Lafayette College | Electrical and Computer Engineering

# Driver Display User Manual

ECE/ME 492 | Spring 2020

Last Revision: 4/3/2020

Prepared by: Leah Diamantides

# Abstract

This document details the technical information for the Driver Display (aka Dashboard Display) subsystem within the FSAE Electric Car project. The physical system, user interface, beginning steps, maintenance, and FAQs are described in the respective sections below.

# Table of Contents

# Introduction

This is a user manual for the driver/dashboard display system developed for the 2019-2020 LFEV team. All software was developed on the Arduino IDE using the ESP32 devkit vl board. A 2.9in epaper display was used and is the same one used by the TSV battery packs. (See the Appendix for information and datasheets for the ESP and display) This system was never integrated with other systems such as the SCADA system, it was only tested with a Raspberry Pi running CAN with a PiCAN2 hat. There is full functionality for receiving CAN data (including the speed, state of charge, motor temperature, motor current, and warnings). Speed, state of charge, and warnings are shown on the display (as described in the USER INTERFACE section). Motor temperature and motor current are represented with RG LEDs (as shown in the GETTING STARTED → HARDWARE SETUP section). There are two buttons included in this design (also described in the GETTING STARTED → HARDWARE SETUP section). However, we did not have the chance to add the functionality to transmit CAN data using CANopen. This should not be too hard to configure as the buttons are already functional in the code.

# Physical System

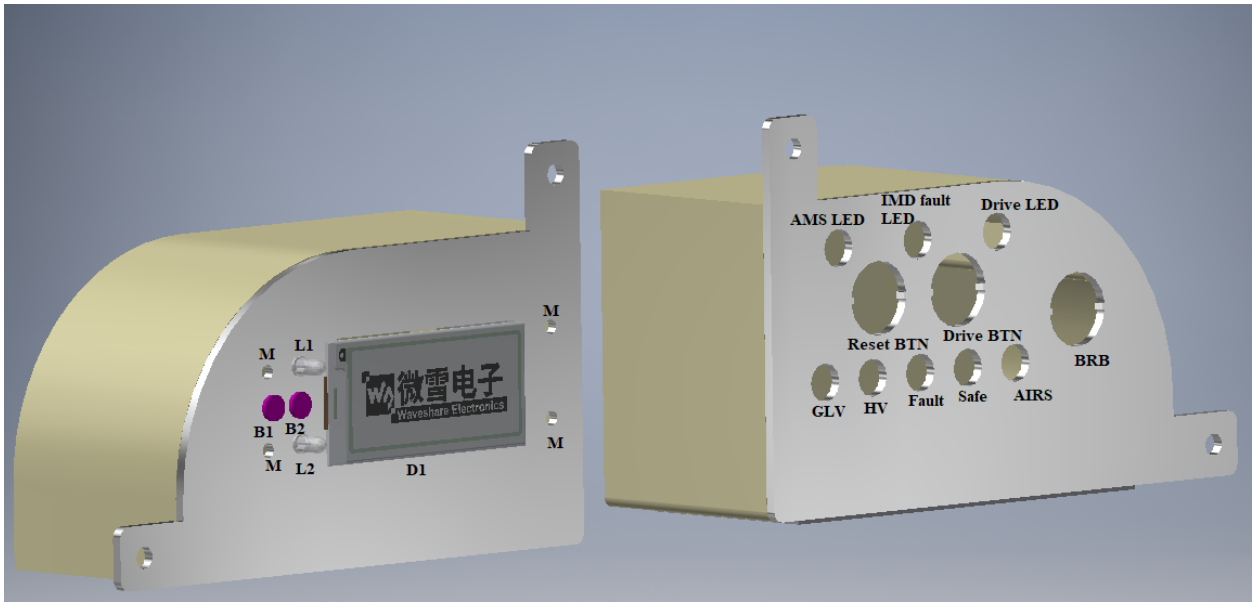



*Figure 1: The annotated mechanical model of the physical system*

This manual focuses on the left panel. The right panel is described in further detail in the Carman User Manual in the Cockpit section. Looking at the left panel, there are several labels:
*M* - these are four screw holes that should be used to mount the driver display PCB to the panel

