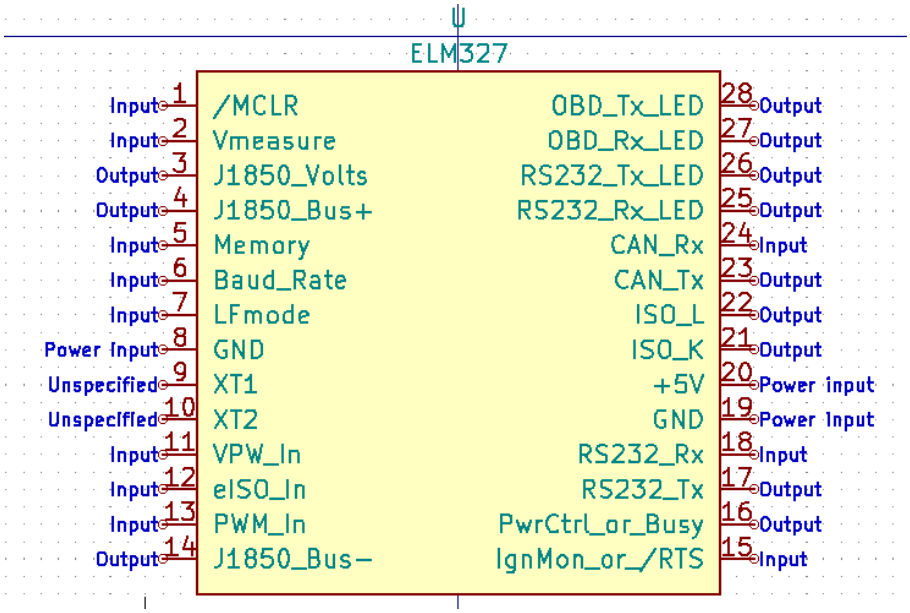
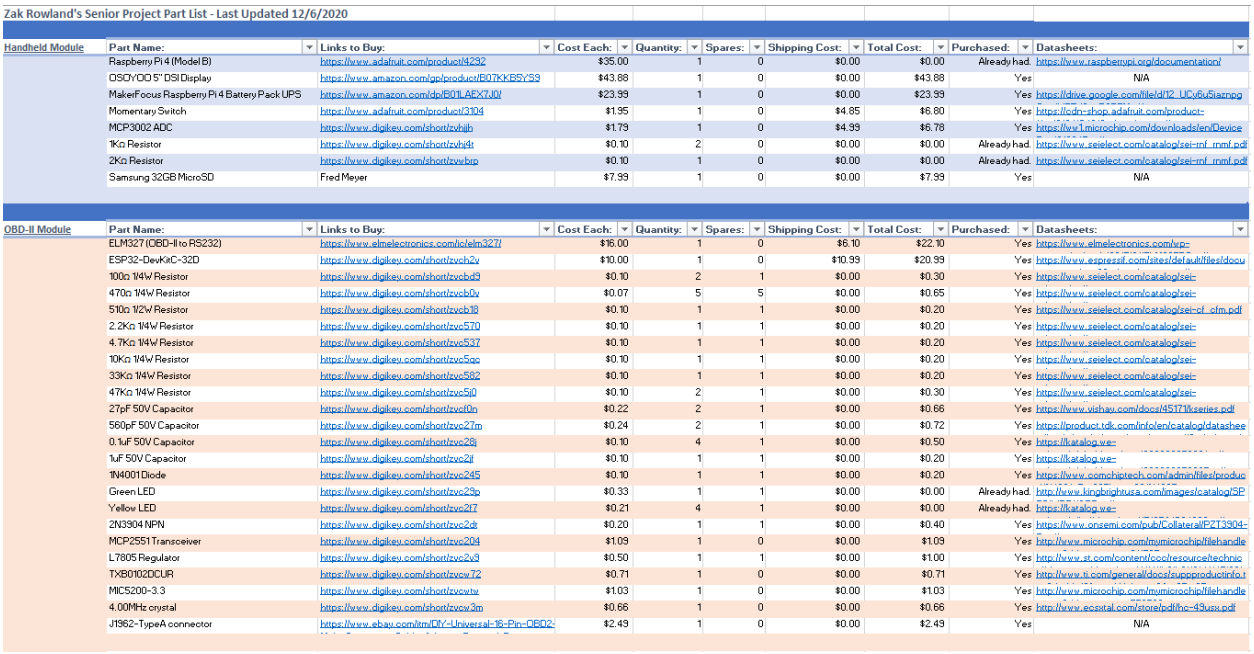



*Figure 8. The ELM327 symbol created for use in schematics.*

The official ELM327 datasheet includes a diagram for an example application circuit for the chip, so the schematic for the OBD-II port module will include pieces of the circuit recommended in said datasheet [6]. Since progress was being made on the schematic, a Bluetooth module had to be chosen. After more research, the ESP32 was chosen as the best fit for the project given the circumstances. This microcontroller uses Bluetooth version 4.2 so it meets requirements, includes TX and RX pins in the GPIO for serial communication, and will allow for much easier debugging.

While developing the schematic, a table of parts was made to make ordering and creating a bill of materials much faster. A screenshot of this Excel spreadsheet can be seen in Figure 9 below.



Zak Rowland's Senior Project Part List - Last Updated 12/6/2020

| Handheld Module | Part Name: | Links to Buy: | Cost Each: | Quantity: | Spares: | Shipping Cost: | Total Cost: | Purchased: | Datasheets: |
|---|---|---|---|---|---|---|---|---|---|
| | Raspberry Pi 4 (Model B) | https://www.adafruit.com/product/4292 | $35.00 | 1 | 0 | $0.00 | $0.00 | Already had | https://www.raspberrypi.org/documentation/ |
| | OSOYOO 5" DSI Display | https://www.amazon.com/gp/product/B07KKB5YS9 | $43.88 | 1 | 0 | $0.00 | $43.88 | Yes | N/A |
| | MakerFocus Raspberry Pi 4 Battery Pack UPS | https://www.amazon.com/dp/B01LAEX7J0/ | $23.99 | 1 | 0 | $0.00 | $23.99 | Yes | https://drive.google.com/file/d/12_UCy6u5iaznpg |
| | Momentary Switch | https://www.adafruit.com/product/3104 | $1.95 | 1 | 0 | $4.85 | $6.80 | Yes | https://cdn-shop.adafruit.com/product- |
| | MCP3002 ADC | https://www.digikey.com/short/zvhjjh | $1.79 | 1 | 0 | $4.99 | $6.78 | Yes | https://ww1.microchip.com/downloads/en/Device |
| | 1KΩ Resistor | https://www.digikey.com/short/zvhjdt | $0.10 | 2 | 0 | $0.00 | $0.00 | Already had | https://www.seielect.com/catalog/sei-rnf_rnmf.pdf |
| | 2KΩ Resistor | https://www.digikey.com/short/zvvbrp | $0.10 | 1 | 0 | $0.00 | $0.00 | Already had | https://www.seielect.com/catalog/sei-rnf_rnmf.pdf |
| | Samsung 32GB MicroSD | Fred Meyer | $7.99 | 1 | 0 | $0.00 | $7.99 | Yes | N/A |

| OBD-II Module | Part Name: | Links to Buy: | Cost Each: | Quantity: | Spares: | Shipping Cost: | Total Cost: | Purchased: | Datasheets: |
|---|---|---|---|---|---|---|---|---|---|
| | ELM327 (OBD-II to RS232) | https://www.elmelectronics.com/ic/elm327/ | $16.00 | 1 | 0 | $6.10 | $22.10 | Yes | https://www.elmelectronics.com/wp- |
| | ESP32-DevKitC-32D | https://www.digikey.com/short/zvch2v | $10.00 | 1 | 0 | $10.99 | $20.99 | Yes | https://www.espressif.com/sites/default/files/docu |
| | 100Ω 1/4W Resistor | https://www.digikey.com/short/zvcbd9 | $0.10 | 2 | 1 | $0.00 | $0.30 | Yes | https://www.seielect.com/catalog/sei- |
| | 470Ω 1/4W Resistor | https://www.digikey.com/short/zvcb0v | $0.07 | 5 | 5 | $0.00 | $0.65 | Yes | https://www.seielect.com/catalog/sei- |
| | 510Ω 1/2W Resistor | https://www.digikey.com/short/zvcb18 | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.seielect.com/catalog/sei-cf_cfm.pdf |
| | 2.2KΩ 1/4W Resistor | https://www.digikey.com/short/zvc570 | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.seielect.com/catalog/sei- |
| | 4.7KΩ 1/4W Resistor | https://www.digikey.com/short/zvc537 | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.seielect.com/catalog/sei- |
| | 10KΩ 1/4W Resistor | https://www.digikey.com/short/zvc5qc | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.seielect.com/catalog/sei- |
| | 33KΩ 1/4W Resistor | https://www.digikey.com/short/zvc582 | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.seielect.com/catalog/sei- |
| | 47KΩ 1/4W Resistor | https://www.digikey.com/short/zvc5j0 | $0.10 | 2 | 1 | $0.00 | $0.30 | Yes | https://www.seielect.com/catalog/sei- |
| | 27pF 50V Capacitor | https://www.digikey.com/short/zvcf0n | $0.22 | 2 | 1 | $0.00 | $0.66 | Yes | https://www.vishay.com/docs/45171/kseries.pdf |
| | 560pF 50V Capacitor | https://www.digikey.com/short/zvc27m | $0.24 | 2 | 1 | $0.00 | $0.72 | Yes | https://product.tdk.com/info/en/catalog/datashee |
| | 0.1uF 50V Capacitor | https://www.digikey.com/short/zvc28j | $0.10 | 4 | 1 | $0.00 | $0.50 | Yes | https://katalog.we- |
| | 1uF 50V Capacitor | https://www.digikey.com/short/zvc2jf | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://katalog.we- |
| | 1N4001 Diode | https://www.digikey.com/short/zvc245 | $0.10 | 1 | 1 | $0.00 | $0.20 | Yes | https://www.comchiptech.com/admin/files/produc |
| | Green LED | https://www.digikey.com/short/zvc29p | $0.33 | 1 | 1 | $0.00 | $0.00 | Already had | http://www.kingbrightusa.com/images/catalog/SP |
| | Yellow LED | https://www.digikey.com/short/zvc217 | $0.21 | 4 | 1 | $0.00 | $0.00 | Already had | https://katalog.we- |
| | 2N3904 NPN | https://www.digikey.com/short/zvc2dt | $0.20 | 1 | 1 | $0.00 | $0.40 | Yes | https://www.onsemi.com/pub/Collateral/P2T3904- |
| | MCP2551 Transceiver | https://www.digikey.com/short/zvc204 | $1.09 | 1 | 0 | $0.00 | $1.09 | Yes | http://www.microchip.com/mymicrochip/filehandle |
| | L7805 Regulator | https://www.digikey.com/short/zvc2v3 | $0.50 | 1 | 1 | $0.00 | $1.00 | Yes | http://www.st.com/content/ccc/resource/technic |
| | TXB0102DCUR | https://www.digikey.com/short/zvcw72 | $0.71 | 1 | 0 | $0.00 | $0.71 | Yes | http://www.ti.com/general/docs/suppproductinfo.t |
| | MIC5200-3.3 | https://www.digikey.com/short/zvcwtw | $1.03 | 1 | 0 | $0.00 | $1.03 | Yes | http://www.microchip.com/mymicrochip/filehandle |
| | 4.00MHz crystal | https://www.digikey.com/short/zvcw3m | $0.66 | 1 | 0 | $0.00 | $0.66 | Yes | http://www.ecsital.com/store/pdf/hc-49usx.pdf |
| | J1962-TypeA connector | https://www.ebay.com/itm/DIY-Universal-16-Pin-OBD2- | $2.49 | 1 | 0 | $0.00 | $2.49 | Yes | N/A |

*Figure 9. The Excel spreadsheet used to track parts and costs.*

After creating more symbols and making the necessary connections, the schematic for the OBD-II port module was completed and can also be seen in Appendix A. Development of the next schematic is currently underway and has involved researching rechargeable battery parts.

Progress on the handheld module's schematic was slow due to preparing documentation and a design review, however this is acceptable because this circuit is much less involved than the OBD-II port module. After the design review, all parts for the OBD-II module were ordered. Choosing resistors and capacitors proved to be more difficult than anticipated. When choosing resistors, there were many different options in composition (carbon film, metal film, etc.,) tolerance, power dissipation, size, and more. After some research, metal film resistors were chosen wherever possible due to the similar/equal prices and superior tolerances when compared to carbon film resistors.

The crystal in the circuit requires loading capacitors on each pin for a total load capacitance ($C_L$) of 20pF. The equation below was used to determine the value of the two loading capacitors.

$$Capacitance_{Load} = \frac{C_1 * C_2}{C_1 + C_2} + C_{Stray}$$

The capacitance load was known, 20pF, and $C_{Stray}$ is stray capacitance from nearby wires and circuitry, typically 3-5pF. With this information, the values for the capacitors (30pF) can be determined.

$$20pF = \frac{C_1 * C_2}{C_1 + C_2} + 5pF \rightarrow 15pF = \frac{C_1 * C_2}{C_1 + C_2} \rightarrow C_1 = C_2 = 30pF$$

27pF capacitors were ordered instead as they were more common and are recommended for use in the ELM327 datasheet.

Once parts for the OBD-II port module were ordered, the parts table spreadsheet seen in Figure 9 above was updated accordingly. Following this began the difficult decision of choosing a rechargeable battery solution for the device. The Raspberry Pi 4 typically consumes about 600mA but can draw up to 1.2A [7]. The OSOYOO touchscreen consumes a maximum of 830mW (or 251.5mA) [8]. Since the maximum possible current draw is about 1.5A, it was determined that the battery would need to be at least 3Ah to power the system for a minimum of two hours as outlined in the requirements. The initial idea was to use a USB-C lithium-polymer battery charger from Adafruit and a compatible battery. However, this idea was scrapped because the battery voltage would need to be boosted to 5V/3A and it was

difficult finding Li-Po batteries with enough capacity and discharge current capable of doing so.

While still deciding on a battery configuration, the ESP32 was tested. After some research on programming environments for the ESP32, it was decided that Visual Studio Code with the official Espressif ESP32 extension (ESP-IDF) would be used. This environment was chosen because the extension is officially supported and provides build, flash, debug, terminal, and monitor functionality all in one place. Setting up this environment proved to be extremely time consuming. After waiting for downloads and installations, it was determined that the ESP-IDF installation path cannot have spaces in it, and the path did have spaces in it due to the username containing a space.

A new user was created to work around this problem, however the "hello world" program was still not building. After searching online and several more attempts uninstalling, downloading, and reinstalling, the issue was an incorrect path to the Python installation that had to be updated in the extensions' settings. Once the path was updated, the example program compiled and successfully flashed to the ESP32 and the result can be seen in Figure 10 below.

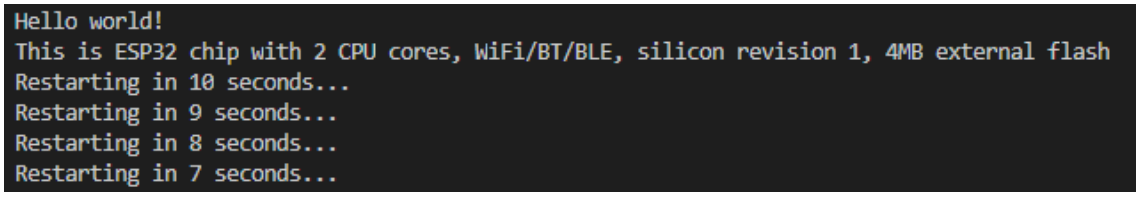



*Figure 10. The output of the "hello world" example program for the ESP32.*

After this test program, the Bluetooth LE examples were browsed. All of the examples were relatively complex and a simple Bluetooth connection was all that

needed to be tested at the moment. The Arduino IDE was temporarily used as the Bluetooth examples included are much simpler. After some configuration, the ESP32 was creating a Bluetooth signal and an Android phone could successfully connect to it and read a characteristic as seen in Figure 11 below.

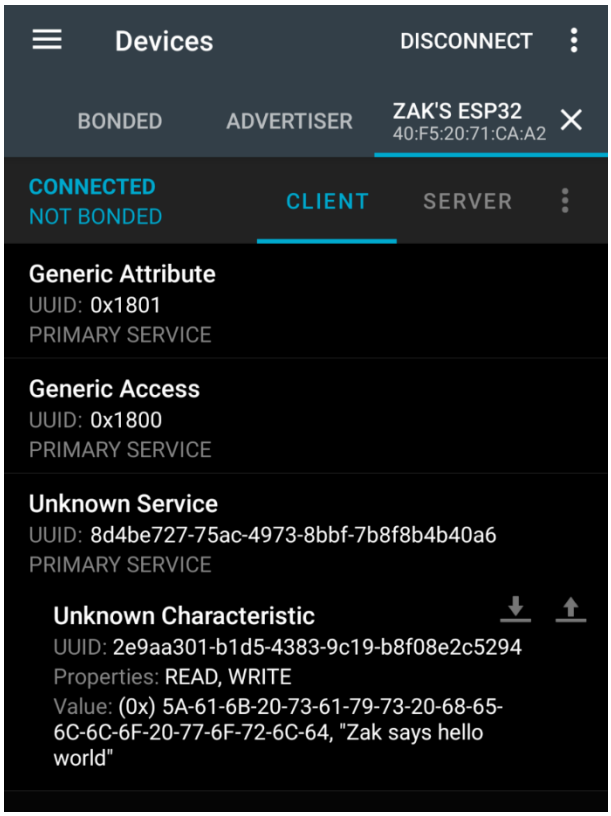



*Figure 11. The ESP32 successfully connected to an Android phone app.*

Once the ESP32's Bluetooth signal was confirmed to work with a phone, the Raspberry Pi 4 was tested next. Since the Pi does not include a Bluetooth stack by default, BlueZ was used as it is popular and open source (although a bit outdated.) After some configuration and troubleshooting on the Pi, the ESP32 was successfully connected to the Pi as seen in Figure 12 below. The same characteristic hex value read in the phone app was successfully read on the Pi.

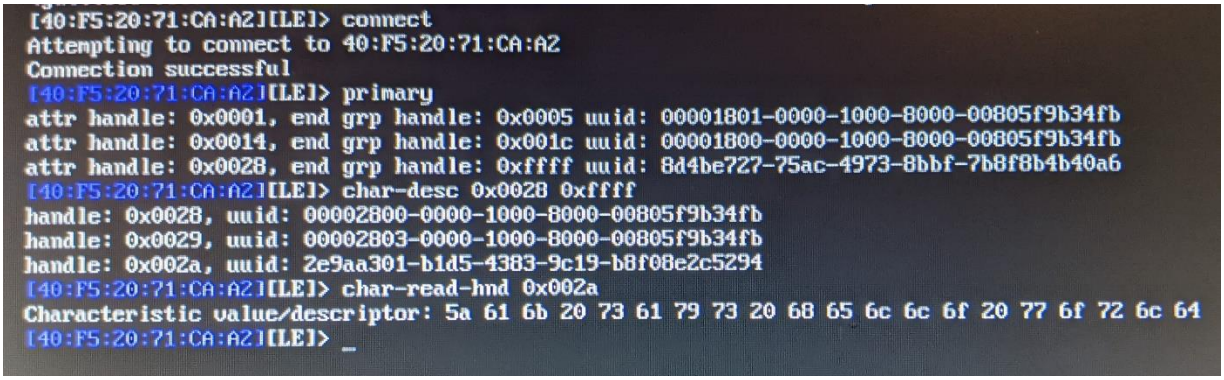



*Figure 12. The ESP32 successfully connected to the Raspberry Pi 4.*

After these Bluetooth tests, the rechargeable battery solution had to be decided. As determined previously, the battery capacity needs to be at least 3Ah and must be capable of 3A output. Researching possible solutions led to battery "hats" for the Raspberry Pi. These hats are separate circuit boards that include a battery and all the charging and protection circuitry required. The MakerFocus Raspberry Pi 4 Battery Pack UPS was chosen because it includes a USB-C charging port, a 4000mAh battery, and it supplies a consistent 5V/3A to the Pi.

