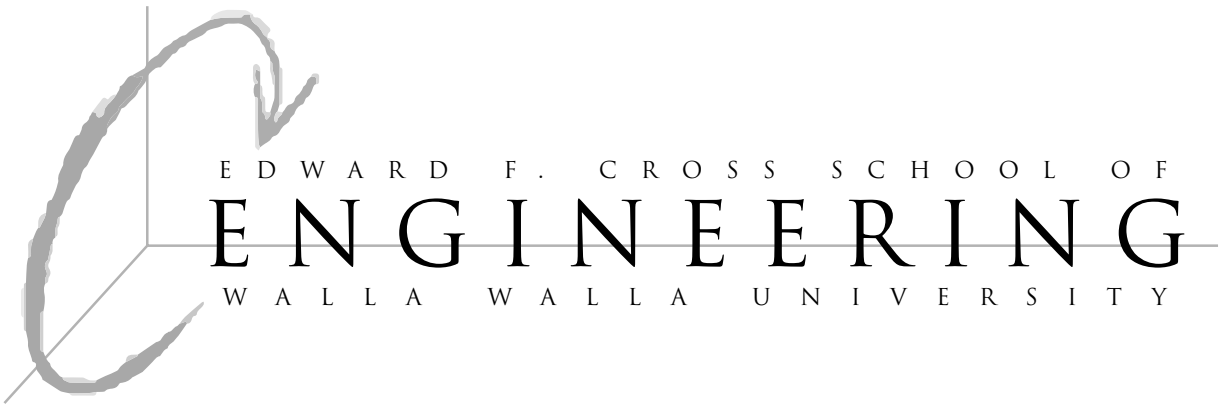


SMART DOMESTIC POWER MONITORING

DANIEL ARLT
Computer Engineering
Walla Walla University

ANDREW BINDER
Computer Engineering
Walla Walla University

MASON WILDE
Computer Engineering
Walla Walla University

Abstract

With the costs and usage of electricity continually on the rise according to the U.S. Energy Information Administration, the average homeowner is now in more need than ever to understand the power they are using and what devices in their home are contributing most to their power usage. With the help of an IOT (Internet of Things) style device that easily integrates with the technology already owned by the average consumer, people may be able to better understand how much power their devices are consuming, adjust their habits accordingly, and thus leave a smaller environmental footprint. To address this need, our project goal was to design a full end-to-end solution in the form of an IOT device consisting of hardware, firmware, and software design elements. Fundamentally, our smart plug would be designed to plug into a standard wall outlet and feature an integrated outlet for other devices to plug into. The power drawn by a device plugged into our smart plug would then be safely measured by the hardware, collected and transmitted wirelessly by the firmware over the Wi-Fi network present in many homes to a local software based web server on the network. To this end, the project was largely successful. A hardware prototype capable of high precision was constructed that was able to communicate with a microcontroller running firmware of our own design, which was then able to collect power measurements from the hardware and transmit them to our software web server. However, in its current state, the system cannot report truly accurate values, as it needs calibration. The calibration process requires precise equipment that was unfortunately not available to us due to conditions outside of our control. With proper calibration, the system created for this project is theoretically fully functional and accomplishes all of our design goals.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Brief Introduction to Topic Area

In 2018, the United States set a record for energy consumption, hitting a total of 101.3 quadrillion British thermal units (Btu), an increase of 4% from 2017, and an increase of 0.3% from the previous record set in 2007 (McFarland 2019). Most of this energy was created by the burning of fossil fuels such as petroleum, natural gas, and coal. With the cost of electricity on the average continuing to rise every year (U.S. Energy Information Administration 2019), it is time that people were made more aware of their energy consumption in the hopes that we may be able to reduce the ever-increasing demands for energy. This awareness is something that this project hopes to achieve.

At its core, the project is a pass-through smart outlet which will plug into one of the existing outlets in a home, and feature another outlet on the front to which other devices can be plugged in. The smart outlet will then monitor the power draw of the device plugged into it, collect this power data, and send it to another device on the home network by means of a Wi-Fi connection. The device that collects the data, such as a home computer, will then host a web page, and it is on this web page that the user will be able to view the collected data. The user could then view various power draw statistics as well as manage multiple smart outlets in the home.

1.2 Source of Project Idea

The original idea for this project came from a collective interest in IOT, or Internet of Things, which is a growing field of smart devices that communicate through the internet. Examples of these IOT devices are products like smart fridges, home security systems, TV's, and voice assistants. We wanted to make a meaningful contribution to this field while demonstrating our individual skills as computer engineers and providing a potential humanitarian benefit. This project is meaningful to us because it will assist people in saving money on their energy bill and help the environment in the process.

We believe that this will be a good demonstration of our individual skills as we have split up our project into three distinct parts and assigned ourselves to the parts where we know we can excel. These parts are hardware design, firmware design, and software design. Each of these pieces are uniquely different in their processes, requirements, and skills required for success and as such, should provide each of us with a unique set of challenges expected of a project at this level.

1.3 Qualitative Definition

Energy consumption in the United States is on the rise, and unless people are made more aware of the energy they are consuming, they will continue to use more power and burn more fossil fuels in the process. In conjunction with finding alternative, cleaner forms of energy production, the carbon footprint can be reduced by consuming less power. In addition, power costs are continually on the rise, which will end up costing homeowners more in yearly living costs.

To help solve this problem, the proposed smart outlet will be designed and constructed for the purpose of monitoring and tracking power usage in a domestic environment to raise a homeowner's awareness of their energy footprint. This will require solving multiple problems relating to hardware, firmware, and software design. The hardware will need to be designed to measure high voltages and current in a safe, accurate manner, the firmware will need to be designed to facilitate communication between the microcontroller and sensors, and the software will need to be designed to be reliable and easy to use.

1.4 Literature Review

After formulating the initial project idea, an online search was conducted to try and find products of a similar nature. A few similar products do exist on the market currently, such as the Stitch by Monoprice (STITCH by Monoprice 2019), a device in a family of various smart home accessories from Monoprice. The Stitch itself is a compact smart plug with useful features such as power monitoring, the ability to remotely turn an outlet on and off, as well as voice assistant integration. All these features are accessible anywhere via the internet and each plug only costs around $15. However, this product is completely reliant on Monoprice's servers which could be shut down at their discretion. In addition, the smartphone app used for setup and control is very buggy according to user reviews of the app.

Another product designed to monitor power usage is the Kill-A-Watt (P3 International 2019). This simple device is inexpensive as well at around $20 and claims to be quite accurate. Unfortunately, the device can only show instant or cumulative power, and only on the screen of the device itself, which would make it quite difficult to read if placed behind furniture. While it is effective at what it is designed to do and is quite popular, the idea that the device represents could be improved with more modern technology and wireless connectivity.

The final significant device on the market that was found is called the Sense (Sense 2019), which was the most fully featured of the three. This device integrates into the consumer's home power distribution panel and identifies different devices based on their "unique power signature" (Sense 2019). This device also allows the consumer to access their power usage statistics from anywhere, such as on a smartphone. On the downside, at $300 this device is quite expensive and could take a lot of time to pay for itself. The installation also requires a certified technician because of the dangerous high voltages associated with the installation into the distribution panel.

Key areas of possible improvement or requirements for the project were noted after studying each of these devices; the project smart plug should be designed to communicate with a controller on the user's local home network such that there is no reliance on external servers operated by a third party or the OEM. The smart plugs also need to be reasonably priced as to increase the overall accessibility of the product, as well as to keep the return on investment time low. Finally, the plugs themselves in addition to the software that controls them needs to be easy to install in a home.

2 Problem Definition

2.1 Hardware

There will be several problems on the hardware front of this project to be addressed. The first will be the selection of the microcontroller, which will require taking things like form factor, necessary processing power, and cost into consideration. These include specs such as CPU speed, RAM size, storage size, IO pins, communication protocol compatibility, and power draw. The second problem will be determining how to measure the high voltages and currents that are used by devices such as refrigerators, all while doing so in a manner that keeps the device and user from being harmed. Finally, the problem of how to combine all the hardware into a compact, integrated, single PCB solution will need to be overcome.

2.2 Firmware

The firmware will consist of two layers. The first layer will be made up of the low-level drivers that directly interface with the hardware components. This initial layer will provide control over the hardware as well as obtain relevant data for power monitoring. A potential struggle here will be understanding the various data sheets well enough to correctly communicate with the hardware in the firmware code. The second layer will be higher-level and will utilize the drivers from the layer below to control the hardware and handle the data appropriately. A challenge for this layer will be communicating with all the hardware components in real time. Another potential challenge will be handling the actual communication of the power monitoring data to the controller over Wi-Fi.

2.3 Software

The software portion of the project has two main components. The main hub will need to accomplish communication with multiple devices, the storage of data from the power monitors, as

well as provide an intuitive user interface. It will be important to choose a lightweight framework for the web server and data storage so that it can run in the background on the user's computer without it consuming excess system resources. Communication will be done using TCP network packets, which allow for maximum compatibility with networking equipment. Ultimately, it will be important to find a set of frameworks and technologies that work together efficiently.

3 Objectives

3.1 Design Criteria

There are a few criteria to be satisfied for the project. The first is accuracy in power measurement, preferably within 1% of actual power draw totals. The second is software that is responsive and easy to use. By easy to use, it is meant that it should not be complicated to an average user to view power data, see the status of each power monitor, or add new power monitoring devices to the system.

There are a few different safety codes that could be satisfied as well, considering that the device will be designed to interact with high currents and voltages within the home. There are a number of certifications for home products, such as UL's PCB Testing and Certification Services (UL 2020) and NSF's Product Certification program (NSF International 2020), which specifically address safety of home devices and appliances by pushing them to their limits. While getting this project officially certified would likely be a time consuming and expensive process, it may be possible to find documentation on the criteria that these independent labs test for, and then try to meet those same safety and performance requirements in the design for this project. Additionally, the IPC-2152 standard (IPC Task Group 1-10b 2020), which specifies guidelines for designing circuit boards with high current loads, may be helpful since the final circuit board will need to be able to tolerate up to fifteen amps. Fifteen amps is the typical breaker value for fuse boxes in most American homes.

3.2 Constraints

3.2.1 Practical

The practical constraints of this project include size and ease of use. The device will need to be relatively compact to avoid physically interfering with objects around it, so it should not be much larger than a standard wall faceplate. It will need to be simple enough for the average home user to integrate into their home and lifestyle, which means it should interface with pre-existing infrastructure in a user's home while not interfering with other existing wireless devices. This would entail connecting to the home Wi-Fi network that likely already exists to interface with the

computer running the controller software that would come with the product. The software setup also needs be relatively simple to install for people with basic experience and knowledge of technology and computers.

3.2.2 Economic

This product needs to make economic sense. Many other similar products that exist on the market are as cheap as $\sim$ \$15 (such as the Monoprice Stitch) or as expensive as $\sim$ \$300 (such as the Sense). Aiming for a price point too low would be difficult but a design that is too expensive would be prohibitive to consumers. As such, a reasonable cost constraint for the potential product that would result from this project would be \$50. This will involve choosing components with adequate specifications to meet performance goals without driving the cost beyond the final price constraint. Factors that will need to be used in determining a final product cost will include price discounts due to buying components in bulk, manufacturing costs, and profit margins. Other, smaller factors such as marketing, research and development costs, distribution, or product support costs could also be taken into consideration.

3.2.3 Safety

Since this device will be interacting with high current and voltage it will need to be safe to use. The two major areas of concern that will need to be addressed are direct harm to the user by electric shock, as well as potential fire risk. As mentioned above in the design criteria, the hardware design should incorporate various standard safety measures such as fuses or varistors, and ideally be constrained by the safety guidelines outlined in the UL and NSF's safety certifications for home appliances. Finally, if the device fails, it needs to do so in a controlled manner.

3.3 Impacts

Due to the nature of this project, the environmental, societal, and global impacts may be less significant than other projects, but they are still existent. This project has an environmental impact by raising awareness of energy consumption in the home. As a result, many homeowners may try to reduce their energy consumption, thereby reducing the amount of consumed energy resources. This project could also practice environmentally conscious manufacturing techniques, such as using lead-free solder, or recycled plastic for the housing.

In terms of societal impact, a proposed device such as this could help to create a society of people who are more conscious of their energy consumption, where that energy is coming from,

and how it is produced. Finally, if technology like this was adopted more globally, the positive environmental and societal impacts could begin to occur on an even larger scale.

4 Hardware Design Development

4.1 Hardware Design Inspiration

The hardware design was partly inspired by the Microchip ATSMARTPLUG, an IOT reference design from Microchip for a smart power monitoring plug with Wi-Fi and Bluetooth connectivity. This reference design accomplishes many of the same goals for this project, namely power monitoring with data reporting via a Wi-Fi connection to a local host. But, more importantly, the hardware components used by the ATSMARTPLUG are disclosed and well documented.

In particular, the ATSMARTPLUG uses the Microchip Smart Connect ATSAMW25 module based on an ARM Cortex M0+ microcontroller for processing and communication, and an ATM90E26 power monitoring IC to collect power usage data. This information served as a great place to start from a hardware design perspective.

4.2 Hardware Design Overview

For this project, the Particle Photon microcontroller was selected for processing and communication along with the Microchip ATM90E26 for collecting data on the power being consumed. The implementation of the ATM90E26 is based on the publicly available reference design from Microchip. From this, a basic diagram of the hardware components was extracted. This basic diagram can be seen in Figure 4.2. Following Figure 4.2 are sections describing each of the hardware components, as well as the analysis and reasoning involved in selecting the components used.

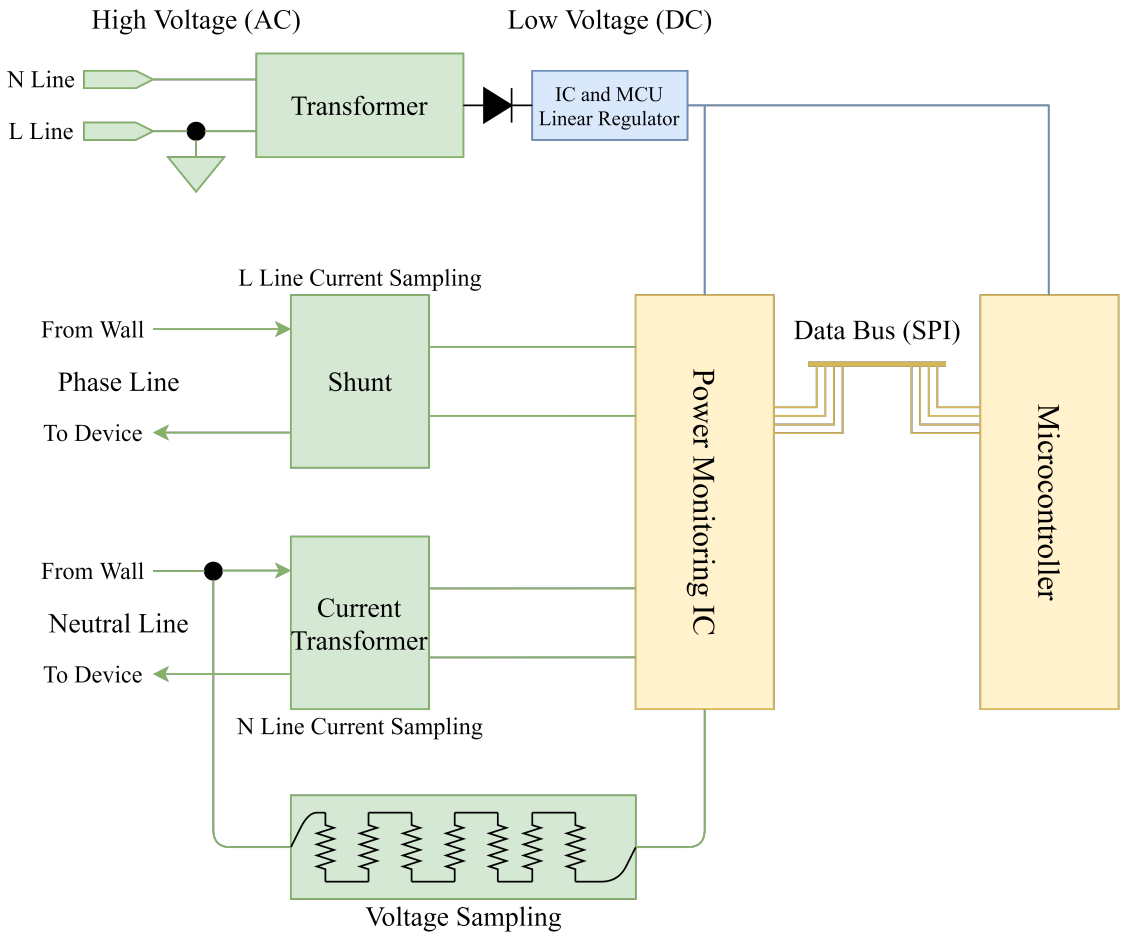


Figure 4.1: Overview of Hardware Design

4.3 Hardware Analysis and Component Selection

4.3.1 The Microcontroller

The microcontroller has two primary roles; retrieving the power usage data from the power monitoring IC as described in Section 4.3.2, and communicating that data to the TCP server, which is further described in Section 6. As such, there were only two core requirements for the microcontroller from a hardware perspective; built in Wi-Fi for communication, and a SPI or UART interface for communicating with the power monitoring IC. Processing power was not a significant concern, since the microcontroller's primary tasks would be retrieving data, processing the data using simple math operations, manipulating the data into the correct format for transmission, and then sending that data over the Wi-Fi interface. Storage capacity was also not a large concern, since long term storage of any data would not be necessary. Ideally, the microcontroller will forward data to the TCP server shortly after it becomes available so that the data can be recorded into the database.

After consideration of multiple microcontrollers such as the Microchip ATSAMW25, Espressif ESP8266 and ESP32, and Particle Photon, the decision was narrowed down to ESP8266 and Photon. Both boards met the hardware requirements, so the ultimate deciding factor was ease of firmware development. This resulted in the final selection of the Particle Photon, and the reasoning for the selection from a firmware development view are further detailed in Section 5.1.

4.3.2 The Power Monitoring IC

The power monitoring IC is responsible for sampling and collecting data on current flow from the phase and neutral lines as well as the voltage from the neutral line. From this collected information, the IC can determine both active and reactive power draw. There are several commercially available power monitoring IC's available on the market, but the final decision was to use the ATM90E26, the same IC used in the Microchip ATSMARTPLUG reference design. This chip has the advantage of being in full compliance with multiple metering requirements, namely IEC62052-1, IEC 62053-21, and IEC62053-23, as well as being very accurate at 0.1% for active energy usage and 0.2% reactive energy usage over a dynamic range of 5000:1. In addition, all of the data from the ATM90E26 is stored in a multitude of digital registers, which means collecting power data is as simple as having the microcontroller read the register(s) that data is needed from. Information such as system status, calibration statistics, energy usage, and measurements can all be found in these registers.

