

EECS 159B Final Report

IoT EVSE Charge Controller

Andy Begey (46433784)
Brandon Metcalf(78186959)
Luis Contreras(93197465)
Shermaine Dayot(19055683)

Mentors: G.P. Li, Michael Klopfer

Contents

Purpose	-	-	-	-	-	-	-	-	-	-	-	2
Introduction	-	-	-	-	-	-	-	-	-	-	-	2
-Background	-	-	-	-	-	-	-	-	-	-	-	2
-Methods	-	-	-	-	-	-	-	-	-	-	-	3
Hardware Design	-	-	-	-	-	-	-	-	-	-	-	4
-Overview	-	-	-	-	-	-	-	-	-	-	-	4
-Phase 1: Microsemi Board	-	-	-	-	-	-	-	-	-	-	-	4
-Design	-	-	-	-	-	-	-	-	-	-	-	4
-Sub-Circuitry	-	-	-	-	-	-	-	-	-	-	-	9
-Pin connection Table	-	-	-	-	-	-	-	-	-	-	-	9
-Phase 2: Smartenit Board	-	-	-	-	-	-	-	-	-	-	-	12
-Design	-	-	-	-	-	-	-	-	-	-	-	12
-Sub-Circuitry	-	-	-	-	-	-	-	-	-	-	-	17
-Pin connection Table	-	-	-	-	-	-	-	-	-	-	-	18
-Flashing the ESP WROOM-02	-	-	-	-	-	-	-	-	-	-	-	19
-EVSE Simulator	-	-	-	-	-	-	-	-	-	-	-	19
-Bill of Materials (BOM)	-	-	-	-	-	-	-	-	-	-	-	21
Software Design	-	-	-	-	-	-	-	-	-	-	-	22
-Overview	-	-	-	-	-	-	-	-	-	-	-	22
-Controller User Interface States	-	-	-	-	-	-	-	-	-	-	-	22
-J1772 Protocol	-	-	-	-	-	-	-	-	-	-	-	23
-ESP-32 ADC	-	-	-	-	-	-	-	-	-	-	-	25
-ADE7953 Metering Chip Interface	-	-	-	-	-	-	-	-	-	-	-	26
-MQTT	-	-	-	-	-	-	-	-	-	-	-	29
-Access Point	-	-	-	-	-	-	-	-	-	-	-	31
-Phase 2: Smartenit Board	-	-	-	-	-	-	-	-	-	-	-	31
Physical Verification	-	-	-	-	-	-	-	-	-	-	-	32
IOT SmartPlug MQTT Guide	-	-	-	-	-	-	-	-	-	-	-	34
References	-	-	-	-	-	-	-	-	-	-	-	40

Purpose

Our plan is a two-phase design. Phase one is development of a hardware interface for the Microsemi Field Programmable Gate Array (FPGA)-based Future Creative Board. Our design is purposed for bringing more powerful computing solutions to electric vehicle charge controllers. The Future Creative Board is physically compatible with Arduino Due. As a result, our design will become the first ever Arduino shield for EVSE charging based application.

Phase two aims to create a “smart” internet connected electric vehicle charging controller that is compatible with the Smartenit back-end and thus allows electrical vehicle owners to easily control the charging schedules and monitor power consumption of their vehicles via a mobile app. This charging solution will be the first of its kind and is a logical evolution of the growing number of IoT connected devices in the home. The system will grow even more useful in time as more electric vehicles hit the road and it becomes increasingly crucial to schedule charging to avoid the more expensive peak rates.

Introduction

Background

Due to the high incentive of government tax credits and rapidly fluctuating oil prices, the amount of battery electric vehicles (solely electric powered, often referred to as BEV's) are on the rise in California and per an executive order from the governor, targeted to reach 1.5 million by 2025 [1]. The introduction of this staggering amount of high-load electric vehicles will begin to cause both price and safety issues as more of them require charging within the same residential space, and BEV owners will require a simple solution for scheduling their vehicles to charge during the most cost-effective times of the day (off peak from 11pm-6am) [2]. Additionally, owners of multiple BEV's will require the ability to plug in both cars, yet have each car charging on a different schedule for both safety and economic reasons. Some modern consumer BEVs have incorporated charge scheduling, making the integration of a charge scheduler redundant. Additionally, some modern consumer BEV chargers already have the ability to be scheduled over Wifi. What these systems currently lack is integration into a smart device network.

With the rise of the Internet of Things (IoT), more devices at home are being connected than ever before [3], in an attempt to localize all electronic devices into a single controllable network. It makes sense that BEV chargers should join the growing list of smart household devices, since it is fast becoming a modern household necessity with the rising number of BEVs in California.

Many companies are breaking into this market and offering solutions for managing “smart home” appliances [4]. One such company is Smartenit- a local company with an API that allows users to monitor a large number of smart household devices via a mobile app, while focusing on

energy saving solutions. By creating an electric vehicle charger that is compatible with their network, smart vehicle chargers could provide users with an intuitive and hassle-free environment for monitoring the charging of their vehicles.

Methods

In order for the extremely wide range of electric vehicles to be able to connect and charge, a standard is needed to regulate plug sizes, charge rates, and communication with the vehicle. In North America, the most common protocol is known as SAE J1772, which is the protocol that this device uses for its microcontroller to communicate with EVs to negotiate charge rate and flag various errors. Essentially J1772 is a simple analog-control protocol that uses one data line (the pilot pin) to output a 1kHz pulse wave modulation (PWM) signal to the car. The duty cycle of the signal indicates to the car how much power it is allowed to draw from the house power line. The corresponding duty cycles are listed below:

AMPS	DUTY CYCLE
5	8.3%
15	25%
30	50%
40	66.6%
65	90%
80	96%

Table 1: J1772 Protocol PWM Duty Cycle and Corresponding Charge Rate

Peak voltage level of the PWM signal on the power pin secondarily is a status indicator of the car's connection to the charger. The status criteria is listed below:

STATE	PILOT HIGH VOLTAGE	PILOT LOW VOLTAGE	FREQUENCY	RESISTANCE	DESCRIPTION
State A	12 V	N/A	DC	N/A	Not connected
State B	9 V	-12 V	1 kHz	2.74 k Ω	EV connected, ready to charge
State C	6 V	-12 V	1 kHz	882 Ω	EV charging
State D	3 V	-12 V	1 kHz	246 Ω	EV charging, ventilation required
State E	0 V	0 V	N/A	—	Error
State F	N/A	-12 V	N/A	—	Unknown error

Table 2: J1772 Protocol Pilot Pin Charge State Diagram

During the charging process, the electric vehicle responds to certain conditions by altering the voltage level of the pilot line, which is then responded to by the microcontroller in the EVSE in order to provide correct and safe charging.

Hardware Design

Overview

Much of the software interfacing work for this project has been developed by peer teams working in Callt2's CalPlug program. Therefore, the basis of this project is largely centered on hardware, which involved the design and testing of two separate PCB's using Autocad's EAGLE PCB design software. Each board functions as a wirelessly controlled EVSE charge controller operating on the J1772 protocol, however each board will run with different processing technologies. The first PCB was developed to be compatible with both the Microsemi Smartfusion2 FPGA as well as Arduino Due, while our second design will work with the Espressif ESP WROOM-32 microcontroller.

The PCB's are installed in a polycarbonate enclosure and connected to DC-controlled AC power relays. The enclosures features an arrangement of button(s) and LEDs to give the user emergency control and indicate device status. LED blinking patterns and color combinations (specified in the software section) indicate device states. The buttons allow the user to override charging functions that were set over MQTT in the event of

The final developed hardware for this project is an EV simulator test bench for simulating the charging of an EV. This simple device is an addition on a previous team's EV Simulator with the added ability to dim and brighten the lights depending on the pulse width on the J1772 pilot line. This ability gives the simulator the ability to visualize changes in charge rate.

Phase 1 Microsemi Board:

-Design

The starting point of our design is the open source OpenEVSE product which was created by Chris Howell and sold online in varying forms of complexity. His design runs on the ATmega328p microcontroller and provided a solid reference for us to build our design from. Looking at his design (schematic shown below in figure 1), there are 4 main building blocks of the circuit we've chosen to adapt to our project:

- 1) The microcontroller stores and runs the code necessary to implement the J1772 protocol and it's safety features.
- 2) The pilot circuitry generates the +/- 12V square wave on the pilot line and provides an input for the microcontroller to determine the state of charging.
- 3) The ground fault interrupt (GFI) circuitry allows for the detection of a short in the system so that the microprocessor can safely shutdown the device in the event of a safety issue arising.
- 4) Level detection and incorporation of circuitry able to connect to CalPlug's own designed wattmeter into the device allows for active power management and logging of the device.

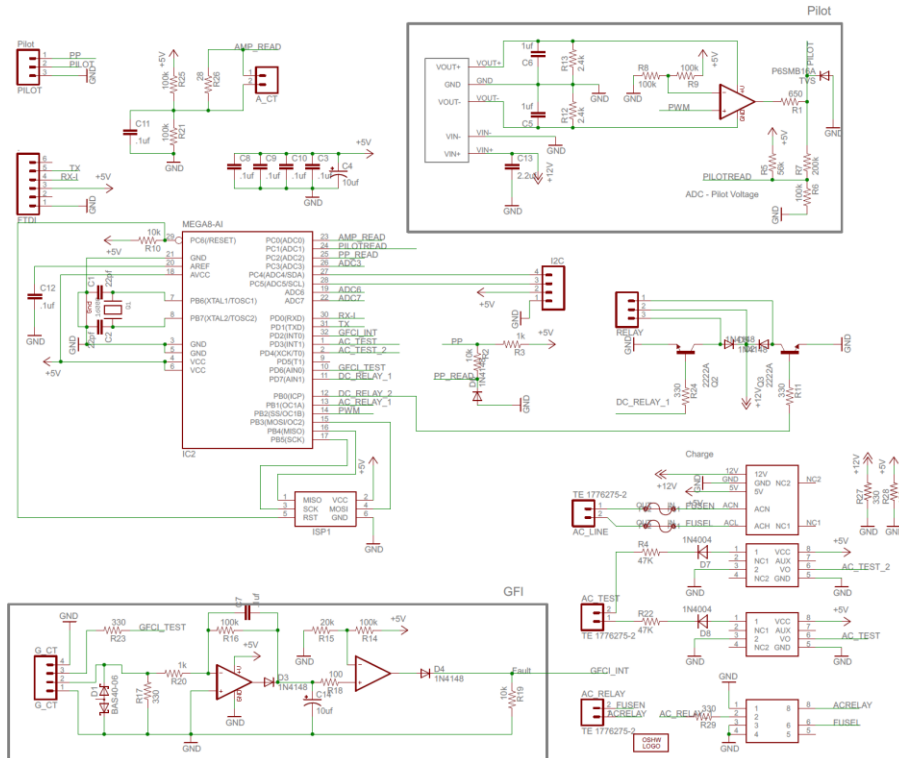


Figure 1: OpenEVSE Schematic

One of the first steps in the redesign process is ensuring that the previous inputs and outputs run at the proper voltage level. The Microsemi FPGA chip is only tolerant up to 3.3V, so all circuitry in our design is adjusted to output and accept 3.3V signals. Backed by LTSpice simulations below (Figure 2), simple voltage dividers and careful manipulation of resistor values from the original OpenEVSE design allow for signals to be adapted to the new needed range.