*L1* - this is the motor temperature RG LED, with green indicating good, yellow indicating that the driver should slow down to reduce the temperature of the motor, and red indicating that the motor temperature is getting so high that the car might shut down (these thresholds are configurable as variables in the code)
*L2* - this is the motor current RG LED, with green indicating good, yellow indicating that the driver should slow down to reduce the strain on the motor, and red indicating that the motor is overcurrenting and the car might shut down (these thresholds are configurable as variables in the code)
*B1 & B2* - these buttons' meanings are configurable in the SCADA *config.xml* file, again this system was never tested with SCADA so the functionality does not exist yet. Some uses for these buttons included certain motor resets that had to be done by the driver or a way of logging a new driver
*D1* - this is the epaper driver display which is mounted onto the PCB behind the panel with screws (as described in the GETTING STARTED → HARDWARE SETUP section)

An important thing to note is that the right-side panel contains an LED labeled *AMS LED*. This light is the only one on the right side that is powered by the ESP. This is discussed further in the GETTING STARTED → HARDWARE SETUP section.

Below is a simplified block diagram of the system showing the inputs and outputs of the microcontroller. The GLV power and CAN connections are further described in the GETTING STARTED → HARDWARE SETUP section.

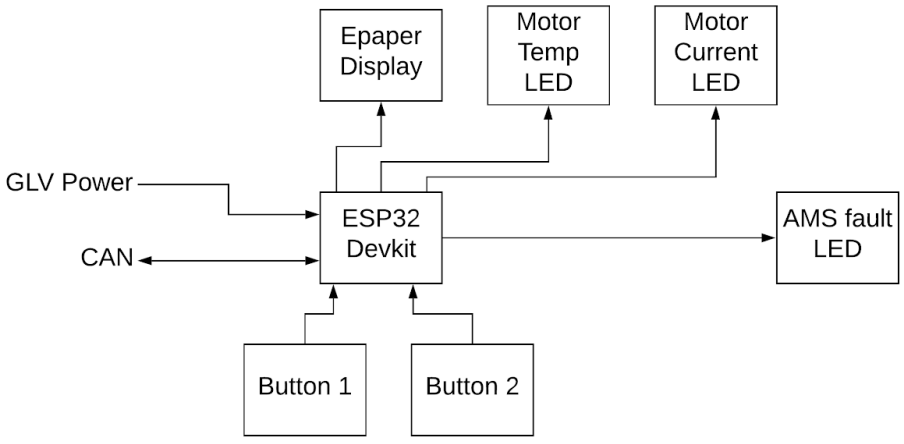



*Figure 2: Simplified block diagram showing inputs and outputs*

# User Interface

The speed and state of charge shown on the display, as well as the motor temperature and current represented as RG LEDs, are all variables defined in the Object Dictionary (for more information on the Object Dictionary, see the Appendix or the MAINTENANCE → SOFTWARE section). When the code is running normally on the board, this is what the display should look like (zeros will appear until CAN data is received):

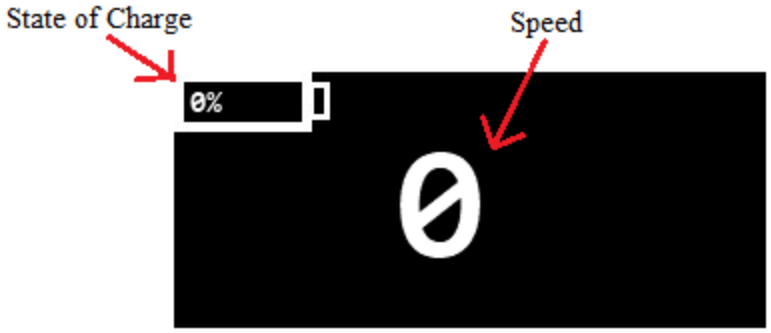



*Figure 3: The annotated normal state of the dashboard display*

Depending on what kind of error occurs, a proper warning will be displayed. “Warning” is one of the variables in the Object Dictionary. These warnings occur when the “warning” variable is changed via CAN from 0 (no warning) to a number ranging 1-3.