On the downside, the ATM90E26 has a reference design that, at first glance, was difficult to interpret. The reference design for this chip is shown below in Figure 4.2.

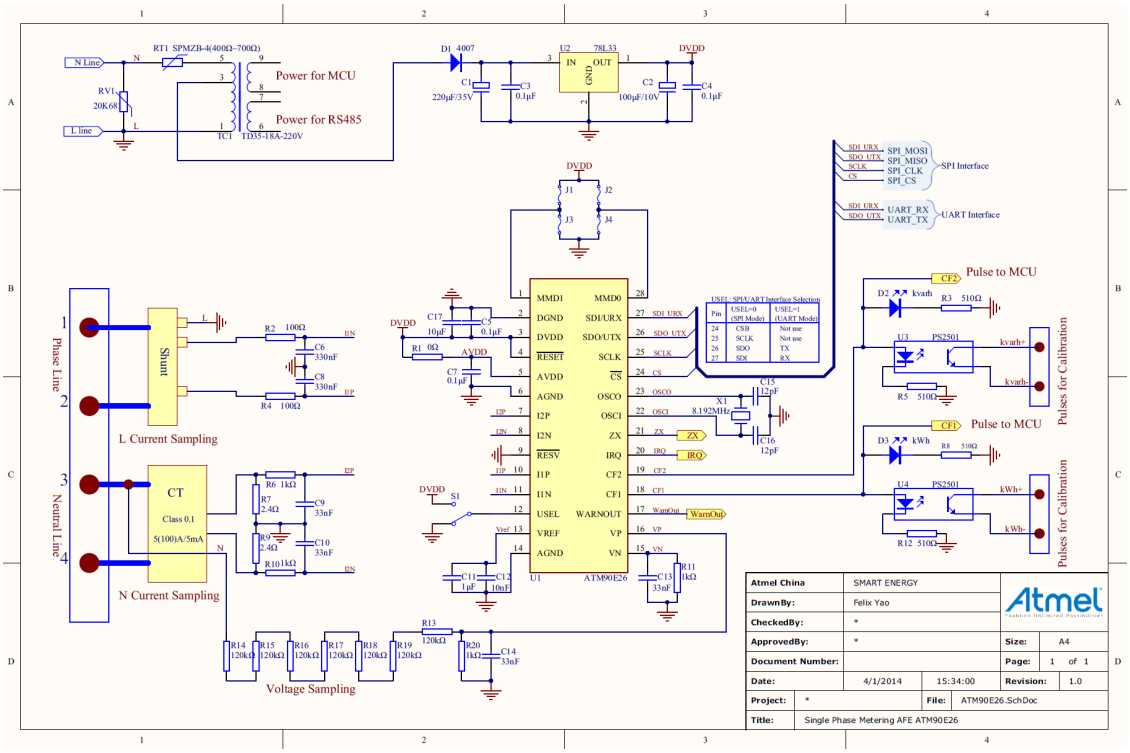


Figure 4.2: Microchip ATM90E26 Reference Design

Some of the problems with this reference design include design choices that were never explained in the application note, values that were not given (such as the shunt resistor and current transformer), as well as a lack of equations or guidelines in the application note for selecting values for these unspecified components. Because of this initial intimidation, some other power monitoring IC's were considered such as the Microchip MCP3905A, which has a much simpler and more complete application note. However, its data collection abilities are not nearly as complete as the ATM90E26, nor does it use a standard method of communication such as UART or SPI like the ATM90E26, instead relying on a rate of pulses to communicate data. These drawbacks would have made for a less capable final result as well as a more difficult firmware development process, so the decision was made to proceed with the ATM90E26.

4.3.3 Selection of the Current Sense Resistor

The current sense resistor, also known as a shunt, is responsible for measuring current on the phase line. Current sense resistors are characterized by their very small and precise value of resistance. By measuring the voltage drop across the resistor, the current flowing through it can be calculated using Ohm's Law. The current sense resistor section from the project schematic along with the calculated value is shown below in Figure 4.3.

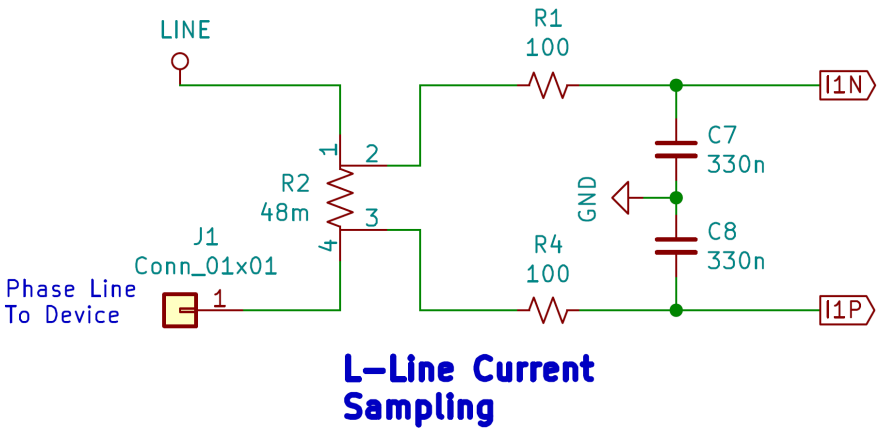


Figure 4.3: Current Sense Resistor Reference Design

A standard value was not supplied for the shunt, so one had to be determined for this application. The meter needed to be able to measure currents up to 15A, since that is the limit of most home outlets, as determined by the circuit breakers used for most outlets (McGregor 2020). I1N and I1P are the inputs to the ATM90E26, and at a programmable gain amplifier setting of 1, voltages of 120 μV_{rms} to 600 mV_{rms} can be accurately measured. Finally, the inputs have an input impedance of 1 kΩ, a value that can be found in the ATM90E26 datasheet. Meaning, that for this application, when there is 15A of current flowing through the shunt, a voltage of 600 mV_{rms} should be seen across the inputs I1N and I1P.

Analyzing this circuit, it is seen that R2, R4, and the input impedance R_{input} form a voltage divider. Capacitors C6 and C8 act as noise filters, and have a relatively high reactance of $X_C = \frac{1}{2\pi fC} = \frac{1}{2\pi(60)(330E-9)} = 8.04$ kΩ, meaning they have negligible effect on the voltage at the IC inputs and can be ignored.

The desired value for the shunt resistor is thus given by:

$$R_{shunt} = \frac{V_{in} * (R_2 + R_4 + R_{in})}{R_{in} * I_L} \tag{4.1}$$

$$= \frac{(0.600\ V_{rms})(1.2\ k\Omega)}{(1\ k\Omega)(15\ A)} \tag{4.2}$$

$$= 0.048\ \Omega \tag{4.3}$$

For the construction of the prototype, a through-hole component was preferred. The closest available value in a through-hole package at the time of purchase was the MP95-0.050-1% from Caddock Electronics. The shunt also needed to be able to withstand up to $P = I^2 * R = (15A)^2(0.050\ \Omega) = 11.25$ W. The MP95-0.050-1% is rated to handle up to 15W of power dissipation, so there are no power concerns here. Using this resistance value and solving

for the minimum measurable current for the smallest allowed voltage at the input of the IC, a minimum measurable current value of 3mA is obtained.

4.3.4 Selection of the Current Transformer

The current transformer is a special kind of transformer that is typically used to safely measure currents that would be too large to measure directly ("The Current Transformer" n.d.). A large current is passed through the center of the transformer (the primary), which generates a much smaller current on the output (the secondary) due to the transformer's very high turns ratio. In the ATM90E26 reference design, the current transformer is used to measure current on the neutral line. The current sense resistor section from the project schematic along with the calculated value is shown below in Figure 4.4.

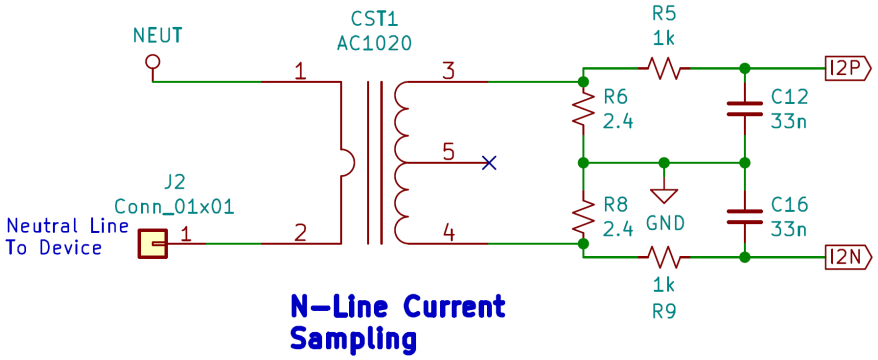


Figure 4.4: Current Transformer Reference Design

Just like in the analysis of the current sense resistor, the smart plug needs to be able to measure currents up to 15A, and the voltage range for the inputs to the IC are 120 μV_{rms} to 600 mV_{rms}. However, for the inputs I2N and I2P, the input impedance is much higher at 400 kΩ. To find the current that the current transformer should output that results in 600 mV_{rms} at the IC inputs for 15A at the input of the current transformer, we can use the node voltage method to create a system of equations from each node of the circuit. In this case, the reactance of the filtering capacitors C9 and C10 will need to be accounted for, since their reactance, at $X_C = \frac{1}{2\pi fC} = \frac{1}{2\pi(60)(330E-9)} = 80.4$ kΩ is relatively small compared to the input impedance of the IC. The following is the system of equations used to determine the current transformer ratio, as well as a diagram of the circuit and nodes used in the analysis as shown in Figure 4.5.

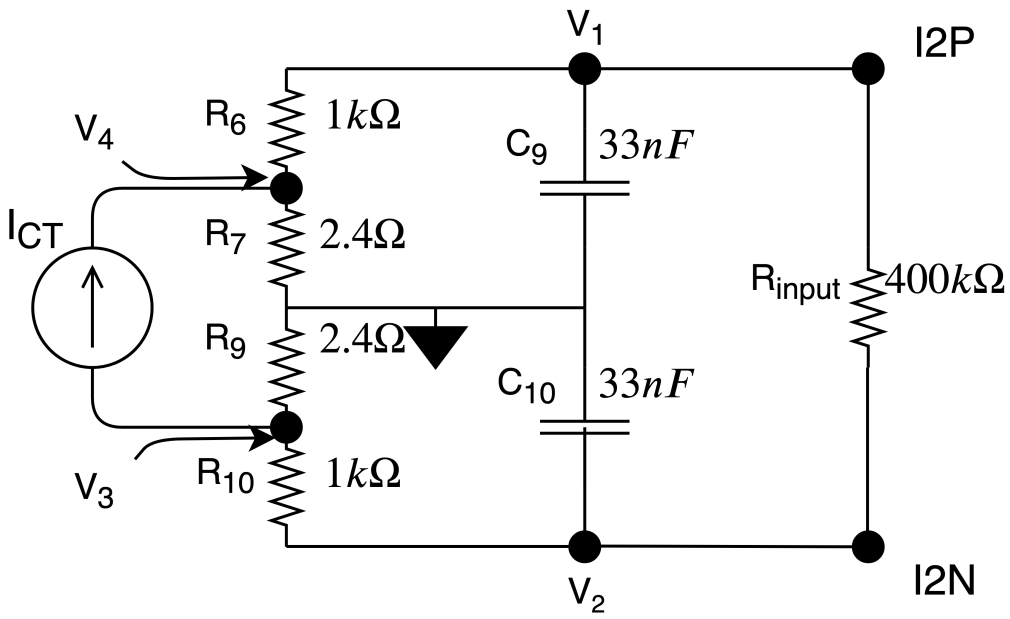


Figure 4.5: Current Transformer Circuit and Analysis Diagram

$$\text{Node 1}: \frac{V_1 - V_4}{R_6} + \frac{V_1}{X_{C9}} + \frac{V_1 - V_4}{R_{input}} = \frac{V_1 - V_4}{1\ \text{k}\Omega} + \frac{V_1}{80.4\ \text{k}\Omega} + \frac{V_1 - V_4}{400\ \text{k}\Omega} = 0 \tag{4.4}$$

$$\text{Node 2}: \frac{V_2 - V_3}{R_{10}} + \frac{V_2}{X_{C10}} + \frac{V_2 - V_1}{R_{input}} = \frac{V_2 - V_3}{1\ \text{k}\Omega} + \frac{V_2}{80.4\ \text{k}\Omega} + \frac{V_2 - V_1}{400\ \text{k}\Omega} = 0 \tag{4.5}$$

$$\text{Node 3}: \frac{V_3}{R_9} + \frac{V_3 - V_2}{R_{10}} + I_{CT} = \frac{V_3}{2.4\ \Omega} + \frac{V_3 - V_2}{1\ \text{k}\Omega} + I_{CT} = 0 \tag{4.6}$$

$$\text{Node 4}: \frac{V_4}{R_7} + \frac{V_4 - V_1}{R_6} - I_{CT} = \frac{V_4}{2.4\ \Omega} + \frac{V_4 - V_1}{1\ \text{k}\Omega} - I_{CT} = 0 \tag{4.7}$$

$$\text{Input Voltage}: V_1 - V_2 = 600\ \text{mV}_{\text{rms}} \tag{4.8}$$

Solving the system of equations, the current transformer needs to supply 127mA of current for a 15A input. This gives a final turns ratio of $n = \frac{I_S}{I_P} = \frac{0.127A}{15A} = 0.0085 \approx 1:120$, where I_P is the current on the primary winding of the current transformer and I_S is the current on the secondary. 1:120 current transformers do exist on the market, but unfortunately their cost is prohibitive at approximately $45. However, 1:1000 current transformers are much more cost effective at approximately $10 or less a piece at small quantities. The ratio of these current transformers can be adjusted by wrapping the primary around the body of the transformer multiple times ("The Current Transformer" n.d.). The number of times needed to wrap around the primary is $\frac{1000}{120} = 8.33 \approx 8$, which results in a final turn ratio of 1:125. Replacing the IC input voltage in equation 4.8 with 120 μV_{rms} will give the smallest current generated by the current transformer, which works out to be $25.4\mu A$. This results in a current input of 3.2mA on the primary, meaning both the phase line and neutral line current sampling circuits will be able to measure up to 15A and as low as $\approx$ 3mA, making neither one a limiting factor.

4.3.5 Design of the Power Supply

In the ATM90E26 reference design is a power supply section for supplying the necessary power for the ATM90E26's operation. This consists of a power transformer to drop the voltage from 120 V_{rms} to a lower voltage for the linear regulator, followed by a 1N4007 rectifier diode, forming a half-wave rectifier. This becomes a near-DC voltage by means of the filter capacitor, with a ripple voltage that has been superimposed by the half-rectified AC from the rectifying diode. The power supply section from the project schematic along with calculated values is shown below in Figure 4.6.

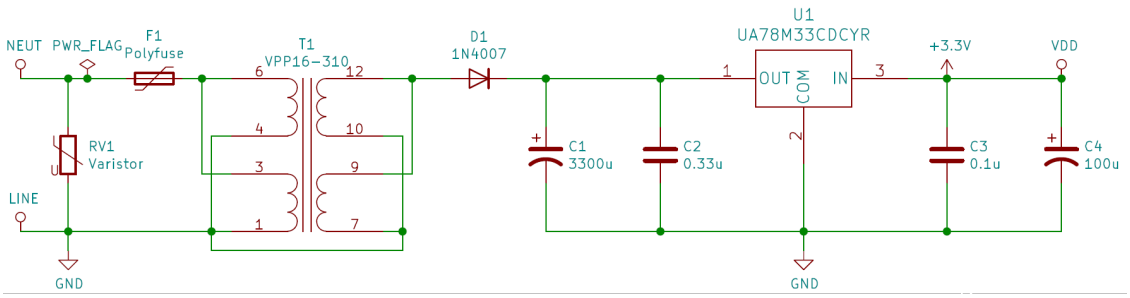


Figure 4.6: Power Supply Reference Design

The design of the power supply for this project will differ in a few fundamental ways; firstly, the transformer in the reference design uses a center tap on the primary for supplying power to the rectifier and linear regulator. Power for the MCU is then derived from one of the two secondary windings. The other secondary winding is for the RS485 serial communication standard. Since these power monitoring IC's are capable of being used in residential power meters, RS485 is likely used in this application for reporting customer power usage back to the power companies that are supplying these homes with electricity. Ultimately, this feature is not necessary for this project.

Looking at the datasheets for the ATM90E26 and Particle Photon MCU, both devices can operate on 3.3V DC. This is rather useful, because it means a single linear voltage regulator can be used to power both pieces of hardware. Since RS485 is not being used, and since the MCU will be powered by the same linear voltage regulator as the ATM90E26, the transformer in the reference design can be replaced with a standard power transformer. The incoming 120VAC will be applied to the primary, and the voltage needed for the linear regulator will be supplied by the secondary. This results in a simpler circuit, as well as a much wider and cost-effective range of choices for component selection.

4.3.6 Determining Power Supply Requirements

The design of the power supply had a plethora of factors to consider, especially when it came to choosing a value for the filter capacitor. Since the output voltage of 3.3V was already determined, other factors to consider included typical current draw, maximum current draw, and allowed output voltage ripple. To determine the current draw parameters, the ATM90E26 and Photon datasheets were consulted. The ATM90E26's operating conditions state that the analog section consumes 3.4mA, while the digital section consumes 2.4mA for a combined 5.8mA. The Photon's datasheet states that typical operating current with Wi-Fi on is 80mA, and 100mA max. However, it also states that the Photon can draw a peak current of 430mA, which represents current bursts when, "transmitting and receiving in 802.11 b/g/n modes at different power levels". Further explanation states that these conditions usually occur (albeit briefly) when connecting to a Wi-Fi network. Although these current bursts may be brief, they still needed to be accounted for. For the design and for margin of error, a maximum current draw of 500mA was chosen to account for the Photon, the ATM90E26, and the current draw in the linear regulator.

In determining allowed output voltage ripple, both the ATM90E26 and Photon accept 3.3 ± 0.3V, which gives a maximum ripple voltage of 0.3V. For a safe margin of error, a maximum allowed ripple of 0.1V was chosen.

4.3.7 Selecting Power Supply Components

Starting with the selection of the power transformer, it needed to be able to supply at least 500mA (or more), and needed to have a high enough output voltage to ensure that the capacitor would be adequately charged to supply power to the system for the time during the cycle that the diode stops conducting, but not so high that the linear regulator has to scrub a large amount of voltage. An overly high voltage output from the transformer could cause a significant amount of power to be lost as heat (not to mention potentially damaging or shortening the lifespan of the linear regulator).