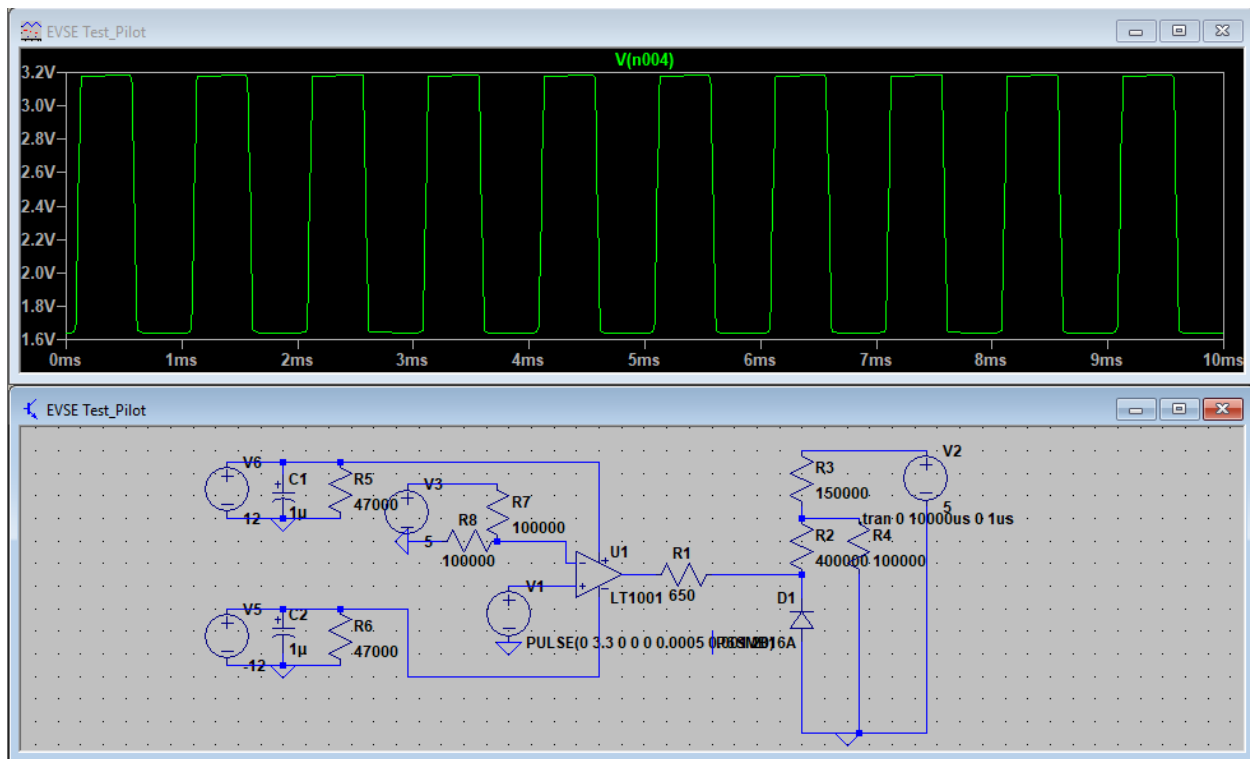
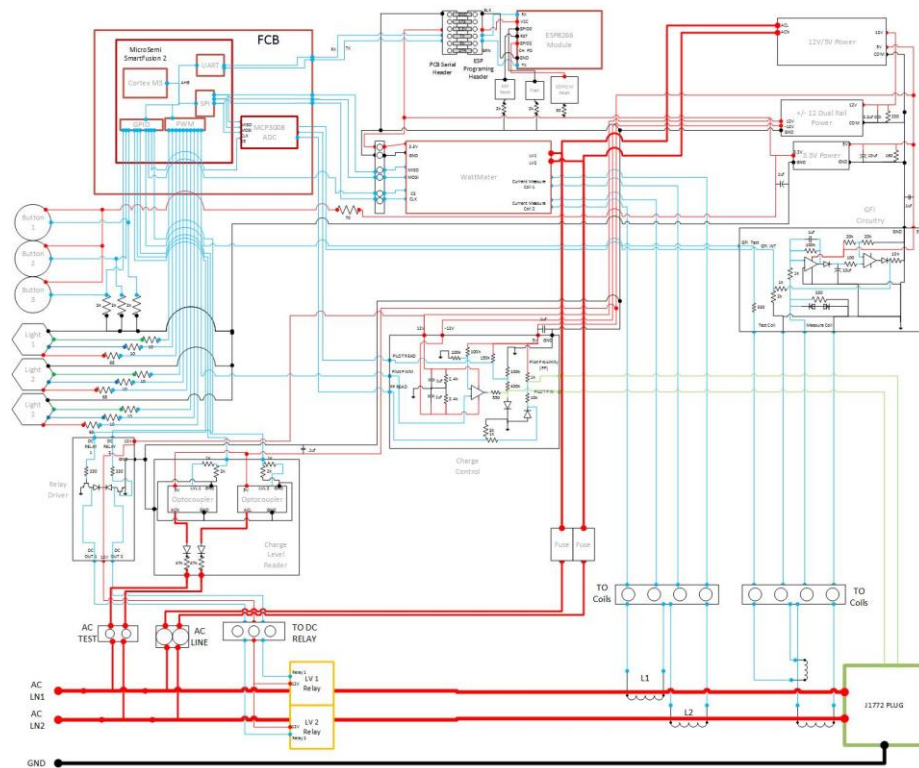


Figure 2: LTSpice Full Sim of Pilot Circuitry outputting 3.3V to ADC

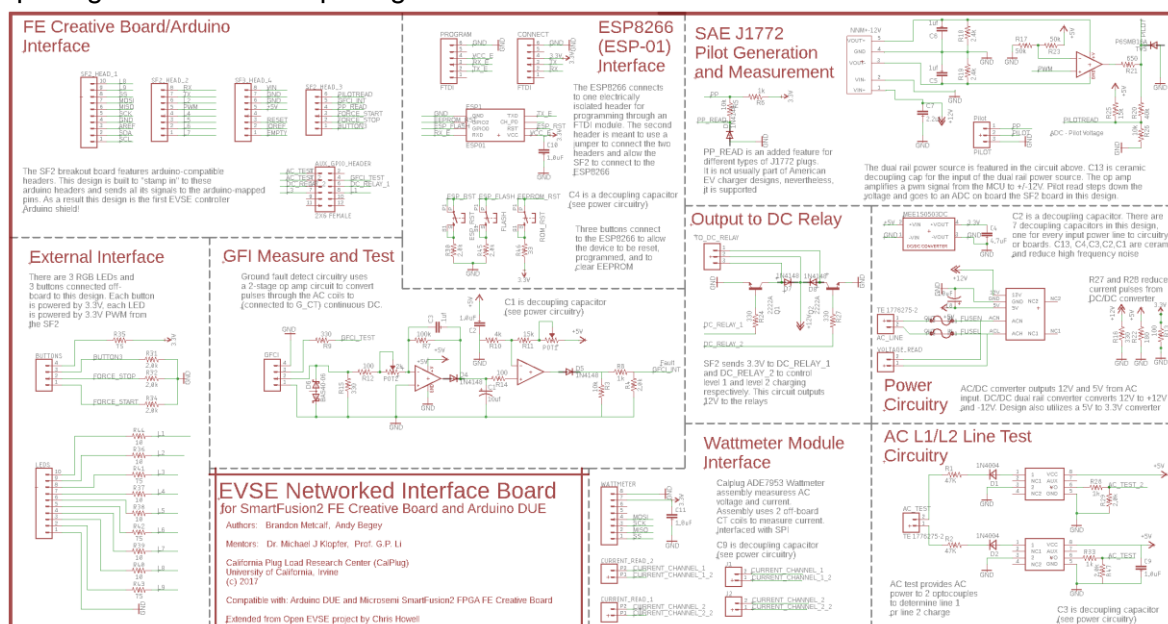
Figure 2 above is a full simulation of the pilot circuitry behavior with re-designed resistor values of R2, R3, and R4 to make sure that the output of the system (at the node where R2, R3, and R4 intersect) outputs 3.3V tolerant signals. Part of the design of this system involved careful examination of the impedances of the input GPIO pins used for reading these values. In order for values to be read correctly by the microcontroller, the input impedance of the GPIOs must be much larger than that of the output of each analog control system, so that almost all voltage is accumulated at the GPIO pin in use. We discovered that the GPIO pins of the microcontroller on Microsemi's FPGA are on the order of several Mega-ohms, so the output impedance values used (~400k ohms max) in this design are tolerable.

In addition to modification of output and input voltages, our design replaces a resistor in the GFI circuitry with a potentiometer to allow modification to the threshold current pulse value that triggers a ground fault detection.

These alterations along with the abstract diagram below (Figure 3) clearly display the design parameters for this device. Each sub circuit and all electronic signal and power connections are displayed in order to outline all hardware constraints for a PCB design.



The completed PCB schematic resulting from Figure 3 above can be seen in Figure 4 below. The finished design and physical layout of the PCB can be seen in Figure 5 (top of the board) and Figure 6 (bottom of the board). Additional hardware constraints for the PCB layout are pin spacing and connector spacing for interface to the Microsemi FPGA.