*Figure 4: This is the motor overheating warning which is defined as warning number 1*


*Figure 5: This is the overcurrent warning which is defined as warning number 2*


*Figure 6: This warning means there's been a break in the safety loo[ which is defined as warning number 3*

***Any of these images, including both the normal state and the warnings, can be changed by following the Img2Lcd instructions documented in the MAINTENANCE → SOFTWARE section***

The display will go back to the normal state when the "warning" variable is reset to 0.

# Getting Started

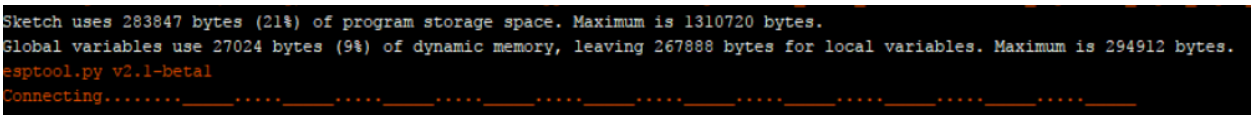
The following subsections contain a detailed guide on how to get started with this system both on a software level and on a hardware level.

## Software Setup

1. Go to the Lafayette-FSAE Github and find the DriverDisplayCode repository or use this link. Download or clone this repository.
2. Download the Arduino IDE from the Arduino website, linked here.
3. Open the **ESP32_Epaper.ino** file from the DriverDisplayCode repository in Arduino. This file can be found in the repository in the folder **ESP32_Epaper**.
4. To upload this code to the board, you must set up the board you are programming. The driver display uses an ESP32 devkit v1. To start, follow these instructions:
    a. In the Arduino IDE, select **File → Preferences**
    b. Where it says "Additional Boards Manager URLs" put the following link: https://dl.espressif.com/dl/package_esp32_index.json
    c. Click "OK" in the Preferences window
    d. To link your board select **Tools → Board: → Boards Manager…**
    e. When the Boards Manager window pops up it might download automatically or ask you if you want to download new boards, click "OK".

f. Once it finishes the download (this may take a minute), type “esp” in the search bar.
g. The “esp32” board library should show up, select a version (I have used the most recent 1.0.4) and click “Install”.
h. Once the library has been installed (this may take a minute), close the window and navigate back to **Tools → Board:**
i. This time, in the drop-down menu, select **ESP32 Dev Module**
j. Once this board is linked and you select the right COM port, you should be able to upload the code to the board with the following instructions

5. In the Arduino IDE, you can use the checkmark in the upper left hand corner to “verify” the code, or comile it without uploading the program to the board.
6. Next to the checkmark there is an arrow pointing right, this will compile and then upload the code to the board. Press this button and follow these steps:
    a. If the code compiles without any error, you should see something like this:




    b. While the “Connecting…..____…..___” is running, hold the **Boot** button, on the board. Not doing so will result in an error (see FAQs section). We realize that this is inconvenient, and we’ve tried to fix it with no luck. However, it only needs to be done when uploading to the board, so it will not be a concern once the board has its final code uploaded and the display is installed on the car.
7. Once, the code is uploaded to the board, you should see the normal state shown in the USER INTERFACE section.

# Hardware Setup

The driver display PCB was ordered but not populated before the end of the semester, so there may need to be some debugging and fixing before this board is ready for the car. The following describes how the board should work and how the pieces should fit on the board. For more information on this PCB, go to the Lafayette-FSAE Github and find the DriverDisplay repository or use this link.

The following shows the front of the PCB, the side that is facing the driver. As mentioned in the PHYSICAL SYSTEM section, this board will go behind the dashboard left-side panel, with only the LEDs, buttons, and display showing through.