To help determine what power transformer to select, the linear regulator was selected first, and then a transformer was selected based on the recommended operating conditions of the linear regulator. For the prototype, the Texas Instruments UA78M33CKCS was selected due to its surface mount and through-hole part availability, low price ($0.79 at a price break of 1 at the time of writing), thermal protection, and ability to output 500mA. From its datasheet, its minimum recommended input voltage is 5.3V, and its electrical characteristics for output voltage were tested with an input voltage range of 8-20V. From this, a transformer with an 8V minimum output would be ideal, as this not only matches Texas Instrument's test conditions, but will also give a margin of safety above the 5.3V minimum voltage input so that the filter capacitor's voltage can drop somewhat when the Photon is drawing peak current.

To this end, the VPP16-310 power transformer from Triad Magnetics was chosen. The

transformer is comprised of two separate windings on both the primary and secondary sides. When the input and output windings are wired in parallel, the transformer can safely supply up to 620mA at 8.0 V_{rms} (11.3V pk) when supplied an input voltage of 115 V_{rms} at 50/60Hz. This exceeds the 500mA peak current requirement, and it exceeds the minimum voltage requirement, especially since the output voltage will be higher when not under full load (up to 25% higher when under no load according to the transformer's datasheet). At $8.00 for one, the transformer is fairly priced compared to some of the alternatives and is also reasonably compact. With the linear regulator and power transformer chosen, the final component of the power supply left to be determined was the filter capacitor.

4.3.8 Determining the Filter Capacitor Value

To find the value of the capacitor, pages 214-217 of *Microelectronics Circuits* by Sedra and Smith were consulted. Shown below in Figure 4.8 is the voltage and current waveforms in a peak rectifier circuit, assuming an ideal diode. From this graph, there are several factors and parameters to consider, namely the peak voltage V_p, the ripple voltage V_r, and the period T. On the lower half of the graph is the current in the diode as well as the load current. It should be noted that during the short conduction interval δt, the diode must supply enough current to charge the capacitor for the entirety of the remaining cycle when it will no longer be forward biased. This also needed to be considered and checked against the specs of the diode to ensure that it could handle the current peaks.

For the purposes of calculation, the power supply circuit was be approximated using the circuit shown below in Figure 4.7, where the resistor R is the load.

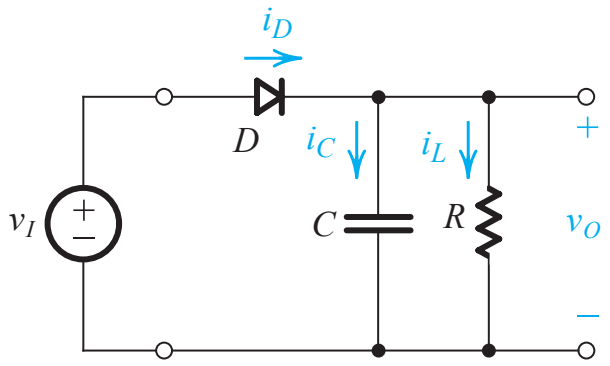


Figure 4.7: Simplified Half-Rectifier Circuit with Filter Capacitor and Resistive Load

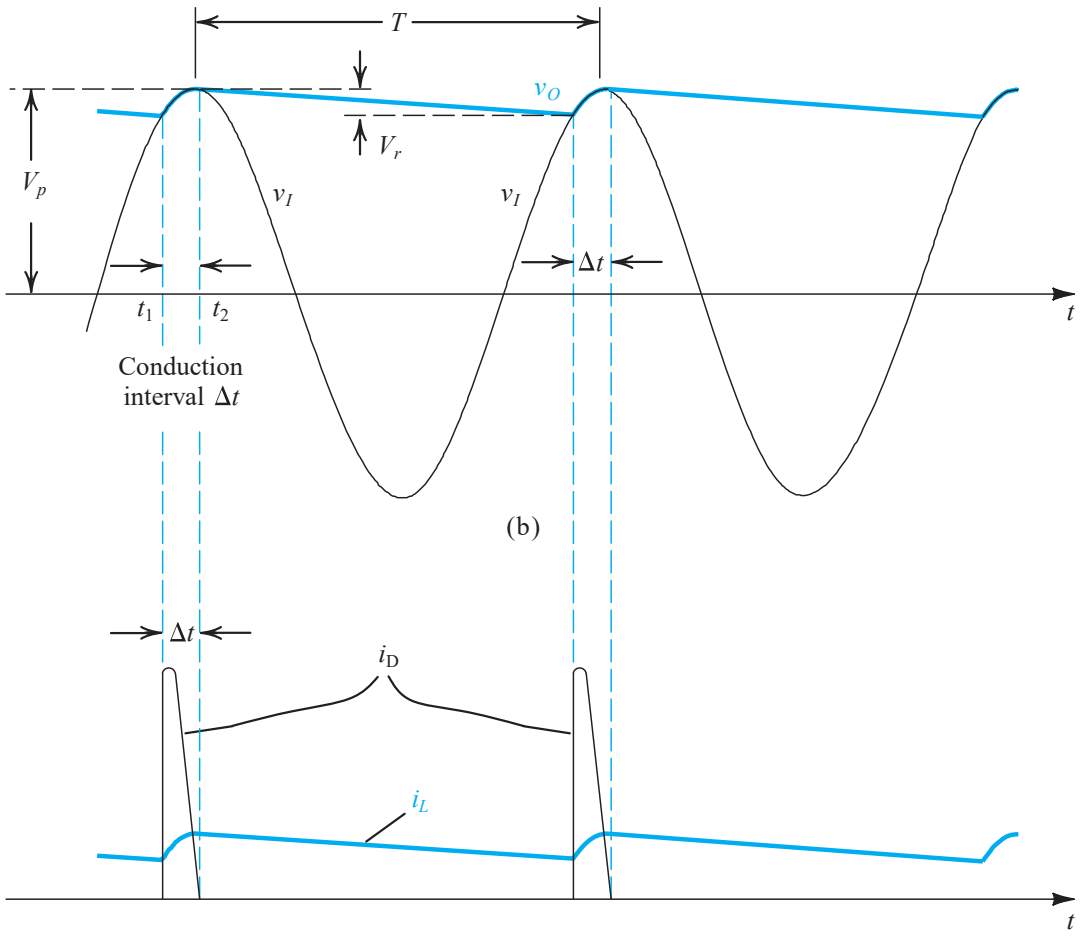


Figure 4.8: Voltage and Current Waveforms in the Peak Rectifier Circuit

Beginning with determining V_p under load conditions, the voltage from the power transformer V_I had to be found first. The VPP16-310 states that in parallel configuration, there will be a potential of 8.0V across the secondary windings at a full load of 620mA. In addition, the datasheet specifies a voltage input of 115VAC, so to find the transformer's output voltage at 120VAC, the turns ratio needed to be approximated. Strangely, this value was not supplied in the datasheet. Since there is a 25% drop in voltage from no load to full load,

$$TurnsRatio = \frac{N_{secondary}}{N_{primary}} = \frac{V_{secondary}}{V_{primary}} = \frac{8 * 125\%}{115} = 0.093 = 1 : 10.8 \tag{4.9}$$

So, for a voltage input of 120VAC, there should be an output of $0.093 = \frac{V_{secondary}}{120VAC} = 11.1VAC$. After a 25% drop for full load conditions, a voltage of output of $11.8VAC * 75\% = 8.34VAC$ can be expected. Converting from RMS to peak-to-peak gives $8.34 * \sqrt{2} = 11.8V = V_I$. Estimating a 1V drop across the diode gives a final peak voltage at the capacitor of $V_p = V_I - V_D = 11.8 - 1 \approx 10.5V$ at the worst.

For the ripple voltage, V_r, a voltage output of no less than 6V at the linear regulator was

desired, since the recommended minimum input voltage was 5.3V, which means that the ripple voltage was $V_r = V_p - V_{o,min} = 10.5 - 6 = 4.5V$. Before a final value for the filter capacitor could be determined, a value for the load resistance needed to be estimated, since the load was not a purely linear, resistive load. Because a lower resistance value would require a larger filter capacitor (due to more current flow in the resistor, thus discharging the capacitor more quickly), the lowest possible equivalent load resistance was used in calculations to determine filter capacitor size. The equivalent load resistance reaches its lowest value when the voltage drops to the lowest desired point in the cycle (6V) under max load conditions (500mA). By Ohm's Law, a good approximate resistance is $R = \frac{V}{I} = \frac{6V}{0.5A} = 12\ \Omega$.

Using Eq. 4.29a from *Microelectronics Circuits*, a value for the filter capacitor can now be determined:

$$C = \frac{V_p}{V_r f R} = \frac{10.5V}{(4.5V)(60Hz)(12\ \Omega)} = 3.24mF \approx 3300\mu F$$

While this is a rather large capacitor, a 3300 µF capacitor rated for 25V could be purchased from Nichicon at $1.01 for a quantity of 1, with the only major cost being the board space taken up (the Nichicon capacitor measures 27mm tall and 16mm in diameter). It may be possible to utilize a smaller capacitor, but it would likely require testing of smaller values, at the risk of the Photon or power monitoring IC powering down during maximum load conditions.

4.3.9 Other Power Supply Considerations

The two factors left to check in the power supply design were the ripple on the output of the linear regulator, and the peak current through the diode. For the linear regulator ripple voltage output, the datasheet for the UA78M33C specifies that under test conditions, the PSRR (Power Supply Ripple Rejection) measures at 62dB. From a Texas Instruments article on PSRR in Low-Dropout (LDO) regulators, PSRR is, "A measure of a circuit's power supply's rejection expressed as a log ratio of output noise to input noise" (Pithadia, Lester, and Verma 2009). From that same article, the basic equation for PSRR is given, which was used to find the ratio of input ripple that gets passed to the output.

$$\begin{aligned} PSRR &= 20\log\frac{\text{Ripple}_{\text{Input}}}{\text{Ripple}_{\text{Output}}} \\ \implies \quad 62dB &= 20\log\frac{\text{Ripple}_{\text{Input}}}{\text{Ripple}_{\text{Output}}} \\ \implies \quad 10^{62/20} &= \frac{\text{Ripple}_{\text{Input}}}{\text{Ripple}_{\text{Output}}} \\ \implies \quad \text{Ripple}_{\text{Output}} &= \frac{\text{Ripple}_{\text{Input}}}{1258.9} \end{aligned} \tag{4.10}$$

From Eq.(4.10), it can be seen that any ripple on the input of the linear regulator will be reduced by a significant amount, easily accomplishing the goal of 0.1V or less of ripple on the 3.3V output of the linear regulator. In addition, the 100 µF capacitor on the output of the linear regulator should smooth any remaining ripple.

A value for the peak current through the rectifying diode is found using Eq.(4.32) in *Microelectronics Circuits*, which determines the peak current by evaluating Eq.(4.25) at the beginning of the moment that the diode conducts. This results in a peak current of:

$$\begin{aligned} i_{Dmax} &= I_L(1+2\pi\sqrt{2V_p/V_r}) \\ &= (0.500A)(1+2\pi\sqrt{2(10.5V)/(4.5V)}) \\ &= 7.29A \end{aligned}$$

While this is a rather large current, this kind of current exists under brief, peak operating conditions when the Photon is connecting to a Wi-Fi network. A 1N4007 general purpose rectifier from Vishay is rated for 1.0A of average rectified current and up to 30A of peak surge current, so the 1N4007 should be adequate for this design. However, for additional safety margin as well as for reliability, a sturdier diode that could handle higher average and peak currents should be considered for a future revision. A diode such as the 1N5401 with 3A maximum average rectified current and 200A of peak surge current would meet the requirements of the project adequately.

5 Firmware Design Development

The firmware for this project acts as the controlling layer on top of the selected microcontroller. The firmware has been abstracted out to three main components as shown in Figure 5.1. The packet handler and packet builder are two components that make up an overall packet factory, and the third component is for the power data handling. The microcontroller is what will ultimately handle the physical side of the direct SPI connection to the power monitoring IC as well as the raw Wi-Fi connection to the TCP server, while the firmware will be handling all the data sent and received through these interfaces.

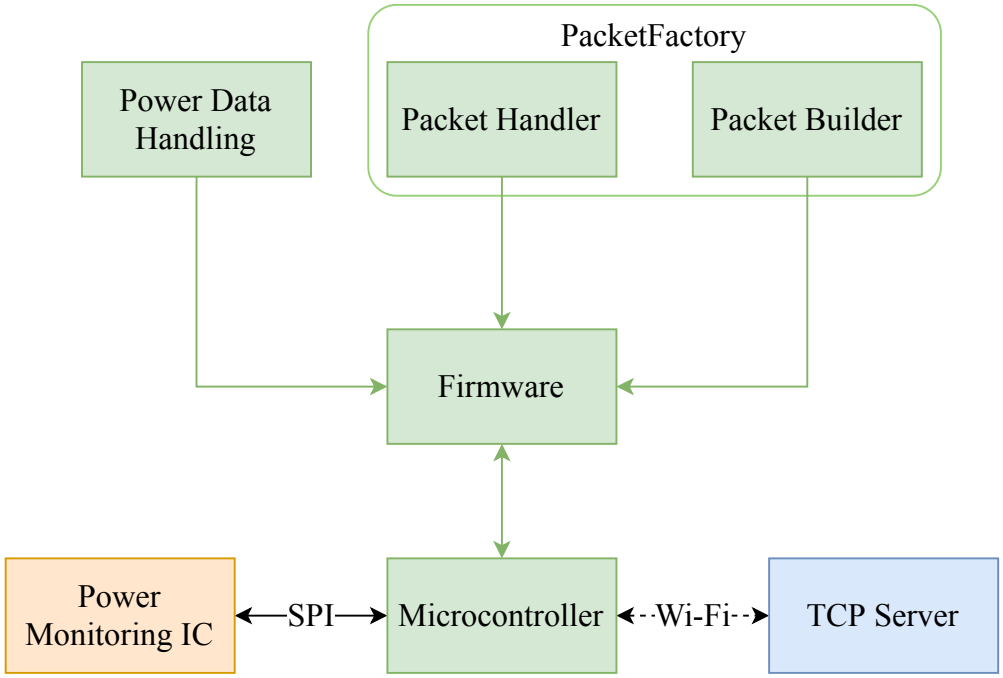


Figure 5.1: Overview of Firmware Design

5.1 Microcontroller Selection

In researching microcontrollers for the project, the selection was narrowed down to two individual microcontrollers before making a final decision. The choice was between the Particle Photon, and the ESP8266-12E shown in Figure 5.2 and Figure 5.3 respectively. The hardware features were essentially the same, so the ultimate decision came down to software features.

The ESP was cheaper than the Photon, which is always a bonus, and it had a very active development community. This active community meant that finding example code and low-level drivers online should be trivial. However, the overall development on this board would be more manual because of the lack of an operating system on the board. The documentation on this board was also a little bit tougher to track down and not as convenient as it could have been.

The Particle Photon was the more expensive option, but it had a lot more software features. The Photon came with an operating system by default called “Device OS” that had a substantial number of API calls that allowed for easy interfacing between the firmware and hardware. The documentation for the Photon and Device OS was very detailed and in-depth which made it invaluable for development.

Ultimately, the Particle Photon was the most compelling option and provided the best chance at making a working prototype in the time allotted. All progress beyond this point utilized the Photon as the microcontroller.

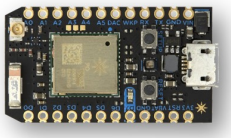


Figure 5.2: Particle Photon MCU

Figure 5.3: ESP8266-12E MCU

5.2 Firmware Tools

By default, the Particle Photon can be developed on with an online IDE that flashes user code to the device over Wi-Fi. This is a great feature for relatively simple or small projects, but for a project of this scope it is more convenient to have more control over the development environment. Thankfully, Particle offers both Particle Workbench, and Particle Dev as options for an IDE. The first exists as plugins on top of VS Code, while the latter exists as plugins on top of Atom. Particle Workbench is the most fully featured of the two and offers easy options for local compilation of code and local flashing of the code over USB to the Photon. However, these features do not always work and seem to take a bit longer than the equivalent options in Particle Dev. Particle Dev does not offer an easy solution to local compilation or local flash, but compiling in the cloud actually seems to be faster than the local compilation in Particle Workbench. Local flash can be easily accomplished with a small Python script which in practice also seems to be faster than the local flash in Particle Workbench. Regardless of these minute details, there is a plethora of development tools available.

The firmware runs on C++, which is a great language for embedded development. C++ provides a lot of power over the low-level aspects of the firmware as well as the ability to abstract more complex firmware structures with classes and object-oriented features.

5.3 Firmware Structure

The firmware has a simple overall structure. When the firmware is booted on the Photon, the setup function is automatically called. This function handles a few basic configuration settings for Device OS and sets up the various required network connections. More specifically, this function is where the Photon initially connects to a WiFi network, opens a serial connection, opens a SPI connection, and connects to the TCP server. After this function has completed, the main loop function is automatically called. The main loop handles all the run-time operations of the Photon. The first thing the main loop does is check if it has a TCP connection. If no connection is found, then it will attempt to renegotiate the TCP connection in every loop of the function until a

connection is made. If a connection is found, then the loop checks if any bytes have been received over the TCP connection. If bytes have been received, then the packet handler is called, and the read buffer is cleared. Regardless of whether a TCP connection exists, a data sample will be taken and stored in the EEPROM every three minutes that the Photon is running. A visual representation of these functions and their responsibilities is found in Figure 5.4.