Once the battery solution was decided, the handheld module's schematic was finished and can be seen in Appendix A. The parts spreadsheet was updated and the last of the required parts were ordered. Testing the various components including resistors, capacitors, and more, as well as building the OBD-II port module's circuit are the next tasks that will begin next term. The state of the Trello board at the end of Fall term can be seen in Figure 13 below.

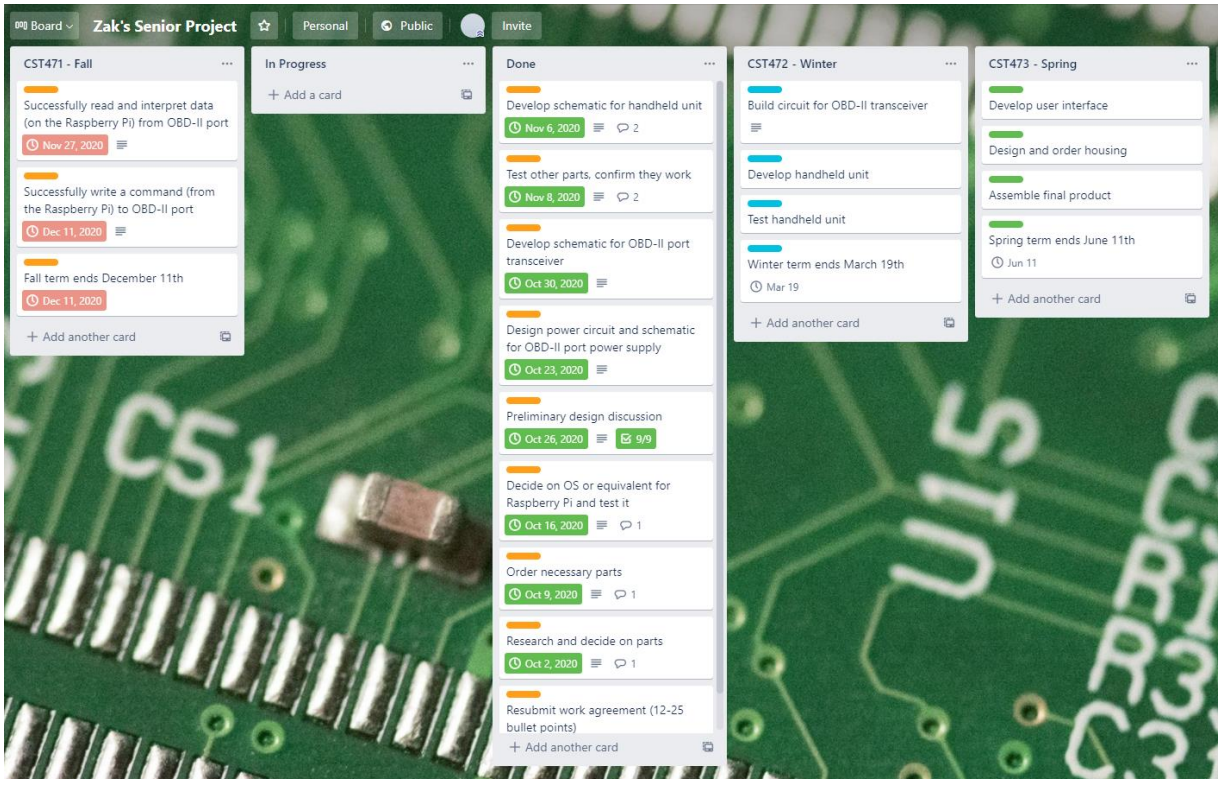



*Figure 13. A screenshot of the Trello board at the end of Fall term.*

The tasks to read and write data to/from the OBD-II port were not completed before the term reached an end because it requires the circuit to be complete; This was an oversight from the beginning of the term when trying to determine feasible deliverables.

## Winter Term Progress, Problems, and Discussion

After a long winter break, progress had to be made on the circuit. The first task completed was measuring the resistors and capacitors using a multimeter to confirm they are the correct values. This turned out to be an unnecessary step, however being certain the values are correct does no harm. At the same time, the push button and LEDs were tested. The button also has an internal red LED that can be permanently powered or controlled by some device. An Arduino was used for

a 5V power source and the button was wired to power an LED while pressed. Figure 14 below shows the testing setup.

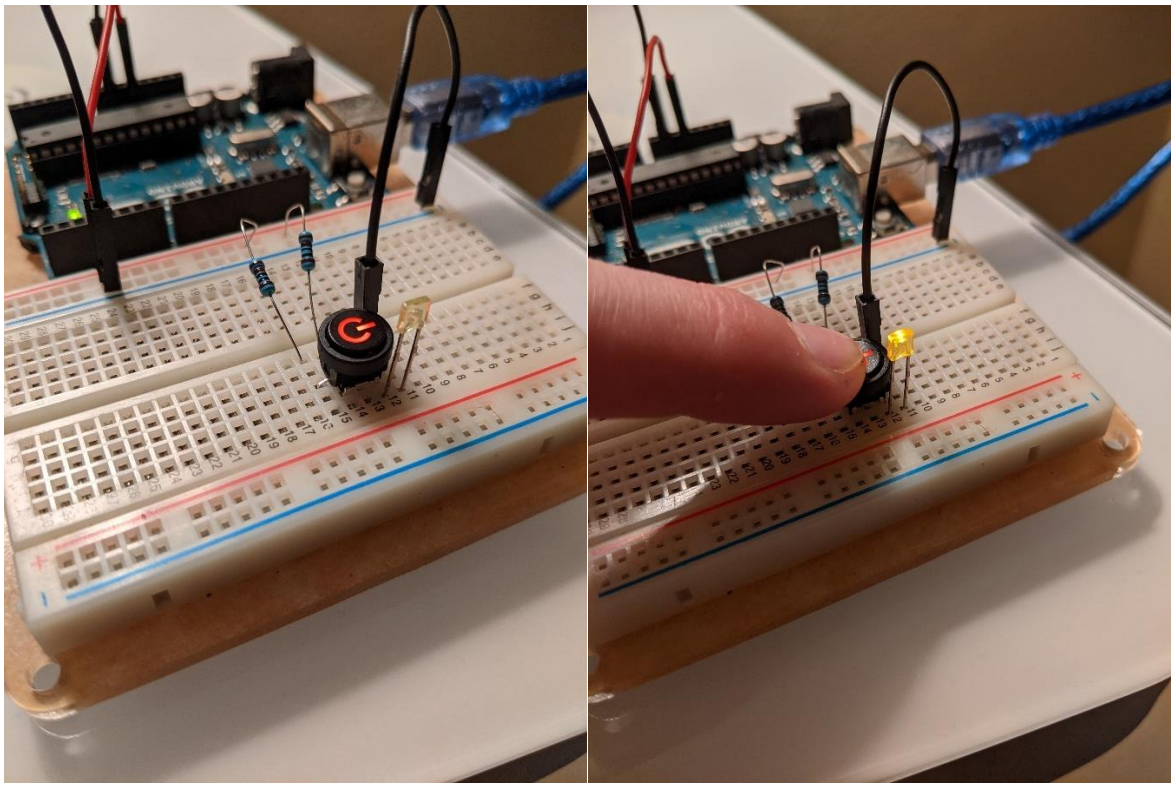


*Figure 14. Testing the button and status LEDs.*

Once these components were confirmed to be working, the circuit could be assembled. At this point, a VSSOP-8 to DIP-8 adapter and a SOT-223 to DIP-6 adapter were ordered to allow the small integrated circuits to be used on a breadboard.

After the first meeting with Kevin Pintong, the work agreement was revised to make the goals more feasible. The Trello board was updated with the term's tasks and due dates as seen in Figure 15 below.

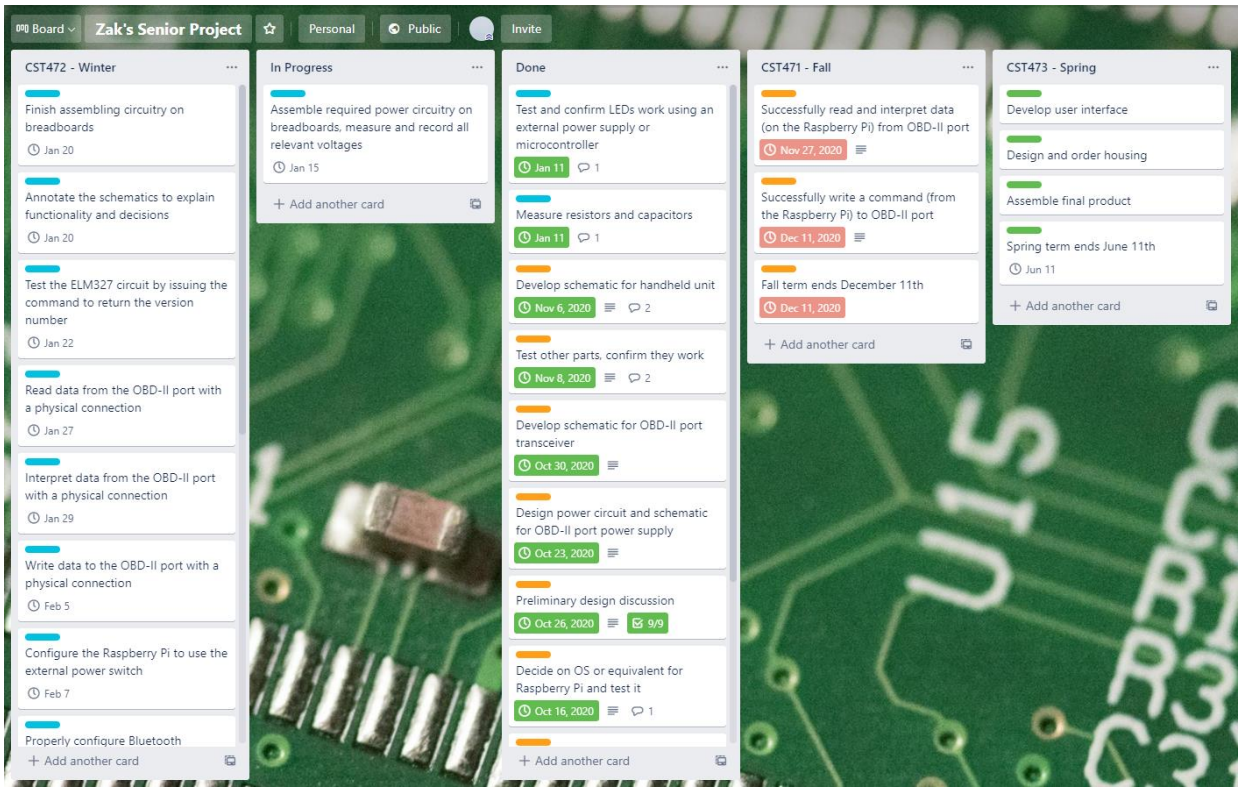



*Figure 15. A screenshot of the [Trello] board at the beginning of Winter term.*

The schematics for the project were also discussed during the first meeting. One of the filtering capacitors for the 5V regulator, C1, was originally a 0.1µF capacitor but after some insight was replaced with a 0.47µF capacitor for increased stability.

Before adding all the components to the breadboard, the power circuitry was assembled and tested first. The 5V portion was built first, and before connecting it the car's battery, the battery was measured at 11.98V. When finished, the regulator produced a consistent 5.05V and illuminated the power LED. Figure 16 below shows the voltage measurements and 5V circuitry.


*Figure 16. Measuring and testing the 5V circuitry.*

The 3.3V circuitry was assembled and tested in the same fashion and resulted in a steady 3.313V. Around this time, the DIP adapters arrived and were soldered together. The smaller IC, the TXB0102, was tricky to solder and one side got quite hot from the iron. After adding some flux, the pins soldered to the adapter nicely, however the possibilty of heat damage cannot be completely ruled out. The assembled adapters can be seen in Figure 17 below.

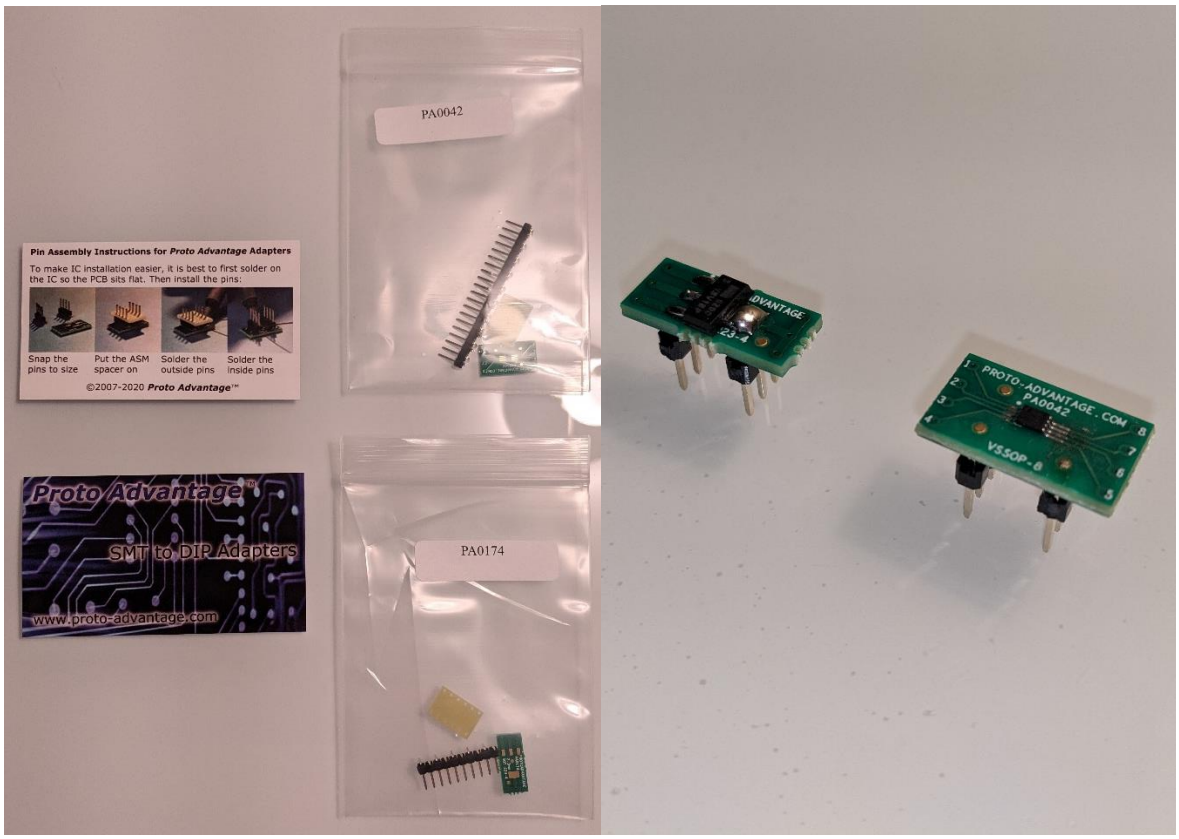



*Figure 17. The DIP adapters before and after assembly.*

While building the circuit, a copy of the schematic was annotated at the same time to clarify the purpose of each portion. Examples of these annotations and the finished circuit can be seen below in Figures 18 and 19 respectively.

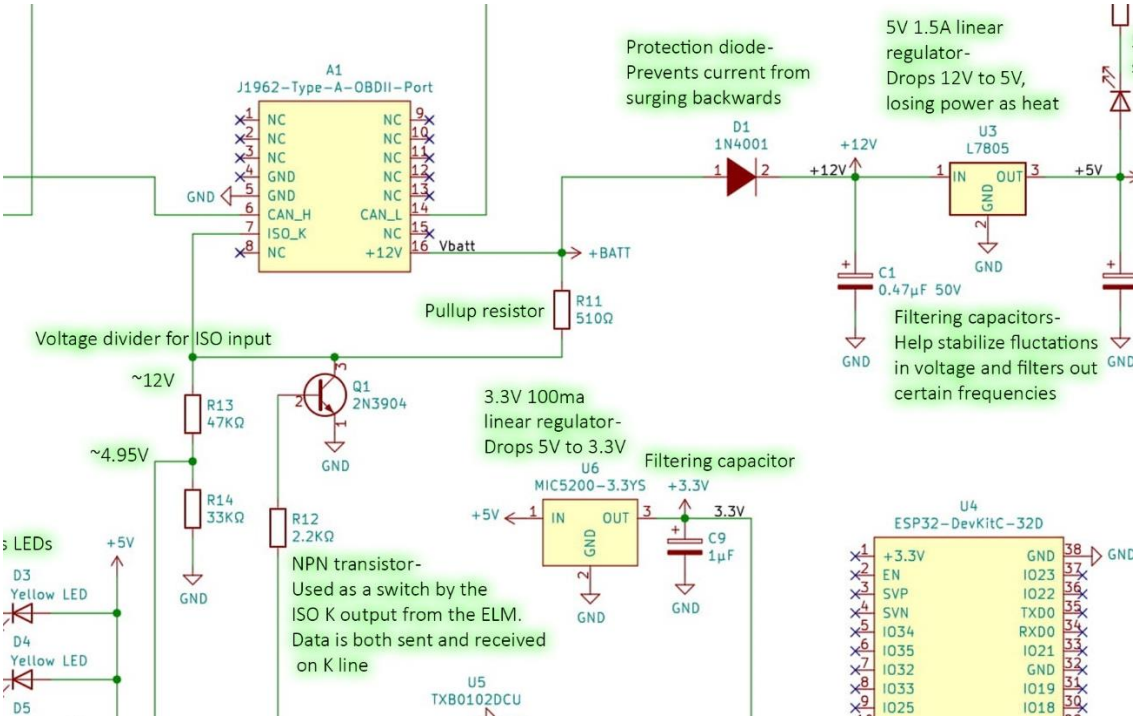



*Figure 18. A section of the annotated schematic.*

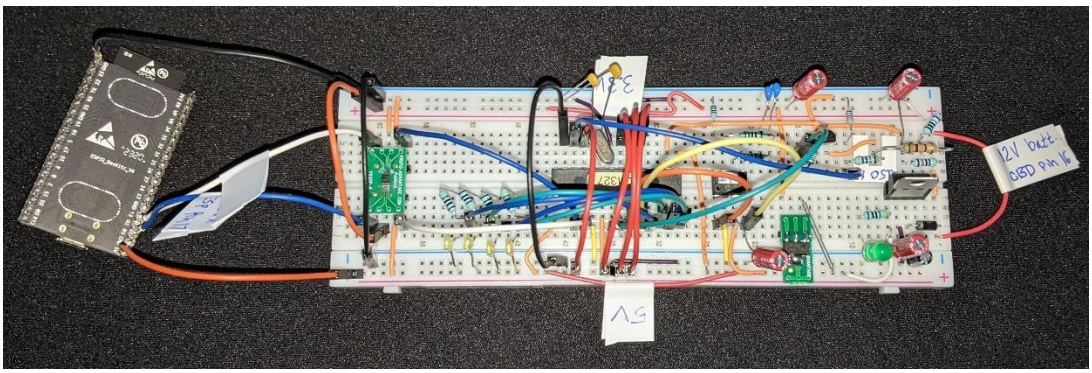


*Figure 19. The finished OBD-II port circuit.*

Once the circuit was assembled, it could be tested in the car. When connected, the status LEDs light in sequence which indicates the ELM327 is functioning at some capacity. However, with the ESP32's UART configured through Arduino IDE and the serial monitor open, the data read from the ELM327 was garbled.