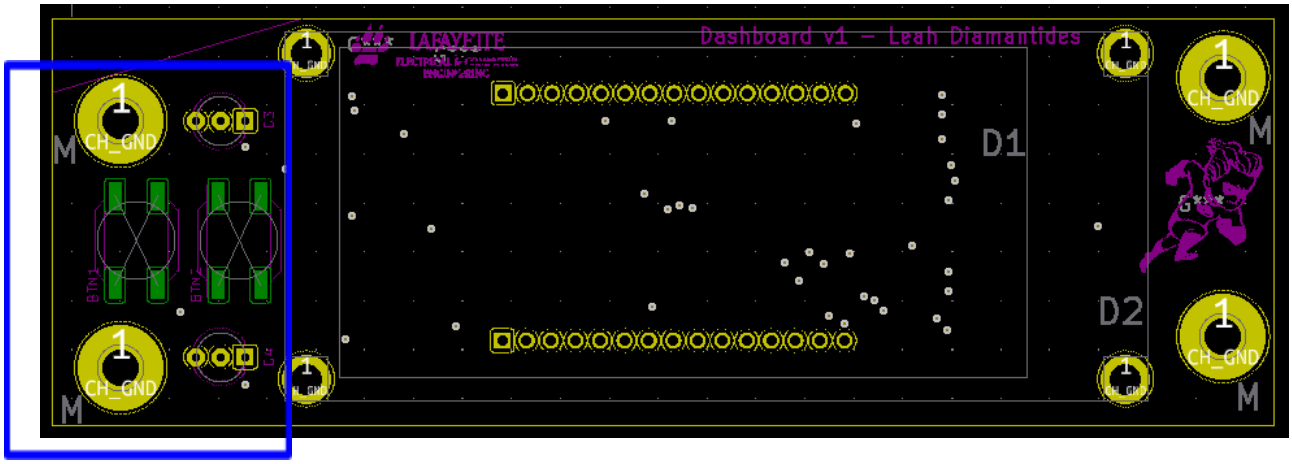



*Figure 7.a: Front view of dashboard PCB*

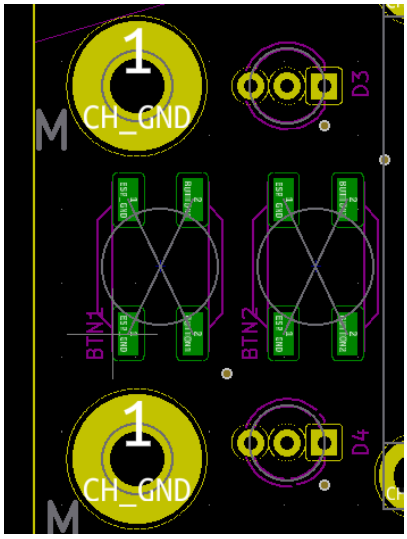



*Figure 7.b: Zoom of blue rectangle in Figure 7.a*

In *Figure 7.a,* the holes labeled *M* represent the mounting holes used to mount the PCB to the dashboard panel (as shown in *Figure 1*). The purple diagonal line in the upper left corner shows where the PCB might need to be cut, in order to better fit behind the panel. The rectangle *D1* corresponds to the *D1* in *Figure 1* and is the part of the screen that should be visible through an opening in the panel. The rectangle *D2* represents the actual size and location of the epaper display, including the four mounting holes needed to mount the display to the PCB.

The buttons and LEDs are more clearly seen in *Figure 7.b*. The buttons are labeled *BTN1* and *BTN2.* The motor temperature LED is labeled *D3* and the motor current LED is labeled *D4*.

If the display, buttons, and LEDs are mounted properly, the four mounting holes should line up with the mounting holes on the dashboard panel. Then the display, buttons, and LEDs should fit smoothly into the spaces shown in *Figure 1*.

The following shows the back of the same PCB, the side that is facing away from the driver.