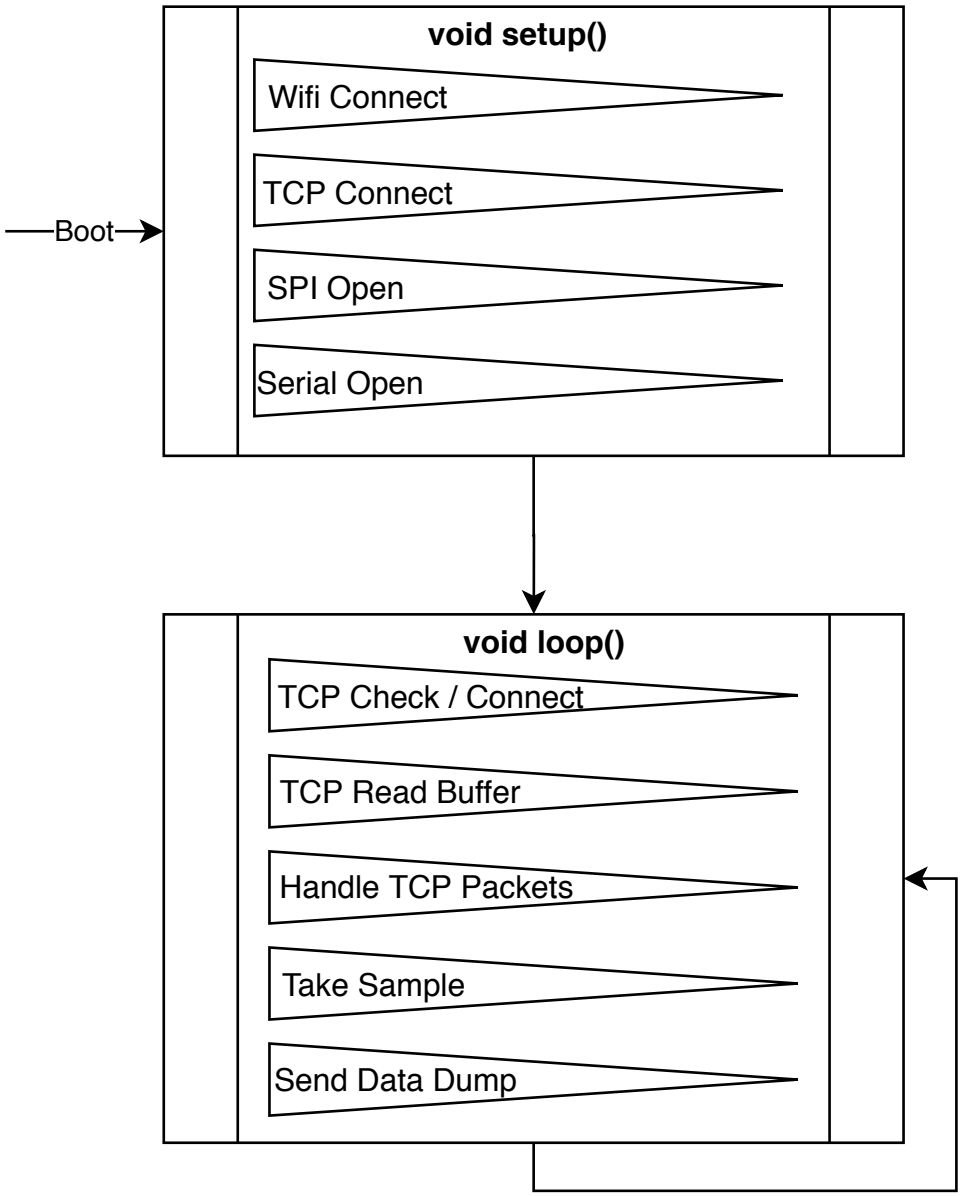


Figure 5.4: Diagram of Firmware Structure

5.4 Packet Format

Table 5.1 provides a basic structure for packets sent and received between the Photon and the TCP server. Each packet will be sent with the first byte indicating the ID number of the packet. The ID is not unique to each device but is rather unique on the network so that each end knows if it is talking to a valid device or not. The second byte indicates the version of the packet. This version field is particularly useful if multiple versions of this packet specification are in use between devices. If the version number of the packet received is not the expected number, then the packet will be thrown out. This prevents either end of the communication from failing to parse a packet of the wrong version. The third byte indicates the subtype of the packet. The subtype is used to determine how the packet will need to be handled and what packet(s) could need to be sent in response. The last field of each packet is set to be the data portion. These data portions are to be set at specific byte offsets, so once the subtype of the packet is known and the packet has been verified then the data portions can be parsed appropriately.

Size (Bytes)	Sender	ID (1)	Version (1)	Subtype (1)	Data (Bytes)
Open Connection (9)	Photon	0xBA	0	0	6
Open Response (7)	Server	0xBA	0	1	4
Status (3)	Server	0xBA	0	2	0
Status Response (11)	Photon	0xBA	0	3	8
Data Dump (129)	Photon	0xBA	0	4	126

Table 5.1: General Packet Format

Table 5.2 provides a basic structure for the data fields of each packet type. In the Open Connection packet, the MAC address is sent as an array of 6 bytes, one byte for each octet of the address. The MAC address is used by the web client to identify the device for future communication data handling and processing. In the Open Response packet, the time is sent as a UNIX timestamp in an unsigned 32-bit integer format. This time value is used by the Photon to set its current UNIX time to be in sync with the one provided. This synchronization of time ensures that data samples tagged with a time value from the Photon will be at the local time expected by the web client. The Status packet is used exclusively by the server to get a Status Response packet from the Photon, so there is no extra data needed. The Status Response packet sends the values of four of the power measuring registers to the server. Pmean is a signed 16-bit integer that indicates the mean active power being measured, Urms is an unsigned 16-bit integer that represents the current RMS voltage load being measured, PowerF is a signed 16-bit integer that indicates the current power factor being calculated, and the frequency is an unsigned 16-bit integer indicating the frequency of the power being measured. The Data Dump packet contains 21 data chunks: each chunk contains a signed 16-bit integer holding the value of Pmean as well as an unsigned 32-bit integer holding the UNIX timestamp when the sample was taken.

Size (Bytes)	Data details (Size)			
Open Connection (9)	MAC (6)			
Open Response (7)	Time (4)			
Status (3)				
Status Response (11)	Pmean (2)	Urms (2)	PowerF (2)	Freq (2)
Data Dump (129)	Pmean (2)	Time (4)	* 21	

Table 5.2: Packet Data Format

5.5 TCP Communication

As seen in Figure 5.1, the communication between the microcontroller and the computer software is accomplished using a TCP server. The communications over this TCP server will be handled by the packet factory in the firmware. Because the plan is to support multiple devices on the same TCP server, each device that connects to the TCP server will broadcast its MAC address to act as an identifier for the software to distinguish between devices and their locations.

The packet factory consists of two main firmware processes: packet handling, and packet building. These processes work together to handle all incoming packets and the creation of any response packets over the TCP wireless connection. The system can distinguish between both types of packets that are sent to the Photon via TCP and handle each accordingly. The server side is also capable of parsing out the messages received from the Photon and displaying relevant data on the web client. More information on the web client can be found in Section 6.1.2

The TCP connection is initially tried during the setup function of the firmware that runs before the main loop. If this connection fails, the Photon will still record data samples, but will be continuously trying to renegotiate a connection to the TCP server. This design ensures that if a TCP connection is lost or not available at any point during the firmware run-time, the firmware will still function as intended and will be able to continue normal operation once a TCP connection is made.

5.5.1 Packet Handler

The overall design for packet handling is straightforward. As mentioned in Section 5.4, the first thing the handler needs to do is verify the integrity of the received packet by parsing the packet ID and version. Once the packet has been validated, the handler can parse out the packet subtype. With that information, it can decide how to deal with the packet; parse out other data fields, and/or send an appropriate response packet. Part of the packet handling function that performs the functions previously outlined can be found in Listing 5.1.

```
void packetHandler(char* buf) {
    // Get packet header details
```

```
uint8_t pID = buf[0];
uint8_t pVersion = buf[1];
subtype_t pSubtype = (subtype_t)buf[2];

// Verify that packet is correct version and ID
if (pID != PACKET_ID || pVersion != CURRENT_VERSION) {
    #ifdef SERIAL_DEBUG
        Serial.println("Error: Incorrect packet ID or version!");
    #endif

    return;
}
// Packet is valid, process based on subtype
else {
    switch (pSubtype) {
```

Listing 5.1: Beginning of Packet Handler

Listing 5.1 outlines how the fully implemented packet handler works. As detailed previously, the initial step is to parse out the ID, version, and subtype which get stored in pID, pVersion, and pSubtype, respectively. Then a simple if-statement verifies that the packet is of correct ID and version against a pair of global defines (for ease of updating). If the packet is validated, then a switch-case statement handles the packet based off the already parsed packet subtype.

Packets are stored in the firmware as C struct objects. These objects have fields for the raw contents of the packet detailed in Table 5.1 and Table 5.2. A UML class diagram that outlines these structures and their member variables can be found in Figure 5.5. Each struct makes use of the "packed" attribute to ensure no memory padding occurs that could result in an incorrect packet format when sent. In the final design, the packet handler follows the outline above and correctly handles both types of incoming packets from the TCP server.

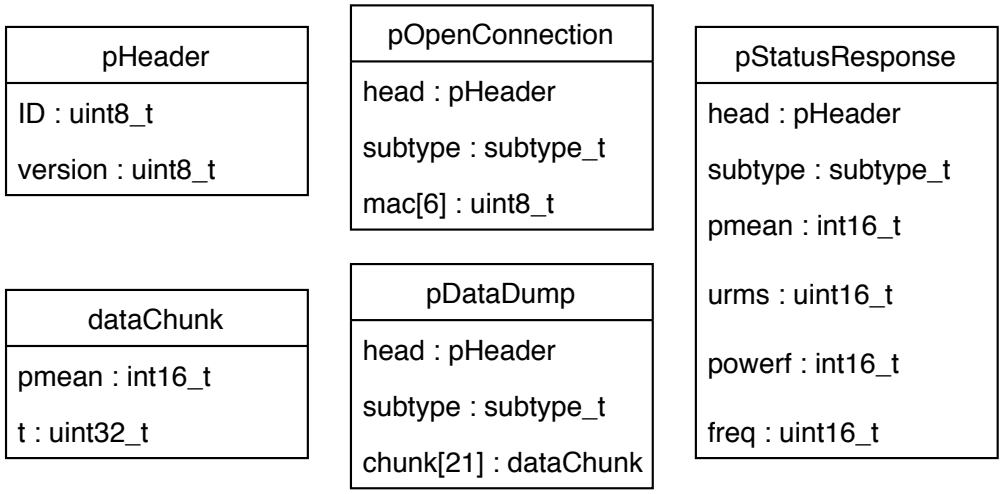


Figure 5.5: UML Diagram of Packet Structures in Firmware

5.5.2 Packet Builder

The design for packet building is straightforward as well. Packets are outlined as struct objects with individual data attributes for each field of the packet as shown in Listing 5.2. Once the struct object is instantiated for a given packet, it is trivial to load the correct data into the corresponding data fields of the object to prepare for transmission. Because each of these struct definitions include the ”packed” attribute, a pointer to the struct object can be passed to the TCP write function along with the size of the struct to send the packets without any memory padding interfering with the expected byte offsets for data field locations. This means that once a struct object for a packet has been instantiated and filled with the correct data, the contents of the packet object can be passed to the TCP API to be sent to the server with ease.

```
struct __attribute__((__packed__)) pStatusResponse {
    struct pHeader head;
    subtype_t subtype = STATUS_RESPONSE;
    int16_t pmean;
    uint16_t urms;
    int16_t powerf;
    uint16_t freq;
};
```

Listing 5.2: Struct for Status Response Packet

This design replaces the initial design of a C++ packet class that would handle all packets. This struct approach is much better for the scope of this project because there are so few packets that need to be sent from the Photon to the TCP server. The C++ class approach would probably have been best for a larger project with more variability in the communication standard. The struct approach is much more user-readable and less complex than using a class. Packet building also happens more on-the-fly then in a function because of how simple it is to do so with struct objects. This also is an alteration of the initial design which elected to have this as a specific function call every time. In the final design, when a packet needs to be built and sent it is simply instantiated, filled, and sent in just a few lines of code.

5.5.3 Communication Flow

A typical communication sequence between the Photon and TCP server can be found in Figure 5.6 and is described as follows. The Photon connects to the TCP server and sends an Open Connection packet with its MAC address enclosed. The server sees this packet, associates the MAC address of the device with the IP address used to connect to the server, and responds with an Open Response packet with the current UNIX timestamp adjusted for the current time zone. Then, while the home page of the web client is open, the server regularly sends Status packets to the Photon. The Photon responds to these Status packets with Status Response packets filled with data retrieved via SPI from the hardware containing mean power, RMS voltage, the power factor, and frequency currently being measured. While the Photon is running, it collects data samples once every three minutes and stores the mean power value retrieved via SPI from the hardware

along with a UNIX timestamp in the emulated EEPROM provided with Device OS. After 20 of these samples have been retrieved (once per hour), the Photon will send a Data Dump packet containing all 20 samples retrieved from the EEPROM to the server. The sent data is parsed by the server and stored in the database. This information can be viewed from the history page of the web client.

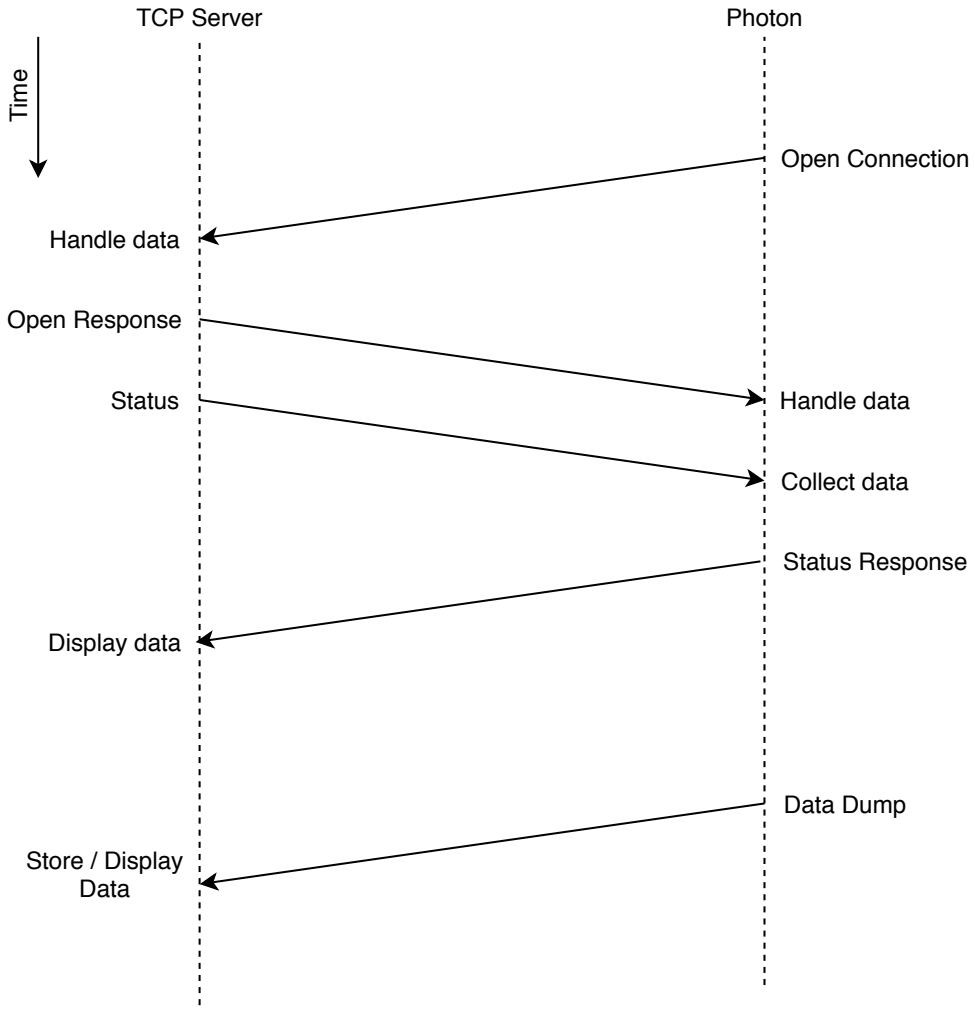


Figure 5.6: Example Communication Sequence

5.6 Power Data Handling

The final component of the firmware is dedicated to handling all the power data received over SPI from the power monitoring IC. This component handles all the receiving, storing, sending, and clearing of power monitoring data as the system runs in real time. The data from the power monitoring IC is sampled at a rate of one sample every three minutes and each Data Dump packet is set to contain 20 valid samples which means data dumps occur hourly. Each sample is stored in the Photon's emulated EEPROM with simple Device OS API calls. Once 20 samples have been

stored in the EEPROM, the memory is scanned through and each sample is stored in a Data Dump packet and sent out. The EEPROM is then cleared to ensure the integrity of future sample collection. This design is shown in Figure 5.7. A feature that has not been implemented is that when requested, the microcontroller would be able to send all the currently collected data over TCP to the server. This feature would be ideal when viewing the power graph on the web client to ensure the most up-to-date information was available to the user. However, there currently is not enough time to develop this feature.

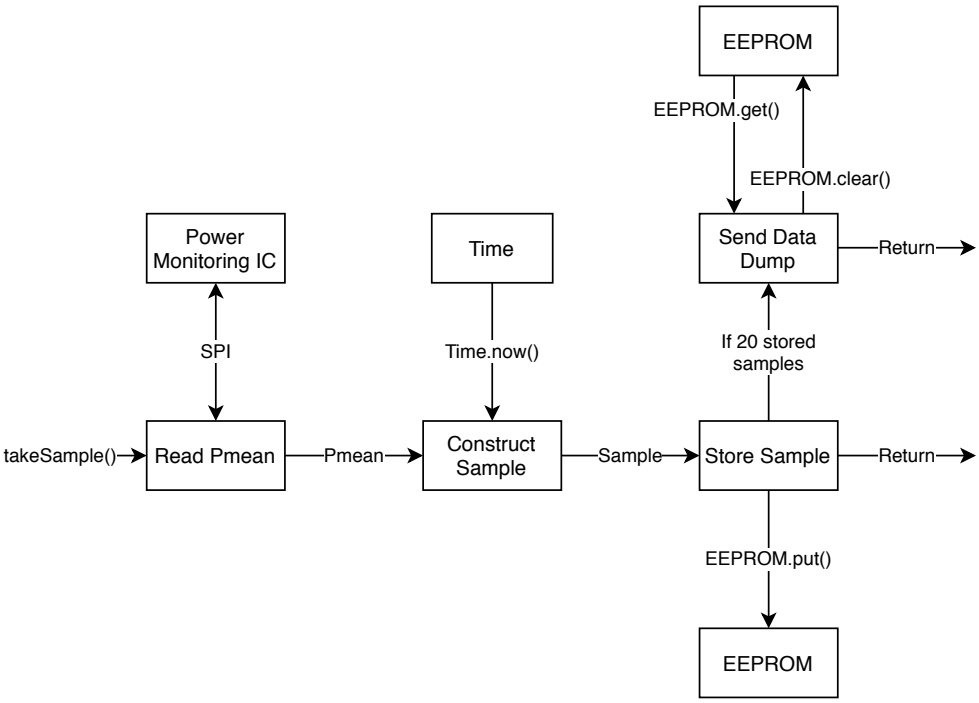


Figure 5.7: Data Flow of Power Data Handling

5.6.1 Data Receiving

As previously mentioned, the power data from the power monitoring IC is sent to the microcontroller through a SPI connection. The power monitoring IC stores all the power monitoring data in an expansive series of registers that can be directly accessed by the microcontroller via the SPI connection. The registers of the power monitoring IC are sampled on demand, and one register is sampled once every three minutes. This data is written to flash memory with a timestamp to be sent out with a Data Dump packet every hour.