Figure

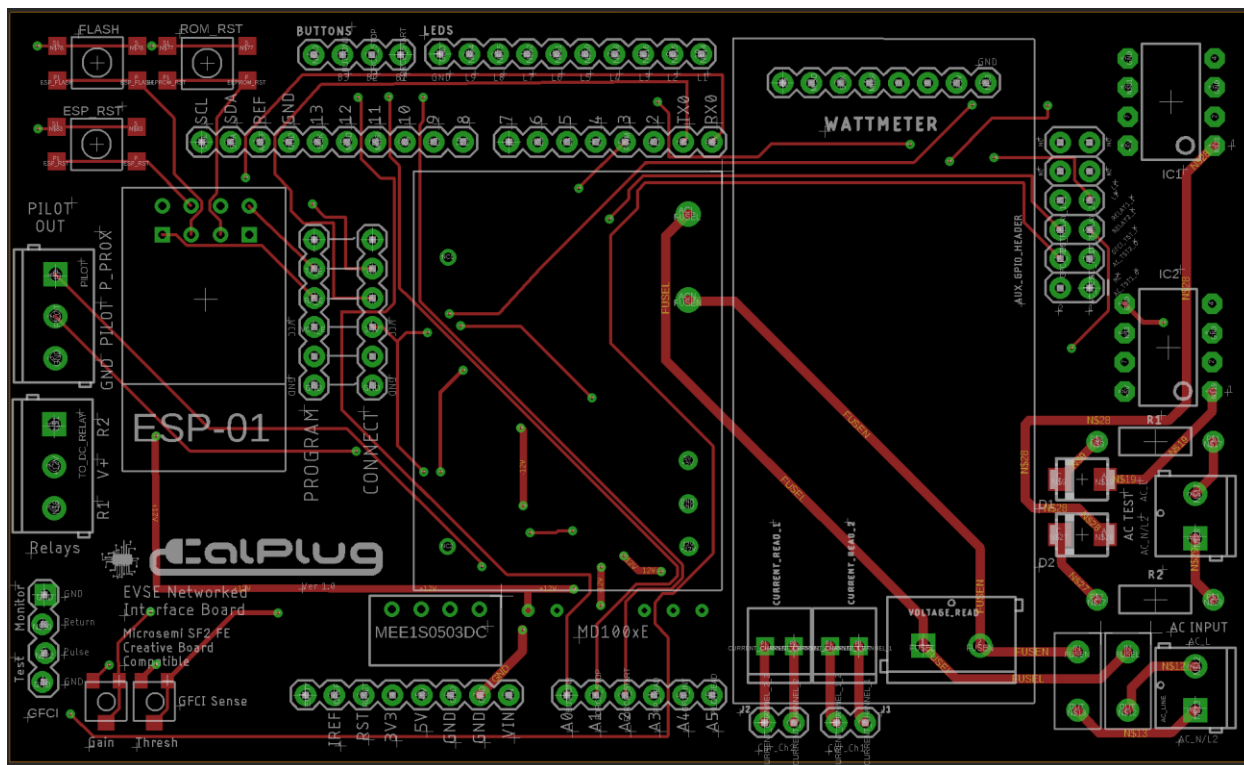


Figure 5: Phase 1 EAGLE Layout Top Layer

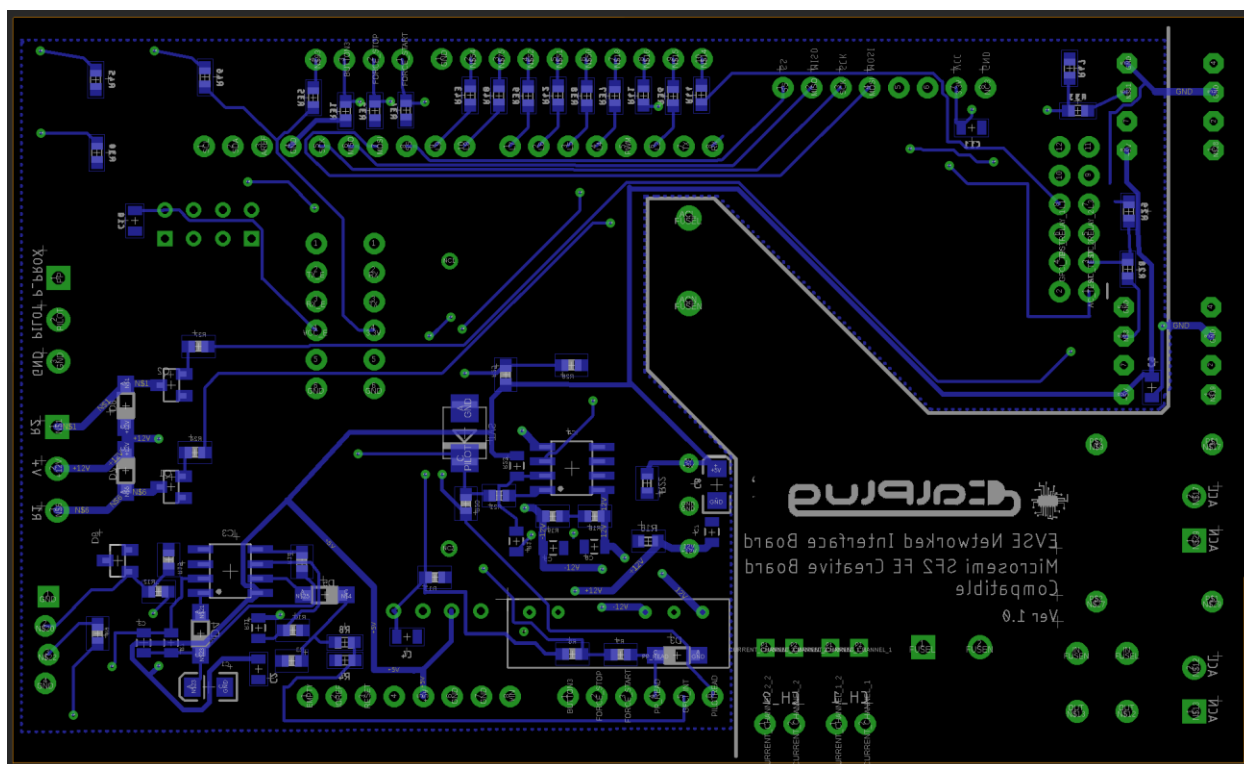


Figure 6: Phase 1 EAGLE Layout Bottom Layer

-Sub Circuitry

There are four major sub circuits used in this design. Details on each circuit are highlighted below. Please note that implementation of these circuits change between board designs in each phase.

1) GFI Detect

Ground Fault Interrupt (GFI) senses any short in the device through measuring the current output from a coil wrapped around the input AC lines. In the case that a short occurs in the device or anywhere on the line, the circuit outputs logic high (3.3V in this) for several seconds so that the MCU can immediately detect it and shut down power.

2) Pilot Generation

Pilot circuitry takes an input 1KHz pwm signal from the MCU and amplifies it to +12/-12V. This signal is then rectified and sent to the car. Simultaneously, this circuitry uses a resistor network to transform the pilot signal to a tolerant voltage for the MCU to read. Reading the pilot line determines the state of the device, as the car uses its own resistor networks and switches to change the voltage of the pilot line on its own. The MCU does not dictate direct charging of an EV. EVs still retain the ability to dictate their own charging limitations as indicated by J1772 protocol.

3) AC Line Test

The AC test line reads the AC lines to determine which lines carry charge. This is done to determine if level 1 or level 2 charging is accessible to the device. This circuit outputs a digital high or low to the MCU for level 1 or level 2 charging accessibility, using two optocouplers.

4) Output to Relay

The DC-operated AC relays take 12V input voltage to activate. This circuit uses transistors to amplify the 3.3V output signals from the MCU to achieve these voltages and activate/deactivate the relays for charging.

Unfortunately, this phase of the project was delayed indefinitely due to loss of contact with the company sponsoring our design. Continuation of the project may resume, if adequate support and time is provided.

-Pin Connection Table

Signal Name	Signal Function	Connection	Output/Input
GFI_INT	Read ground fault interrupt	MCU (A4)	INTPUT
GFI_TEST	Generate GFI Test signal	MCU (IO2)	OUTPUT
PILOT_READ	Read pilot signal voltage	MCU (A5)	INPUT

PILOT	Pilot signal to car	Charge Cable	OUTPUT
PP_READ	Read output line for PP Signal	MCU (A3)	INPUT
PP	Used to find max current for cable	Charge Cable	OUTPUT
PWM	PWM signal for Pilot circuitry	MCU (IO3)	OUTPUT
DC_RELAY_1	Open/close relay 1	MCU AUX_GPIO (6)	OUTPUT
DC_RELAY_2	Open/close relay 2	MCU AUX_GPIO (5)	OUTPUT
AC_TEST	Test for level 1 availability	MCU AUX_GPIO (1)	INPUT
AC_TEST_2	Test for level 2 availability	MCU AUX_GPIO (3)	INPUT
CURRENT_CHANNEL_1	Input current channel 1	Wattmeter	INPUT
CURRENT_CHANNEL_1_2	Output current channel 1	Wattmeter	INPUT
CURRENT_CHANNEL_2	Input current channel 2	Wattmeter	INPUT
CURRENT_CHANNEL_2_2	Output current channel 2	Wattmeter	INPUT
BUTTON3		MCU (A0)	INPUT
FORCE STOP	UI button to stop charging	MCU (A1)	INPUT
FORCE START	UI button to start charging	MCU (A2)	INPUT
L1	activate/deactivate red LED1	MCU AUX_GPIO (8)	OUTPUT
L2	activate/deactivate green LED1	MCU (IO2)	OUTPUT
L3	activate/deactivate blue LED1	MCU AUX_GPIO (7)	OUTPUT
L4	activate/deactivate red LED2	MCU (IO4)	OUTPUT
L5	activate/deactivate green LED2	MCU (IO5)	OUTPUT

L6	activate/deactivate blue LED2	MCU (IO6)	OUTPUT
L7	activate/deactivate red LED3	MCU (IO7)	OUTPUT
L8	activate/deactivate green LED3	MCU (IO8)	OUTPUT
L9	activate/deactivate blue LED3	MCU (IO9)	OUTPUT
5V	5V power	MCU (5V)	INPUT
GND	Ground	MCU (GND)	INPUT

Phase 2 Smartenit Board:

-Design

Phase 2 of this design was intended to use the NXP JN5168 ZigBee enabled microcontroller, and connect to a Smartenit ZigBee hub device for use in a smart home network. However, in cooperation with Smartenit, the project was redesigned for integration into a different enclosure using the ESP WROOM-32 MCU.

The initial design of the NXP board is heavily based on the design of the phase one board, the main difference being the removal of the ESP8266 and Smartfusion board connector pins and addition of the new NXP module. The EAGLE Schematic and layouts for this board are shown below in figures 7, 8, and 9.

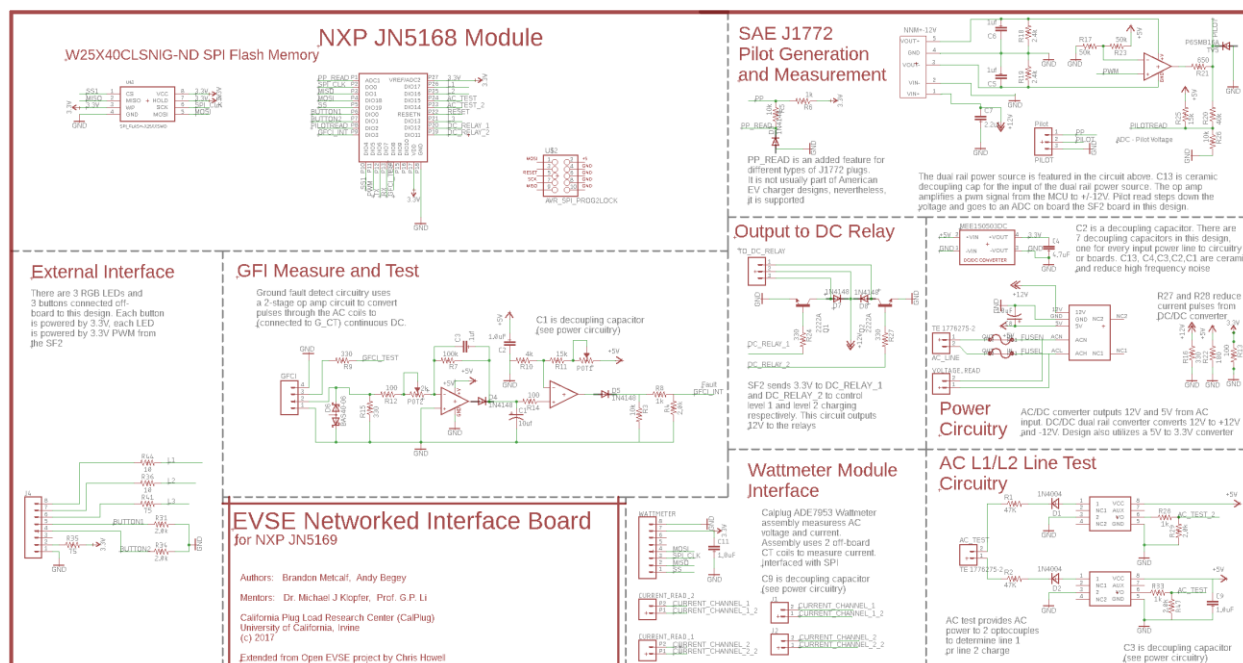


Figure 7: NXP board EAGLE Schematic

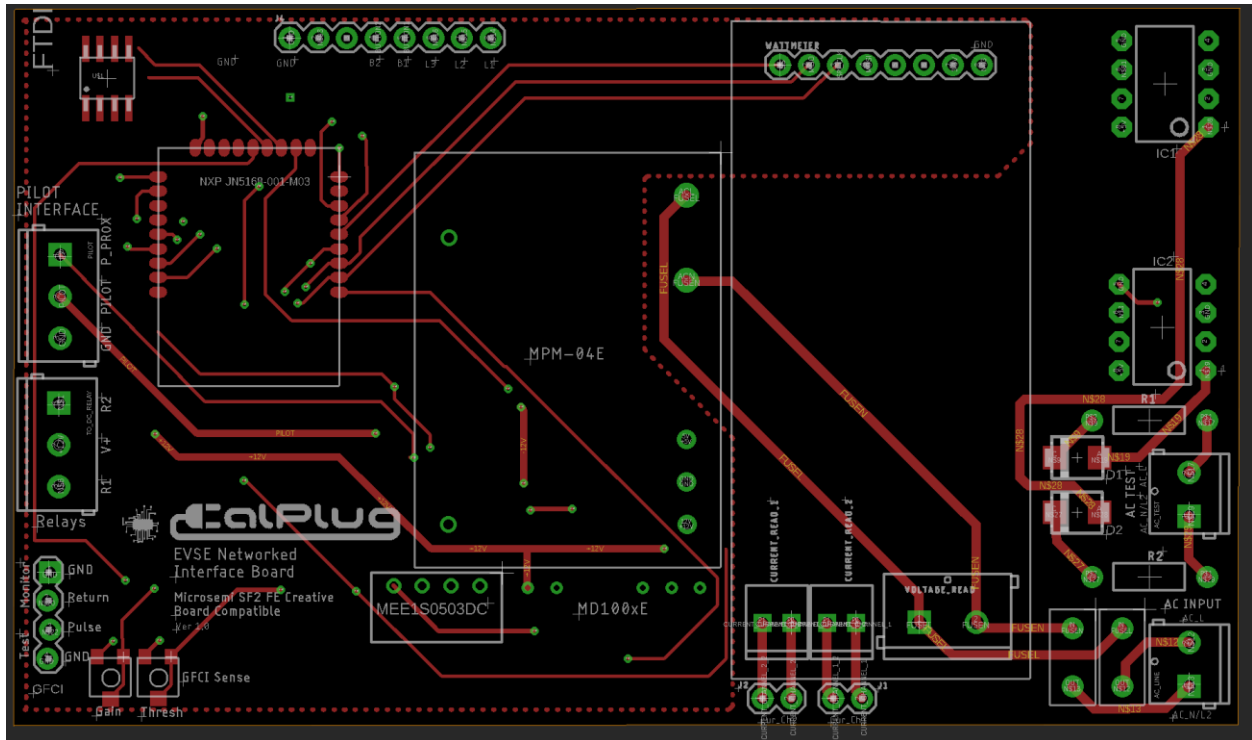


Figure 8: NXP board layout (top layer)