The ELM327 was removed from the circuit and the voltages were measured once more since some components were moved around, but all voltages were still as expected, so the ELM was placed back into the circuit. In order to test if the ESP32 UART configuration or the logic level translator was the issue, a USB to serial adapter (UM232H-B) was used to communicate directly to the ELM327. Using this connection, the “AT Z” command to reset the chip and return the version number was issued successfully as seen in Figure 20 below.

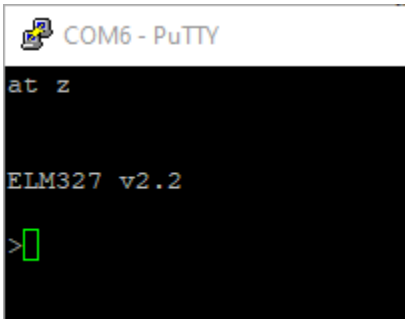



*Figure 20. A command successfully issued to the ELM327 from a terminal.*

The Espressif documentation includes some examples for configuring the UART for the ESP through the Visual Studio Code extension. After trying some of these examples, changing settings, and setting the baud rate in the ELM327 to 115.2K, the data was still garbled as seen in Figure 21 below.

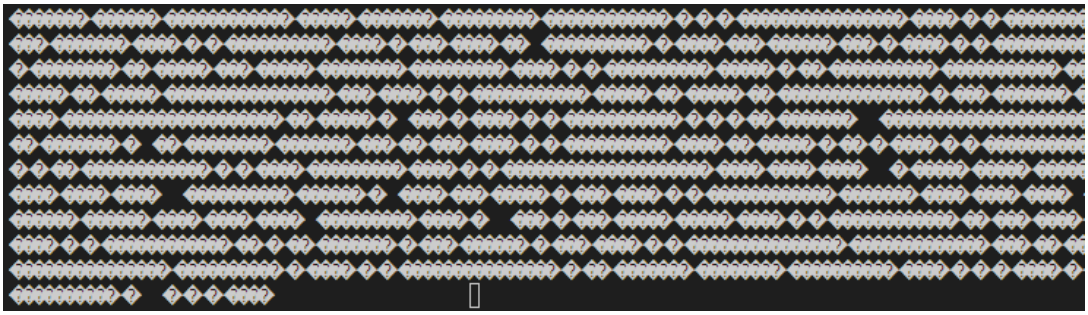


*Figure 21. An example of the gibberish data read through the ESP32 UART.*

This data was still present after commenting out any UART reads or writes, indicating it could be a software configuration issue. After failing to get the ESP32's UART to communicate correctly, the module was tested in the vehicle using the same USB to serial connection mentioned earlier. Using this method, the ELM327 successfully interfaced with the vehicle and was able to issue commands and read the response including checking coolant temperature, engine RPM, and more. The screenshots of the terminal session were saved and annotated to interpret all the different data retrieved and can be seen in Figures 22 and 23 below.

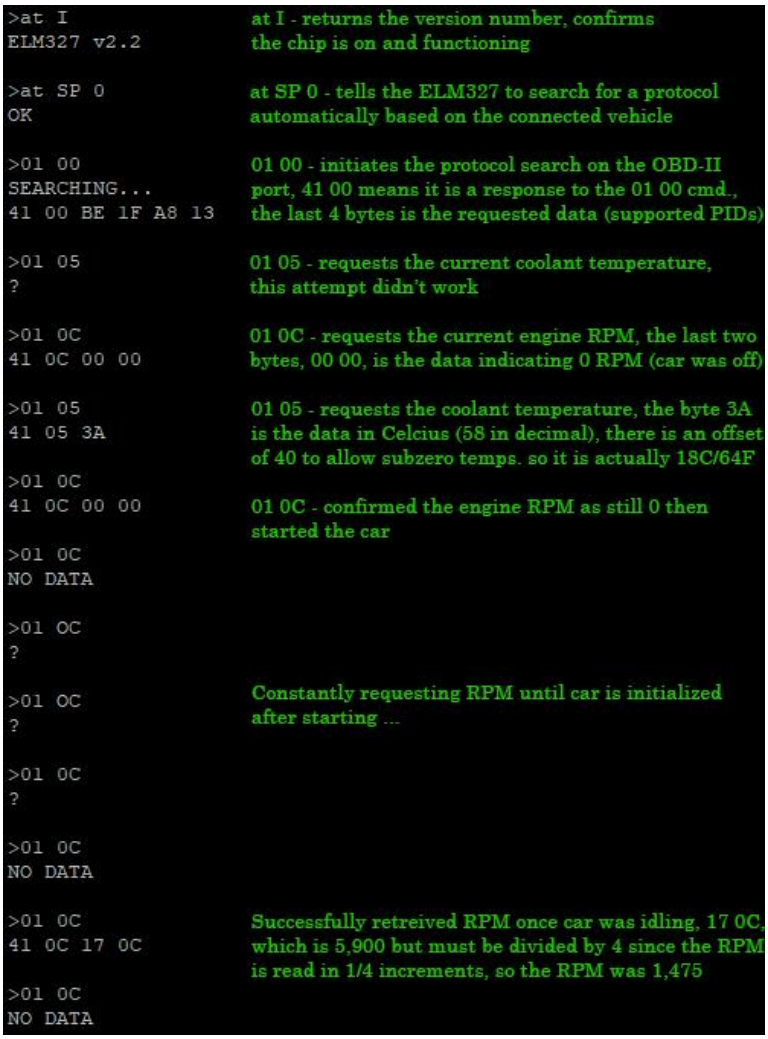



*Figure 22. Annotated screenshot of the first connection to the vehicle.*

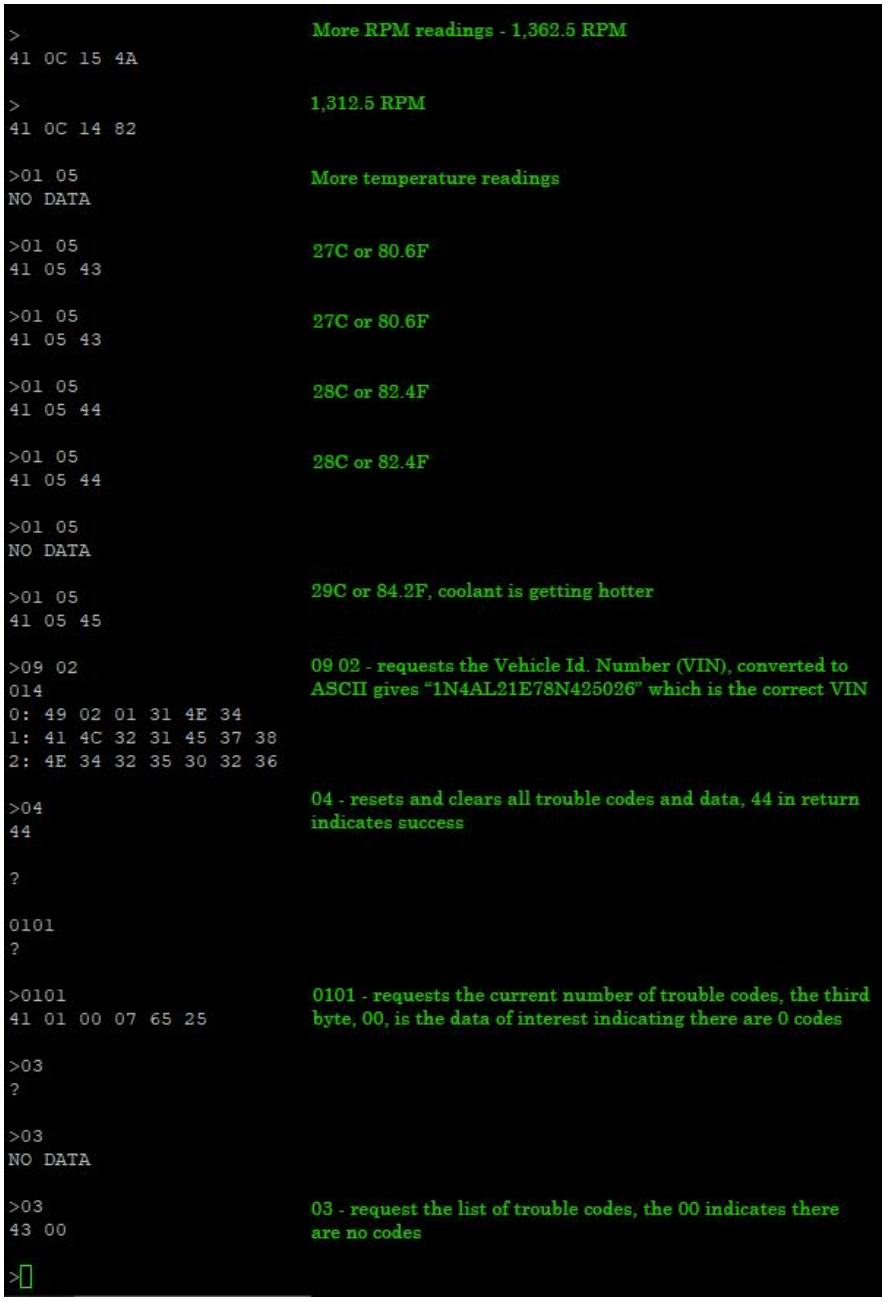



*Figure 23. More data from the first connection to the vehicle.*

With everything else working as intended, the next step is to debug the ESP32's UART configuration. It is also entirely possible that excess heat ruined the logic level translator when soldering, but an oscilloscope may be the only way to confirm the operation.

The state of the Trello board can be seen in Figure 24 below.

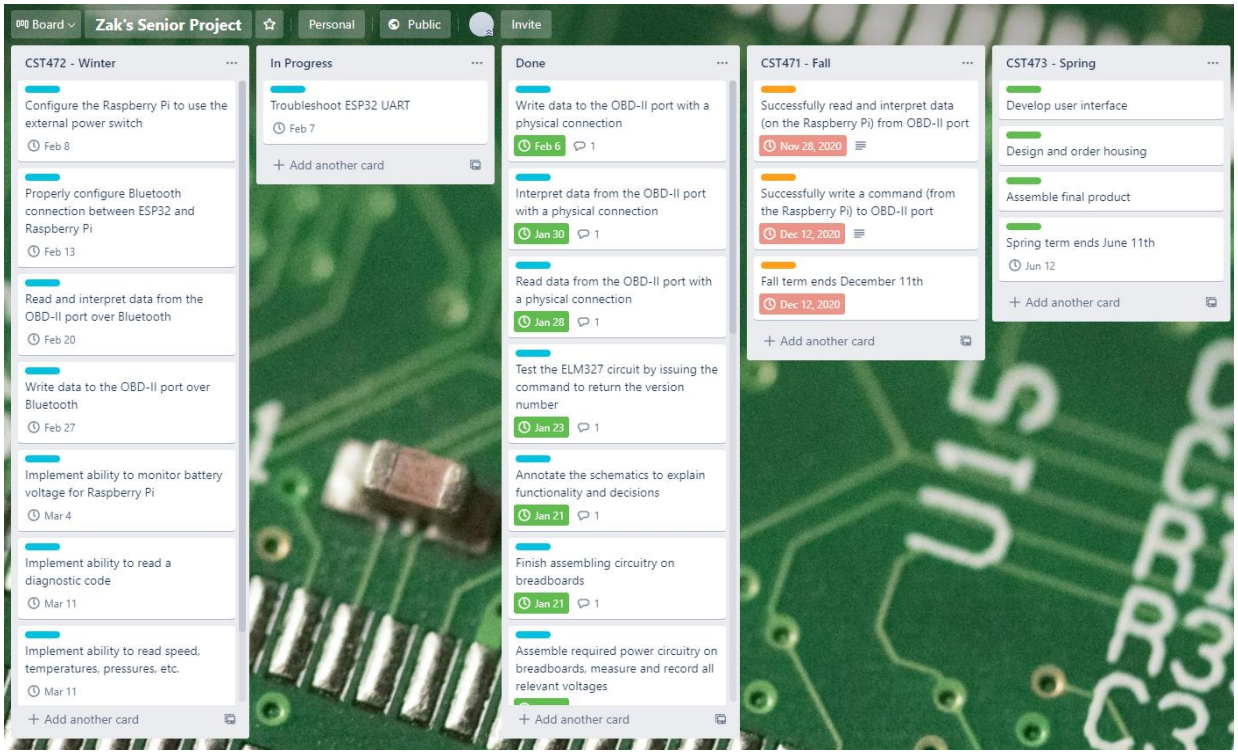



*Figure 24. A screenshot of the Trello board.*

Adding an external power button to the Raspberry Pi 4 turned out to be trivial. Following a tutorial from Embedded Computing Design [9], a line was added to the boot configuration file of the Pi to allow a switch connected to GPIO to shut it down using, power on is enabled by default. The internal LED was not connected yet, but it will be connected to a GPIO pin later to be controlled in a script. Figure 25 below shows the button connected to the Pi.

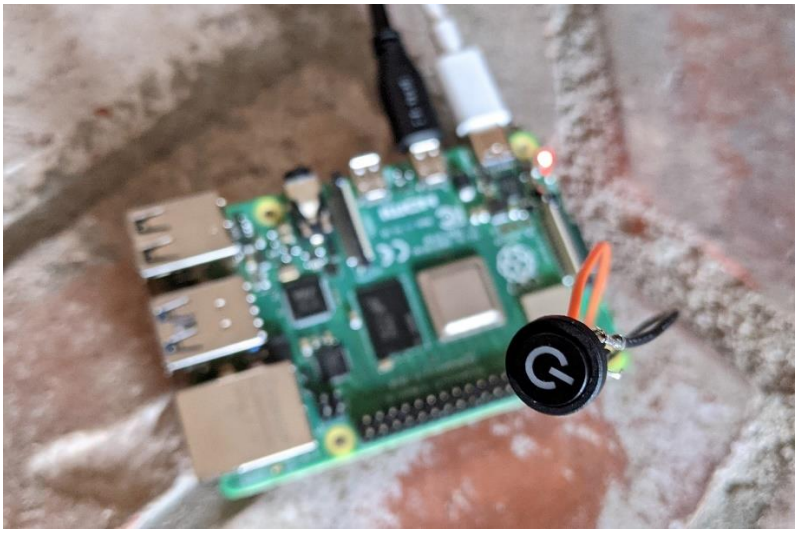


*Figure 25. An external power switch connected to the Raspberry Pi 4.*

The serial connection between the ESP32 and the ELM327 still required debugging, so a Hantek 2D72 handheld oscilloscope was borrowed from Oregon Tech for this purpose. While reading through the ESP32 documentation, it was discovered that the pins used for UART1 are reserved for SPI by default and are not recommended for other uses. After switching to UART0, a proper data signal was captured on the oscilloscope and can be seen in Figure 26 below.

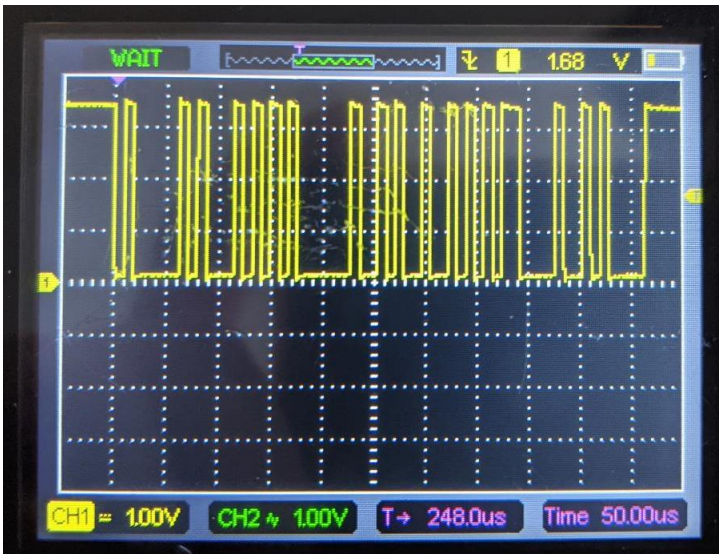



*Figure 26. An image of the serial output from the ESP32.*

The data was initially perceived to be incorrect since the decoded value was gibberish, however after a design discussion, it was realized that the data is sent least significant bit (LSB) first. To confirm this, the UART from an Arduino UNO was used to send the command “AT I”. The decoded oscilloscope reading from the UNO, which verifies the data, can be seen in Figure 27 below.

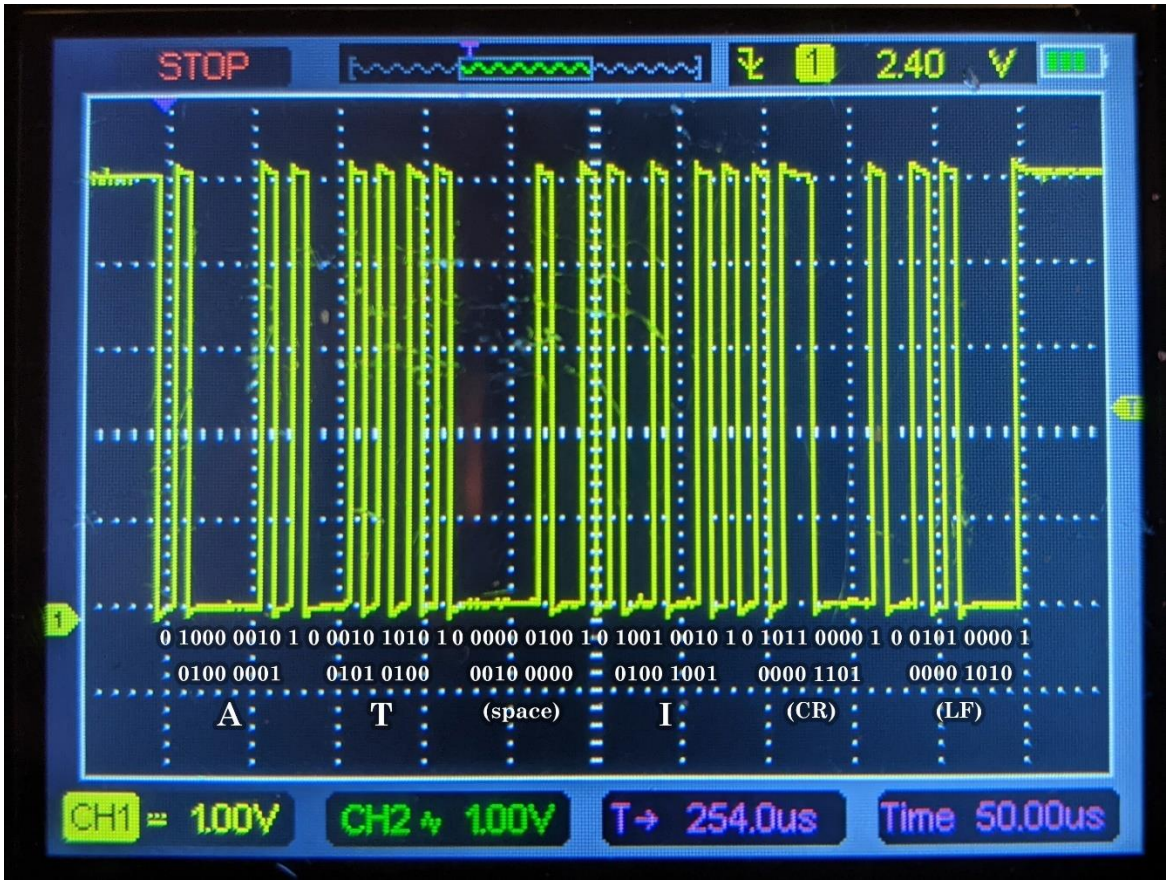



*Figure 27. The decoded oscilloscope reading from the Arduino Uno.*

Once the data was confirmed, the same test was performed on the ESP32 again. The signal produced by the ESP32, while at 3.3V instead of 5V, was identical to the signal produced by the Arduino Uno. This confirmed the ESP32 was producing the correct signal, and the oscilloscope reading can be seen in Figure 28 below.