Listing 5.3 shows a sample SPI read from the Pmean register of the power monitoring IC. The main component shown here is the SPI1.transfer() function call. This function takes in transmit (tx) and receive (rx) buffers as well as a size (both buffers must be this size) and callback function. The power monitoring IC has 8-bit addresses and 16-bit measuring registers. Because of this, the size used is two bytes for both the transmit and receive buffers. The eight lowest bits

of the transmit buffer are set to be the address of the measurement register that will be read from (in this case 0x4A). The callback function is not utilized in this design as it allows asynchronous completion of the SPI transmission which is not necessary for this system. Ultimately this function returns the contents of the specified register in the receive buffer which is then converted to the correct 16-bit data type for either immediate sending to the server or immediate storing in the EEPROM.

```
// SPI transmit/receive buffers
char rx[2] = {0,0};
char tx[2] = {0,0};
int16_t sStore;
uint16_t uStore;
// Read registers and pack
tx[1] = 74; // Pmean register at 4AH
SPI1.transfer(tx, rx, sizeof(rx), NULL);
sStore = (rx[0] << 8) | rx[1];
```

Listing 5.3: Sample SPI Read from Pmean Register

5.6.2 Data Storing

The power monitoring data received through SPI is stored in the Photon's flash memory. Through Device OS, the Photon offers an emulated EEPROM that can be dedicated to keeping the data memory organized and accessible when required. The firmware is setup to store 20 samples in the EEPROM before sending them to the server. The EEPROM is 2 kB in size and can store 340 total samples before running out of storage, so 20 samples are nowhere near this limit. Because we only sample at a rate of one sample every three minutes, the storing of data is not particularly time dependent and thus the default Device OS API calls work well.

Listing 5.4 shows the section of code that stores a sample in the EEPROM. The main component shown here is the EEPROM.put() function call. This function takes in an integer that represents the address into the EEPROM memory as well as the data object that is to be stored at the provided address. Once the memory is stored, the variable that stores the address of the EEPROM is incremented by the size of the data to ensure that none of the data is overwritten. The data samples are stored in a dataChunk structure as shown in the second line of Listing 5.4. The dataChunk structure has fields for the Pmean register (which is retrieved via the SPI connection) and the time which is obtained with a Device OS call to Time.now() that returns the current UNIX timestamp.

```
// Get dataChunk to store sample
struct dataChunk data;
data.pmean = endianswap_16_signed(sStore);
data.t = endianswap_32_unsigned(Time.now());

// Store dataChunk in the EEPROM at the specified address
EEPROM.put(eeAddress, data);
// Increment EEPROM address to prevent data overwriting
```

```
eeAddress += sizeof(struct dataChunk);
```

Listing 5.4: EEPROM Data Storing

5.6.3 Data Sending and Clearing

The stored power monitoring data is sent out once 20 samples have been stored in the EEPROM. This is done automatically and is automatically handled by the TCP server once it is sent. Once the samples have been sent to the server, the EEPROM needs to be cleared to make sure old samples are not sent accidentally.

Listing 5.5 shows what happens once 20 samples have been stored in the EEPROM. Initially the static variables for the EEPROM address and count of samples need to be reset. Then the EEPROM is iterated through and each collected sample is stored in a Data Dump packet. After all the samples are collected, an empty sample is appended onto the end to act as an endpoint for the server-side parsing. Finally, the Data Dump pack is sent and the EEPROM is wiped to ensure the validity of future data collection.

```
// Reset statics
eeAddress = 0;
sampleCount = 0;

// Get dataDump for storing samples from EEPROM
struct pDataDump toSend;
for (int j = 0; j < 20; j++) {
    // Get all the samples and store in packet
    EEPROM.get(eeAddress, toSend.chunk[j]);
    eeAddress += sizeof(struct dataChunk);
}
// Append an empty sample for software parsing
struct dataChunk empty;
toSend.chunk[21] = empty;
// Send filled dataDump packet
sendPacket(&toSend, SIZE_DATA_DUMP);
// Clear EEPROM and reset the address one more time for future collection
EEPROM.clear();
eeAddress = 0;
```

Listing 5.5: EEPROM Data Sending and Clearing

5.7 First Time Setup

Setting up this device for the first time has proven to be one of the more difficult tasks for the team to overcome. This is because the power monitoring device itself can only communicate to a computer over a wireless connection, which is not possible without first setting up said wireless

connection. Because of this restriction, that the best way to set up the device for the first time would be over a USB connection to a computer. The user would then need to download some extra software onto their computer to communicate with the device. This software would act as a nice user interface layer over the top of some Device OS API calls that would be used to get the device connected to a network and the TCP server for future communications and configuration. As it stands, this feature will not be implemented for this project. We have decided that in our efforts to make a working prototype, first time setup should not be a priority and thus is outside the scope of this project.

6 Software Design Development

The software portion of this project has three main components: the front-end, the back-end, and the database, as seen in Figure 6.1. This also shows the general flow of information between each component, as well as the communication methods. Each component runs independently of the others. The front-end is the user interface, the back-end is responsible for communication with the power devices and database, and the database is where usage information from the power devices will be stored. This project is developed using an Agile software development due to changing requirements during development.

The database and back-end will be installed on a user's computer, which should be running constantly, which is important to successfully allow saving of all usage data. This could be accomplished on a desktop computer or theoretically a Raspberry Pi. A Raspberry Pi would make more sense if the user's computer is shutting down or being moved. The front-end would be hosted on this computer too but would be accessible anywhere on the local network the host computer is available at.

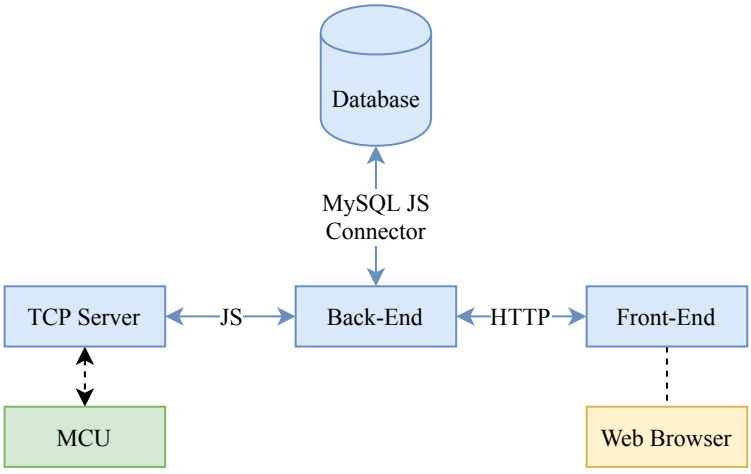


Figure 6.1: Overview of software system

6.1 Front-End Development

The front-end is based on JavaScript for its compatibility with many browsers. The main tool being used is React, a popular library designed for building user interfaces designed and maintained by Facebook. It supports web-applications and mobile applications, which means there is potential that this user interface could be ported to a mobile application environment, making it more mobile friendly. There is plentiful support from both community and documentation which makes the library easy to use.

Another popular option for building user interfaces is Angular, which is developed by Google. Angular is fuller featured out of the box, where React requires libraries for many features Angular includes. This means that Angular can take more time to learn about all the libraries, while React includes the necessities and other features are added by the developer, which makes documentation easier to digest. Angular can also be more complex than React, making it tougher to learn. In research, it felt like there was more information about React than Angular, so React was chosen.

Bootstrap was chosen as an HTML/CSS framework. It includes a lot of commonly used user interface components, such as buttons, tables, navigation bars, and more. This was important to use so that more time could be spent on development of the project itself as opposed to creating UI components.

Chart.js is another important tool used for creating charts. A chart is the best way to view usage history of power usage of specific devices. Chart.js allows dynamic building of charts, and accepts JSON files for data, which is useful as that is what is returned from the back-end from database requests. It also supports dynamic updating of data, which may be used to implement features on the home page, such as a real time power usage pie chart.

These tools together are used to create the website. They are all open source and managed through npm – a package manager included with Node.js. This helps with future revisions: packages can be updated or keep specific versions for compatibility reasons. Since they are all JavaScript tools with community React support, using them all together is made easy.

A nice feature of the tools being used is they support dynamic sizing of user interface elements. For example, Figure 6.2 shows screenshots of an early version of the web application running on a tablet and phone. No manual changes were made to make these application pages work properly on these other devices. This is important for user accessibility, since one of the objectives of this project was the interface should be easy to use and accessible.

This is a feature which comes from React that libraries also support. The best example of this is the navigation bar on the phone screen. In Figure 6.2, the navigation bar has been minimized to a drop-down menu, which automatically slides down as the button is tapped. The other UI elements also resize automatically so they do not overload the screen, such as massive buttons or oversized charts.

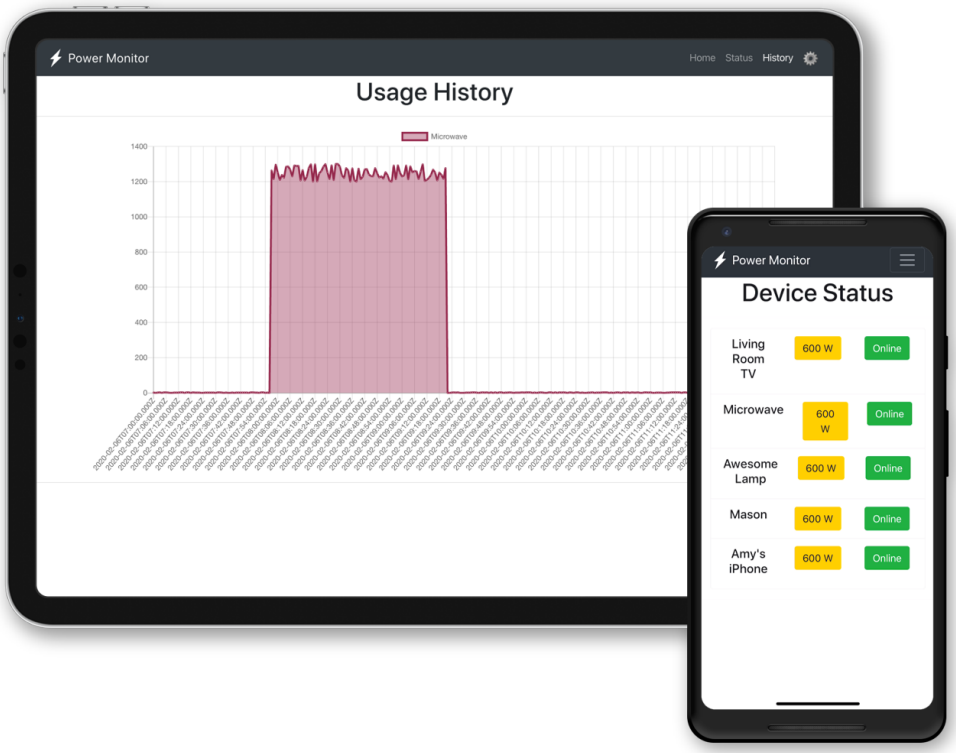


Figure 6.2: Early application version dynamic sizing

6.1.1 User Interface Mock-up

Figure 6.3 shows a mock-up created in Adobe Illustrator before any programming was completed. This allowed for a clear design direction that gave specific elements to work on. It also gave a general layout and allowed thought to be put in about how the elements themselves would work and what order they should be displayed in to be most intuitive to a user.

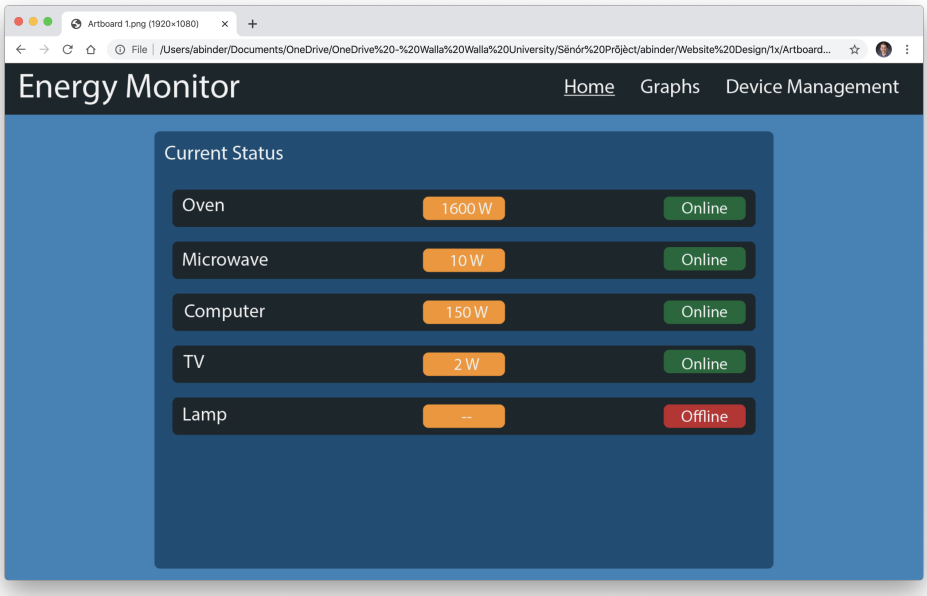


Figure 6.3: Website homepage mock-up.

The first component needing designed was the navigation bar. In the mock-up, three different tabs were selected. The homepage (shown) gives all devices in the home, their instantaneous energy usage, and a device status. The graphs page would show usage history, and device management would manage the addition/removal and other configuration of devices in the network.

On the homepage, the mock-up showed a collection of other components that needed implementation. To generate each object, a method was needed for retrieval of each device and its respective status. The database design has a table for each device, so each device will be a database result (more on database design in Section 6.3). The rest of the status information would be an API call to the back-end, which would ultimately make TCP calls to each device asking for a status update. The power draw should also update in real-time to reflect current power draw.

6.1.2 Website Implementation

Navigation Bar

In implementation, one of the more important components is the navigation bar. While all information could technically be made visible on one screen, separating it into different pages will considerably clean up the visual design. Bootstrap has a component for navigation bar and navigation links, which was a good starting point. However, these are only visual elements and do not handle any routing. So, a router was included which is responsible for taking different URL's

and routing the user to the correct UI elements. For the UI elements to properly display (highlighting the link of the present page), they needed to be passed through the router.

6.1.3 Home Page

Using these tools and the mock-up, a web application was created. During implementation, it was decided that the status page would be the main home page. An implementation was created and was more of a learning exercise than final implementation. It showed successful interaction between an object of devices returned from the database and UI components from the Bootstrap library. From here, HTTP routes (discussed in Section 6.2.1) are used to get device information from the back-end. The general flow of information is shown in Figure 6.4.

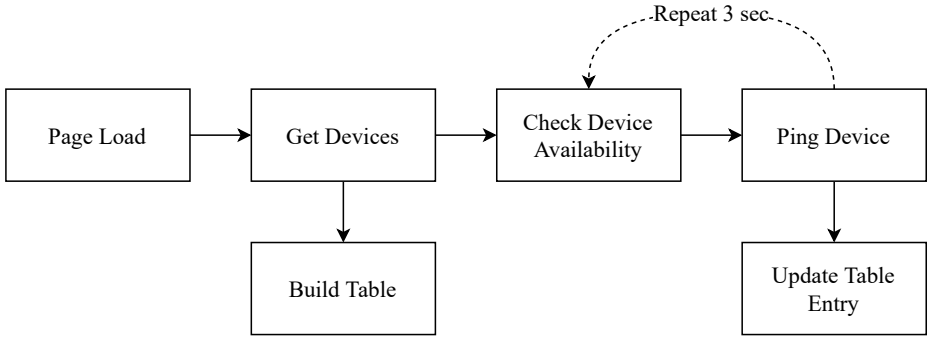


Figure 6.4: Information flow of the home page

On page load, each device returned from the database makes a request to the back-end's TCP server to check its status. This status is shown in the third column and shows either "Online" or "Offline." When the page is loading, the default text and variant of each label is "Loading" and `'warning'` and stays that way until a response is returned. This way, if the back-end or network is a little slow, the page shows the user that the status is still loading.

Once the device's status is confirmed to be online, an additional request is made to get an instantaneous power reading from the device. If the device is available, this action is repeated every three seconds, to give the user an idea of live power draw from the device. For each device, the power reading label can be pressed to see this additional data. It displays the power, voltage, frequency, and power factor. Figure 6.5 shows the final implementation featuring all UI components.

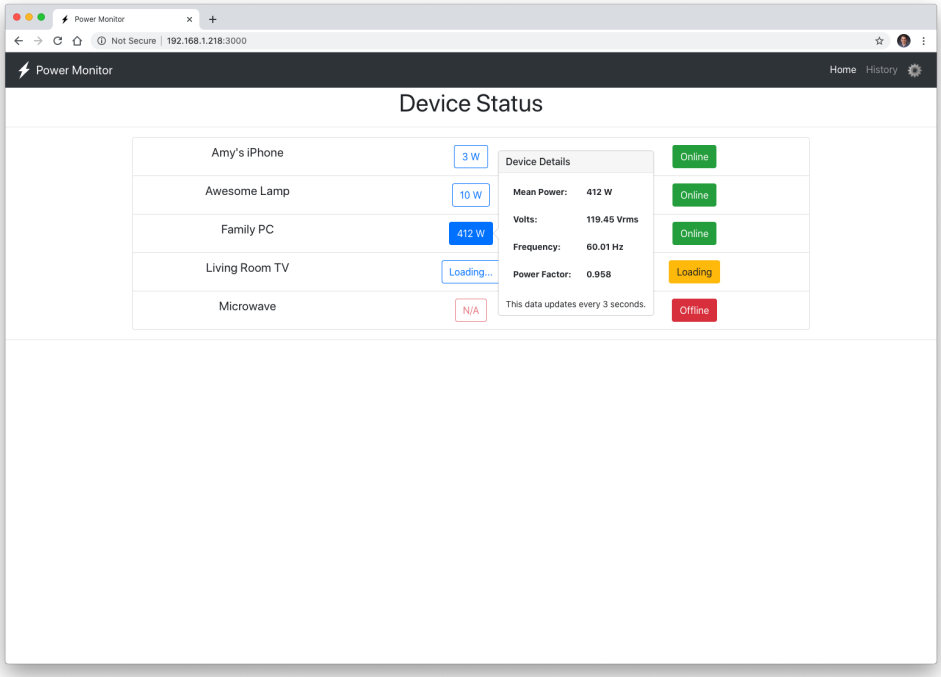


Figure 6.5: Implemented homepage showing device details

6.1.4 History Page

Another page, the *Usage History* page, was implemented. This page has a drop-down list of available and enabled devices and a date-time range picker, for customization of the chart data. This will be done exclusively using SQL commands through the database. When the page is first loaded, an API request will be made which runs a SQL query that requests a list of the devices the database has information about. That data will be sent to the drop-down, where a user can select a device to show in the graph. Then, a user can select a date and time range. The front-end will generate an API request that has information about devices and date ranges, which the back-end will generate a SQL query from and return that data. That data will then be shown on the graph to the user. This generalized flow of information can be seen in Figure 6.6.