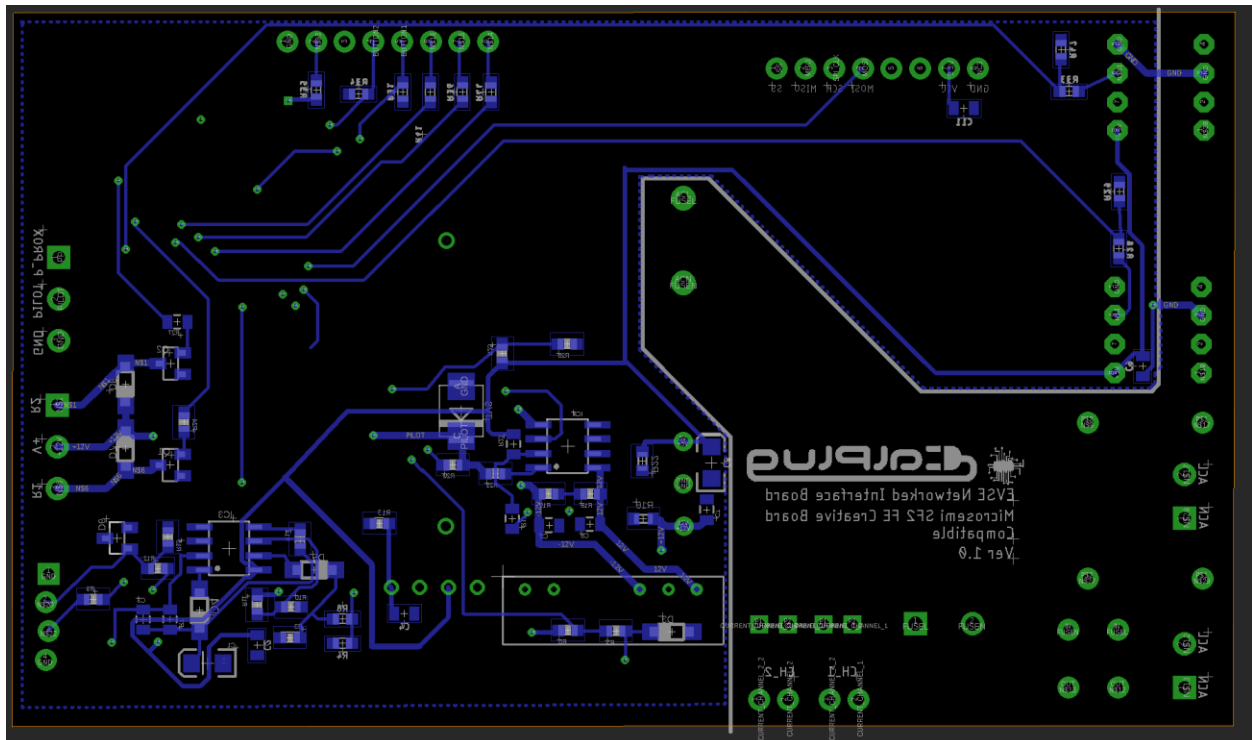


Figure 9: NXP board layout (bottom layer)

As mentioned previously, this version of the board was cancelled in order to fulfill Smartenit's request to develop a Wifi capable board based on the ESP WROOM-32 module.

The resulting phase 2 board is based on a reference design supplied to our team by Smartenit and used in their own prototype EVSE controller. The board schematic and layout in Figures 10, 11, and 12 is reverse-engineered for use with the ESP-32 WROOM MCU for ease of interfacing with the device for software integration. It is important to note that this controller board is designed to be used with a secondary power board designed by Smartenit. This board incorporates the DC relays, power reading coils, and charge output circuitry, so the board sub-systems are very different from the phase 1 design. These components were all provided to us by Smartenit, and we worked closely with them during the rest of development of this board.

Important note: since the design of this board it has been discovered that 2 additional loading resistors are required between the inputs B1 and B2 of the multiplexer and ground. A value of 4.7k Ohms has been shown to be sufficient in this case.