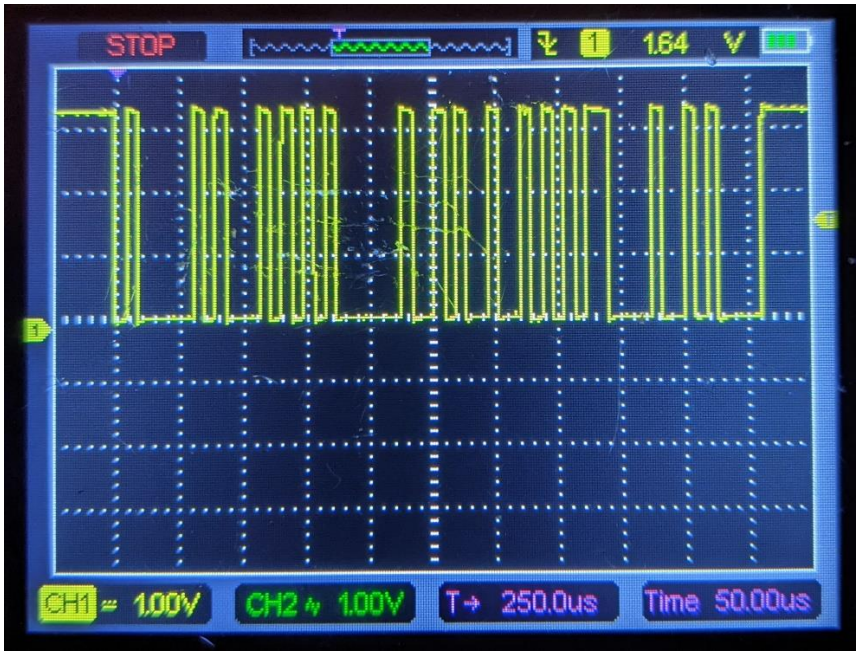



*Figure 28. Another image of the serial output from the ESP32.*

After confirming the ESP32's UART was functioning properly, the signal after 3.3V to 5V translation was tested with the oscilloscope. The signal quality was degraded, but the data appeared to be in tact as seen in Figure 29 below.

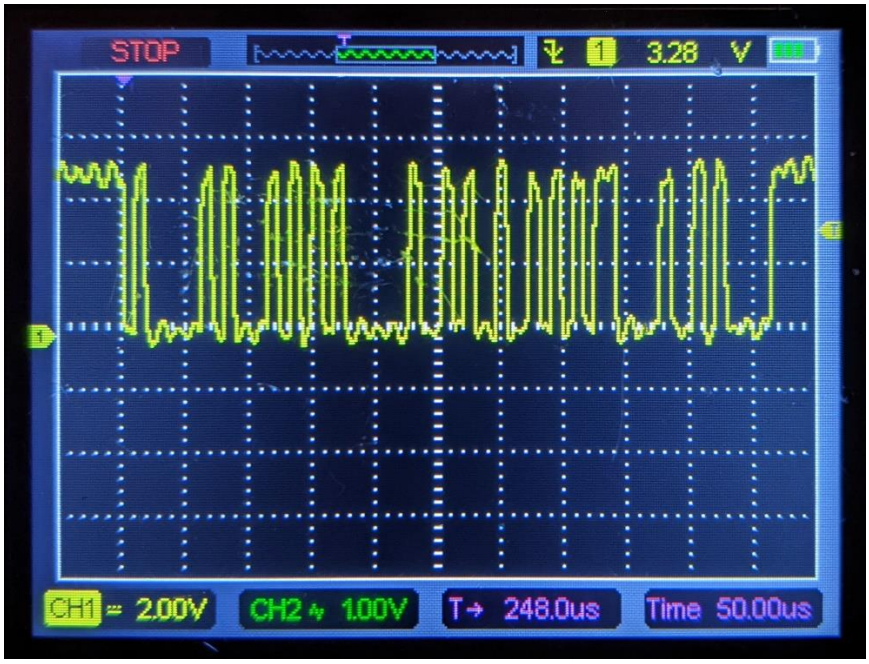



*Figure 29. The serial output from the ESP32 after being translated to 5V.*

Since the ELM327 was receiving the correct data from the ESP32, the ELM327's response was tested with the oscilloscope as seen below in Figure 30.

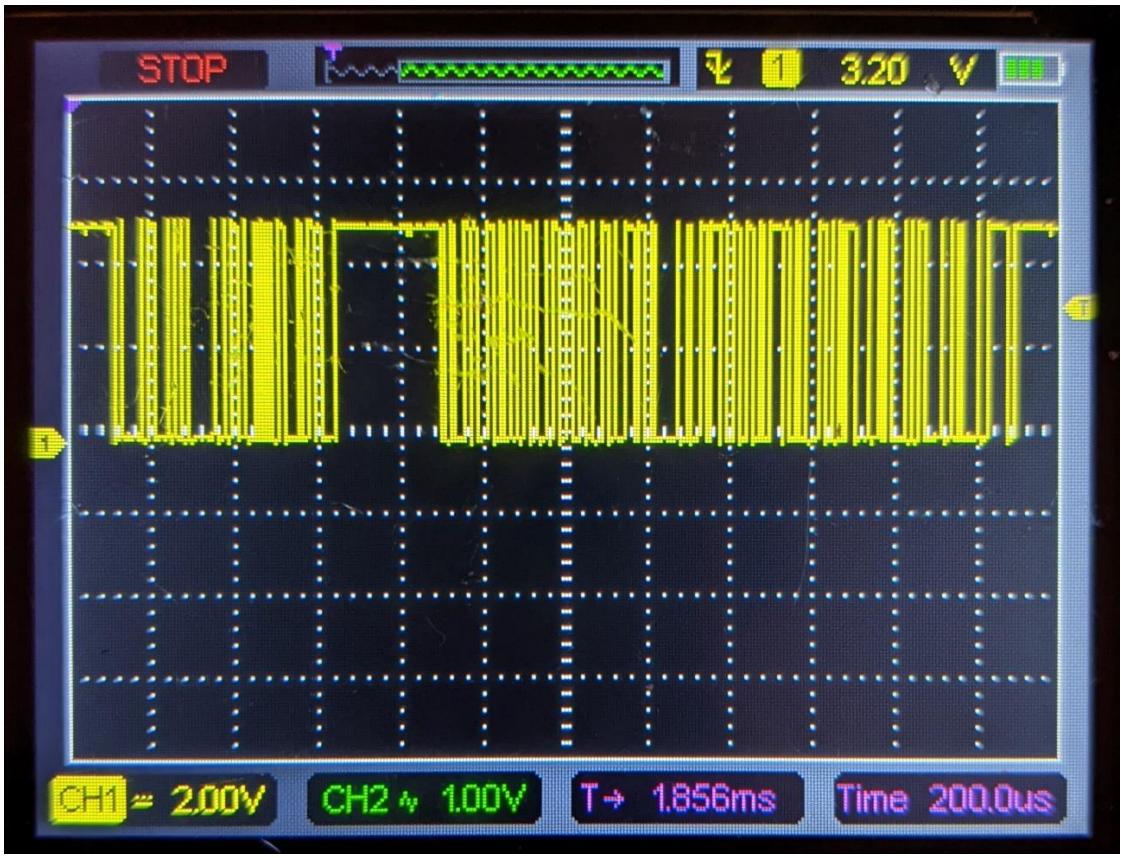



*Figure 30. The ELM327's response to the "AT I" command.*

This data was not decoded since the real point of interest was testing the functionality of the TXB0102 logic level translator. The signal from the ELM327 after translation from 5V to 3.3V appeared to be identical but with lost quality, similar to the 3.3V to 5V translation. The oscilloscope reading of the 5V to 3.3V conversion can be seen in Figure 31 below.

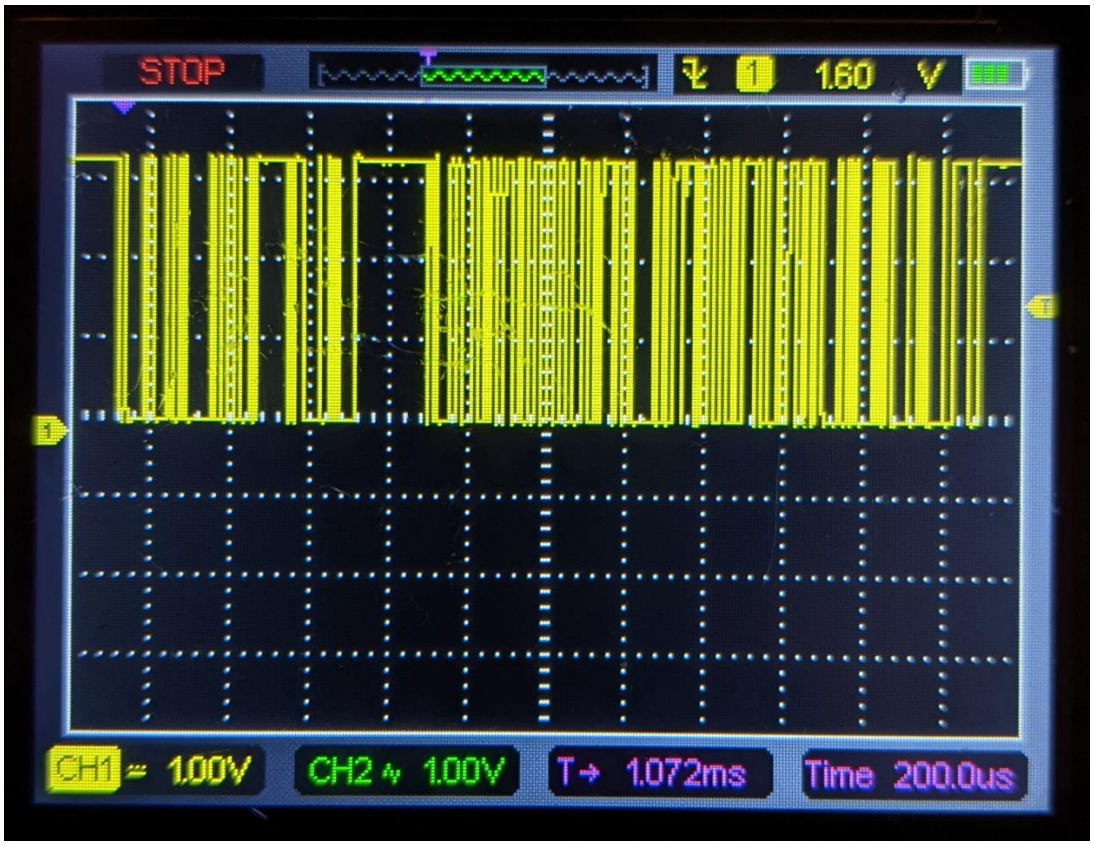



*Figure 31. The ELM327's response after translation to 3.3V.*

Since the logic level translator appears to be functioning correctly, the next task is configuring the Bluetooth connection between the ESP32 and the Raspberry Pi 4 so this data can be transferred wirelessly. The state of the Trello board can be seen in Figure 32 below.

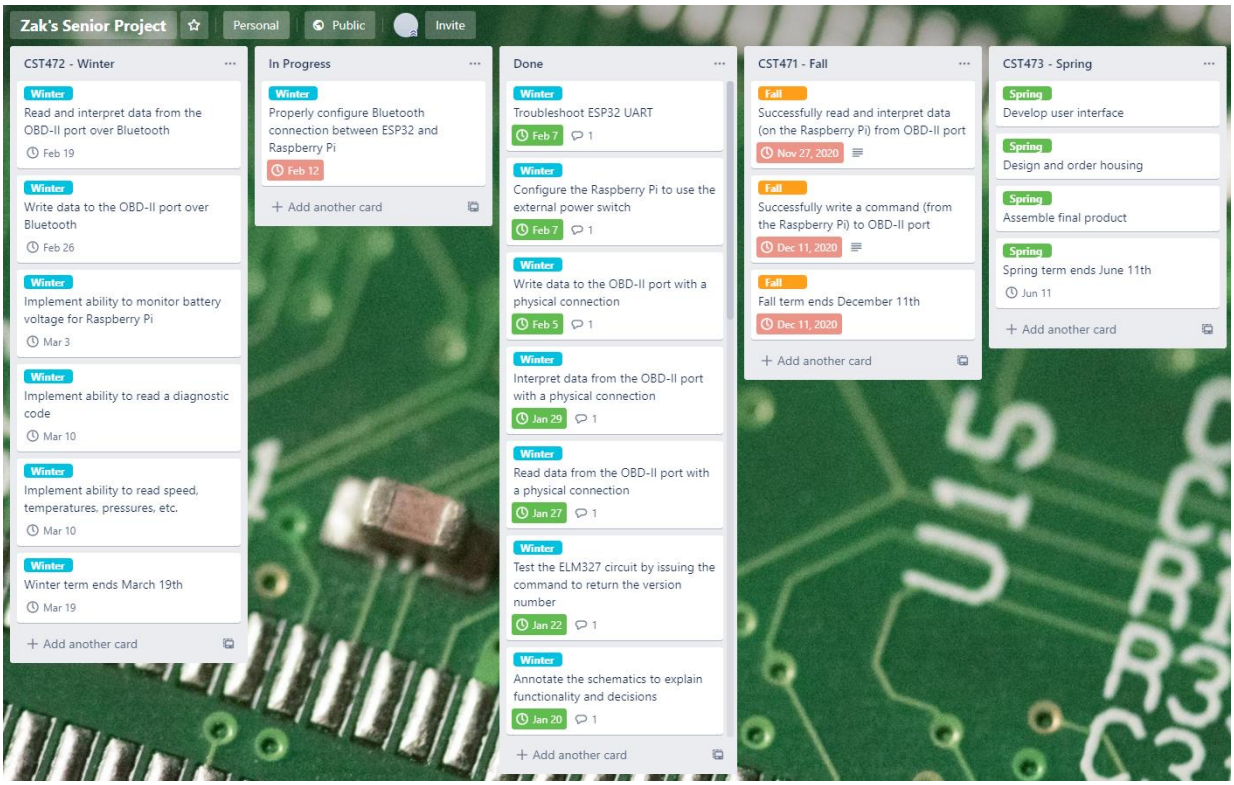



*Figure 32. A screenshot of the Trello board.*

Minimal progress was made during the last three weeks of the term, but some work was done for the Bluetooth connection. A GitHub repository was also created for source control, however it is still empty. After confirming the IDE and ESP32 were still working correctly, research began on Bluetooth Low Energy. A tutorial from Adafruit provided useful information in an easy to understand format [10]. Bluetooth Low Energy uses GAP (Generic Access Profile) and GATT (Generic Attribute Profile) to communicate; GAP specifies how a device (peripheral) advertises itself and how a central device scans and connects to it.

Once the connection is made, the advertising stops and the GATT profile is used to transfer data in transactions. A peripheral device, in this case the ESP32 microcontroller, acts as the GATT server. The GATT client, the Raspberry Pi 4,

initiates the transactions by sending requests to the server. A diagram from Adafruit visualizing this relationship can be seen in Figure 33 below.

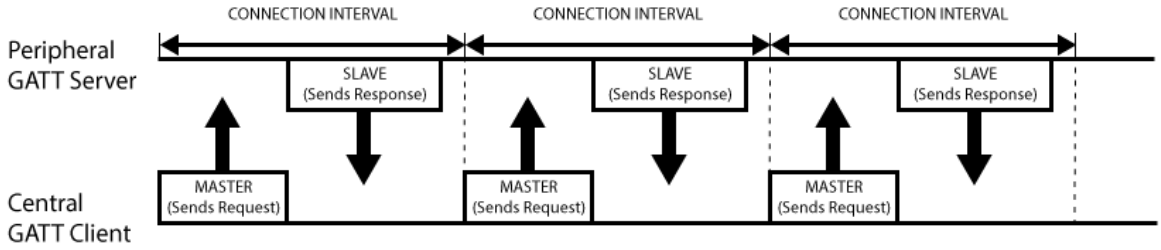



*Figure 33. A diagram of the GATT server/client relationship from Adafruit [10].*

Transactions use nested objects called Profiles, Services, and Characteristics to accomplish data transfer. Profiles are simply a collection of Services. Some Profiles are official and compiled by Bluetooth SIG, but Profiles can also be custom made by the engineer. Services are used to separate data into different groups, and the actual data comes in the form of Characteristics. A diagram from Adafruit of these nested objects can be seen in Figure 34 below.

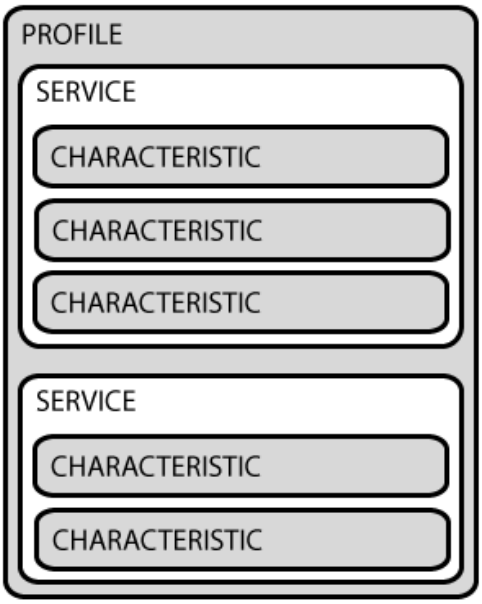



*Figure 34. A diagram of the nested Services and Characteristics from Adafruit [10].*

After learning more on how the protocol works, an example project included in the Espressif extension was used to test the BLE connection. The example is difficult to follow and utilizes FreeRTOS to handle the different tasks, which will require more research to understand. Figure 35 below shows the ESP32 connected to an Android app over BLE.