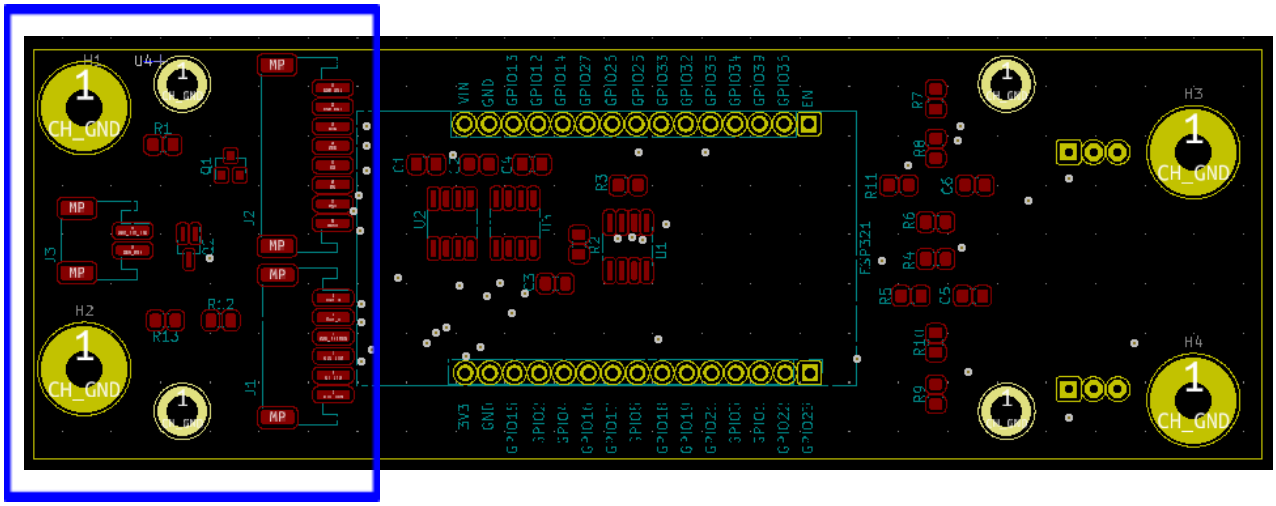



*Figure 8.a: Back view of dashboard PCB*

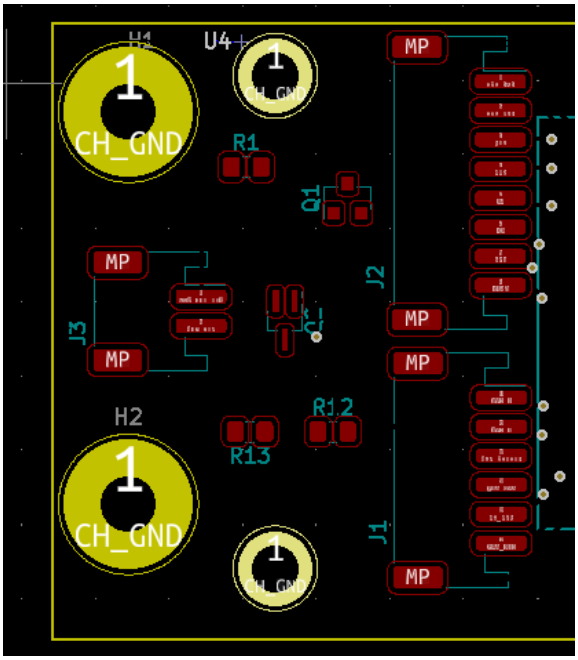



*Figure 8.b: Zoom of blue rectangle in Figure 8.a*

The important thing to note on the back of this PCB, as shown in *Figure 8.a*, is the location and orientation of the ESP32 devkit. Make sure that the ESP gets mounted on the back (away from the driver) and not the front, otherwise, the I/O pins will not be wired properly.

The connectors can be more easily seen in *Figure 8.b*. There are three important connectors for this board, labeled as *J1, J2,* and *J3.* Connector *J1* gives the board power from GLV as well as access to the CANbus in the car. Connector *J2* is all the connections that are needed to interface with the epaper display (further described in FAQs section with breadboard diagram). Lastly, *J3* connects to the AMS LED shown in *Figure 1*. The trigger for the AMS LED will come from SCADA via the CANbus, but this functionality was never tested.

Another important note is that these connections will likely need to be attached *before* the display is mounted to the front of the board. There is very little space between the mounting holes and the connectors, so I expect this to be a solution.

# Maintenance

The following explains how to maintain this system. The software can be fairly easily maintained by editing the existing code. As mentioned, there are a few things that did not get finished on the software side that will need to be added. Additionally, the PCB has never been tested so instructions on how to set up the system on a breadboard are included.