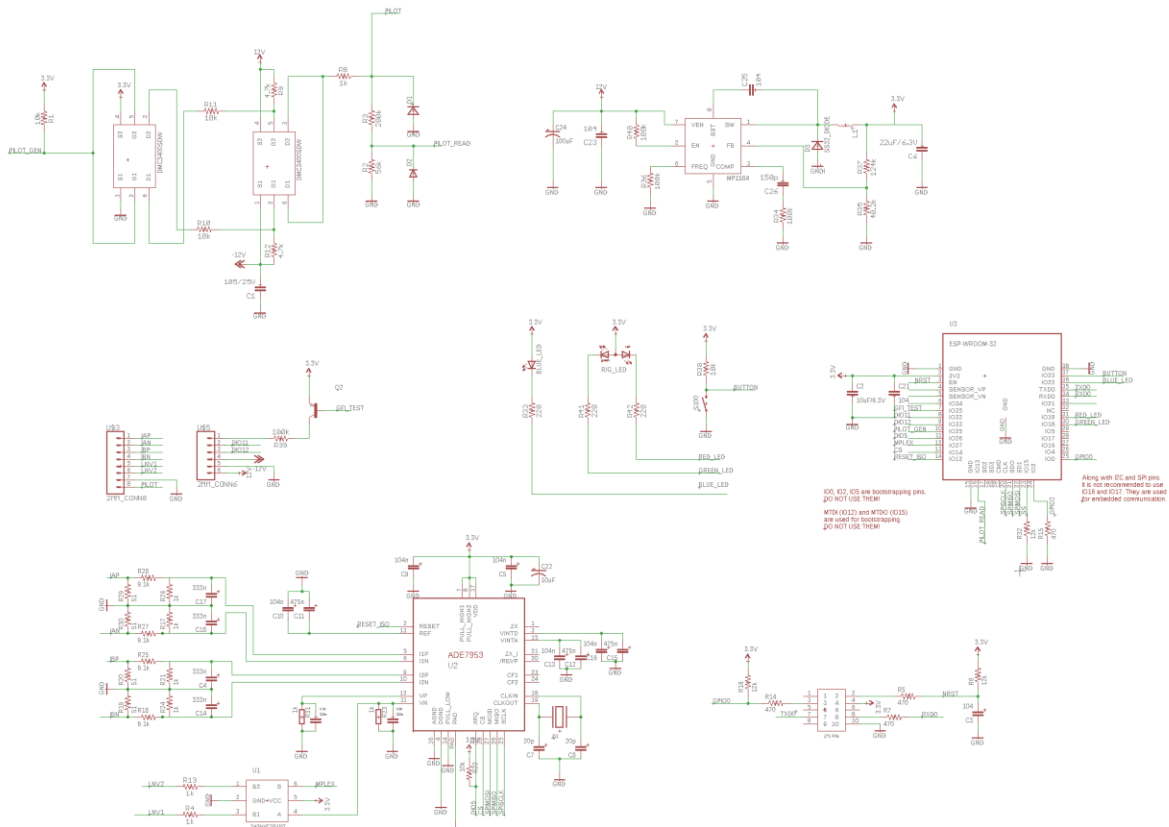


Figure 10: EAGLE Schematic for Phase 2 Design

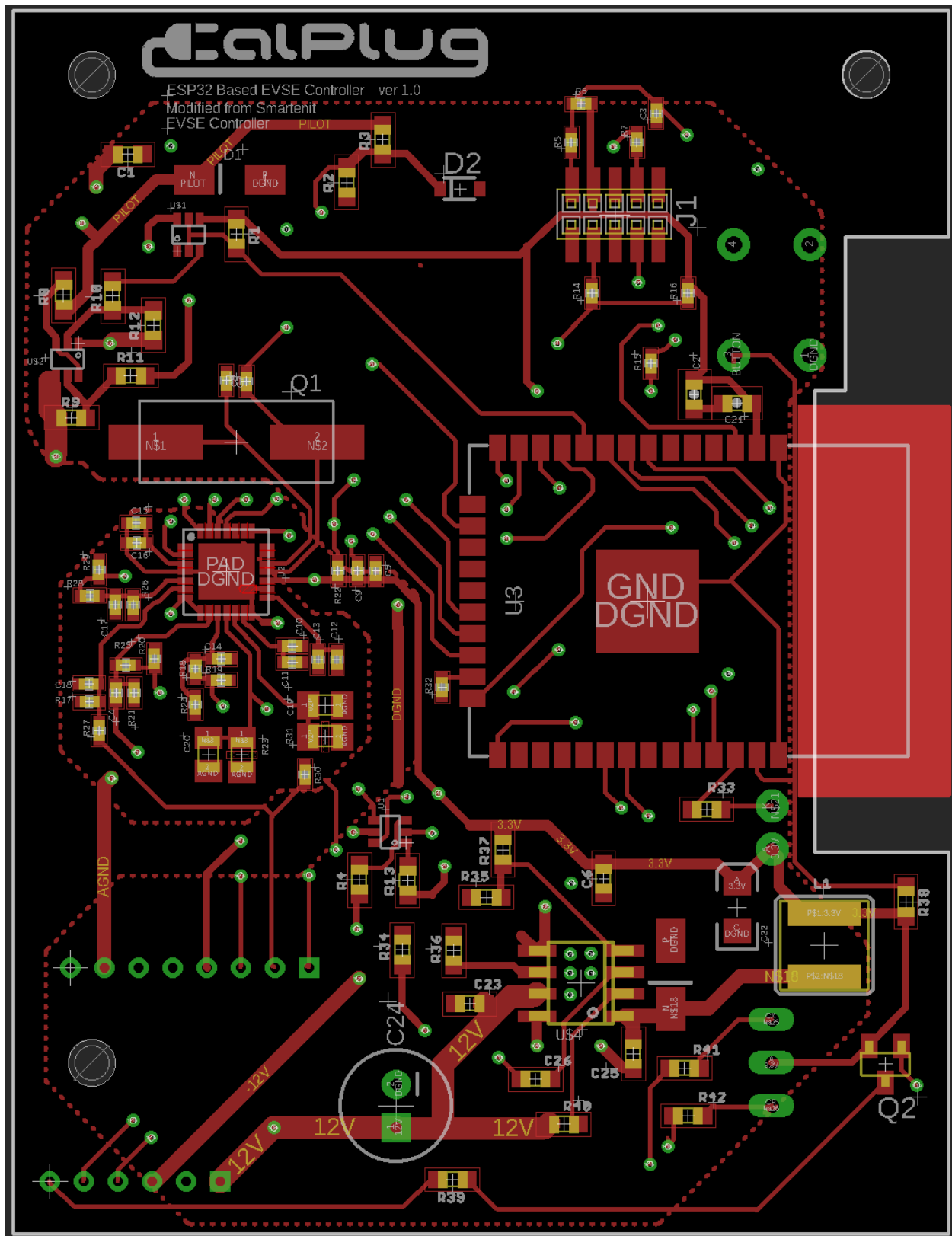


Figure 11: Phase 2 EAGLE layout top layer

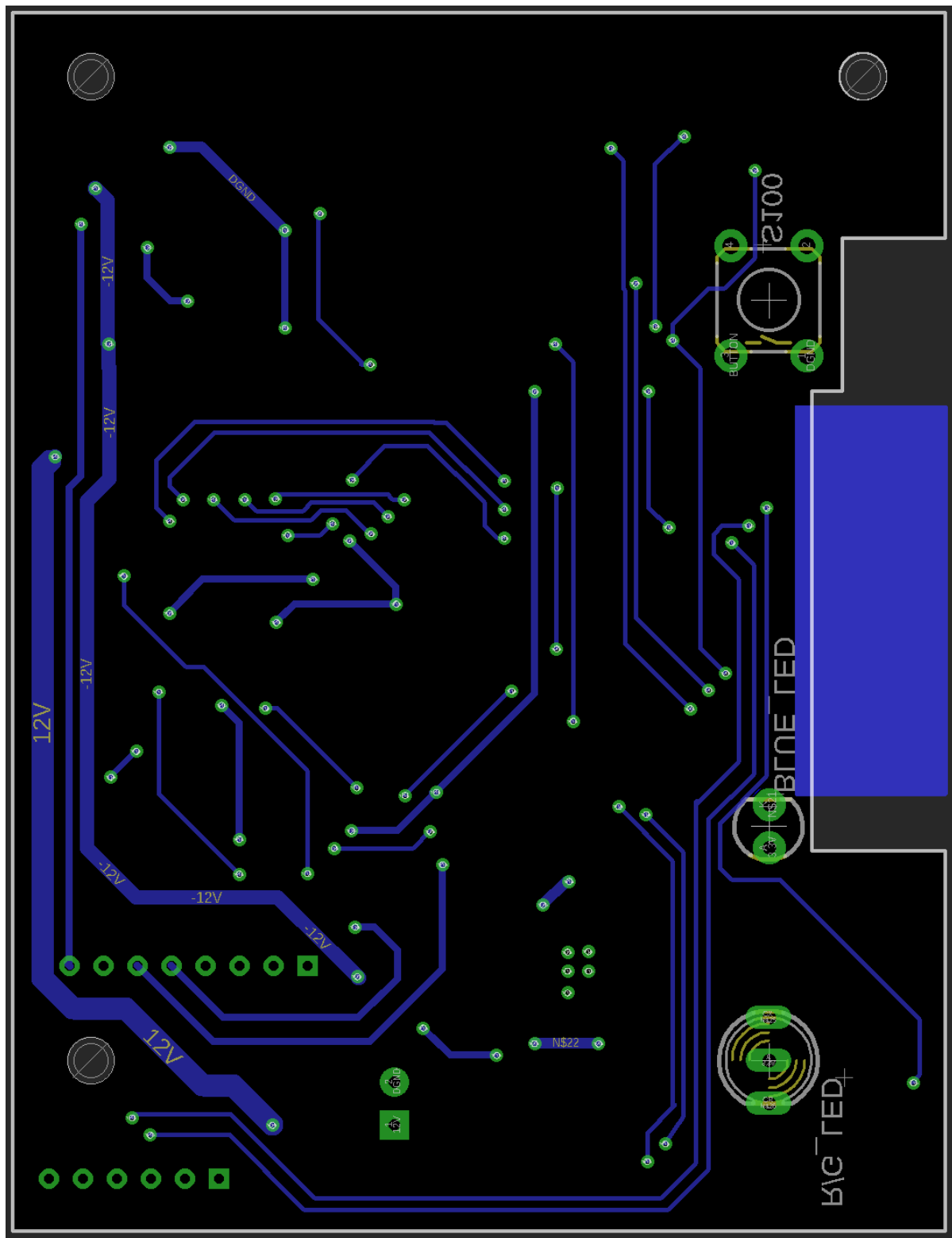


Figure 12: Phase 2 EAGLE layout bottom layer

-Sub Circuitry

There are three major sub circuits used in this design. Details on each circuit are highlighted below:

1) Pilot Generation

Pilot circuitry takes an input 1KHz pwm signal from the MCU and amplifies it to +12/-12V. This signal is then rectified and sent to the car. Simultaneously, this circuitry uses a resistor network to transform the pilot signal to a tolerant voltage for the MCU to read. Reading the pilot line determines the state of the device, as the car uses its own resistor networks and switches to change the voltage of the pilot line on its own.

2) Level 1/Level 2 Detection

The ADE chip is allowed to read two AC line voltages through the use of a multiplexer. This multiplexer is controlled by the MCU to allow the ADE7953 chip access to two AC lines, in the case that level 2 is available. Setting the mux pin (pin “S” according to the datasheet) to zero sends level one voltage to the ADE. Setting the mux to digital high sends level two voltage to the ADE chip.

3) Output to Relay

The power relays for this design are on the power board, not the control board. However, the relays are controlled through the use of 3 pins. The power board features an amplifier (LB1909) to amplify the MCU’s signals to 12V for the relay. One pin from the MCU is used for enabling the amplifier, while the other two pins control which relays are turned on or off. The relays are latching, so operation of them is much like writing to memory. Outputting low on the enable pin activates the chip to write the values of the other two pins to the relays. If the two relay pins are high and the enable pin is low, the relays switch on. If the two relay pins are low, and the enable pin is low, the relays switch off. However, if the enable pin is high, the relays will not open or shut, since the driver chip is disabled when enable is high.

Ground Fault Interrupt (GFI) detection in this design is done through the ADE7953 chip. This is done by subtracting the current readings between the two current channels that the ADE chip reads off of the power board. If the subtraction results in something other than zero, a ground fault has occurred.

One major improvement of this board over the phase one board is the introduction of multiple ground planes. Digital circuitry is inherently extremely noisy from switching. As a result, power readings and other more noise-sensitive circuitry can become inefficient, especially in the application of many analog control systems, as this design is abundant in. As a result, there are two ground planes in this design: analog ground and digital ground. The analog ground plane is primarily used for grounding the components taking power readings from the power board and is located nearest to the ADE7953 chip. Digital ground connects all other circuitry together. Both ground planes meet underneath the ADE chip, as suggested by its datasheet.

-Pin Connection Table

Signal Name	Signal Function	Connection	Output/Input
PILOT_READ	Read pilot signal voltage	MCU (IO39)	INPUT
PILOT	Pilot signal to car	Charge Cable	OUTPUT
PILOT_GEN	PWM signal for Pilot circuitry	MCU (IO25)	OUTPUT
DC_RELAY_1	Open/close relay 1	MCU (IO32)	OUTPUT
DC_RELAY_2	Open/close relay 2	MCU (IO33)	OUTPUT
DC_RELAY_ENABLE	Enables amplifier on power board	MCU (IO21)	OUTPUT
VP	Test for level 1 availability	ADE (VP)	INPUT
IPA	Input current channel 1	ADE (IPA)	INPUT
INA	Output current channel 1	ADE (INA)	INPUT
IPB	Input current channel 2	ADE (IPB)	INPUT
INB	Output current channel 2	ADE (INB)	INPUT
BUTTON	System configure button	MCU (IO34)	INPUT
BLUE_LED	activate/deactivate blue LED1	MCU (IO22)	OUTPUT
GREEN_LED	activate/deactivate green LED1	MCU (IO4)	OUTPUT
RED_LED	activate/deactivate blue LED1	MCU (IO16)	OUTPUT
12V	12V power	MP5184/Pilot Amplifier	INPUT
-12V	-12V power for pilot signal	Pilot Amplifier	INPUT
DGND	Digital Ground	MCU (GND)	INPUT
MUX	MCU selector signal for mux	MCU (IO27)	OUTPUT

Just as in the phase 1 board, the ESP32 module communicates with the ADE7953 through SPI. SPI for the ESP32 module for connected peripherals typically use 3 and 4 of the ESP32 SPI modules. The first two SPI modules are used for EEPROM flashing of instructions to the MCU, so they cannot typically be used.

Flashing the ESP WROOM-02

A few minor adjustments have been made to successfully flash code to this device. The J1 connector on the board is intended for flashing purposes, however a proprietary flashing module is used by Smartenit to flash code to the device. If that cable is missing, the following steps can be taken to flash the ESP board manually. First, connect the power, TX and RX pins of an FTDI breakout board to the correct pins on the PCB J1 connector (labelled on the schematic). Then during flashing connect both the GPIO0 and NRST pins to ground. Release the NRST while still holding GPIO0 to ground and the flashing process should begin. Holding NRST to ground resets the chip and holding GPIO0 to ground, after resetting, puts the chip into boot mode. GPIO0 can be released after the WROOM-32 successfully enters boot-mode. Note, that in order for the chip to enter normal operation, it must be reset again after successfully flashing.

EVSE Simulator

In order to demonstrate proper operation of our controller, we constructed a simple electric vehicle simulator which uses lights and space heaters to simulate the load of an electric vehicle. By plugging in the J1772 plug and flipping the ON switch, charge is initiated with the controller and AC power will flow into the simulator. A triac was used in concert with an Arduino Mini Pro, in order to dim the lights according to the duty cycle of the incoming pilot line pulse. There are 4 charging modes built into the simulator, which are controlled by 2 SPST switches on the J1772 connector box. They control how bright the light bulbs will get as the “car” is charging with between 6-40 Amps. The power modes are summarized in the table below, and the simulator and switches can be seen in the following pictures as well.

Table 3: Operation of EV simulator power modes		
Switch A	Switch B	Operation
off	off	AC off -No power
off	on	50% power: 20 Amps at max (half brightness at max charge)
on	off	100% power: 40 Amps at max (full brightness at max charge)
on	on	AC always on max power

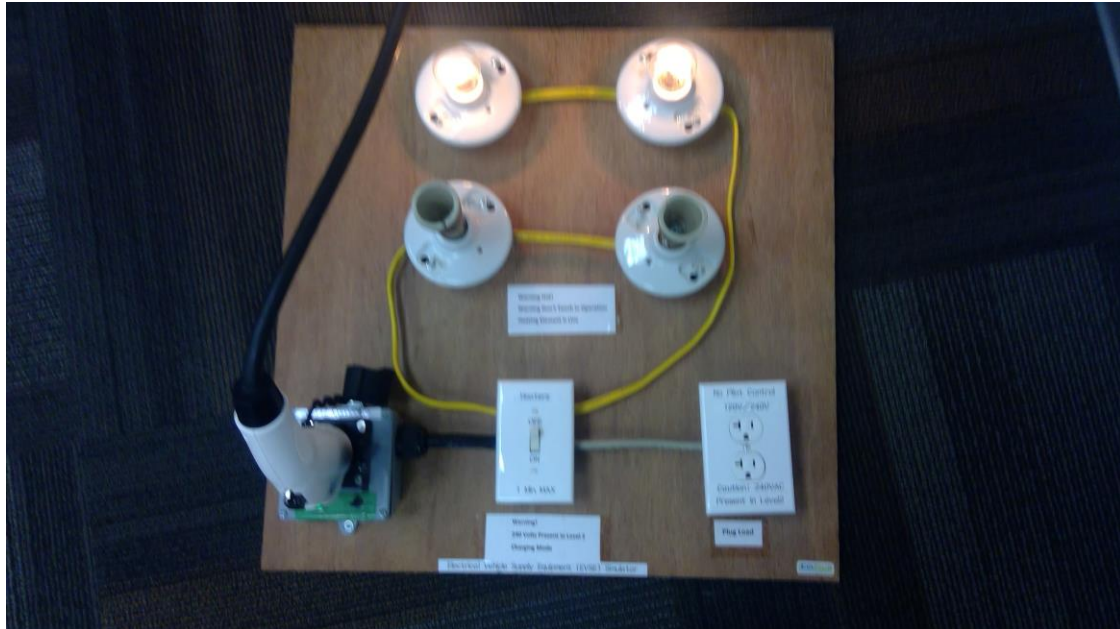


Figure 13: EV Simulator



Figure 14: Charge mode switches on EV Simulator

Bill of Materials for PCBs

The parts lists for both the phase one and phase two boards can be found at the following Google Sheets links. They include part names and URLs to each of the components on the PCBs. Additionally, the excel files for these have been attached to this report for reference.