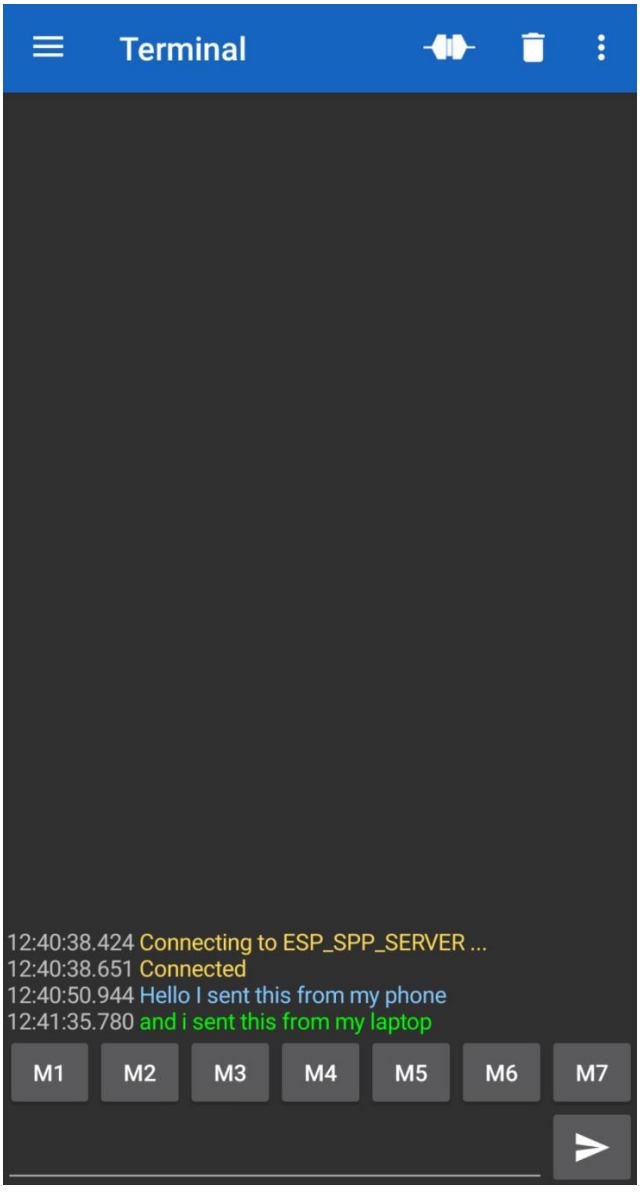



*Figure 35. The ESP32 connected to an Android app over BLE.*

Figure 36 below shows the same connection but from the terminal in Visual Studio Code.

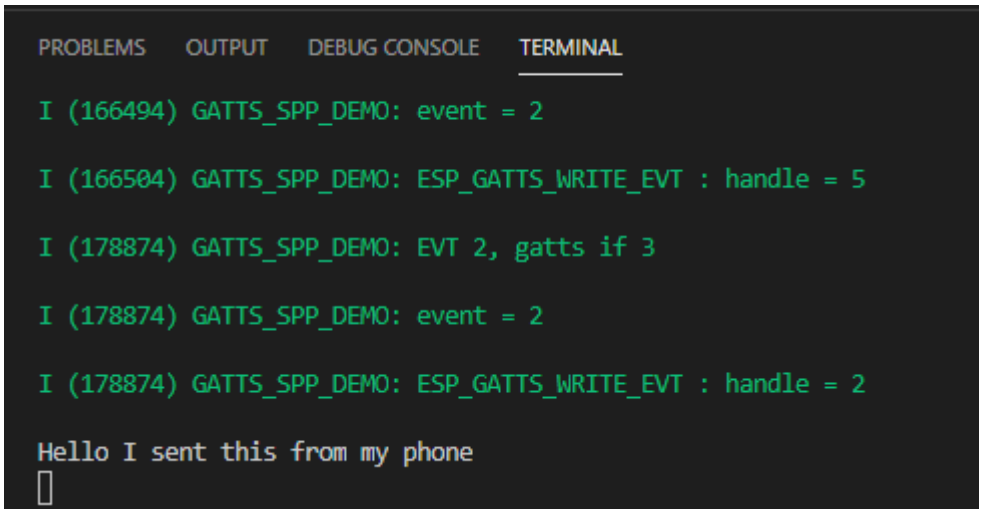



*Figure 36. The terminal in Visual Studio code used to send data over BLE.*

After confirming the example was functioning, it was to be connected to the Raspberry Pi. Before doing so, the existing Raspberry Pi OS was replaced with Raspberry Pi OS Lite. The reason for this was to reduce CPU, memory, and storage use which reduces load times and decreases power consumption. The standard package also includes many services and packages that are unnecessary for this project, including a desktop environment. SSH was also configured on the Pi to allow a remote connection. Since Raspberry Pi OS Lite defaults to the command line, a GUI will have to be added later for use with the touchscreen.

Once the Pi was setup, the battery board was assembled and tested. The board included headers to solder in place for easy access to 5V and ground, and there is also a header for a UART. The battery board, attached to the back of the touchscreen with the battery in between, can be seen in Figure 37 below.

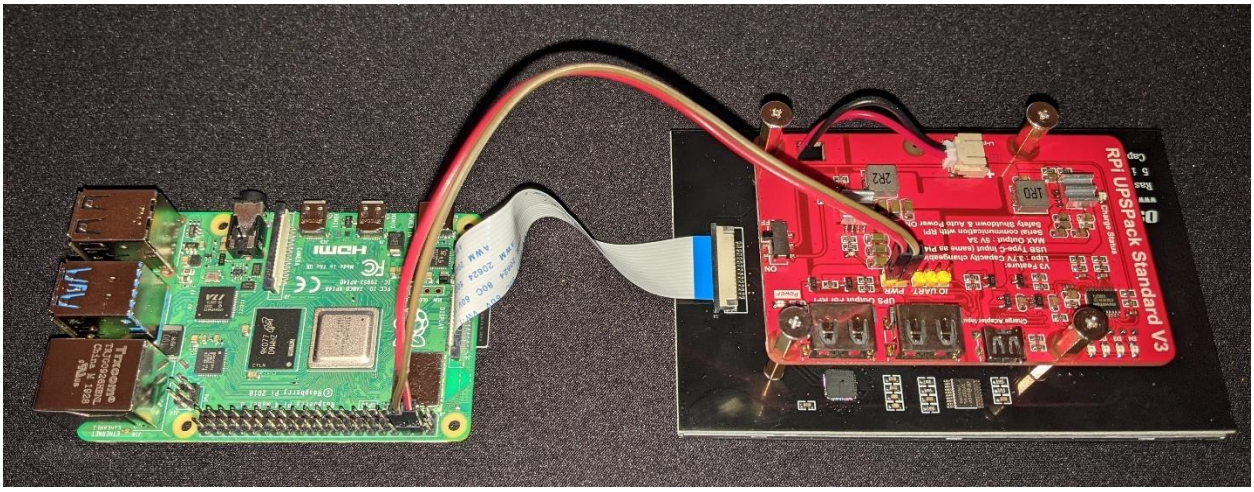



*Figure 37. The assembled battery circuit attached to the Raspberry Pi.*

The board had no issues powering the Raspberry Pi and touchscreen, however having an external keyboard connected caused some undervoltage warnings. Figure 38 below shows the Raspberry Pi and touchscreen powered on with the battery.

*Figure 38. The Raspberry Pi and screen powered by the battery circuit.*

After testing the battery with the Pi, a tutorial from Howchoo was followed on connecting to and communicating with Bluetooth devices [11]. Once scanning

and pairing was complete, a list of available Profiles, Services, and Characteristics from ESP32 was provided as seen in Figure 39 below.

```
pi@raspberrypi: ~
sudo: startx: command not found
pi@raspberrypi:~ $ sudo bluetoothctl
Agent registered
[bluetooth]# scan on
Discovery started
[CHG] Controller DC:A6:32:44:B0:EE Discovering: yes
[NEW] Device 40:F5:20:71:CA:A2 ESP_SPP_SERVER
[NEW] Device 72:96:33:DB:CE:57 72-96-33-DB-CE-57
[bluetooth]# pair 40:F5:20:71:CA:A2
Attempting to pair with 40:F5:20:71:CA:A2
[CHG] Device 40:F5:20:71:CA:A2 Connected: yes
[NEW] Primary Service
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0001
        00001801-0000-1000-8000-00805f9b34fb
        Generic Attribute Profile
[NEW] Characteristic
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0001/char0002
        00002a05-0000-1000-8000-00805f9b34fb
        Service Changed
[NEW] Descriptor
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0001/char0002/desc0004
        00002902-0000-1000-8000-00805f9b34fb
        Client Characteristic Configuration
[NEW] Primary Service
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028
        0000abf0-0000-1000-8000-00805f9b34fb
        Unknown
[NEW] Characteristic
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0029
        0000abf1-0000-1000-8000-00805f9b34fb
        Unknown
[NEW] Characteristic
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char002b
        0000abf2-0000-1000-8000-00805f9b34fb
        Unknown
[NEW] Descriptor
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char002b/desc002d
        00002902-0000-1000-8000-00805f9b34fb
        Client Characteristic Configuration
[NEW] Characteristic
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char002e
        0000abf3-0000-1000-8000-00805f9b34fb
        Unknown
[NEW] Characteristic
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0030
        0000abf4-0000-1000-8000-00805f9b34fb
        Unknown
[NEW] Descriptor
        /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0030/desc0032
        00002902-0000-1000-8000-00805f9b34fb
        Client Characteristic Configuration
```

*Figure 39. Establishing the Bluetooth connection.*

Through trial and error of reading from all available Characteristics, the text "hey" was successfully sent from the Visual Studio Code terminal and read on the Raspberry Pi as seen in Figure 40 below.

```
[ESP_SPP_SERVER:/service0028/char002b/desc002d]# select-attribute /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0029
[ESP_SPP_SERVER:/service0028/char0029]# read
Attempting to read /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0029
[CHG] Attribute /org/bluez/hci0/dev_40_F5_20_71_CA_A2/service0028/char0029 Value:
  68 65 79 0d 0a                                   hey..
  68 65 79 0d 0a                                   hey..
```

*Figure 40. Reading "hey" on the Raspberry Pi over Bluetooth LE*

The next step is to use the provided examples to develop the necessary functionality for the system. The current state of the Trello board can be seen in Figure 41 below.

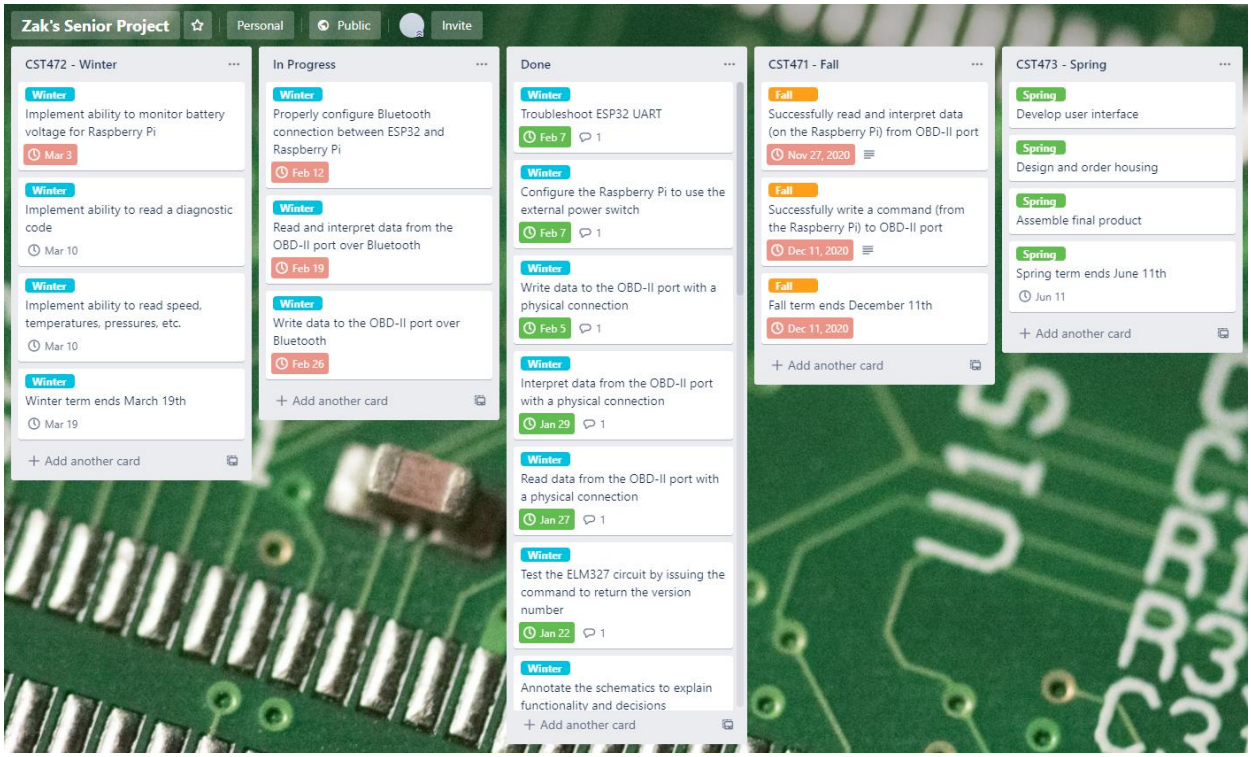



*Figure 41. The current state of the Trello board.*

## Spring Term Problems, Progress, and Discussion

Little progress was made during spring term aside from the required documentation. One of these documents is a testing plan which can be found in Appendix C. Some progress was made towards a GUI for the Raspberry Pi, however it is just a layout. The buttons on screen have no functionality, but it was a good start at getting the different elements laid out such as the list of trouble codes, battery percentage, and controls. Figure 42 below shows the layout of the GUI.

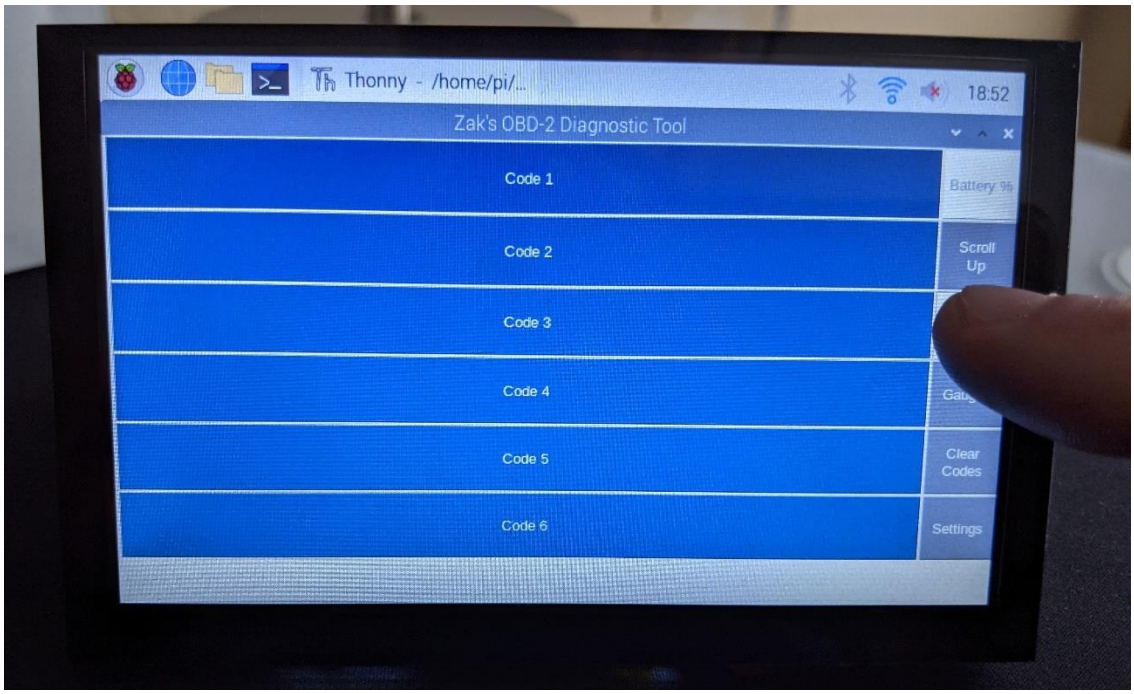



*Figure 42. The layout of the GUI*

The GUI was written in Python using a library called PySimpleGUI. This framework utilizes a simple row and column structure for placing elements. See Appendix B for the full Python script.

## Conclusion

The COVID-19 pandemic created a unique working environment that made realizing the project to its full potential a difficult task. Between no access to proper lab equipment and remote learning, many challenges were met and there was a struggle for motivation. The project is missing two key features that make it unique; the Bluetooth connection and the GUI on the handheld device. In this sense, the project is not successful. However, designing and building the circuits, assembling them, and successfully communicating with the car was a big milestone. The Bluetooth connection was difficult due to the decision to use Visual Studio Code and the Espressif extension. This programming environment provides a lot of control over the hardware, however this level of control was not required for the project. The Arduino IDE should have been used instead.

There are many lessons to be learned from this project. The most important being to try and contribute to the project every day, or at least in some consistent manner. It is easy to assume everything will work smoothly, but that is almost never the case and some work takes hours longer than expected. Another lesson is to reach out for help and communicate more, especially in a remote working environment.

## Requirements

1. The system shall work with the 2008 Nissan Altima OBD-II protocol, ISO15765-4 (CAN-BUS.)
    a. Other vehicles that use the same protocol may work with the system, however it is not required.
2. The system shall be able to read and clear diagnostic (trouble) codes.
    a. The user interface will display the diagnostic codes in list form with buttons to scroll up and down through the list.
        i. The list will display the diagnostic codes (e.g. P0011) only.
        ii. The user must touch one of the diagnostic codes to read the description or possible cause.
    b. The user interface will provide a button to clear all diagnostic codes.
3. The system shall have the ability to read sensor data at minimum 30 times per second including speed, coolant temperature, and oil pressure.
    a. The user interface will display the data in decimal format.
        i. The option for digital gauges may be implemented.
    b. The data will be displayed by default in units of miles per hour for speed, Fahrenheit for temperature, and pounds per square inch for pressure.
        i. The option for metric units shall be implemented.
4. The system shall use Bluetooth 4.0 or greater for data transfer.

    a. The Bluetooth version shall be 4.0 or greater because previous versions do not support Bluetooth Low Energy (BLE.) The range of the connection is also improved with newer versions.

5. The Bluetooth transceiver circuit that interfaces with the OBD-II port shall be powered by the OBD-II port.
    a. A power circuit will be designed to ensure power from the port is reliable.
6. The handheld unit shall use a touchscreen to display information and receive user input.
    a. The touchscreen of the handheld unit shall be 3.5” or greater diagonally.
    b. The touchscreen of the handheld unit shall operate at a resolution of at least 320 x 480 pixels.
7. The user interface shall not require multi-touch or swiping.
    a. Operation with single presses simplifies the user interface and is more accessible to those with disabilities.
    b. The option to scroll through a list by swiping up or down may be implemented.
8. The rechargeable battery shall be recharged via a USB-C port.
9. The rechargeable battery shall power for the handheld unit at full load for at least 2 hours.

    a. The rechargeable battery must include protection and charging circuitry.

10. The charge level of the rechargeable battery shall be displayed on the user interface.
    a. The charge level will be displayed as a percentage, 100% being fully charged.
11. The brightness level of the touchscreen shall be adjustable in the user interface in 10% increments, 100% being maximum brightness.
12. The handheld unit shall include a physical ON/OFF power switch.
13. The system shall use a single board computer or microcontroller.
14. The code and documentation developed shall be open source.