## Software

To update the software, follow the instructions under the GETTING STARTED → SOFTWARE SETUP section.

All code for editing the user interface is in the *Dashboard.cpp* file in the existing repository. The code is well commented, so it should be simple to make adjustments on your own. There are a few aesthetic things that are not as simple to figure out and those are: displaying an image, making a new font for the display, and editing the Object Dictionary.

### Displaying an image

The main screen for the display uses a background image that may be found here. This image was created using Microsoft Paint and then saved as a .jpg file. In order to configure an image to be used on the display:

1. Download Img2Lcd. Which can be found in the DriverDisplayCode repository in the **Image2Lcd** folder
2. Open your .jpg image
3. Set the following parameters:

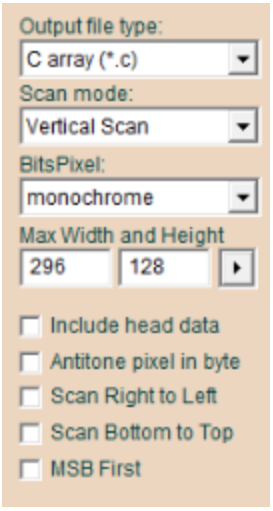



*Figure 9: Parameters for Img2Lcd program*

*** You can also check the *Reverse Color* option, but I wanted mine to invert***

4. Once this is done, click *Save*
5. Open your new .c file, copy all of the contents, and paste them into the *BitmapGraphics.h* file, which can be found in the repository under the **ESP32_Epaper** folder
6. You can now use the drawEampleBitmap function to display your image. You can see many examples of this function being used in the *Dashboard.cpp* file

## Making a new font for the display

If you would like a font that does not currently exist in the DriverDisplayCode repository under **ESP32_Epaper → Fonts**, then you can add it using the following steps:

1. Go to http://oleddisplay.squix.ch/#/home
2. Use the following settings (but change Font Family, Style, and Size to your preference):

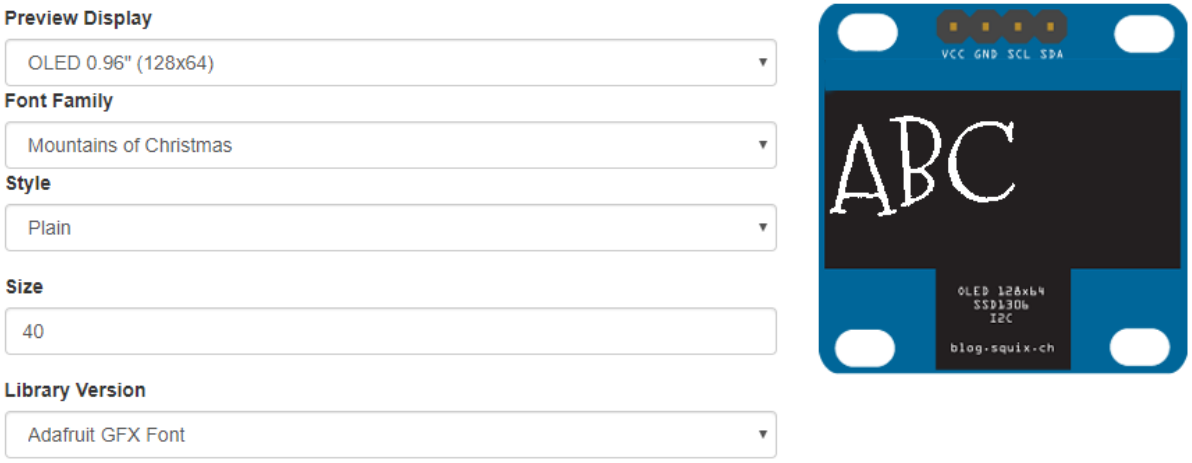



*Figure 10: Font Converter Settings*

3. Click *Create*
4. Copy and paste the text created, and make it into a .h file
5. Put this file in the **ESP32_Epaper → Fonts** folder and check the *Dashboard.cpp* file for examples on how to use a different font.