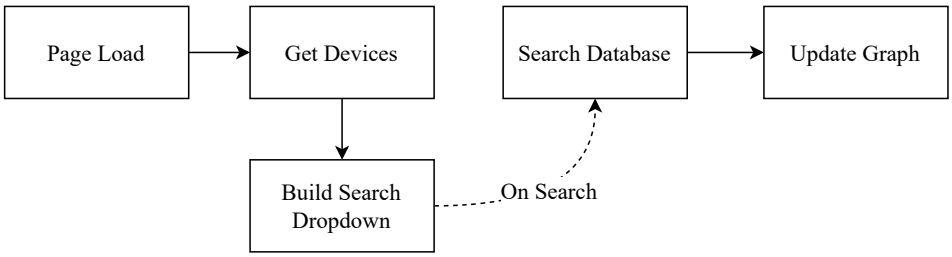


Figure 6.6: History page information flow

The page automatically creates a new API request when a different device is selected, which updates the chart dynamically. The date-time pickers do not follow this behavior, as they are the first user input that is changed in making a new query. It would not make sense for the page to update when the start date is changed, but not the ending date. These date-time pickers default to showing the last 24 hours of data but can be adjusted.

Since the database only returns samples between the two dates, the first and last returned samples correspond to the closest sample taken within the given range of dates. The chart uses the first and last sample to set the limits on displayed data, so there is no empty white space on the ends of the chart. The completed history page is shown in Figure 6.7.

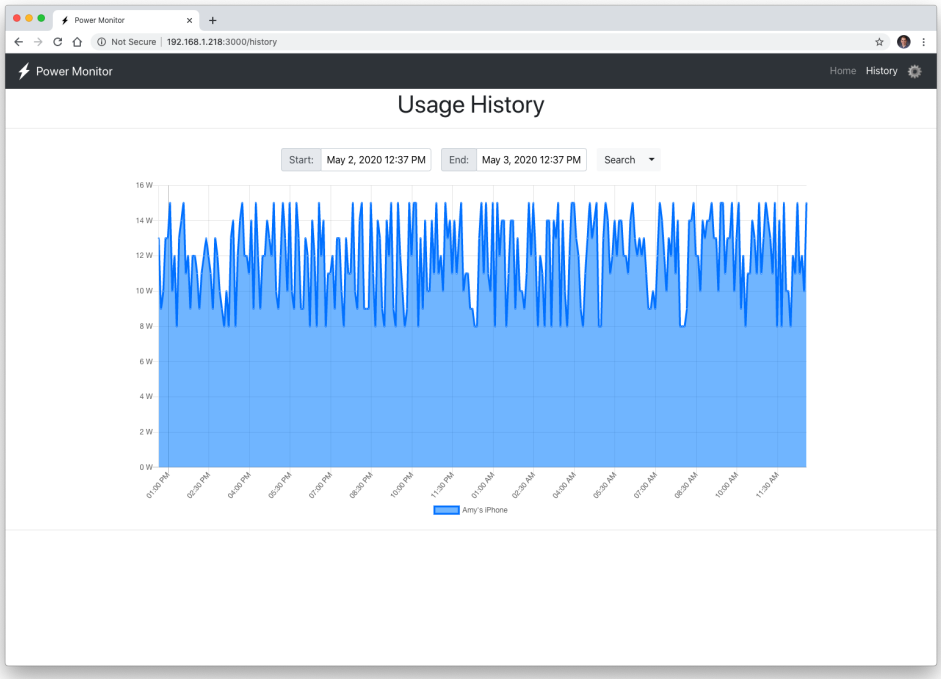


Figure 6.7: Implemented Usage History page

6.1.5 Settings Page

The *Settings* page was added to manage devices. When a new device needs to be added, or a device renamed, there should be a user-friendly way to accomplish that. Currently, there are three actions which can be done on this page. Users can temporarily deactivate devices, delete devices, and add new devices. All actions use the same API call with different header properties, and that information flow can be seen in Figure 6.8.

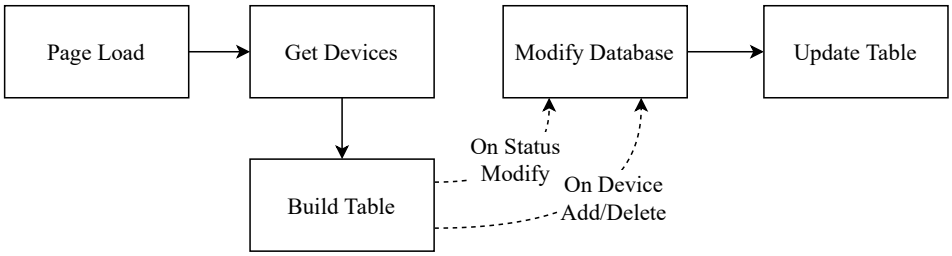


Figure 6.8: History page information flow

Device deactivation can be useful when a device is not wanted on the homepage or the chart page. It changes one column of the database, without removing any usage history. This is done with an API call to the `'devices'` table of the database.

Similarly, devices can be added and removed easily too. All known devices (active or not) are shown on the page, with a dropdown option. This holds the delete option, which removes a device from the `'devices'` table of the database. Upon clicking the dropdown, an additional modal pops up to confirm the action, to avoid accidental clicks.

Devices are added with a group input at the bottom of the page. It takes a device name and ID and makes a call to the API with this information. There is some light data validation done on the front-end, mainly making sure the text boxes are not blank. Most of the data validation is done by the database, including field length and character stripping. Figure 6.9 shows the implemented settings page. This is not the best solution to adding new devices, as there is some setup that needs to be done on the device itself, such as getting the appropriate IP address. Ideally, this setup would be completed on this page with a wired connection to the device. This challenge is discussed more in Section 5.7.

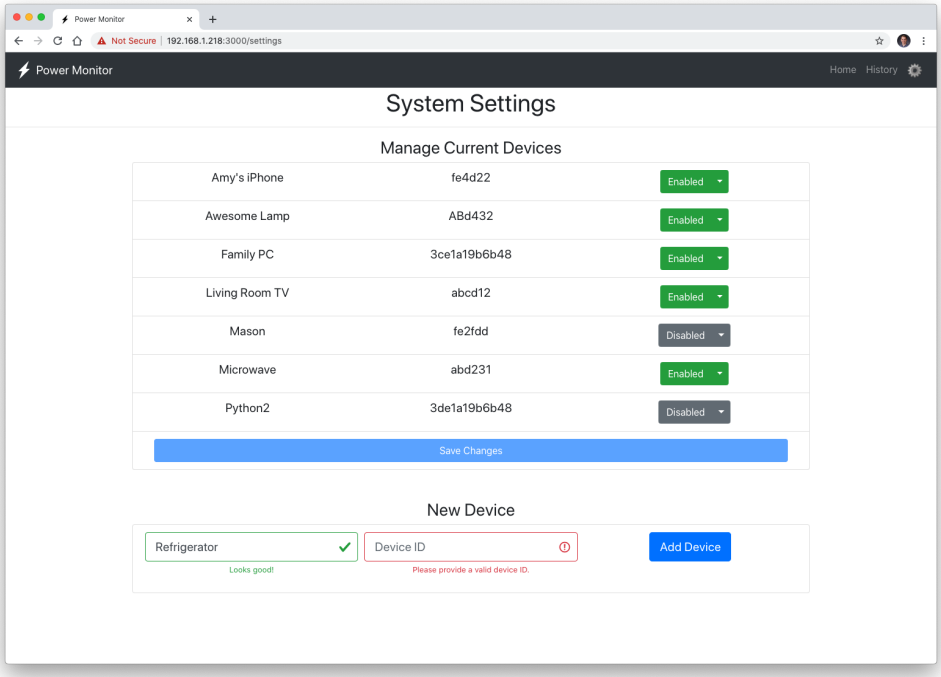


Figure 6.9: Implemented Settings page

6.2 Back-End

The back-end is based around Node.js, a JavaScript runtime built from Chrome's JavaScript engine. Express.js runs on top of this, which gives web tools that make it easier to create web applications and API's, which will be important when integrating with the front-end. Node.js includes the Net library, which has a useful component for sending UDP packets and hosting a TCP server. This is nice because it allows for seamless integration with hosting a TCP server on the back-end.

Another option that was considered was using PHP for the back-end. PHP is very popular for server back-ends since it makes accessing databases much easier. It has also been around longer than the Node.js/express.js method. While PHP is slower than JavaScript, ultimately performance will not matter too much for this application, since relatively few requests will be made in general use. JavaScript as a web development platform has been increasing in popularity for years, so there is lots of support and documentation that is helpful for the project.

JavaScript was ultimately chosen for simplicity reasons. API calls will return JSON objects, which are JavaScript objects, which makes front-end implementation simpler. It supports the front-end tools being used better than PHP, and it only requires using one language for the software of the project instead of two: JavaScript and PHP. The host computer will then only need JavaScript dependencies, not PHP dependencies.

6.2.1 HTTP Routes

All communication between the front-end and the back-end will happen with HTTP requests. This allows any browser running the front-end to access data from the back-end, without compatibility issues.

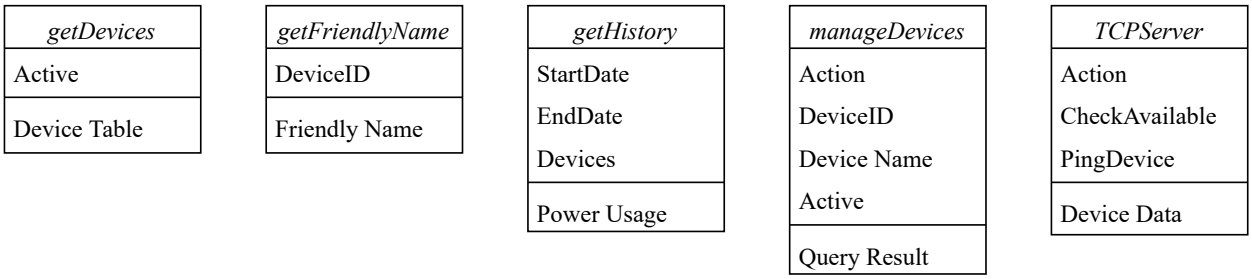


Figure 6.10: Summary of HTTP Routes

There are five main routes used which are summarized in Figure 6.10. Each route has a few expected headers that are called by the front-end, which are summarized in the upper boxes. The lowest box gives the data the back-end returns to the front-end when a given request is made. The next section will go in-depth on each route and how it is configured.

getDevices Route

The getDevices route is used by every page implemented in the front-end. It is responsible for communicating with the database to produce an object of the Devices table from the database.

There are two possible queries that this route will run, depending on the `active` header. When the `active` header is not present or not equal to `'all'`, the query in Listing 6.1 is used. Each query will alphabetize the results based on the friendly name of the device, making UI elements more user friendly as they are sorted in a recognizable way.

```
SELECT * FROM powermonitoring.devices ORDER BY FriendlyName;
```

Listing 6.1: Query for retrieving all devices

When the header `active` is present and equal to `'all'`, then a slightly modified version of the above query is used. By enforcing a true active field, the database returns only devices that are set active by the user. The query is listed in Listing 6.2

```
SELECT * FROM powermonitoring.devices
    WHERE active = '1' ORDER BY FriendlyName;
```

Listing 6.2: Query for retrieving active devices only

Both queries are important; for example, if the user wants to hide a device by setting it as not active, then it will not be returned in most requests to this route. It is important to have deactivated devices still available, in case data history is wanted, so the settings page will use the `active` header to get all devices and allow for activated and deactivating devices.

getFriendlyName Route

The getFriendlyName route is used on the usage history page. Only a Device ID is needed to get usage history on a device. Since that is the only data the chart element receives, a route is needed to convert that ID into the friendly name set by the user.

Only one header is expected by this route: `device`. The value of this device is used as a parameter to fill in the SQL query shown in Listing 6.3.

```
SELECT FriendlyName FROM powermonitoring.devices WHERE DeviceID = ?;
```

Listing 6.3: Query to get FriendlyName of a device

This query returns a single string containing the "Friendly Name" of the requested device. In normal use, the requested device will always exist in the table, since the requested device will come from a list given by the getDevices route.

It is important to note that the mySQL connector library includes a function to format a SQL string with parameters given in an array of strings. Parameters are filled in place of question marks. This function strips all potentially malicious characters from the string and parameters array, which blocks SQL injection. While this is not a concern here, it is in other routes, where direct user input is used as a parameter. All SQL queries run on the back-end use this formatting function, regardless of their content.

getHistory Route

The getHistory route is responsible for querying the database about a specific device and returning all measurements it has from that device. It takes three header fields: `'startdate', 'enddate', 'devices'`. Before feeding it into the SQL format function, the function converts the `startdate` and `enddate` header fields to the `datetime` SQL format. It expects that the given dates are in UTC ISO format: `2020-04-28T04:42:07Z`. The function will convert the time to the local time zone before inserting the dates as parameters.

Once the parameters are in order, the route queries the database using the SQL of Listing 6.4.

```
SELECT * FROM powermonitoring.powerusage
    WHERE MeasurementTime
    BETWEEN CAST(? AS DATETIME) AND CAST(? AS DATETIME)
```

```
    AND Device = ?;
```

Listing 6.4: Query to get FriendlyName of a device

This query will return a list of measurements containing the measurement time and the power usage at that time. This data is passed to the chart element of the History page, where the data is graphed. If the device or date/time range is not valid, the database will return an empty array of results.

manageDevices Route

The manageDevices route is used to handle most requests from the settings page. It is mainly responsible for controlling the device table of the database: adding devices, deleting devices, and disabling devices. There are three actions that can be accomplished on this route, as set by the `'action'` header. Valid values are `'insert'`, `'delete'`, and `'changeactive'`.

The `'insert'` action takes two inputs: `'deviceid'` and `'devicename'`, which is the information needed to add a new entry to the device table in the database. In this case, it is very import to escape malicious characters with the mySQL connector format function since raw user input is used in the query. There are also some hard limits set on the database data types, including that the MAC device ID cannot be more than 26 characters. Since the user is adding a new device, it is assumed that the device should be activated, so it is hard coded into the query. The query is shown in Listing 6.5. This function returns whether the query was successful.

```
INSERT INTO powermonitoring.devices VALUES ( ?, ?, 1);
```

Listing 6.5: Query to add new device

The `'delete'` action takes one input: `'deviceid'`. This value is passed to the HTTP request by the front-end, based on the item the delete was called from. This function also returns whether the query was successful. This only removes it from the recognized devices table, so if the device were still connected and sending data, the data would still exist in the usage database table. This is useful if a device needs to be renamed. The query is shown in Listing 6.6.

```
DELETE FROM powermonitoring.devices WHERE DeviceID = ?;
```

Listing 6.6: Query to delete device

The final action, `'changeactive'`, is used to disable or enable a device. It takes two inputs: `'deviceid'` and `'active'`. Both values are fed from the front-end, so it is assumed these will always be valid inputs. The two inputs are formatted to the query shown in Listing 6.7. This function also returns the success of the query.

```
UPDATE powermonitoring.devices SET active = ? WHERE DeviceID = ?;
```

Listing 6.7: Query to modify device's active state

TCPServer Route

The TCPServer route is used mainly to get live updates from the power monitoring devices themselves. The file containing this route also contains the main TCP server, which is discussed in Section 6.2.3. This route makes no calls to the database when requested from the front-end. Instead, it has the TCP server make calls to the devices. This route is called through the home/status page, which shows live data from the devices. One call is made for each device, with one of two possible actions.

The first action is `'checkavailable'`. It is used to see if a specified `'device'` is currently connected to the TCP server. Every connection is stored in an array of sockets including the device's ID. When this route and action is called, the array is checked to see if that requested device exists in the list of socket connections. It returns a simple true/false depending on the status of the connection.

The other action available is `'pingdevice'`. This is used to get some more detailed information about the current reading from the power devices, such that the home/status page can show the most up-to-date data possible. By sending a packet with the `ping` subtype, the device will take a reading and respond with a reading of the appropriate data (see Section 5.4 for more packet details).

Once a packet is generated and sent to the specified device's connection socket, a listener event is created. This listens on the same socket for a response from the device, which ensures the response is routed correctly and not mistaken for another device's response. Once the listener reads a response, the data packet is sent to a function that creates a JSON object with appropriately labeled fields which is then returned in a response to the front-end.

If the listener times out (after 5 seconds), the function will return a note telling the front-end the socket has become unavailable. This would most likely occur from a device crashing, as the front-end checks that the given device is available, using the other action in this route, before requesting a ping be sent.

6.2.2 TCP Server Background

The TCP Server is a significant portion of this project. It is the main communication method for the multiple power devices with the host running the back-end. This project essentially building on application on top of the TCP protocol (see Section 5.4 for details about packet formatting), which requires plentiful consideration on its own. Since the project aims the be compatible with existing Wi-Fi networks, anything lower on the network stack would start discarding ensured compatibility with little technical gain.

There were two main options considered for network communication: UDP (User Datagram

Protocol) and TCP (Transmission Control Protocol). While both have their pros and cons, TCP was chosen over UDP for several reasons. The two main reasons are reliability and connection style, which are discussed below:

TCP has a plethora of properties that make the data sent more reliably reach its destination. For example, it automatically will guarantee delivery of a sent packet. If the packet is lost anywhere, it will resend the packet. It does this by completing a "handshake" whenever data is sent, which tells the sending client that the packet has reached the destination. It also ensures multiple packets are received in the order that they were sent. TCP also includes error checking and handling: if a packet becomes corrupt, it will try to fix it or request that it is resent. UDP does not have any of these tools, which makes it a better tool to use. Many of these features would need to be implemented in the application layer if UDP was used.

Another important reason TCP was chosen was connection style. TCP is connection-oriented, while UDP is connection-less. A connection-oriented technology makes more sense for this project. The host computer is the only thing that will communicate with any given monitoring device; the devices will never communicate with each other. It therefore makes more sense to host a TCP server on the host computer which the devices connect to as clients for communication. This can also be used to track what devices are connected and which are not. If a device is connected to the TCP server, it can send a status request, which can be used to determine status. If a device is not connected to the server, it is obvious the device is not online, and a status packet does not need to be sent.

A challenge this brings, which also applies to the some of the other components, is IP addressing. If addresses are changing, that needs to be communicated to all the usage devices, so they know where to find the TCP server. This could be fixed with static addressing, but that is something the user should not have to worry about configuring. It would be more user friendly if the user could avoid adjusting their home networking setup.