Phase 1 board:

https://docs.google.com/spreadsheets/d/1wdp51o8JV1KYRSJ3N7ixFmwc_FVero3h5fXNqp7kQkU/edit#gid=0

Phase 2 board:

https://docs.google.com/spreadsheets/d/11fM1nPDNoOCqIDIU4FM2bKbLwWIEnj_bDYVJhdz3DHc/edit#gid=0

Section 2: Software Design

Overview

This component of the project mostly entails the software created to control the hardware through various stages of its use. The first stage runs the hardware through the initialization phase. During this stage, various inputs and outputs are setup to provide functionality for the hardware such as light-emitting diodes, a button, Wi-Fi internet connection with MQTT functionality, a multiplexer switch, relays with an relay enable pin, serial peripheral bus initialization with a wattmeter module, and a pulse width modulated signal.

Controller User Interface States

The light-emitting diodes are attached to a pulse width modulated signal with a frequency of 1Hz. This allows the charger to turn on the LEDs for a customized amount of time in order to denote the different states of the charger. The red LED denotes that the charger has disconnected from the MQTT server or when the charger has detected a hardware error (State F). A blinking green LED (State A) indicates that the charger has passed all hardware safety checks and is ready to be connected to the electric vehicle. A static green LED (State B) indicates that the charger has detected that the J1772 plug has been inserted into the vehicle and is ready to provide charge. Upon detecting that the car has been plugged in, the microcontroller turns on the relays to produce the charging signal on the pilot line. Once in charging mode (State C), the blue LED turns on in conjunction with a static green LED. The blue LED utilizes the same duty cycle that is being provided to the car as a visual cue for the user. A blue LED that is on for a long time before turning off again demonstrates that the car is currently being provided with a large amount of current. Likewise, if the blue LED is on for a short amount of time before turning off, that is an indicator that the car is currently provided with a small amount of charge. If the charger is in a failstate or reads a faulty voltage, only the red LED will turn on and flash on and off at an equal rate.

LED	State	Description
Red	F	disconnect/error
Blinking green	A	Ready to connect
Static green	B	Charger is plugged
Blue and green	C	Charging state

Table 4: Corresponding states and descriptions for each LED

ON duration	OFF duration	Current
--------------------	---------------------	----------------

Short	Long	6A
Medium	Medium	18A
Long	Short	40A

Table 5: Blue LED on and off duration for different currents

The following code snippet provides an example of the pulse width modulation signal generated on the LED pins. It is also used to generate the duty cycle needed to throttle the amount of charge being provided to the electric vehicle. This can be found in the hardware core libraries for the esp32 under esp32-hal-led.c.

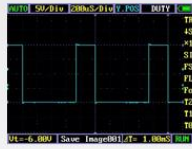
```
ledcAttachPin(LED_PIN_BLUE, 1);
ledcSetup(1, freq, resolution);
ledcWrite(2, 500);
```

The esp32-hal-led library has 16 channels available to produce a pulse width modulated signal. The first step requires the user to select the GPIO pin that will be utilized to output the PWM signal. This is selected within the first parameter of the **ledcAttachPin()** function. The second parameter selects the channel from the 16 channels available to provide this signal. Once attached, the **ledcSetup()** function determines the frequency of the signal as denoted by the variable **freq** above in addition to the resolution of the produced wave. The **resolution** variable denotes the number of bits used to adjust the duty cycle on the channel. A higher resolution results in a PWM that can be adjusted with greater accuracy. Lastly, the **ledcWrite()** function generates the signal on a defined pin. The second parameter of this function represents the duty cycle to be produced on the line. Using an example resolution of ten bits, writing 1023 into this parameter will fully turn on the signal, whereas writing 511 will adjust the signal to approximately 50% duty cycle.

J1772 Protocol

A 1kHz square wave signal is required by the J1772 during charging cycles. The duty cycle present on this signal denotes the amount of current the electric vehicle shall receive during its charge. The following chart [5] provides several examples of the duty cycle and current exchange rate based on a standardized formula.

J1772 Duty Cycle



The J1772 Pilot is a 1kHz +12V to -12V square wave, the Duty cycle (ratio high state to low state) determined the maximum available current. The EVSE sets the duty cycle the EV must comply to original setting or changes to the duty cycle.

6A - 51A

$$\text{Amps} = \text{Duty cycle} \times 0.6$$

$$\text{Duty cycle} = \text{Amps} / 0.6$$

51A - 80A

$$\text{Amps} = (\text{Duty Cycle} - 64) \times 2.5$$

$$\text{Duty cycle} = (\text{Amps} / 2.5) + 64$$

Amp	Duty Cycle	Amp	Duty Cycle
6A	10%	40A	66%
12A	20%	48A	80%
18A	30%	65A	90%
24A	40%	75A	94%
30A	50%	80A	96%

Figure 15: J1772 corresponding duty cycles for different current levels

Additionally, the charging level detected by the hardware determines the ceiling cap that can be generated by the charger. Level 1 charging can provide only up to a maximum of 16 amperes during charging whereas level 2 charging can increase that ampere output up to 80. However, for the purposes of this project, this charger can provide no higher than 40 amps to the vehicle. Requests for currents higher than 40A or less than 6A will be ignored by the device. [6]

	Voltage	Max Current
Level 1 (L1)	120VAC	16A max per J1772
Level 2 (L2)	208-240VAC	80A

Table 6: Current mappings to Voltage Levels

This duty cycle output is generated using the same functionality to create the blinking LEDs. If this value is updated through the MQTT messaging service or through interactions with the onboard button interface, it is applied immediately to the output simply by calling the `ledcWrite()` function during the device's next loop cycle.

The hardware needs to be able to detect changes in states within the charger. Initially, the J1772 interface produces a 1kHz square wave signal with a 12 Vpp. This sends a signal to the hardware that nothing is currently connected to the plug, thus no changes need to be made within the hardware. The following chart denotes more states that can be generated by the pilot signal and their significance.

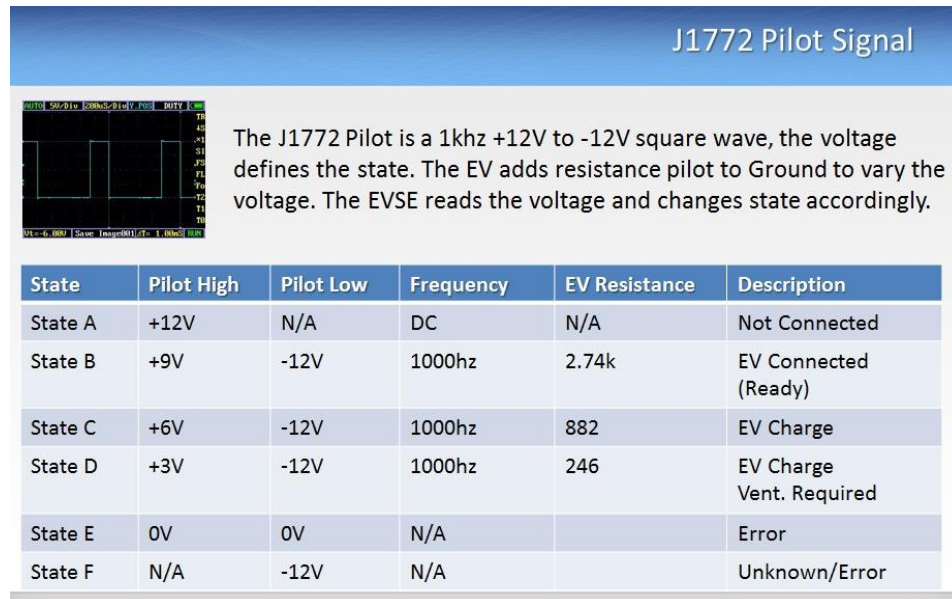


Figure 16: The different car states and their descriptions

ESP-32 ADC

The ESP32 module has a multitude of analog to digital converter channels available to measure small input signals. These channels gauge which state the pilot signal is currently outputting in order to make adjustments within the hardware. However, there are certain limitations when using the onboard ADCs on the ESP32. Firstly, there are two sets of channels; ADC1 and ADC2. ADC1 channels are attached to GPIOs 32-39 and can be used freely. ADC2 channels are attached to GPIOs, 0,2,4,12,13,14,15,25,26, and 27. However, configuring channel ADC2 limits the functionality on the ESP32, specifically when it comes to using the Wi-Fi driver. The Wi-Fi driver has higher priority on the ESP32 module, and prevents ADC voltage readings attached to the ADC2 channel whenever the Wi-Fi driver is active. [7] Individuals interested in programming both the Wi-Fi driver and the ADC in conjunction need to be wary of the fact that the GPIOs attached to the ADC2 channel are unusable when using the ESP32 to connect online. Secondly, the attenuation on the ADC can be adjusted to read up to 3.3V. Unfortunately, the ADC has a non-linear response to these inputs which makes automatic mapping difficult. This results in a need to map out the results from the ADC manually by adjusting the input voltage and producing an average from these readings. Alternatively, a separate ADC module can be connected to the ESP32 instead to obtain a linear response from the inputs [8].

The following code snippet shows an example of how to setup a GPIO on the ADC1 channel for the ESP32 module. Note that this only applies to GPIO pins available on the ADC1 channel. ADC2 requires different functions to be called and the Wi-Fi driver to be off.

```
adc1_config_width(ADC_WIDTH_BIT_12);  
adc1_config_channel_atten(ADC1_CHANNEL_3, ADC_ATTEN_DB_11);  
int x = adc1_get_raw(ADC1_CHANNEL_3);
```

The **adc1_config_width()** specifies the width of the ADC capture, maxing out at 12 bits. The second function **adc1_config_channel_atten()** configures which GPIO pin is going to read the voltage values in the first parameter while the second parameter sets the attenuation of the ADC capture. Finally, after the initialization is completed, the ADC channel is capable of reading inputs by calling **adc1_get_raw()**. This function returns an integer that is dependent on the `adc_width_bit` set in the configuration process.

For the purposes of this project, an ADC width bit of 12 was used to capture inputs from the pilot pin. This results in a resolution capable of displaying up to 4095. However, since the pilot signal is a square wave that is also dependent on the duty cycle being generated by the charger, the ADC picks up the voltage when it is low and cannot adjust setting since most other states use the same pilot low voltage. To remedy this, the charging device polls the ADC once during every loop cycle and averages out the values received after every one-tenth of a second. The average determines which state the pilot signal is generating and adjusts the hardware accordingly.

ADE7953 Metering Chip Interface

The ADE7953 is a hardware module that measures electrical energy for single phase programs. It can measure the voltage and current on the line and calculate a variety of energies, including active, reactive and apparent energy. Furthermore it is also capable of obtaining the rms values for voltage and current. This ADE7953 chip can communicate with other microcontrollers through a variety of communication interfaces including SPI, I²C, and UART. SPI was used for the purposes of this project since an example library exists on the CalPlug GitHub. This library provides a sample of the communication interface. It also includes several functions that can return certain values from the ADE chip such as instant voltage, V_{rms} , instant current, I_{rms} , reactive energy, and apparent power for starters. The SPI master passes pre-determined addresses through SPI to the ADE slave via the MOSI line. These addresses are mapped to specific measurements on the ADE. Once accepted, the ADE responds and returns the respective value back to the SPI master where it can be utilized further.

There are three SPI lines available on the ESP32 module including FSPI, VSPI, and HSPI. The FSPI bus is normally attached to the flash and uses GPIOs 6, 7, 8, and 11. However, using this bus may cause interference within the ESP32 module when it is attempting to read from the EEPROM and the ADE chip at the same time. This may result in undesirable results within the ESP32 module including software panics which causes it to reset, or provide faulty readings when communicating with the ADE. Thusly, it is strongly recommended to use VSPI or HSPI pins instead for SPI interfacing. The Arduino IDE has libraries included for interfacing through

SPI and provides several examples that showcase its functionality. However, this built-in library is not naturally compatible with the ESP32. Using this library may cause issues within the ESP32 and force it to reset because mutex-locks are not implemented. A separate SPI library for esp32 titled **esp32-hal-spi.h** and **esp32-hal-spi.c** come included when installing the ESP32 resources on a local machine. This library implements mutex-locks on a system level, allowing it to utilize the SPI interface. A version of the ADE7953 library has been ported using the ESP32 SPI library and is available on this project's GitHub for reference.