## Editing the Object Dictionary

The Object Dictionary has been adopted from a GitHub Repository by robincornelius. If looking to add a variable to the Object Dictionary:

1. Download the EDS Editor linked above
2. Open the .XDD file found in the DriverDisplayCode repository or linked here
3. In the EDS editor to the *Object Dictionary* tab.
4. Right click anywhere in the box called *Manufacturer Specific Objects*. Click *Add New Object*. Add your new variable. If looking to edit a variable that already exists, follow these same steps but rather than right clicking, choose the variable you are editing and make changes accordingly.
5. Be sure to save this .XDD file.

To then add this functionality of the Object Dictionary to your code, use the following steps:

1. In the EDS Editor go to **File → Export CanOpenNode c/h**
2. This will create a CO_OD.h and CO_OD.c file. These already exist in the current repository, so you can replace them or edit them. I **strongly** recommend editing them. There were a lot of variables we had to add to get it to work, so try editing the current file instead.

## Hardware

As mentioned before, because the dashboard PCB was never populated or tested, there may be issues that need to be debugged. All of my development was done on a breadboard, so here is a detailed diagram of the I/O connections. The MCP2551 is the CAN transceiver (see Appendix), PacMan uses this chip as well. Hopefully, this should help when testing and debugging the system.

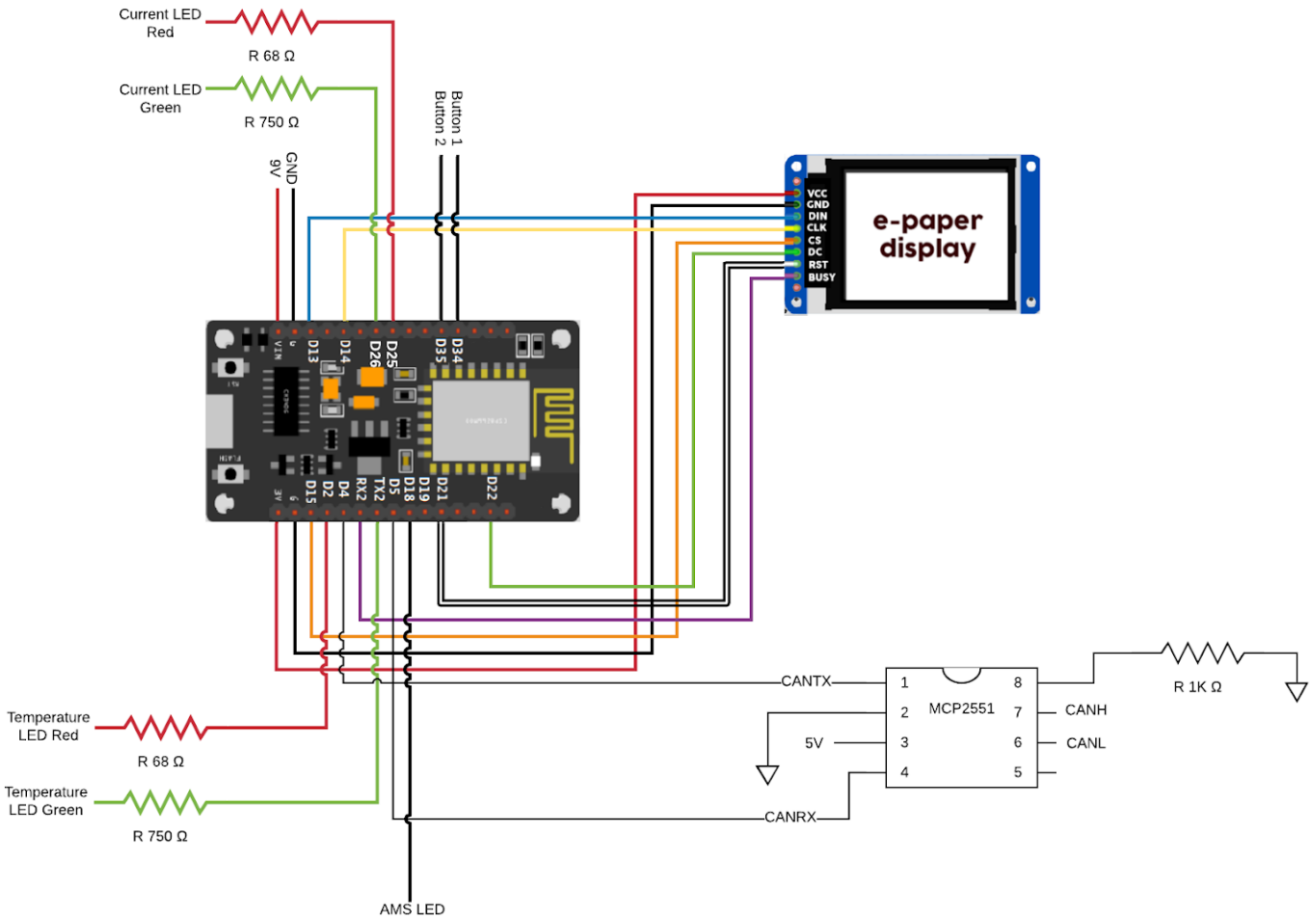



*Figure 11: Wiring diagram for debugging purposes*

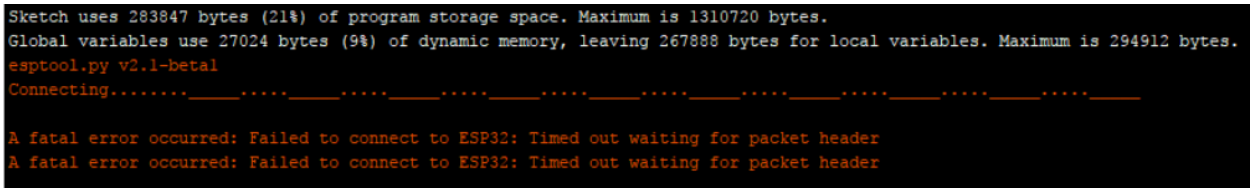
# FAQs

### Why are we using an epaper display?

Epaper displays are very low power, as is the ESP that powers it. This is a great solution for the car because we needed something low-power and sunlight readable. There's no glare on an epaper display, so the driver will always be able to see it.

### Why can't I upload to the ESP32?

Check to make sure you installed the board correctly using the steps outlined in GETTING STARTED → SOFTWARE SETUP. If that's done correctly, you may be getting an error that looks like:




Make sure you hold the **Boot** button on the devkit while the "Connecting…..____…..___" process is running, this should fix your problem.

### Why isn't the partial update working on the display?

Early on, we bought 2.9in epaper displays that displayed in black, white, and red. We later found out that these displays **do not support partial update**. They look very similar to the one that is supposed to be used, so make sure you are using a display that only supports black and white images.

### Are there any other resources for using CAN?

All CAN code for this system was borrowed and adapted from PacMan firmware. The Github for PacMan firmware is a good place to look.

# Appendix

### ESP32 devkit v1

This is the microcontroller used as the brains behind this system. All information about it can be found at this link.

## Waveshare 2.9in epaper display

This is the display we are using which supports partial update. This is the black and white version **not** the black, white, and red version. All information about it can be found at this link.

## MCP2551

This is the CAN transceiver we are using. The ESP32 does not have CAN capabilities without it, and this is the same one that PacMan uses. In general, if there are problems with CAN you should look at the PacMan documentation; all CAN functions for this system were borrowed from that.

## Object Dictionary

The system currently keeps track of only 5 variables, the Object Dictionary looks like this:

| | |
|---|---|
| 2000 | speed |
| 2001 | temperature |
| 2002 | current |
| 2003 | warningNum |
| 2004 | SOC (state of charge) |
| 2005 | AMS |

These variables are all set up as uint16 and initialized as 0. The *speed*, *temperature*, and *current* can really be any uint16 number, whereas the *warningNum* should only be between 0 and 3 (as mentioned in the USER INTERFACE section), the *SOC* should be between 0 and 100, and *AMS* should only be 0 or 1 (no fault or fault).