6.2.3 TCP Server Implementation

Thanks to the packet design discussed in Section 5.4, implementation of the TCP server was straight forward. The packets were simplified into three main actions, shown in Table 6.1. The only setup needed to get the TCP server running in node.js is setting the IP address of the host computer, assuming the default port is available. If run on a dedicated machine (such as a Raspberry Pi), these setup steps are trivial.

Action	Initiator	Expected Response
New Connection	Monitor Device	Current Time
Current Status	Back-End	Instantaneous Measurement
Data Dump	Monitor Device	Insert to Database

Table 6.1: Summary of possible TCP actions

The first action brought some challenges. When a device first connects to the server, the back-end will have no idea what device it should correspond to. While it would be possible to use the IP address it connected from to link to a given device in the known devices database table, this approach brings some problems. All home networks assign IP addresses with a DHCP server in the router, which usually do a good job at reserving an IP address for a device when it disconnects. However, if the device gets a new IP address, the back-end has no way of re-linking that device.

The design approach to solve this problem involved creating a dedicated packet subtype for new connections. When the power monitoring device successfully connects to the TCP server, the first step it completes is generate and send an identifier packet. This packet contains the normal packet identifier, version, and subtype, and most importantly this packet subtype contains the MAC address of the device, which is the "device ID" the back-end knows and looks for. When this packet is received in the TCP server, the TCP server creates a socket uniquely for that device. The MAC address received is appended to this socket, and the socket is pushed onto an array of known sockets.

After sending a new packet connection, the device expects a response. The packet subtype for this response contains a different subtype and the current time. The subtype acts as the acknowledge to the new connection. The current time is generated as an unsigned integer of the Unix epoch, and then converted to seconds, since that is easiest for the device to process. Since the device is known and the socket has been identified, it is trivial to reply to only that socket with this data. This flow of information is summarized in the block diagram of Figure 6.11.

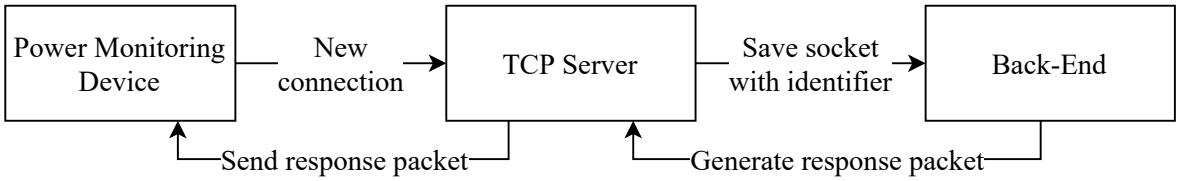


Figure 6.11: Data flow of new device connection

The next action considered was the "Current Status" action. When the status page is loaded, the front-end will make an individual request for data from each device. Each request needs to be responded to with the data from the requested device, so that data from devices is not mixed up.

This was solved by creating a listener event for each socket/device a request is sent to. This listener waits for *one* response from the given socket, and then ends the listener. On response, the packet is parsed according to the pre-determined packet format, and a response is generated using the socket's saved MAC address identifier and the data from the power reading. A block diagram is shown in Figure 6.11.

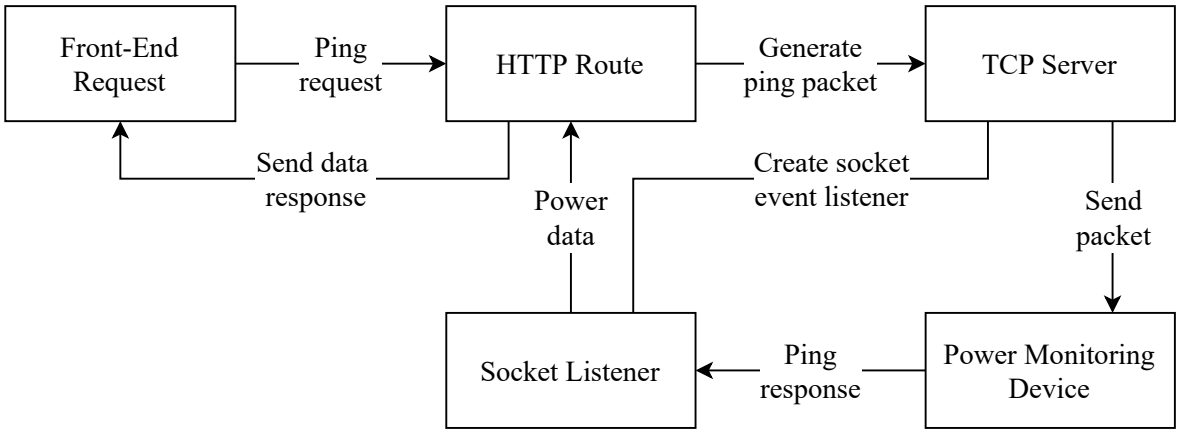


Figure 6.12: Data flow of ping request

In this process, the raw hex data from the packet needs to be processed from the format the power IC is sending to the microcontroller. This was a challenge because each measurement register stores the data in a different format. It appears that some are (two's complement) signed, some are unsigned, and some are unsigned with a sign bit. Once the datasheet was fully understood, it was possible to correctly interpret the register values and return them as signed integers in an object for the front-end.

The final action is the data dump. The power monitoring device will save several samples to its local storage, to save on network traffic. This packet is sent automatically by the devices, so the back-end needs to always be available. The TCP server is set up in a way that listens to all packets and decodes them as they are received. When the data dump subtype packet is received, the data is parsed and sent to the database. Each sample from the device contains the time of the sample, which is sent to the database for each power usage sample.

When the packet is received, a SQL query is generated for each sample that inserts it into the database. It may be beneficial to convert this to one large SQL query in the future, but in testing the single SQL queries have been working well. This simple flow of data is shown in Figure 6.13.

Figure 6.13: Data flow of data dump packet

One challenge with the TCP server and communication was the byte-packing needed to build packets. To keep network overhead low, the packets were designed to be as condensed as

possible. JavaScript does not have an included method for densely packing data into a buffer. There are two issues this creates: receiving packets and sending packets.

The included net package of node.js, which is responsible for creating and running a TCP server, brings a method for setting the encoding of a received method. By setting it to `'hex'`, the message is treated as a string of hex values. This is easy to work with, as the individual bytes can be extracted from the string based on the predetermined packet structure. The bytes can be cast as hex to JavaScript-friendly integers.

For packet sending, there is a package available on npm that allows declaration and use of C-style structs that was chosen for this project. It creates a byte-packed object that can be referenced like a normal JavaScript object. An example can be seen in Listing 6.8, where an initial open connection response is generated to send to a newly connected device.

```
let message = Struct()
    .word8('PACKET_ID')
    .word8('VERSION_ID')
    .word8('SUBTYPE')
    .word32Ube('TIME');  // big-endian for network!!
```

Listing 6.8: Example JavaScript struct declaration

6.3 Database Design

The database is where the "big data" saved from the individual power monitoring devices will be stored. At a given time interval, the monitoring devices will send the average of the last time interval's power usage to the database to be stored. A database was chosen to aid in efficient data retrieval. This data could easily be appended to a text file or a JSON object, but efficiently searching that data would take extra implementation that a database does on its own.

Oracle's MySQL database was chosen as the database framework. MySQL is a relational database based on structured query language (SQL). It was mainly chosen due to past familiarity with the product. Interestingly, MySQL has out of the box support for MySQL, while with Node.js, a connector library is needed. A benefit of MySQL is that MariaDB is also being maintained as a drop-in replacement by the original developers of MySQL. So, if anything were to happen MySQL, MariaDB could be swapped in trivially.

There are two of main tables needed in this database: one to keep track of devices and one to store the actual data in. The ER Diagram shown in Figure 6.14 explains in more detail the entities and how they are related. It is a relatively simple design with two tables, but as requirements change and features are added, it is trivial to adjust the database to store different power options without rebuilding the entire database.

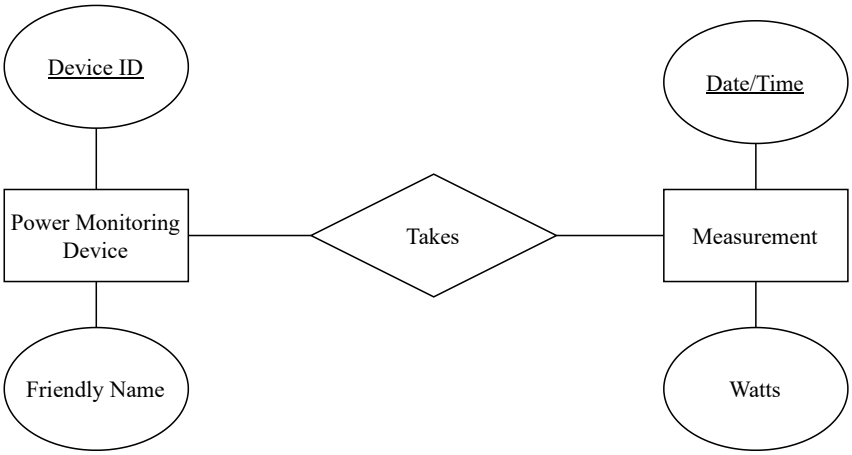


Figure 6.14: Entity Relationship Diagram

One table will be used to store a list of devices in an installation. Each device will have a unique ID (MAC Address) which needs to be translated to a more user-friendly name. The user will only know a specific device by the name they assign it. It is important that this name can be changed at the user's discretion, such as devices being moved between appliances.

Another table will be used for data storage. For each time interval, a power value will be stored with the device's unique ID and the date/time the value was read. When this data is requested by the API, a natural join on the two tables will be done, which will return each reading between two-time intervals with the friendly name of the device. An issue with this might be that if a device is repurposed, suddenly it has the data history of a different device, which may be a problem. It may be worth modifying the device table to store a history of names, and then join the tables using dates from each table.

6.3.1 Database Implementation

A MySQL community server must be run on the back-end host machine for data storage. It contains one database with two tables: one with devices, and one with the usage history. The two tables are shown in Table 6.2. The back-end connects to this database through a node.js MySQL connector, which allows the back-end to run queries on the table.

PowerUsage	
MeasurementTime	DATETIME
Device	CHAR(26)
WattValue	SMALLINT

Devices	
DeviceID	CHAR(26)
FriendlyName	VARCHAR(45)
Active	TINYINT

Table 6.2: Database tables and columns

The data types were chosen to keep the database as compact as possible. The more exact data types allow the database to pack the information as densely as possible, which is important mainly for the power usage table, which will grow over time.

The devices table is used to store the device IDs known to the system. A known device will exist in this table with a device ID and a friendly name set by the user. The friendly name is the only name a user will know a device by, excepting when managing the devices. This is done to keep the system more user friendly.

Originally, the power usage table used a foreign key constraint to ensure the device was known in the devices table, but that was since removed. This decision was made to protect the data coming into the system. When there is data, it should be stored even if at that instant the device is not known. It could be that the user mistyped the MAC address when adding a new device, or a device is being renamed.

A security implementation mentioned in Section 6.2.1 had to do with SQL injection and malicious queries. In some implementations of databases and user input, it is possible to put an apostrophe or semicolon and a different query in to maliciously modify the database. In the implementation of this project, all malicious characters are stripped out before the query is generated and run on the database. This is done with a function included in the MySQL node.js library. Queries are built dynamically in a way that SQL injection is not possible.

7 Testing

7.1 Hardware

7.1.1 Construction of the Prototype

Based on the reference design and values calculated for the various components that did not have specified values, a prototype was constructed by hand using through-hole components ordered from DigiKey and assembled on generic proto-board. This was done for the purpose of initial testing, as well as to root out any errors in the design before a custom PCB was potentially ordered. The prototype in its current state can be seen below, with an IEC 60320 C14 socket for powering the prototype, and a NEMA 5-15 to plug whatever device is desired to have its power draw measured.

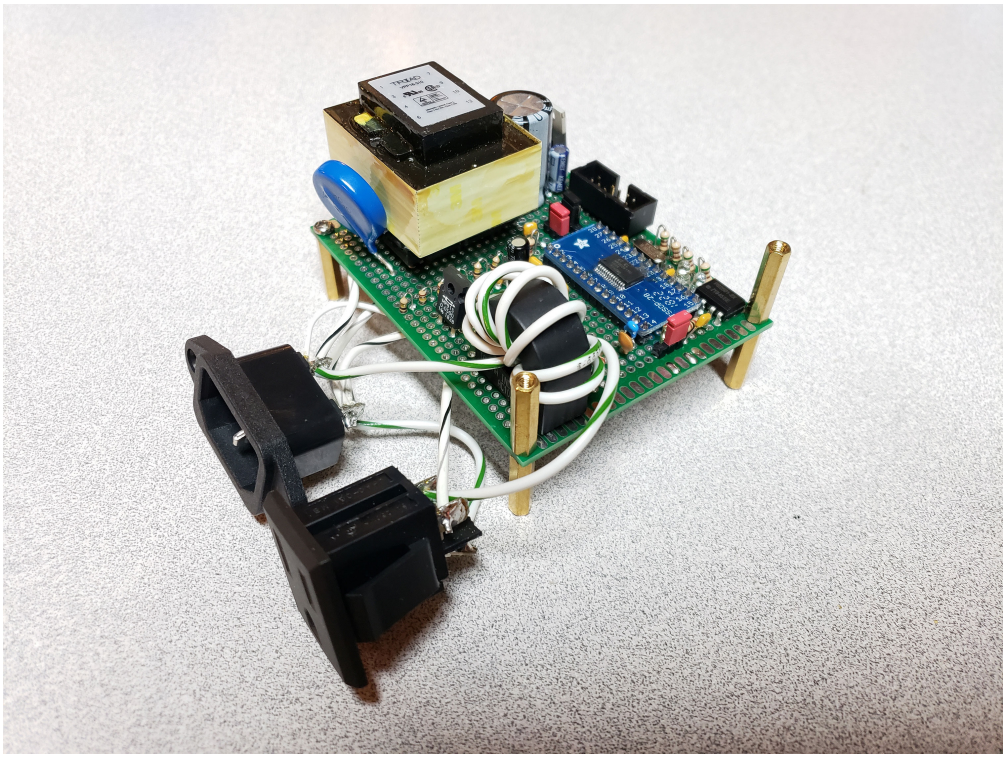


Figure 7.1: The Hand Built Prototype

The construction of the prototype was not without its challenges, particularly due to all the ground points in the circuit, individual capacitors and resistors, and care that needed to be taken to avoid creating a short. In addition, some components were accidentally not ordered the first time, since the reference design's BOM had missed some components such as the 12nF decoupling capacitors for the crystal oscillator as well as a 10μF decoupling capacitor between the digital voltage and ground input.

7.1.2 Current Measurement Testing

To do some initial tests on the prototype, the prototype was plugged into a standard GFI outlet for an additional layer of safety. Ideally, the testing would be done with the prototype plugged into an isolation transformer to completely isolate it from earth ground, but without access to the isolation transformers kept in the University labs, this was not possible. For a load, a simple plug in light bulb was used. This made for an ideal load, as different wattage incandescent bulbs could be used to change the load, and in theory the power draw of an incandescent light bulb will be constant. A 100W bulb was used to provide a measurable load that the ATM90E26 could easily detect.

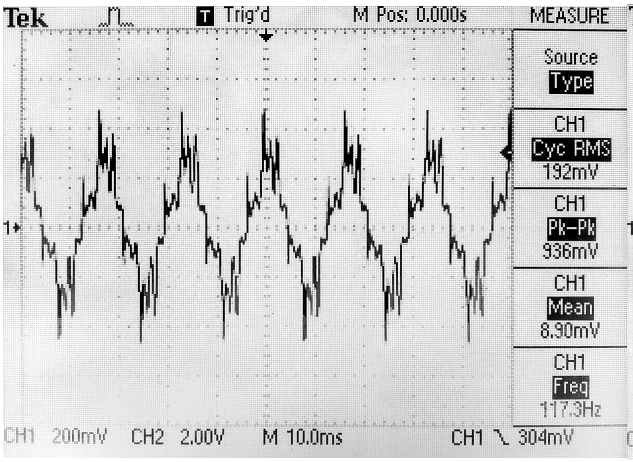


Figure 7.2: Neutral Line Current Sampling

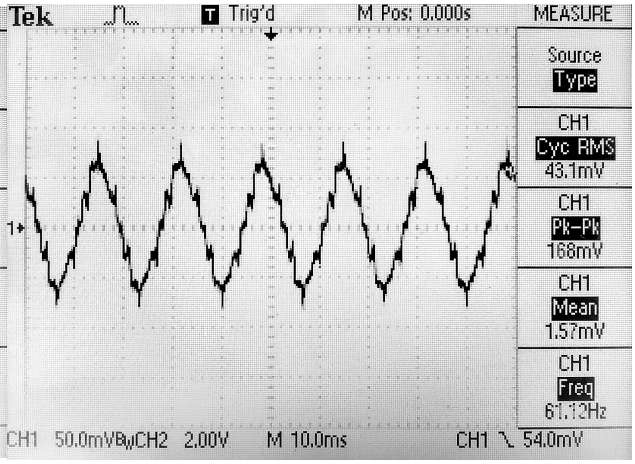


Figure 7.3: Phase Line Current Sampling

Above are two pictures taken of oscilloscope readouts from the neutral line current sampling (which comes from the current transformer) and the phase line current sampling (which comes from the shunt resistor). Both measurements were taken at the points where the waveforms entered the ATM90E26. Table 7.1 below documents the expected results of the components of the current measuring sections of the circuit as well as the actual results.

Design Goal	**Selected Values**	**Testing Conditions**
600mVrms @ 15A	Resistor: 50mOhm Transformer: 1:125 Turn Ratio	100W Bulb => ∼850mArms @ 120Vrms
Simulation Results	**Actual Results**	
Resistor: 35mVrms Transformer: 33mVrms	Resistor: 43.1 mVrms Transformer: 192mVrms	

Table 7.1: Current Measurement Results

Observing the results, the current sense resistor's values are well within margin of error, especially considering that the parts used in the prototype were not high precision. The current transformer however was well outside of the expected results. Observing Figure 7.2, it can be seen the waveform is rather erratic. In Section 4.3.4, it was determined that a 1:120 ratio would be best, but due to the cost of a 1:120 current transformer, an alternative solution was found by using a much more affordable 1:1000 current transformer and wrapping the primary wire around the body eight times to get a close 1:125 turns ratio. While the RMS value is not out of bounds of the ATM90E26's measuring range (192mVrms according to the scope), the signal is rather erratic, and would likely become out of bounds with higher currents. As such, the eight-hand-wound turns around the current transformer not transferring the flux as efficiently as expected. If a project like this were to be further developed into an actual product, it is likely that a custom current transformer would be used to get more desirable results.