## Satisfied Requirements

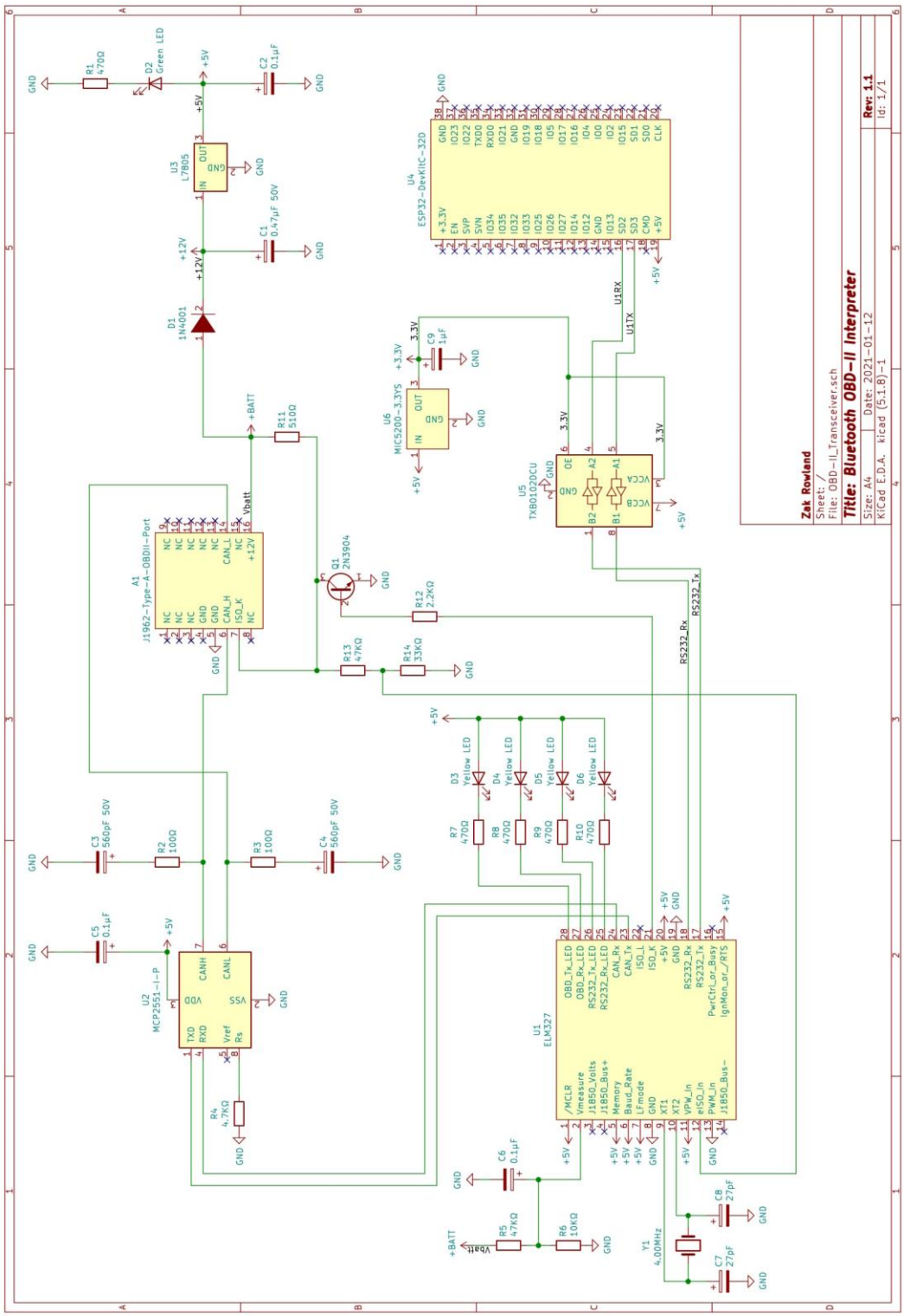
1. Successfully communicated with a 2008 Nissan Altima to read/clear diagnostic codes, read the VIN, and other sensor data.
2. Successfully received and decoded a diagnostic code then cleared it afterwards.
    a. Incomplete.
    b. Incomplete.
3. Incomplete.
    a. Incomplete.
    b. Incomplete.
4. The Raspberry Pi uses Bluetooth 5.0 and the ESP32 uses Bluetooth 4.2.
5. The OBD-II port circuit is powered through the 12V car battery. This voltage is stepped-down through linear regulators to 5V and 3.3V.
    a. The power circuitry includes a diode to prevent current from flowing backwards and multiple capacitors to filter noise.
6. The OSOYOO display purchased for the system is touchscreen and uses a DSI connection.
    a. It is a 5 inch screen.
    b. The resolution is 800 x 480.
7. The GUI layout only requires single button presses.
8. The battery pack purchased for the project is charged via a USB-C port.

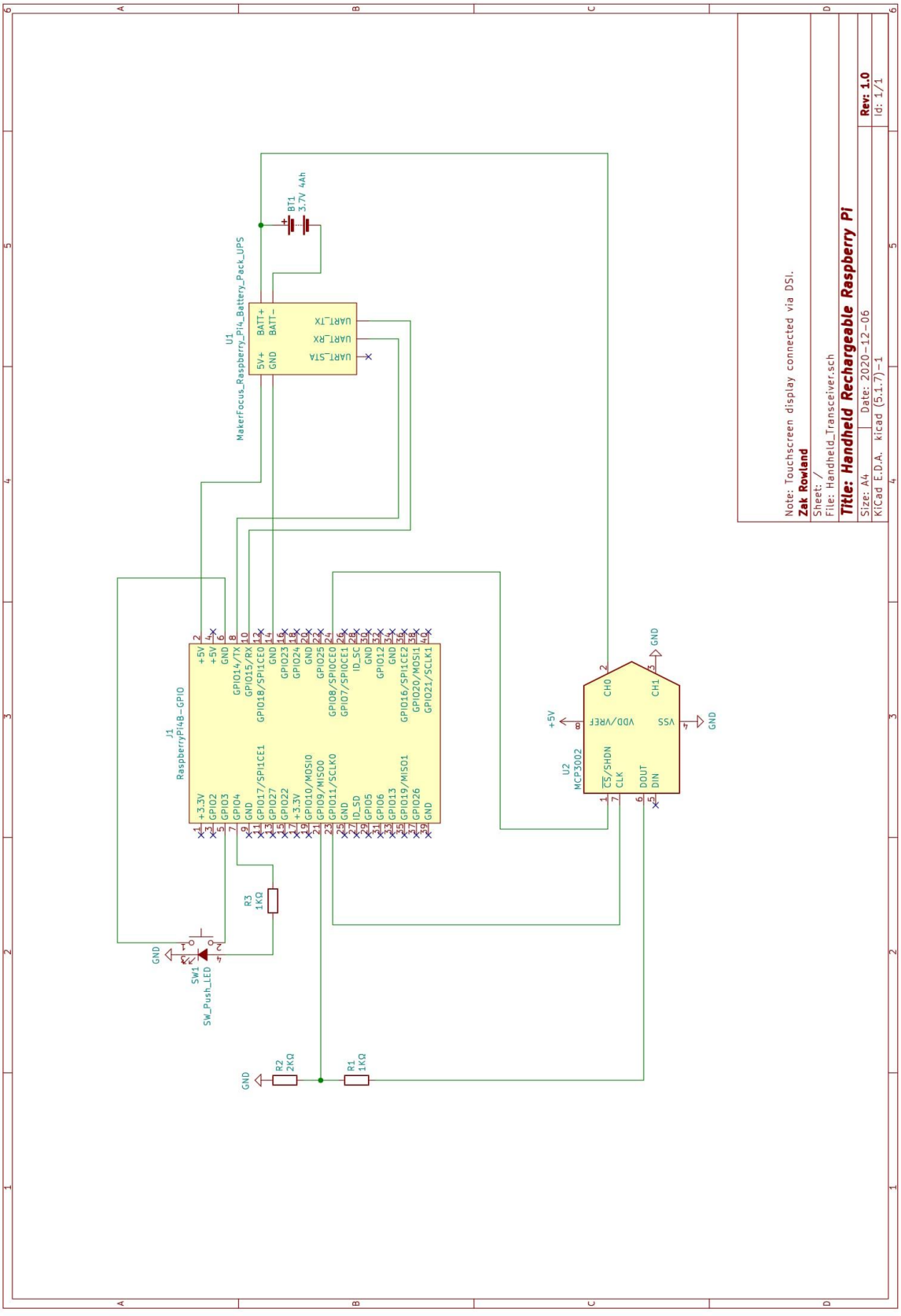
9. The battery pack has a capacity of 4,000mAh, the Raspberry Pi 4B can consume at most 1.8A without USB devices, and the touchscreen display consumes about 200mA. In the worst possible conditions, the Pi would stay powered for a minimum of two hours.
    a. Protection and charging circuitry is included with the battery.
10. Incomplete.
11. Incomplete.
12. An external power switch was connected to specific pins in the GPIO and successfully powered the device on and off with a press.
13. The systems uses both a single board computer, the Raspberry Pi 4B, and a microcontroller, the ESP32.
14. Code and documentation is available at github.com/ZR-OIT/Senior-Project

# Glossary

| **Term** | **Definition** |
|---|---|
| OBD-II | OBD-II stands for On-Board Diagnostics (version 2) and is the interface used in all vehicles model year 1996 or newer. |
| CAN Bus | CAN stands for Controller Area Network and it is the bus protocol used in vehicles and other embedded systems to send and receive data to and from all the various devices (nodes) connected to it. The OBD-II port has connections to the CAN Bus. |

# Appendix A


Zak Rowland
Sheet: /
File: OBD-II_Transceiver.sch
Title: Bluetooth OBD-II Interpreter
Size: A4
Date: 2021-01-12
Rev: 1.1
KiCad E.D.A. kicad (5.1.8)-1
Id: 1/1
U4 ESP32-DevKitC-32D
U1 ELM327
U2 MCP2551-I-P
U5 TXB0102DCU
U6 MIC5200-3.3YS
U3 L7805
A1 J1962-Type-A-OBDII-Port
Q1 2N3904
D1 1N4001
D2 Green LED
D3 Yellow LED
D4 Yellow LED
D5 Yellow LED
D6 Yellow LED
Y1 4.00MHz
U1RX
U1TX
RS232_Rx
RS232_Tx
Vbatt
+BATT
+5V
+3.3V
+12V
GND


Note: Touchscreen display connected via DSI.
Zak Rowland
Sheet: /
File: Handheld_Transceiver.sch
Title: Handheld Rechargeable Raspberry Pi
Size: A4
Date: 2020-12-06
Rev: 1.0
KiCad E.D.A. kicad (5.1.7)-1
Id: 1/1
BT1
3.7V 4Ah
U1
MakerFocus_Raspberry_Pi4_Battery_Pack_UPS
5V+
GND
BATT+
BATT-
UART_STA
UART_RX
UART_TX
J1
RaspberryPi4B-GPIO
U2
MCP3002
VDD/VREF
CH0
CH1
VSS
CS/SHDN
CLK
DOUT
DIN
R1
1KΩ
R2
2KΩ
R3
1KΩ
SW1
SW_Push_LED
GND
+5V
+3.3V

# Appendix B

main.py

```
import PySimpleGUI as sg

# Retrieve screen resolution
screen_width, screen_height = sg.Window.get_screen_size()
#print(screen_width, screen_height)

sg.theme('SandyBeach')

code_button_size = (100, 3)
option_button_size = (30, 3)

# Column layout
codes = [[sg.Button(button_text='Code 1', size=code_button_size)],
         [sg.Button(button_text='Code 2', size=code_button_size)],
         [sg.Button(button_text='Code 3', size=code_button_size)],
         [sg.Button(button_text='Code 4', size=code_button_size)],
         [sg.Button(button_text='Code 5', size=code_button_size)],
         [sg.Button(button_text='Code 6', size=code_button_size)]]

options = [[sg.Button(button_text='Battery %', disabled=True, button_color=('Black', 'White'),
 size=option_button_size)],
          [sg.Button(button_text='Scroll\nUp', button_color=('White', 'Grey'), size=
          option_button_size)],
          [sg.Button(button_text='Scroll\nDown', button_color=('White', 'Grey'), size=
          option_button_size)],
          [sg.Button(button_text='Gauges', button_color=('White', 'Grey'), size=
          option_button_size)],
          [sg.Button(button_text='Clear\nCodes', button_color=('White', 'Grey'), size=
          option_button_size)],
          [sg.Button(button_text='Settings', button_color=('White', 'Grey'), size=
          option_button_size)]]

# Window layout
layout = [[sg.Column(codes, element_justification='l'), sg.Column(options,
element_justification='r')]]

# Display the window and get values
window = sg.Window('Zak\'s OBD-2 Diagnostic Tool', layout, size=(screen_width, screen_height),
 margins=(0,0), element_padding=(0,0), no_titlebar=True)

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Settings':
        break

window.close()
```

# Appendix C

## Test Plan

To test the first requirement, the OBD-II circuit should be able to successfully communicate with a 2008 Nissan Altima. A USB to serial device can be used to send commands and receive responses through a terminal such as PuTTY running on a laptop computer. The system should be able to request data and clear diagnostic codes. An oscilloscope may be required to monitor serial lines.

The second requirement can be tested by issuing the commands to read and clear trouble codes. The quantity of trouble codes present and a list of them should be successfully received and decoded. To test the user interface, tap all available buttons and ensure they function as intended. Also ensure the trouble codes are displayed as a list and can be selected for a better description.

To test the third requirement, the Bluetooth connection should be monitored as it attempts to send commands and receive data 30 times per second. This could be accomplished by printing the appropriate variables at the same rate and ensuring the values change each time.

To verify that the Bluetooth devices use version 4.0 or later for the fourth requirement, the datasheets of said devices should be checked.

The fifth requirement is tested by using a multimeter to ensure there is 12V from the car's battery, 5V after the regulator, and 3.3V after the regulator. An oscilloscope can be used to monitor the noise and ripple to ensure the power source is sufficient.

Testing the sixth requirement requires checking the screen manufacturer's datasheet to ensure all specifications are met.

To test the seventh requirement, the GUI should be thoroughly tested to ensure it can be operated with only taps or button presses.

The eighth requirement is tested by viewing the schematic of the system if a charging module is custom designed, or by viewing the datasheet from the manufacturer, to ensure a USB-C port is utilized.

To test the ninth requirement, the ESP32 could be powered by a computer and a Bluetooth connection will be established with the Raspberry Pi. With a timer, the running time will be measured with the handheld device constantly transferring Bluetooth data with the screen at full brightness. To ensure the rechargeable battery includes proper protection and charging circuitry, check the datasheet.

Testing the tenth requirement involves watching the user interface and verifying the battery percentage falls accurately as the device stays powered. The status should be 100% at full charge, and the device should power off once the percentage reaches 0.

To test the eleventh requirement, verify there are buttons to adjust the brightness in 10% increments. Tap these buttons and ensure the brightness changes accordingly each time.

The twelfth requirement can be tested by pressing the power switch when the device is off and ensuring it boots up, as well as pressing the power switch when the device is on and ensuring it shuts down.

The thirteenth requirement can be tested by checking the schematics to ensure either a single board computer or microcontroller is used in the system.

The last requirement can be tested by viewing the GitHub repository and verifying all documentation and code is open source.

## Memos

**CST471**

# Memo

**To:** Kevin Pintong

**From:** Zak Rowland

**Date:** October 9, 2020

**Re:** Memo 1

---

The first few weeks of progress has mainly involved researching and deciding on parts that I will likely use. Before the pandemic hit, I wanted to go for the lowest cost parts available that would still satisfy the requirements in order to keep the project affordable for anyone. Since cheap parts usually don't have the best documentation, the class was advised against doing this and instead going for the most commonly used and well documented parts. This factor has removed one of the big purposes of my project, which is keeping the cost as low as possible, but I think the benefits of completing the project remain the same. I plan on finding theoretical low-cost alternatives to the parts I will be using to highlight what I likely would have used if access to lab equipment was permitted as usual.

In these first three weeks of the term, I have worked on the project for about 11 hours and 15 minutes. The first week, I spent 3 hours revising the requirements and schedule from last term, as well as developing a work deliverables agreement. The second week I spent more time researching parts, had a short meeting with Kevin, began building a parts table, and revised the work deliverables agreement. This progress from the second week took about 6 hours and 15 minutes. Finally, this week I spent more time researching parts, and ordered the ELM327 OBD-II to RS232 interpreter IC as well as a 5-inch touchscreen with a DSI cable. I want to develop my schematics before ordering any other parts, but these parts are essential and will not change so I wanted to get them shipped to start using and testing. I also got a MicroSD to store the Raspberry Pi's operating system.

During the second week, I spent a lot of time researching all the different Bluetooth modules and thought would settle on the HC-06 module. However, after reviewing the part documentation and my own requirements, the HC-06 uses Bluetooth 2.0 and my requirements require Bluetooth 4.0+ for Bluetooth Low Energy (BLE). I could take the "easy" route and remove the 4.0+ requirement, however I want to learn how to use BLE. The HM-19 is Bluetooth 5.0 (which means BLE is supported,) and the pinout is the same as all the other common Bluetooth modules like HC-05 or -06, so I think this is the one I will finally settle on. However, I

am going to wait until I work on schematics before I order a Bluetooth module to be sure I get one that will suit my needs. Another problem I noticed is that according to the ELM327 datasheet, it uses 5V logic for RX/TX instead of 3.3V, so I will likely need a logic level adjusting chip to go along with the Bluetooth module I order (recommendations from Elm are included in the datasheet) unless it can handle 5V logic, which most don't from what I have seen.

Over the next week, I plan to start tinkering with the Raspberry Pi 4 and becoming familiar with how to use it and develop for it. This should help me decide the best operating system to use for the project. If time permits, I will start working on schematics as well.

Schedule / project management (Trello):

https://trello.com/b/57kp2fr0/zaks-senior-project

Work-in-progress parts table (Excel – View Only):

https://oregontech-my.sharepoint.com/:x:/g/personal/zak_rowland_oit_edu/ESNJ9B0jKnFEjfNFRXb084MBa_BKUOOE3Skfk8lLYKfS-Q?e=bcle9j

CST471

# Memo

**To:** Kevin Pintong

**From:** Zak Rowland

**Date:** October 23, 2020

**Re:** Memo 2

---

During the past two weeks, I haven't made much progress. Last week, I used the Raspberry Pi 4 for the first time. I installed Raspberry Pi OS onto the SD card and booted it up. While it was updating, I noticed an undervoltage warning appeared quite a lot. Hopefully supplying power isn't an issue, however I can eventually design the battery circuit to supply enough.

I looked through the OS and tried out all the included applications including the Thonny Python IDE where tested "hello world." I began thinking about how making a GUI would work so I started the Python GUI tutorial on the official Pi website. I had to install a library called "guizero." I was able to make a window with a title and text inside, and it seems it will be simple to make a text-based GUI. However, animations for something like a digital gauge could be tricky.

I also worked on the preliminary design discussion and almost finished all the material needed for it. My next priority is finishing the preparation for the discussion and start making schematics. I'm worried making schematics will be a struggle since I'm getting pretty burnt out from the term already.