A certain issue arises when the ESP32 attempts to communicate with the ADE7953 chip on the PCB. The reset pin on the ADE7953 is normally held **LOW**, preventing it from establishing communication with the ESP32 master. This was an issue that plagued the team for several days and thus prevented progress. It's important to note that the GPIO reset pin for the ADE needs to be pulled **HIGH** on the software level in order to establish communications. If an attempt to interface with the ADE through SPI is made when the reset pin has not been pulled down, odd data will be returned to the ESP32 master. Such scenarios include receiving a negative 0 for a voltage reading. This is a good indication that the ADE is not communicating correctly and needs to be remedied.

The ADE7953 is critical to many of the functions implemented on the software. For starters, it provides voltage and power measurements on the line that are useful for the purposes of this project. However, this particular chip can only read one line at a time. In the event that level 2 charging is available, the device will switch the readings using a built-in multiplexer and read the voltage on that line as well. This software detects that the voltage readings provided to the microcontroller are within valid ranges otherwise it sets an internal flag to enable the level detection failure feature. For a level one reading to be valid, the first line needs to be within a 120V voltage range and the second line reading needs to be near or at 0V. If these voltages are switched, an internal flag representing the **voltages are backwards** is set and charge is prevented from occurring. Additionally, if the voltage readings are floating between 0 - 120V or are significantly higher than 120V, this is counted as a **voltage reading failure** and charge cannot occur. When attempting to decipher if level 2 charging is available, both lines need to be approximately 120V. If both lines read 0V, an internal flag representing **ground check failure** is set to true and charge cannot occur until this is remedied.

Upon verifying that the charger is in either a level one or level two charging mode, the relays need to be set accordingly. Level one charging requires that only one of the relays be on for charging purposes whereas level two charging requires that both relays be on. In the event that a hardware failure is detected, both relays are shut off and are prevented from changing state until the hardware recovers through a system reset. In order to switch the states of the relays, the relay enable pin must be set to low first. Afterwards, the state of the relays can be set simply through a digitalWrite() to their respective pins. Writing **HIGH** turns on the relay while writing **LOW** turns it back off. After adjusting the states of the relays, it is strongly recommended to disable the enable pin for the relays for safety purposes. Additionally, a stuck relay check function needs to be implemented for the same reasons. If all the hardware tests pass and the load has not been turned off manually by the user or by an MQTT broker message, the

microcontroller will automatically turn on the relays when it detects that the charging state has changed to State B. If the

The onboard button provides limited functionality to interface with the device. A five second timer is started whenever the button is pressed for the first time. During this five second interval, subsequent button presses increases the tally stored by the microcontroller. Once the five second timer expires, the charger checks the tally to generate a specific outcome. One to five presses causes the charger to toggle relays on and off. If the charger is in state C when this is pressed, this causes the charger to revert to state B and vice versa. It can be pressed in whichever state the charger is in. However, no changes will be made if the device encounters a hardware test failure. Six to eight button presses makes adjustments to the load. The charger delivers 10 amps of current when pressed six times, 20 amps of current when pressed seven times, and 40 amps of current when pressed eight times. Pressing the button between 9 or 10 times generates an MQTT message to the server the device is currently connected to. Pressing the button more than 11 times will cause a hardware reset to the ESP32. It will revert to its factory state, clearing any saved credentials for Wifi and MQTT that was stored during the time the device was on. An interrupt function is tied to each button press. However, due to the nature of button presses, a significant amount of bouncing is produced every time the button is pressed. This can cause false readings on the device and cause the wrong function to be called. A debouncing feature is implemented to decrease the number of false inputs. This is done by comparing the time the button is initially pressed and the second time it is pressed. If the second reading occurs within the debouncing range, then the device knows this is most likely an invalid reading and disregards the input.

Note that in order to obtain correct readings on the voltage and current channels, a linear normalization function must be used. Following is a table of the linear functions used for each channel. Input x is raw analog measurement from ADE chip, and output is the correct voltage/current reading.

Normalization Values for ADE7953	
ADE7953 Channel	Applied function (mx+b)
IrmsA	12.6x-17.8
IrmsB	12.6x-17.8
Vrms channel 1	
Vrms channel 2	1.24x-51.8

MQTT

MQTT is a lightweight messaging protocol which allows embedded devices to communicate with each other or the outside world. By interfacing our device with an MQTT server, the J1772 charger is able to communicate with the user via the internet. We've implemented functions that will allow our device to connect to a network and then connect with the MQTT server. Our device will initially connect to the hard-coded network, which is the **UCInet Mobile Access** network for testing purposes. Although it will automatically connect to this network, we have implemented a function that will also scan networks that are available to use, and prints out these available networks onto the serial monitor.

```
wifiscan();  
connectToWiFi(networkName, networkPswd);  
client.setServer(mqtt_server, mqttPort);  
client.setCallback(callback);
```

The fragment of code above depicts the order of connectivity. On the first line, we are scanning for any networks available and the second line will connect to the hard-coded network name and its password using the function **connectToWiFi** that is provided by the **WiFi.h** library in Arduino.

Once it has been successfully connected to the internet, it will connect to the mqtt server that has also been hard-coded for testing. For MQTT server connection, we have to provide the name of the MQTT server, port, user and password, and these information can be obtained from the CloudMQTT account. In the above code, the third function takes the provided MQTT server and MQTT port to set which server will be in use. After this setup, we connect to the server using the function below.

```
client.connect("ESP32Client", mqttUser, mqttPassword)
```

In this function provided by the **PubSubClient.h** library for Arduino, we want to connect to the ESP32Client using the hard-coded user and password. A message of connectivity verification, i.e. "Hello from ESP32!" will appear on the websocket user interface of the MQTT broker and the serial monitor if it has been successfully connected. When this message appears, the device is ready to communicate with the user. If it has been disconnected it is able to reconnect to the original hard-coded MQTT server.

MQTT transports messages via publish/subscribe method in which the user inputs a topic and a payload. The topic is where information is stored, and payload is the desired information from the topic. In order for our device to transmit information about a certain topic, topic subscriptions on the device must be made. The following fragment of code is what we used to subscribe to a topic:

```
client.subscribe("in/devices/1/SimpleMeteringServer/LVVoltage");
```

In this piece of code, we have subscribed to the topic called, "in/devices/1/SimpleMeteringService/LVVoltage" for getting the voltage at a certain charge level. The function, **client.subscribe** can be used under the **PubSubClient.h** library. Once the topics needed have been subscribed, we implemented functionalities for specific payloads that the user may input so that when a topic and payload has been requested, the device transmits the

correct data needed. The following bit of code is our function for the functionalities of subscribed topics and payload.

```
void callback(char * topic, byte* payload, unsigned int length)
```

In a typical MQTT operation, the user inputs a topic that they want to get information from, for example, if the user wants information about the voltage when the car is charging at level 1 they must input **in/devices/SMARTXXXXXX/1/SimpleMeteringService/LVoltage** because the information about level voltage is under this topic. Then, the message corresponding to obtaining the voltage at level 1 in particular, would be

{"method": "get", "params": {"value": "L1"}}. The topic and message will then be acknowledged, and our device transmits the desired data back to the MQTT server. A flowchart representation of this example is shown below:

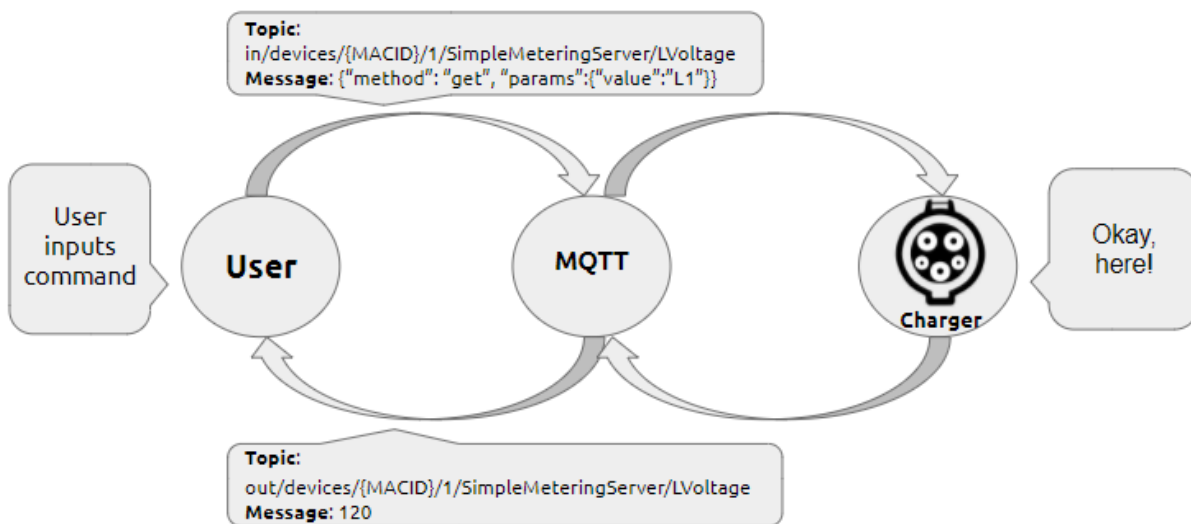


Figure 17: Flowchart of MQTT operation

In the case that the user wants to change the network to another available network, they may do so by going to the MQTT server and inputting the following topic and payload:

Topic: in/devices/SMARTXXXXXX/0/cdo/reset
Payload: {"method": "post", "params": {"value": "wifi"} {"wifiname: password"}}

The user must replace the parameters **wifiname** and **password** to their desired ones in order to connect to the correct network.

The user may also change the MQTT server they wanted to use, and they may do so by inputting the following topic and payload:

Topic: in/devices/SMARTXXXXXX/0/cdo/reset
Payload:
 {"method": "post", "params": {"value": "mqtt"} {"mqttuser: mqttpassword: mqttserver: mqttport"}}

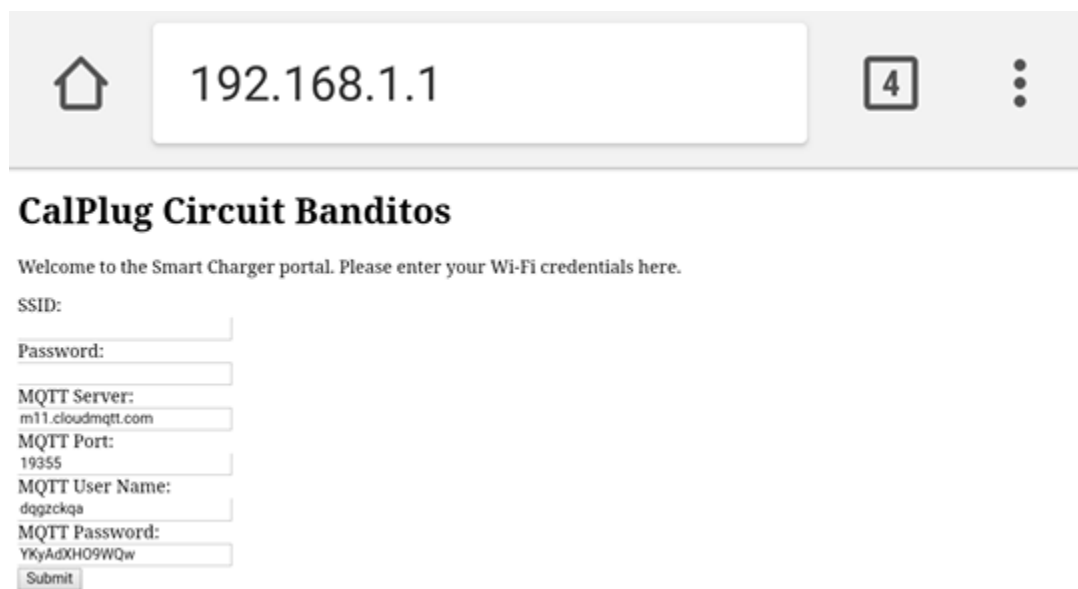
Similar to resetting the network, the user must replace the parameters **mqttuser**, **mqttpassword**, **mqttserver**, and **mqttport** to their own in order to connect to the correct MQTT server.