Total time worked: 4 hours

Schedule / project management (Trello):

https://trello.com/b/57kp2fr0/zaks-senior-project

CST471

# Memo

**To:** Kevin Pintong

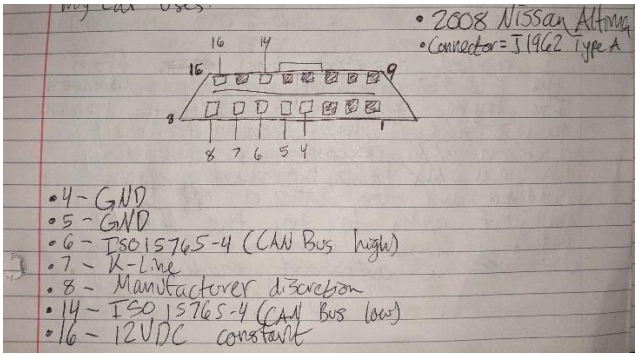
**From:** Zak Rowland

**Date:** November 24, 2020

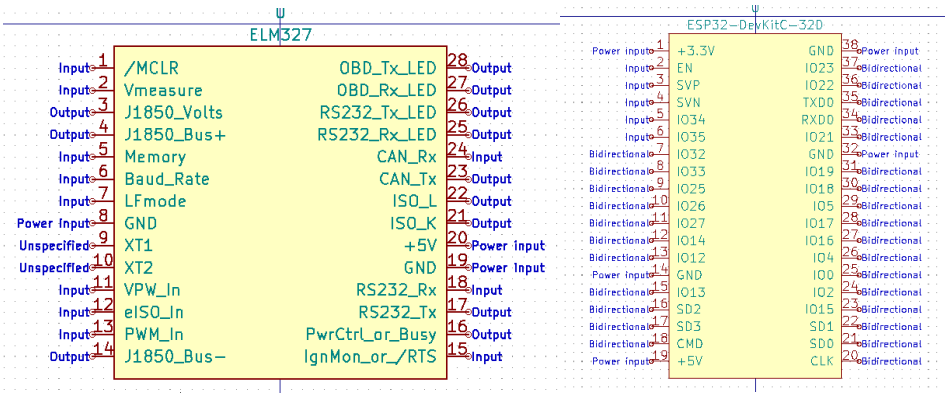
**Re:** Memo 3

---

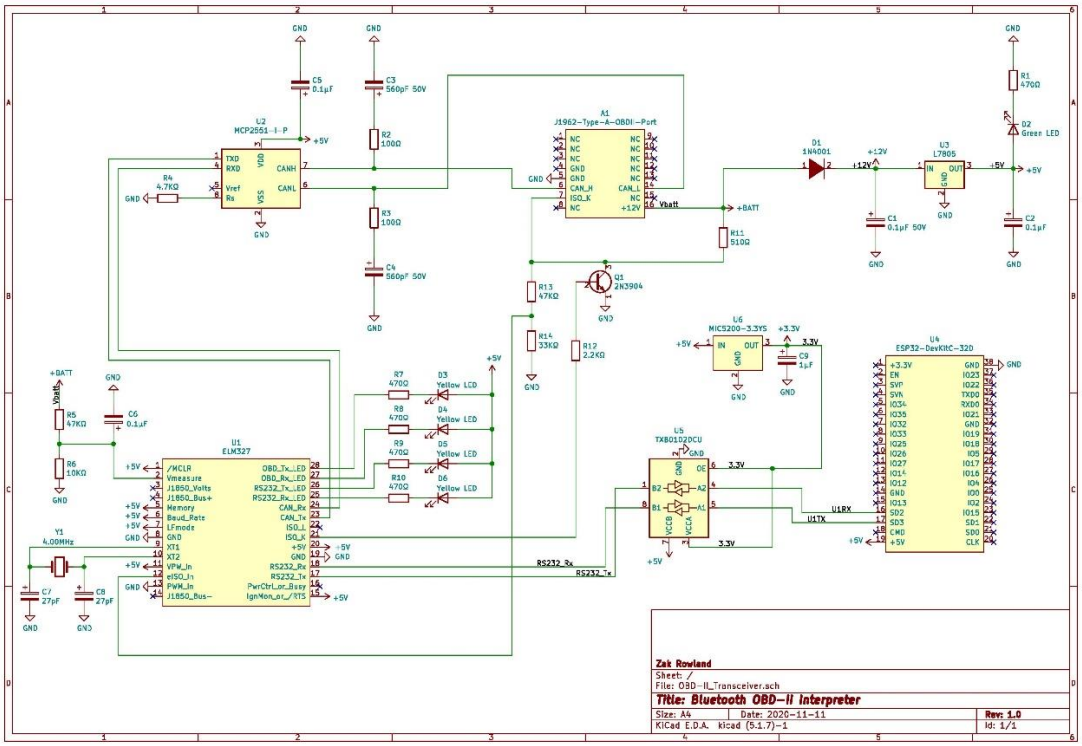
Over the past month since the last memo, I have made good progress. While working on preliminary design review documentation, I verified the pinout of my car's OBD-II port as seen in the image below.




I needed to verify this pinout because depending on the vehicle, some pins of the ELM327 don't need to be used. I began developing a library of schematic symbols I will need for the project including the ELM327, OBD-II port, ESP32, and others. Examples of these symbols can be seen in the images below.




Once I had the ELM327 symbol, I began working on the schematic for the OBD-II port module. The ELM327 datasheet provides a diagram of how the chip could be wired, so that helped a lot when creating the schematic. One problem I had was choosing a Bluetooth module. I compared many different modules including the HC-05, HC-06, HM-11, HM-19, and others, but I just could not come to a confident decision on the module to use. Because of this, I decided to go with the ESP32 because I have used it before, it has Bluetooth 4.2, and is the same price as the modules mentioned previously. After fixing some symbols in my library, choosing a few more parts, and making the necessary connections, the schematic was finished as seen in the image below.




Once the OBD-II port module's schematic was finished, I began looking into parts for the handheld module. This schematic will be much easier to create since there are far fewer parts. I need to decide on a power switch, battery, battery charger, and an A/D converter to monitor battery voltage. I already have some options in mind from Adafruit, but I don't want to

go with them just yet until I compare to other options. Since I haven't decided on these parts, this schematic doesn't have much progress besides a symbol for the RPi4 GPIO header.

The past week or so has involved mostly working on the draft report and design review presentation. Today I am finishing ordering all parts I'll need for the OBD-II port module. Although I made good progress, I am still slightly behind schedule concerning finishing schematics and ordering parts. If the parts come in a reasonable time, I'll be able to finish most or all of my deliverables for the term.

Total time worked: 35 hours

Schedule / project management (Trello):

https://trello.com/b/57kp2fr0/zaks-senior-project

CST 472

# Memo

**To:** Kevin Pintong

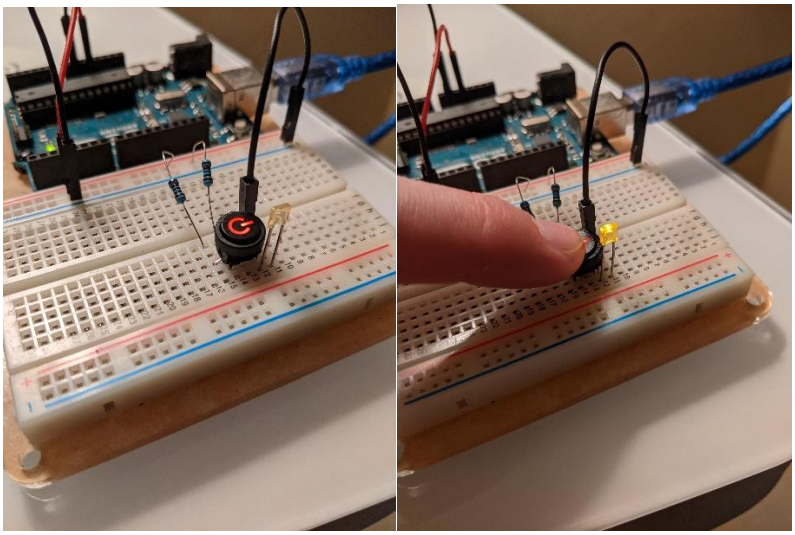
**From:** Zak Rowland

**Date:** January 23, 2021

**Re:** Memo 1

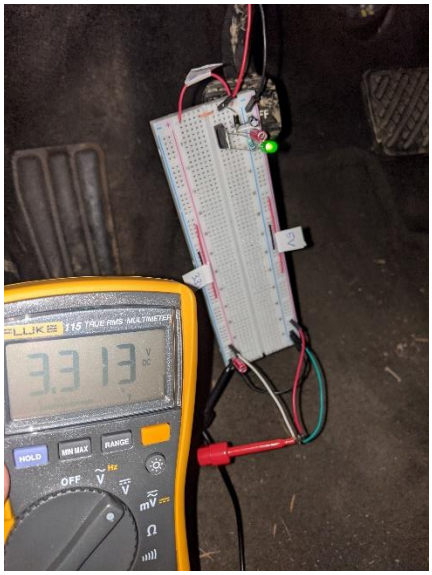
---

After last term ended, I wasn't working on the project for almost a month as I was visiting family. Once the term started again, I would dive back into the project. But, motivation has been extremely hard to come by, and not just in this class. The first thing I worked on was testing resistors and capacitors with a multimeter as well as the LEDs and push button. The push button I got for the Pi has a LED built in, so I tested the external LEDs using this button and an Arduino for power.

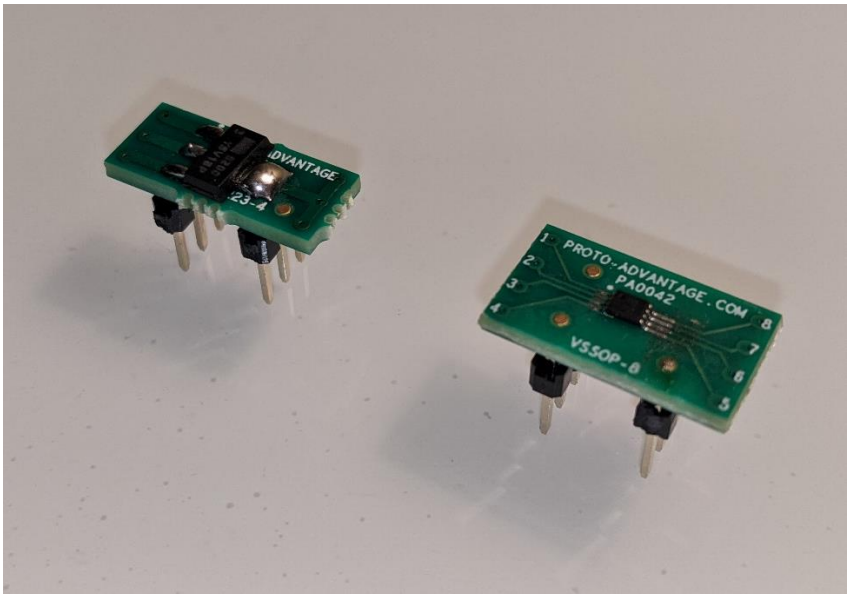


While testing and looking through my parts, I realized I forgot to order surface-mount to dual-inline-pin adapters for my small ICs that will allow me to breadboard the circuit easier, so I ordered some. After meeting with you and discussing my schematic, I changed the 5V regulator to use a bigger input capacitor and began assembling the power delivery portions of the circuit. I built and tested the 12V to 5V circuitry first which worked as expected. I also attached breadboard wires to my OBD-II connector.

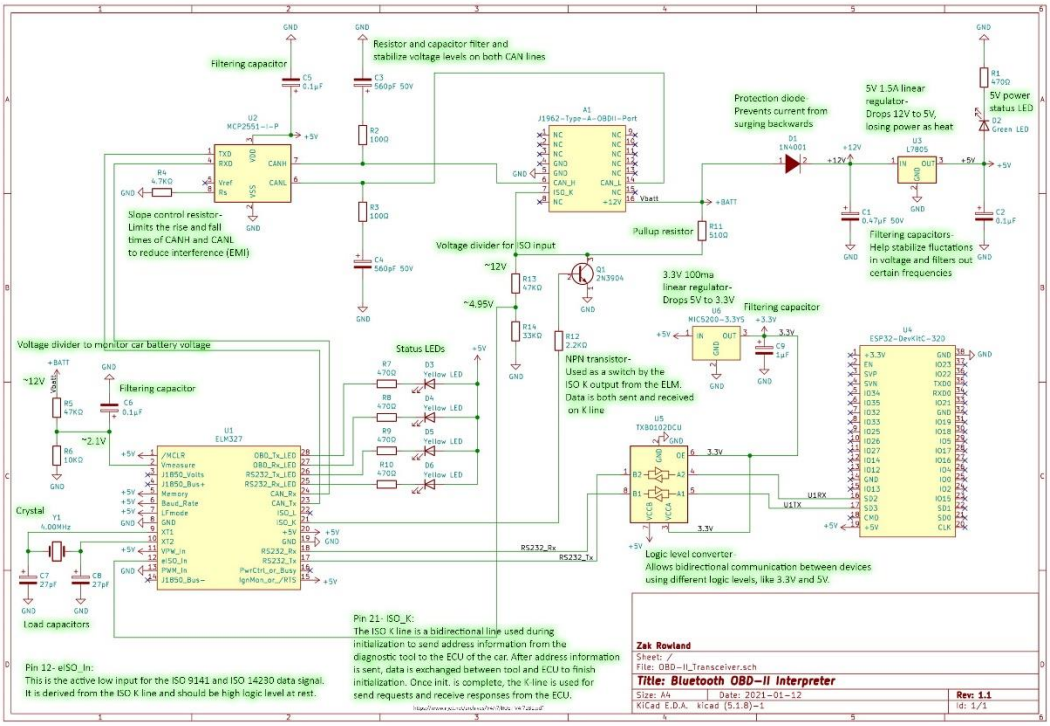

Once I had 5V power, I added and tested the 3.3V circuitry which also worked as expected.



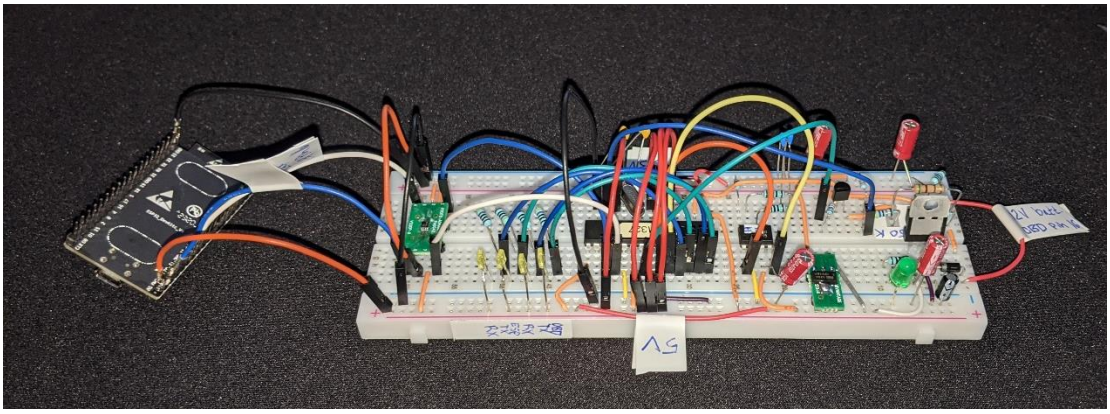

After I had the power circuitry, I began assembling the rest of the circuit. Once the SMD to DIP adapters came, I soldered them together with the ICs. I had trouble at first soldering the legs of the tiny 8-pin package (I used too much solder,) but eventually got the first side soldered and the other side went much smoother. The excess heat from messing up could have damaged the IC but it is unlikely.

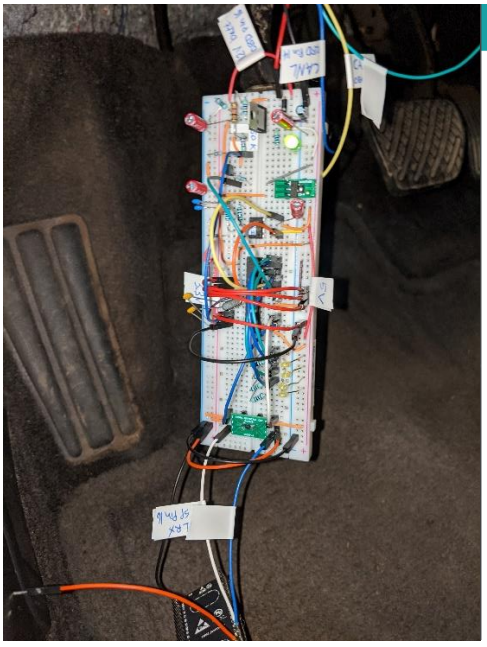



While building the circuit, I annotated the schematic to make sure myself and others understand how it functions and what things are for.




Once the circuit was assembled, I tried to test the ELM327 by reading the version number through a terminal. I used the UART of the ESP32 using the correct settings, however the data being read is complete gibberish, so I cannot confirm the circuit works yet. One good sign is that the four status LEDs light up in sequence when power is provided to the circuit, which as explained in the ELM327 datasheet means the chip is initializing.





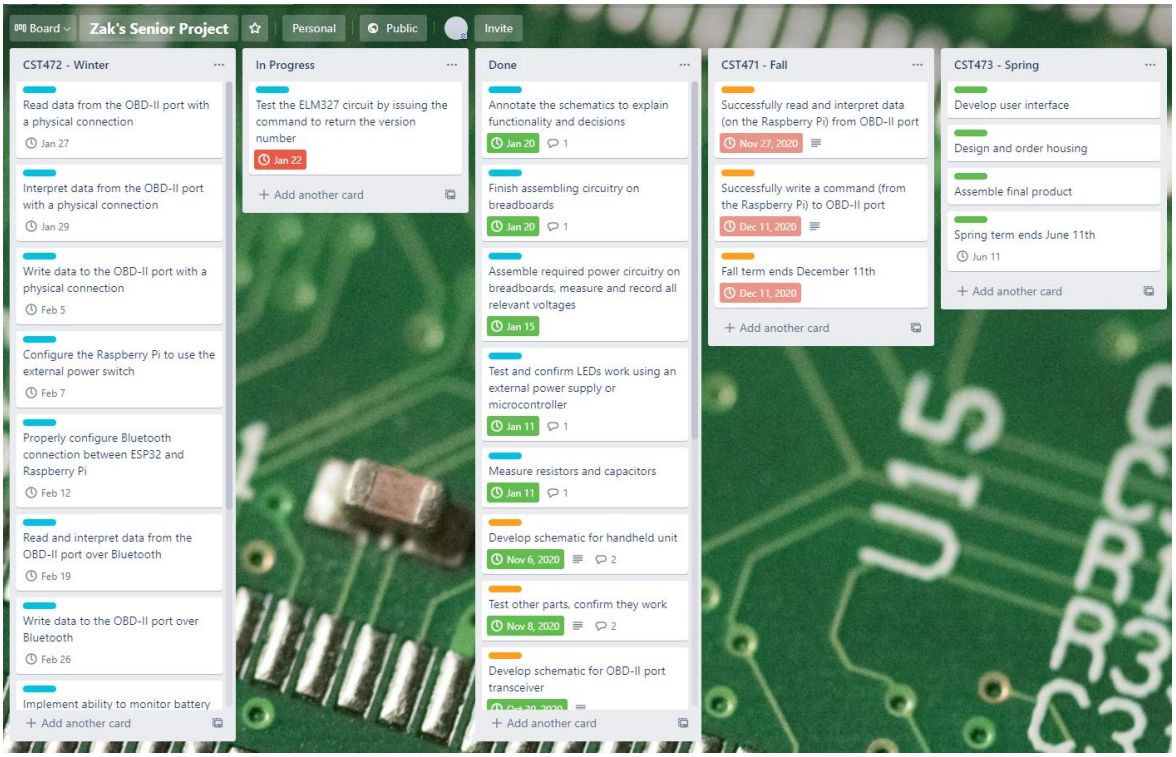
```
ELM_test_1
char buffer[64];
volatile byte count = 0;

void setup() {
  // put your setup code here, to run once:
  Serial1.begin(38400);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial1.available() > 0){ //if there is data to be read
    byte temp = Serial1.read();
    if(count < sizeof buffer){
      buffer[count++] = temp;
      if(temp == '\r'){
        count = 0;
        Serial1.print(buffer);
      }
    }
  }
  //else{
    //Serial1.print("AT Z\r"); //Reset command
  //}
}
```

The next step is obviously to debug the circuit. One way to do this is connecting the ELM327 directly to a RS232 to USB adapter (which I have already) to test the version number, bypassing the ESP32 for now. I should also go back through the circuit and double check everything is connected properly, measuring voltages if I need to. Debugging this circuit may be tough without an oscilloscope, so I could have to order one.