Access Point

We implemented the ESP32 as its own access point for its initial state. To create its own server, we used the ESP32Server example for Arduino which includes the **DNSServer.h** library. In order to connect to it a wifi enabled device, we had to create an IP address and connect to it on a web browser.

```
IPAddress apIP(192, 168, 1, 1);
```

In this code, we have the IP address as **192.168.1.1**, and it can be any arbitrary number. This IP address is the number to be inputted on the search bar of the web browser to access the portal. An example of what should be expected after entering the IP address is shown on the figure below:



192.168.1.1

CalPlug Circuit Banditos

Welcome to the Smart Charger portal. Please enter your Wi-Fi credentials here.

SSID:

Password:

MQTT Server:

MQTT Port:

MQTT User Name:

MQTT Password:

Figure 18: An example of ESP32 server for the user to enter their WIFI and MQTT credentials

In our code, we've implemented this AP to run through the use of a button press so that we have the option of running the charger with and without the wifi credentials initialized.

Smartenit Board

For the Smartenit board, we wanted to use zigbees to provide the wireless communication. We have configured the Zigbees as router and coordinator to achieve communication between the two modules. We've been able to turn on an LED wirelessly via point-to-point communication

using two Xbee series 2 modules. It is crucial that they have the same PAN ID, otherwise they will not communicate.

Our Zigbee modules have been configured in the following manner:

Coordinator		Router	
ID PAN ID	1234	ID PAN ID	1234
DL	FFFF	DL	0
NI (Node Identifier)	COORDINATOR	JV Channel Verification	Enabled [1]
AP API enable	Transparent mode [0]	CE Coordinator enable	Disabled [0]
CE Coordinator enable	Enabled [1]	NI (Node Identifier)	Router

Table 7: Current mappings to Voltage Levels

Physical Verification & Testing

This device has successfully been used to charge an electric vehicle at currents ranging from 5 to 12 Amps. The method of altering charge rates over MQTT has been proved successful, as long as the J1772 plug is disconnected from the vehicle in between changes of duty cycle of the device. Current code implementation has been proven to trigger a GFI interrupt for a 10mA current imbalance. However, once tested on a vehicle the GFI setting proved too sensitive and was moved up to trigger at 100mA. The image below shows the charger successfully charging a vehicle at 5.7 Amps (level one).



IoT SmartPlug MQTT Guide ^[9]

1. API Information

Topic	Payload	Use	Response Values
in/devices/SMARTXX XXXX/1/OnOff/OnOff	{"method": "get", "params": {}}	Reads status of load	Returns: 1: On (represented as a bool) – a state of 1 is True to indicate charging state 0: Off (represented as a bool) – a state of 0 is False to indicate no charging state
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/GeneralFault	{"method": "get", "params": {}}	Reads status of general faults	Returns: 0: OK -(represented as an integer) – a state of 0 indicates no problem was found with charger 1: GFI Fault - (represented as an integer) – a state of 1 indicates that the charger detects a failure with the GFI 2: Lvl Detection Fault – (represented as an integer) – a state of 2 indicates that the charger could not properly detect charging levels 3: Stuck Relay - (represented as an integer) – a state of 3 indicates that the charger did not

			pass stuck relay test 4: Diode Check Fail (represented as an integer) – a state of 4 indicates that the charger did not pass the diode check test
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/SUPLevel	{"method": "get", "params": {}}	Checks charging level of device	Returns: 1: L1 (Hot/Neutral) connected (represented as an integer) - a state of 1 indicates that the charger is in level 1 charging 2: L2 (Phase 1/Phase 2) connected (represented as an integer) – a state of 2 indicates that the charger is in level 2 charging 0: FAIL (Error detected) (represented as an integer) – a state of 0 indicates that the charger could not properly determine which charging state it is in.
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/GFIState	{"method": "get", "params": {}}	Reads status of the GFI test	Returns: 1: OK (represented as a bool) - a state of 1 indicates no problems with the GFI 0: FAIL (represented as a bool) – a state of 0

			indicates that the charger failed to pass the GFI test
in/devices/SMARTXX XXXX/1/SimpleMeteri ngServer/RequestCu rrent	{"method": "post", "p arams": {"value": "C urrent"}}	Sets the charging current to supply	Returns: 1: OK (represented as a bool) – a state of 1 indicates that provided value for charging is valid 0: FAIL (represented as a bool) – A state of 0 indicates that provided was not in within valid range Provide: Current: 0 – 4000 (unsigned integer, fixed point representation, divided by 100) – value represents the current in steps of 10 milliamps.
in/devices/SMARTXX XXXX/1/SimpleMeteri ngServer/RmsCurren t	{"method": "get", "pa rams": {}}	Obtains the amount of current being supplied	Returns: 0 - 4000: Current (unsigned integer, fixed point representation, divided by 100) – value represents the current in steps of 10 milliamps
in/devices/SMARTXX XXXX/1/SimpleMeteri ngServer/ChargeStat e	{"method": "get", "pa rams": {}}	Determines the state the charger is currently in	Returns: 1: B (represented as an integer) – a state of 1 indicates connected but not charging 2: C (represented as an integer) – a

			<p>state of 2 indicates connected and charging</p> <p>0: A, D, E, F (represented an integer) – a state of 0 indicates charger is not connected and not charging</p>
--	--	--	--

in/devices/SMARTXX XXXX/1/SimpleMeteringServer/LVoltage	{“method”: “get”, “params”: {“value”: “L1”}}	Checks level 1 voltage	<p>Returns:</p> <p>####: Voltage with respect to ground (unsigned integer, fixed point representation)</p>
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/LVoltage	{“method”: “get”, “params”: {“value”: “L2”}}	Checks level 2 voltage	<p>Returns:</p> <p>####: Voltage with respect to ground (unsigned integer)</p> <p>*Returns 0 if level 1 charging</p>
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/CurrentSummation/AccumulatedDemandCharge	{“method”: “get”, “params”: {}}	Read Metering summation kWh	<p>Returns:</p> <p>####: total power delivered during current cycle (unsigned integer)</p>
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/InstantaneousDemand	{“method”: “get”, “params”: {}}	Read Power kW	<p>Returns:</p> <p>####: instantaneous demand in watts being delivered (unsigned integer)</p>
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/AccumulatedDemandtotal	{“method”: “get”, “params”: {}}	Total accumulated charge in kWh	<p>Returns:</p> <p>####: Total accumulated charge delivered (unsigned integer)</p>
in/devices/SMARTXX XXXX/1/SimpleMeteringServer/GROUND0	{“method”: “get”, “params”: {}}	Checks status of ground and wiring	<p>Returns:</p> <p>1: OK (represented as a bool) - a state</p>

K			<p>of 1 indicates no problems with the ground and wiring</p> <p>0: FAIL (represented as a bool) – a state of 0 indicates that the charger failed to pass the Ground test</p>
in/devices/SMARTXX XXXX/1/SimpleMeteri ngServer/INSTCurre nt	{“method”: “get”, “params”:{}}	instantaneous supplied current	<p>Returns: ####: Instantaneous supplied current (unsigned integer)</p>

2. Load Control

Topic	Payload	Use	Response Values
in/devices/SMARTX XXXXX/1/OnOff/Tog gle	{“method”: “post”, “params”:{}}	Toggle Relay	<p>Returns: 0: OK (represented as an integer) – state of 0 indicates that the relay toggle was a success</p> <p>1: FAILSTATE (represented as an integer) – state of 1 indicates that the toggle relay failed because charger is not in appropriate state</p> <p>2: GENFAIL (represented as an integer) – state of 2 indicates that relay cannot be toggled because a safety check failed</p>
in/devices/SMARTX	{“method”:	Turn Off Relay	Returns:

XXXXXX/1/OnOff/On	"post", "params":{}}		0: OK (represented as a bool)
in/devices/SMARTX XXXXXX/1/OnOff/Off	{"method": "post", "params":{}}	Turn On Relay	Returns: 0: OK (represented as an integer) – state of 0 indicates that the relay toggle was a success 1: FAILSTATE (represented as an integer) – state of 1 indicates that the toggle relay failed because charger is not in appropriate state 2: GENFAIL (represented as an integer) – state of 2 indicates that relay cannot be toggled because a safety check failed

3. Factory Reset

Topic	Payload	Use	Response Values
in/devices/SMARTX XXXXXX/0/cdo/reset	{"method": "post", "params": {"value": "all"}}	Resets the device	Returns: 0: OK (represented as a bool)
in/devices/SMARTX XXXXXX/0/cdo/reset	{"method": "post", "params": {"value": "wifi"}, {"wifiname": "password"}}	Reset the wifi settings of device	Returns: 0: OK (represented as a bool)
in/devices/SMARTX XXXXXX/0/cdo/reset	{"method": "post", "params": {"value": "mqtt"}, {"mqttuser": "mqttpassword", "mqttserver": "mqttport"}}	Reset the MQTT Settings of Device	Returns: 0: OK (represented as a bool)

in/devices/SMARTX XXXXX/0/cdo/reset	{"method": "post", "params": {"value": "device"}}	Delete information set by user such as device name and image	Returns: 0: OK (represented as a bool)
--	--	---	--

4. Device Information (extra implementation)

Topic	Payload	Use	Response Values
devicename	name	Sets name of the device	Returns: 0: OK (represented as a bool)

Special Thanks

We would like to thank Smartenit for their involvement and support during the course of this project.

References

- [1] Mitchell, Russ. "Funding for \$3-Billion Electric Car Rebate Bill Is up in the Air." *Los Angeles Times*, August 31, 2017. <http://www.latimes.com/business/autos/la-fi-hy-ev-rebate-bill-20170830-story.html>.
- [2] "Making Sense of the Rates." Accessed November 13, 2017. https://www.pge.com/en_US/residential/rate-plans/rate-plan-options/electric-vehicle-base-plan/electric-vehicle-base-plan.page.
- [3] Meola, Andrew. "How IoT & Smart Home Automation Will Change the Way We Live." *Business Insider*. Accessed November 13, 2017. <http://www.businessinsider.com/internet-of-things-smart-home-automation-2016>
- [4] Porter, Michael E., and James E. Heppelmann. "How Smart, Connected Products Are Transforming Competition." *Harvard Business Review*, November 1, 2014. <https://hbr.org/2014/11/how-smart-connected-products-are-transforming-competition>.
- [5] "OpenEVSE Support." *Basics of SAE J1772 : OpenEVSE Support*, 28 Aug. 2015, openev.freshdesk.com/support/solutions/articles/6000052074-basics-of-sae-j1772.
- [6] "OpenEVSE Support." *Basics of SAE J1772 : OpenEVSE Support*, 28 Aug. 2015, openev.freshdesk.com/support/solutions/articles/6000052074-basics-of-sae-j1772.
- [7] "Analog to Digital Converter¶." *Analog to Digital Converter - ESP-IDF Programming Guide v3.0-Dev-1979-g2935e95 Documentation*, 2016, esp-idf.readthedocs.io/en/latest/api-reference/peripherals/adc.html.
- [8] "[TW#12287] ESP32 ADC Accuracy · Issue #164 · Espressif/Esp-Idf." *GitHub*, 24 Dec. 2016, github.com/espressif/esp-idf/issues/164.
- [9] Smartenit. "Smartenit IoT Metering SmartPlug Product Guide." *IoT-SmartPlugProductGuide*, Compacta International, 2016, drive.google.com/file/d/0BzHXEJi5UbkmVHlvOUdUdUJLTkdYTfVhU0VYY1NPNXh6N1Vr/view.
- [10] Klopfer, Michael, et al. "CalPlug/EVSE_Smart_Charger." *GitHub*, Mar. 2018, github.com/CalPlug/EVSE_Smart_Charger.
- [11] Pillai, Vysakh, et al. "Espressif/Arduino-esp32." *GitHub*, github.com/espressif/arduino-esp32/blob/master/docs/arduino-ide/windows.md.