Hours worked since term start: 14




https://trello.com/b/57kp2fr0/zaks-senior-project

**CST 472**

# Memo

**To:** Kevin Pintong

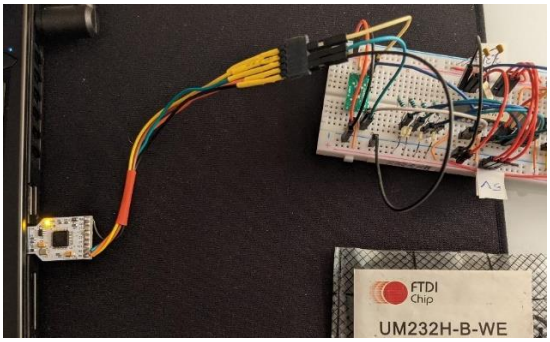
**From:** Zak Rowland

**Date:** February 6, 2021

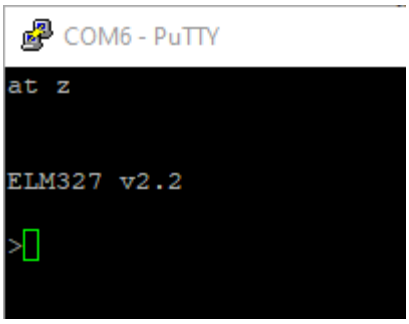
**Re:** Memo 2

---

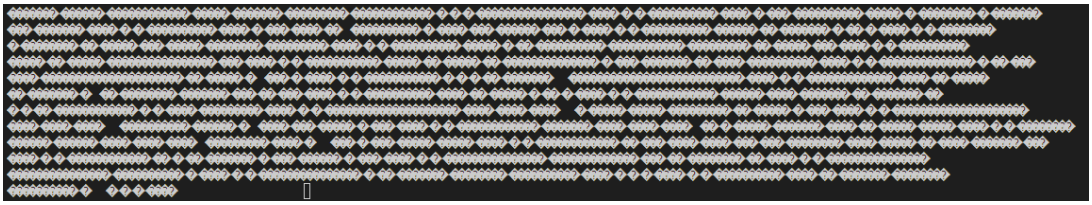
Since I would be doing some debugging on my circuit, I found a 15V wall power brick I could use in place of my car's battery. The 7805 regulator I'm using can tolerate up to about 30V so this worked out just fine. I disconnected the ELM327 chip from power and measured the 5V and 3.3V sources again to confirm nothing broke in the power circuit. Both sources were fine so I plugged the ELM back in. I used a serial to USB device from a previous course to communicate with the ELM through a terminal.



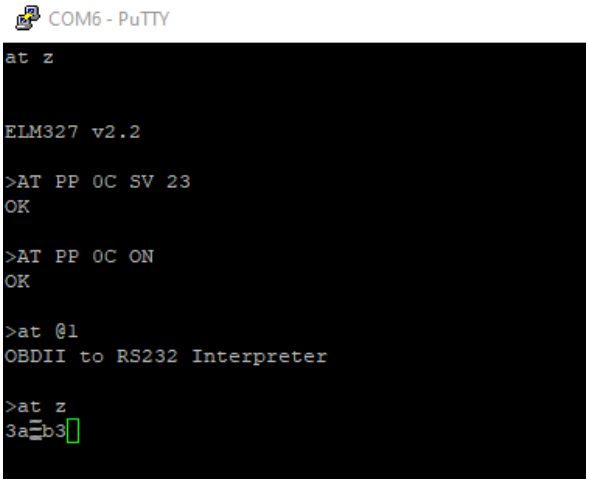

After installing PuTTY and setting up the serial settings, I was able to issue commands and view responses from the ELM327.




This confirms the rest of the circuit is working is expected, but the issue lies in the ESP32's UART or the logic level conversion chip. The logic level conversion chip is the one I may have overheated while soldering. I set up a simple UART configuration in the Arduino IDE and also tried an example UART code in the Visual Studio Code ESP extension, both ended in the same result of garbage data in the serial monitor.

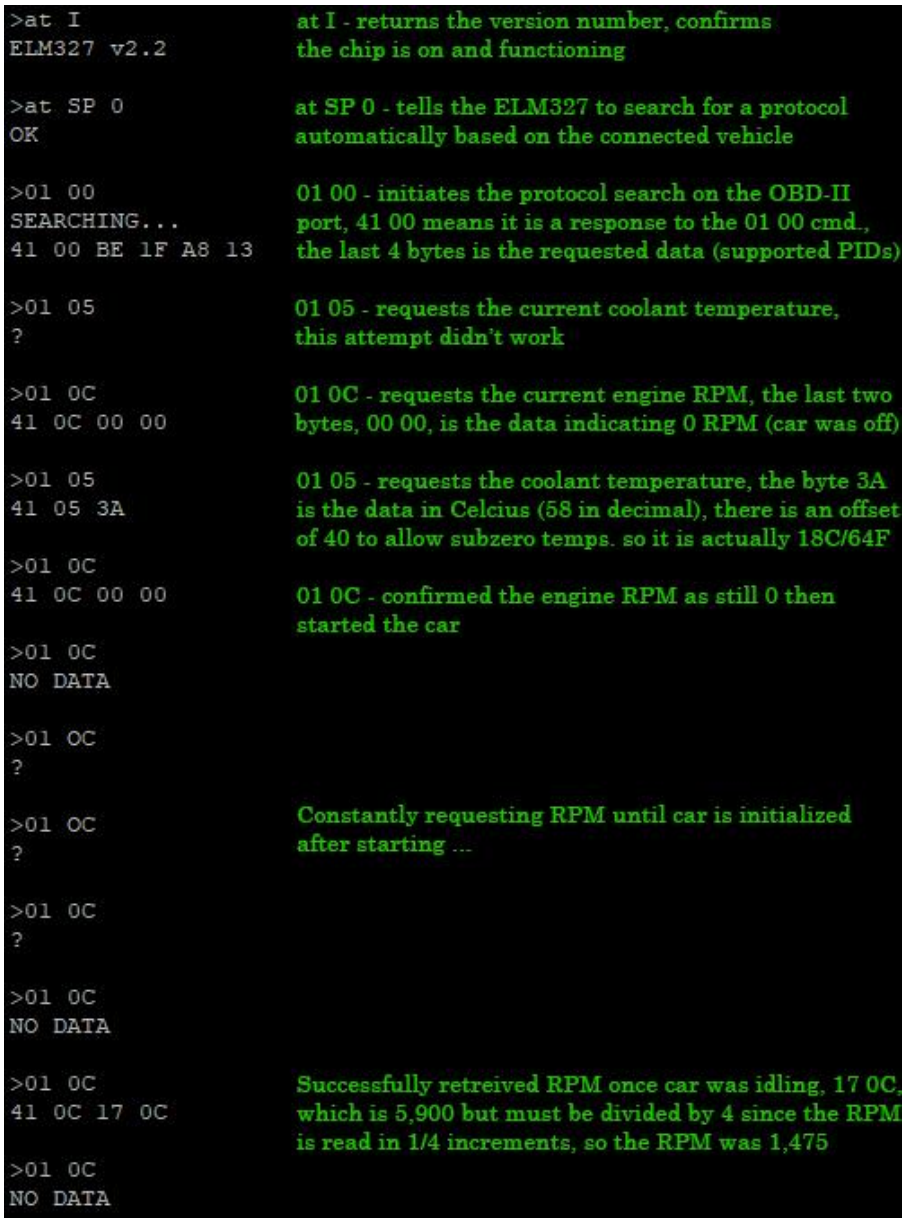


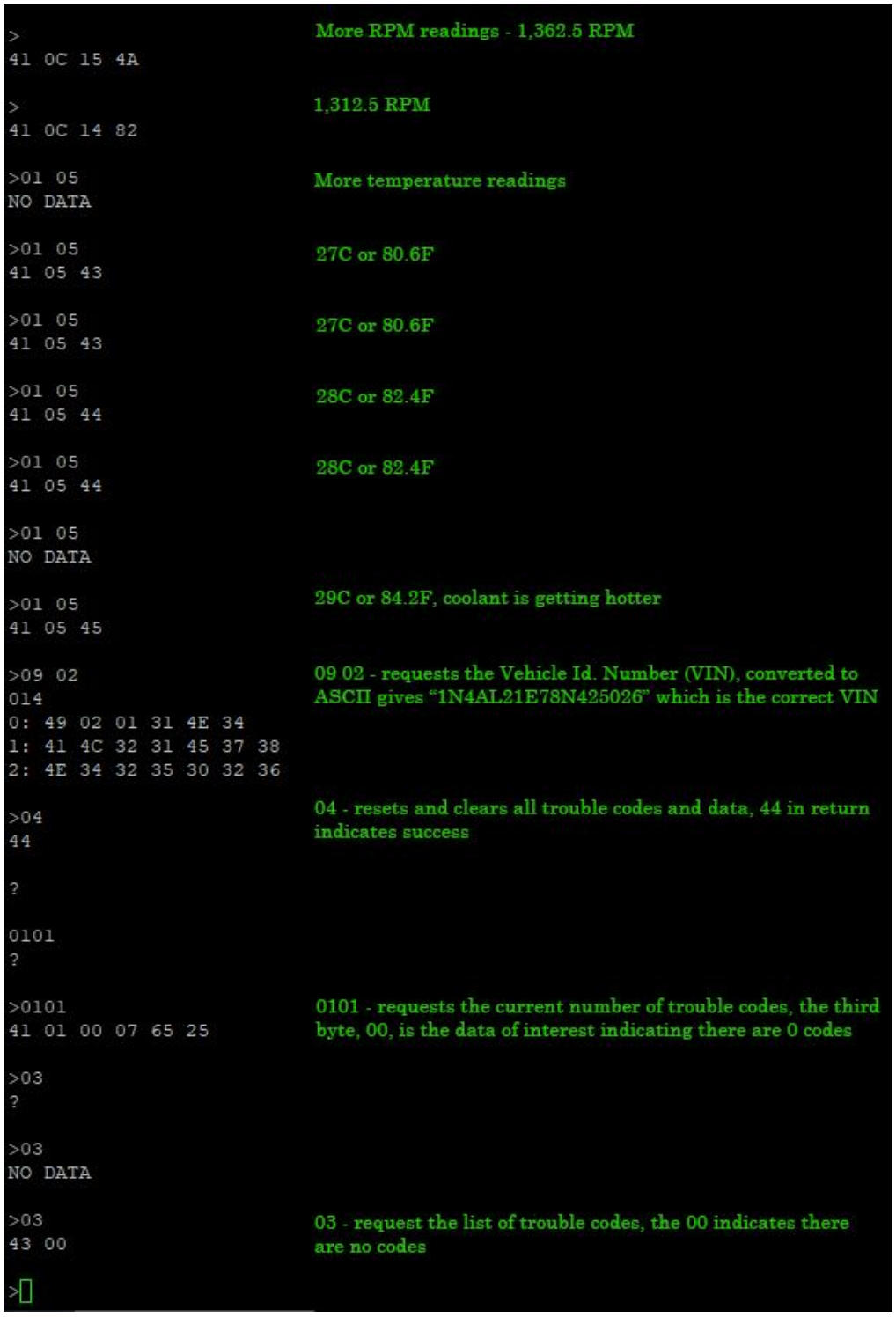
One weird thing I noticed was when I commented out all reads and writes to the UART, the serial monitor still filled with gibberish – indicating something else may be going on. I also configured the ELM327 to use 115.2K baud instead of the default 38,400 as seen below. The AT PP 0C X commands are setting the new baud rate, and once AT Z (reset) is issued, the configuration takes effect.




After updating the UART tests to use this new baud rate, the results were still the same. At this point I determined I would probably need an oscilloscope to debug the ESP side of the circuit and moved on to testing the functionality with my car. With the serial to USB device, I connected the circuit to my car's OBD-II port and was able to successfully interface with it. I was able to

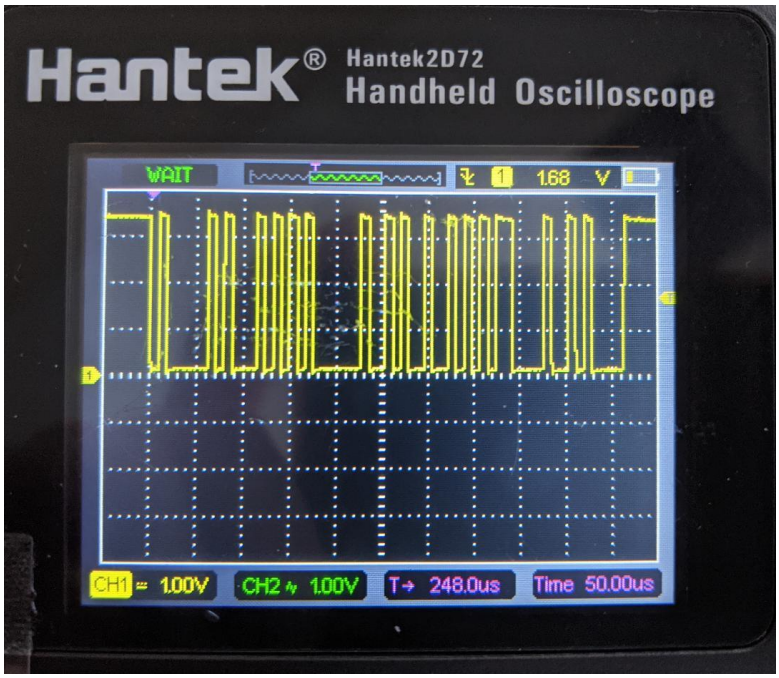
read coolant temps., engine RPM, and the VIN number and others. I also issued the commands to clear diagnostic codes and read diagnostic codes. Most of this data comes back as hex, so I had to interpret the different pieces of data I requested as seen in the screenshots below.





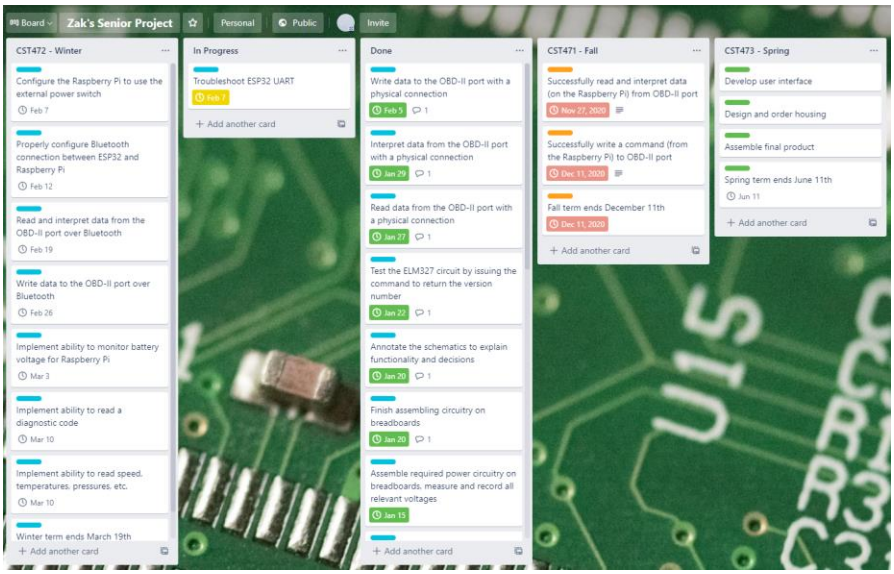

I recently picked up an oscilloscope from Oregon Tech to debug the circuit. It's a handheld unit that is a little confusing to use but I think it will work for the project. I connected it the ESP's UART1 TX line to see what was happening. I noticed in the serial monitor that the ESP gets stuck in a short boot loop with SPI errors if the scope is connected while it boots. After looking through the Espressif documentation, it mentioned UART1 pins can have unexpected

behavior since they are used for SPI by default. Instead of changing the config., I just changed pins to UART0 TX. I was able to get a screenshot of the signal and it appears to be 8N1 serial data as it should, but when I tried decoding the data it didn't make sense and start/stop bits didn't line up.




My next steps are continuing to debug the serial connection and begin working on the Bluetooth connection with the Pi.

Hours worked since Memo 1: 9.5




https://trello.com/b/57kp2fr0/zaks-senior-project

CST 472

# Memo

**To:** Kevin Pintong

**From:** Zak Rowland

**Date:** February 11, 2021

**Re:** Peer feedback

---

**Garrison Peacock –**
Great presentation and project overall. Your diagrams are neat, detailed, and you explained them thorougly. Regarding your API, maybe you could combine color selection and drawing a shape into one function call, further simplifying things.

**Ston Yackamouih –**
Your slow servo issue is probably just a small error in the configuration somewhere, try a basic example included in the Arduino IDE if you haven't already (or a different example if you have.) Option 2 is a good choice since the servos will consume the most power when they are moving. When they reach their set position, they consume little power unless a force tries to move it.

**Jonah Cross –**
I like your project, and if I remember correctly you chose to go with Wi-Fi over Bluetooth which I think is a good idea in your case since Wi-Fi has much better range. Being able to use and charge it at the same time would be a nice addition though.

**Alex Nguyen –**
It looks like you are making good progress on your code which is great, the demo showing what the possible moves are was nice too. One tip I have is adding due dates to your Trello cards to make your board a bit more like a schedule.

**Jose Martinez –**
You have made great progress on the app so far, the interface is simple and clean. The hardware is going to be a bit challenging though, so I think limiting the tilt to something like 180º or 270º would be more feasible. A 3D printed design would be cool and there should be some

resources on campus for that, or you could always send your design to a company to print for you but I'm sure that's relatively expensive.

**Hayden Hutsell –**
Your project is cool and I think a lot of people would find it useful to have in the kitchen. The AWS/Alexa integration is a nice addition and you did a good job explaining how it works. Your sketches of the design were helpful as well.

# References

[1] Trello, Zak's Senior Project, https://trello.com/b/57kp2fr0/zaks-senior-project

[2] Amazon, Autel MaxiCOM MK808BT Diagnostic Scan Tool, https://www.amazon.com/dp/B07MBSZH48/

[3] Amazon, Friencity Bluetooth Car OBD ii 2 OBD2 Scanner Adapter, https://www.amazon.com/dp/B082SMZTFL

[4] Instructables, OBD-Pi, https://www.instructables.com/OBD-Pi/

[5] Raspberry Projects, M3 Pi: Raspberry Pi OBD-II Touchscreen Car Computer, https://projects-raspberry.com/m3-pi-raspberry-pi-obd-ii-touchscreen-car-computer/

[6] Elm Electronics, ELM327 OBD to RS232 Interpreter, https://www.elmelectronics.com/wp-content/uploads/2016/07/ELM327DS.pdf

[7] Raspberry Pi, Power Supply, https://www.raspberrypi.org/documentation/hardware/raspberrypi/power/README.md

[8] OSOYOO, Instruction for Raspberry Pi 5″ DSI Touch Screen, https://osoyoo.com/2019/09/20/instruction-for-raspberry-pi-5-dsi-touch-screen/

[9] J. Cook, Raspberry Pi Power-Up and Shutdown with a Physical Button, *Embedded Computing Design,* https://www.embeddedcomputing.com/technology/open-source/development-kits/raspberry-pi-power-up-and-shutdown-with-a-physical-button

[10] K. Townsend, Introduction to Bluetooth Low Energy, *Adafruit*, https://learn.adafruit.com/introduction-to-bluetooth-low-energy

[11] B. Stockton, How to Set Up Bluetooth on a Raspberry Pi, *howchoo*, https://howchoo.com/pi/bluetooth-raspberry-pi