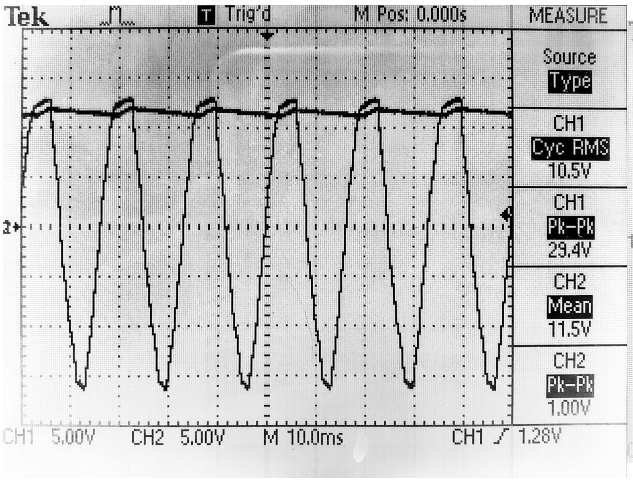


Figure 7.4: Neutral Line Current Sampling

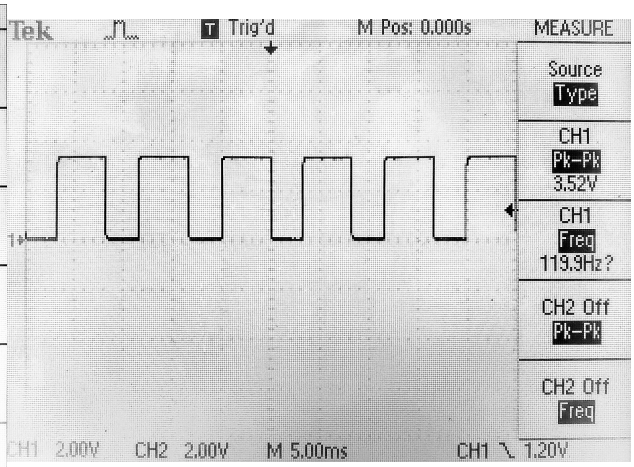


Figure 7.5: Phase Line Current Sampling

The power supply section of the design was the section that differed most from the reference design for the ATM90E26. In order to be considered to be operating correctly, it needed to provide a steady 3.3V output for the MCU and ATM90E26, especially when the Photon MCU was connecting to a Wi-Fi network, which is the time that the Photon draws the most current according to its datasheet. Observing Figure 7.4, the output from the power transformer (the sine wave, Ch.1) and the output of the filter capacitor (the line at the top that rises with the positive peaks of the power transformer, Ch.2) can be seen. This picture was taken while the Photon was already connected to a Wi-Fi network and actively sending packets to our TCP server, as the startup time when it is connecting was difficult to capture. There is very little ripple across the filter capacitor (about 1.00V according to the oscilloscope), and the output from the linear regulator was a steady 3.27V, meaning the design goals of the power supply were achieved.

Finally, to check if the ATM90E26 was functioning, the output of pin 21, the ZX (zero crossing) pin was probed. This pin is asserted every time the voltage sampling input crosses zero. As can be seen in Figure 7.5, there is a consistent square wave that the oscilloscope measures as having a frequency of 120Hz, which would make sense considering the line voltage is at 60Hz, and the voltage crosses 0 twice in every cycle.

7.2 Firmware

To prototype the firmware for the packet factory without requiring a fully functional TCP server, Python was used to create a script that could send UDP packets to the Photon to check how it responded. The differences between TCP and UDP are discussed in more detail in Section 6.2.2 but were minimal enough that this was a very effective prototyping method. The python script was capable of building packets with multiple different data types represented. The script would send packets requesting different Device OS API calls to be run on the Photon, and the Photon in response would run these API calls and output the results of them over serial. By

running a serial monitor while this script was sending packets, the responses were able to be seen and debugged. This process proved to be invaluable when converting the firmware code to work with TCP and different request packets. A lot of the API calls used here will need to be called during first time setup mentioned in Section 5.7, so having learned how these API calls work proved useful for future work.

As mentioned in Section 5.5.2, a C++ class was initially created to handle creation and loading of packets. This design first became an issue when packets needed to be sent. It was not apparent how each individual packet type could be accurately described by a single class and sent as an array of bytes. At first, dynamic memory was explored to allocate memory space on the fly for the different data fields represented in the various packet types. The dynamic memory would then be used to create an array capable of being sent as intended. Dynamic memory proved to be quite challenging to implement with the working design and after multiple hours of testing, it was clear a different approach needed to be researched. It was discovered that C++ vectors could be used in place of manual dynamic memory, and utilizing these would have worked for the original goal, but even vectors became an issue in prototyping when it was discovered that a single packet class would not have been sufficient to store the various data fields found in the multiple packet types that had been outlined. This problem could have been circumvented by creating multiple packet classes that inherited from a base packet class. While this fix seemed doable, the details were still unclear in how exactly these objects could be packed and sent out as packets. Considering this uncertainty, the design was shifted to use simple struct objects with the "packed" attribute to ensure there was no data padding when sending these structs as packets. It was clear from even initial testing that this approach would simply work as intended and required much less code. It was not the most robust solution but given the scope of this project it was not required to be.

Once a fully functional TCP server was provided, more in-depth and complex communication prototyping could begin. This process involved writing the firmware code that would handle incoming packets sent from the server as well as sending out packets in response and packets full of power monitoring data. Once that code was written, Wireshark is a simple program that can be used to sniff packets on a wired or wireless connection, and it was used to sniff the TCP packets on the network to verify that the data was formatted and sending correctly. Serial outputs were also utilized from the firmware to ensure that incoming TCP packet data was interpreted and handled correctly. This portion of prototyping occurred in close communication with Andrew as to guarantee that both sides of the communications agreed on how each respective side would run. Despite the challenges of having to work remotely, we were able to effectively debug both ends of the communication protocol and get each system working together as intended.

7.3 Software

During development of the front-end, there came a point where some testing had to be accomplished to continue development. Regarding the front-end alone, nothing special was needed to complete tests. However, communication with the microcontroller over the TCP server needed additional work. Some testing was able to be completed before the Design Review and

COVID-19 pandemic, however that was more proof-of-concept than development.

Some method was needed to create a power monitoring device stand-in while progress on the actual firmware was still being completed. Similarly, to the firmware testing, a Python script was written to act as a power monitoring device. The packet format (defined in Section 5.4) was used to create a script which sent accurate packets to the back-end software to decode and handle. This script was instrumental in creating a working system.

When the firmware for the microcontroller was ready, the code was shared with Daniel who was able to get the front-end software running and microcontroller communicating with the software seamlessly. There was some data interpretation that needed to be changed, but that was simple.

Another sort of user-testing completed was sharing the software portion of the code with the team. Helping them get it installed brought up some possible issues that were able to be fixed. One example is that the database user needs to be set up with standard security, not the "high-encryption" security type. This is something that needs to be specified in documentation.

7.4 Manufacturing Costs

To get an estimate of how much our project would cost to build if it were put into production, the unit cost of ordering surface mount parts in bulk was added up. While the following list is not 100% comprehensive, it includes some of the most expensive parts such as the transformers.

SMT Parts (Capacitors and Resistors)	$0.72
Transformers	$7.56
MCU, ATM90E26, and Linear Regulator	$2.58
Other Parts	$0.97
Total:	**$11.83**

Table 7.2: Estimated Materials Cost

The final total comes to about $11.84 per unit. While it is difficult to predict or estimate other costs per unit from sources such as software development, labor, manufacturing, advertising, and customer support if this project were developed into a marketable product, it isn't unreasonable to assume that due to the low cost for the hardware, devices based on our design could be sold for under $50 (our economic constraint for this project) while still maintaining a reasonable profit margin, assuming the devices could be sold in high enough volumes.

8 Project Evaluation

8.1 Design Criteria

There were several design criteria for this project on the hardware, firmware, and software fronts that the project aimed to achieve. On the hardware front, the hardware needed to be accurate, safe, cost effective, and able to communicate wirelessly over a local network. On the firmware front, the firmware needed to be able to collect information from the hardware, and reliably communicate that information to the software. And finally, on the software front, the software needed to be reliable, easy to use, display helpful information, and run on a local machine within the home network with little reliance on outside resources or 3rd party resources.

The software goals were met. The software is reliable easy to use, thanks to its simple design and user interface. It displays the most user-friendly data from the power monitoring device, such as power, voltage, frequency, and power factor. Any more information likely would have detracted from the experience for most users. Additionally, it is a lightweight package which runs easily on a desktop computer or a Raspberry Pi, and requires no outside resources once installed.

8.2 Practical, Economic and Safety Constraints

It was decided that the device should be safe and easy to use. A relatively small design would be beneficial. With the electronics design moved to a PCB and with some custom parts, the already relatively compact design shrinks. Once encapsulated in a box, the size of the device should be small enough to fit this constraint. It was designed to be used with an existing Wi-Fi network, which makes it easy to integrate into most homes. The software the user interfaces with the system was designed to be easy to use, which meets the practical constraints set.

Economically, a budget was set at less than $50 per device. The hardware costs (discussed in Section 7.4) are well under this budgeted price, but parts alone do not account for all the costs. Some additional costs might include ongoing development, manufacturing, marketing, research, testing, etc. All factors will play into the final cost of the device and cover much more than the engineering done and discussed in the scope of this project. It would be reasonable to say that the original economic constraint of less than $50 per device was met.

The safety constraints tie in closely with the standards and codes section. The main safety considerations had to do with keeping users and their homes safe from high current and voltage, which is the main purpose of standards followed. The hardware was designed with thought put towards over-current protection, excessive transient voltage protection, and a physical case around the device to protect the user from electric shock.

8.3 Environmental, Societal and Global Impacts

As mentioned in the introduction, the environmental, societal, and global impacts may be less significant in this project. If implemented and widely adopted, it has the potential to make homeowners more conscious of their energy usage and habits and potentially help lower their power usage. A society of people who are conscious of their energy consumption could bring positive environmental and societal impacts globally.

8.4 Standards and Codes

The main codes and standards used had to do with hardware design. The NSF and UL tests are certifications used to ensure that hardware is designed in such a way that protects consumers. While it was not feasible to have these tests done on the hardware, some of the stress tests done in these certifications were taken into consideration during hardware design. The main goal of the hardware was to keep the user and device safe under normal use.

9 Conclusions

9.1 Project Objectives

In the hardware portion of the project, the two main goals mentioned in *Design Criteria* were that the hardware needed to be accurate, safe, cost effective, and capable of wireless communication. These objectives were largely achieved. For the accuracy objective, the power monitoring IC that was chosen (the ATM90E26) was capable of high accuracy (within 0.1% for active energy measurement), and part values were chosen for the various measurement sections that would theoretically give the highest dynamic range of measurement for currents between 0 and 15A. While the current transformer did not exhibit expected behavior in practice, this could be remedied with a current transformer that was specifically designed for the application. The safety goal was addressed as well, with a resettable polyfuse that will trip if the current exceeds 15A (assuming the outlet breaker doesn't get tripped first), and a varistor to protect the circuit from any dangerously high transient voltages. The cost of the parts is reasonably low, particularly when purchased in bulk like they would be in a manufacturing scenario, and finally, the hardware incorporates the parts needed to communicate wirelessly with the firmware and software elements of the project, and was successful at doing so in testing.

The firmware had two main objectives defined for the project to function as intended. The first of these was to efficiently collect and handle the power monitoring data from the power monitoring IC. The second was to send power monitoring data to the TCP server when required to be displayed to the user. This was intended to be done with a layer of low-level firmware that

communicated with the physical hardware beneath a layer of high-level firmware that handled the overall flow of operation as well as the TCP communication. In the result, both main objectives were completed, although SPI communication remains relatively untested. The low-level firmware layer ended up being handled by the built-in operating system provided by the selected microcontroller, and the high-level firmware layer is where most of the development time was spent. The firmware is theoretically capable of receiving all the power monitoring information it needs over SPI from the power monitoring IC. The firmware is also capable of sending the necessary data it has collected to the software to be seen by the user. With the help of the standardized packet format that was created, this TCP communication is confirmed to be valid on both the sending and receiving fronts to ensure end-to-end reliability.

The main objectives defined for the software portion of the project had to do with ease of use and use of lightweight frameworks. The user interface was designed to be intuitive and easy to learn. In the light usability testing done within the group, it appeared the system was easy to learn and use. The frameworks chosen were also lightweight. In testing, the entire system ran transparently on the desktop computer and on a Raspberry Pi, as discussed in Section 7.3. Overall, the software portion of the project met its main objectives.

9.2 Future Work

9.2.1 Current Weaknesses

The biggest weakness with the hardware currently is that it is uncalibrated. The calibration process needs to be completed for the ATM90E26 power monitoring IC to measure accurate power data. For this to be done, the communication between the power IC and MCU has to work perfectly, calculations must be performed in the MCU, and proper values must be written to specific digital registers within the power IC so that it knows what value of current to associate with the voltages it receives from the current and voltage measuring sections. Unfortunately, there were issues with the SPI communication that are described in more detail below. In addition, this calibration process is best done with equipment that can provide a very stable, precise voltage and create a very stable, precise load, so that a well-known base current can be established. Working on this project remotely away from the University made equipment like this difficult to acquire. While all three parts of the project (hardware, firmware, and software) are communicating with each other as intended, the data is unfortunately unusable without calibration. Finally, while safety elements are incorporated into the hardware design, it cannot be guaranteed that the design meets professional standards such as UL's 62368-1 Hazard Based Safety Standard, as the documents that detail these standards are prohibitively expensive.

As it stands, the firmware suffers from a few weaknesses involving communication with both the TCP server as well as the power monitoring IC. The firmware communicates to the power monitoring IC over SPI, but the code in place to do this has been relatively untested, so it is unclear if any problems exist. SPI communication in the firmware is handled by Device OS

function calls. The functions provided use transmit and receive buffers that must always be the same size. This restriction is problematic because the power monitoring IC expects an 8-bit value transmitted as the address and will return a 16-bit value from the register at the specified address. Without proper testing we cannot be sure that the power monitoring IC receives the address as expected in the 16-bit transmit buffer.

The firmware communication with the TCP server is also less than ideal. The firmware samples the values from the power monitoring IC once every three minutes, and stores 20 samples before sending the stored samples in a Data Dump packet. These samples are used to update the usage history graph discussed in Section 6.1.4, but because of how the firmware currently sends these Data Dump packets, this graph would only be able to update once every hour.

Currently, the software portion's main weakness has to do with the communication between the front-end and the back-end. Currently, the back-end returns bare-bone responses with only the data necessary. It would be beneficial for troubleshooting if the back-end returned more information with the response. One example is that when the `devices` table of the database is empty, the front-end cannot generate tables for the empty response. A user would not be able to easily debug that problem. A more functional response could include information about the back-end as a whole: whether the database is online, whether the query ran successfully, or whether there was some other problem. If needed, the front-end could then display warnings or errors gracefully instead of failing to render the page.

9.2.2 Continued Work

Continued work for the hardware of the project include a custom PCB design, additional cost analysis, and check for certification compliance. The original plan for the project was to complete a hand-built prototype and fully test it before designing and ordering a custom PCB, but the hand-built prototype took quite some time to complete and test, making a custom PCB design outside of the scope of the project. On the front of cost analysis, basic cost analysis of the hardware was completed, but additional cost analysis could be done from a manufacturing, distribution, product support, and marketing standpoint. Finally, checks to make sure the hardware is compatible with official safety certifications and standards from independent laboratories such as UL and NSF should be done, assuming there is a good way to access the documents detailing these standards without having to pay the high prices the laboratories charge.

Continued work for the firmware would mainly address the weaknesses noted in Section 9.2.1. Setting up an environment where the SPI communication to the power monitoring IC would be the top priority to ensure that the power monitoring data is being accurately collected as is expected. Modifying the packet standard in a way that allowed the firmware to send a Data Dump packet on demand would also be a useful addition. This would allow the graph on the usage history page to update whenever it was loaded by a user to provide the most up-to-date data. A final bit of additional work could be done to provide a first-time-setup experience for the firmware. This setup would probably involve a physical USB connection to the finished product

with a software interface to allow the device to be configured. This configuration would include things like setting up the wireless network and setting a device name to be seen on the web client.

Some continued work for the software portion includes setting up an installer for all software components. Currently, it needs to be installed and built manually, which is not too hard thanks to node package manager (npm), but that is more only an option for people used to working on software. Another issue is the database is installed using Oracle's developer tools, which could be simplified. By creating an automated installer, it would take a lot of strain from the user during the installation process.

In addition, there are always other less technical features that could be added. For example, it would be interesting to see a real-time updating graph of devices on the home page. In addition, the history page only graphs one device at a time and supporting multiple devices could be another added feature. This would make it easier for the user to compare usage between devices.

References

"The Current Transformer." n.d. https://www.electronics-tutorials.ws/transformer/current-transformer.html.

IPC Task Group 1-10b. 2020. *Standard for Determining Current-Carrying Capacity In Printed Board Design.* Accessed January 13. http://electronica.ugr.es/~amroldan/cursos/2014/pcb/modulos/temas/IPC2152.pdf.

McFarland, Allen. 2019. "In 2018, the United States consumed more energy than ever before." Today In Energy. April 16. Accessed October 30, 2019. https://www.eia.gov/todayinenergy/detail.php?id=39092.

McGregor, Conrad H. 2020. "Power Plug & Outlet Types A & B." May. https://www.worldstandards.eu/electricity/plugs-and-sockets/ab/.

NSF International. 2020. "Home Product Certification Program - NSF International." NSF International Home Product Certification. Accessed January 14. http://www.nsf.org/services/by-industry/consumer-products/home-products-appliances/home-products.

P3 International. 2019. "P3 P4400 Kill A Watt Electricity Usage Monitor." Accessed October 30. https://www.amazon.com/P3-P4400-Electricity-Usage-Monitor/dp/B00009MDBU.

Pithadia, Sanjay, Scot Lester, and Ankur Verma. 2009. "LDO PSRR Measurement Simplified." July. http://www.ti.com/lit/an/slaa414a/slaa414a.pdf?&ts=1588882155187.

Scherz, Paul, and Simon Monk. 2013. *Practical electronics for inventors.* McGraw-Hill Education.

Sense. 2019. "Sense Technology." Sense.com. Accessed October 30. https://sense.com/technology.

STITCH by Monoprice. 2019. "STITCH by Monoprice Wireless Smart Plug with Energy Monitoring & Reporting." Accessed October 30. https://www.monoprice.com/product?p_id=27937.

U.S. Energy Information Administration. 2019. "Electric Power Monthly." EIA - Electricity Data. October 24. Accessed October 30, 2019. https://www.eia.gov/electricity/monthly/epm_table_grapher.php?t=epmt_5_6_a.

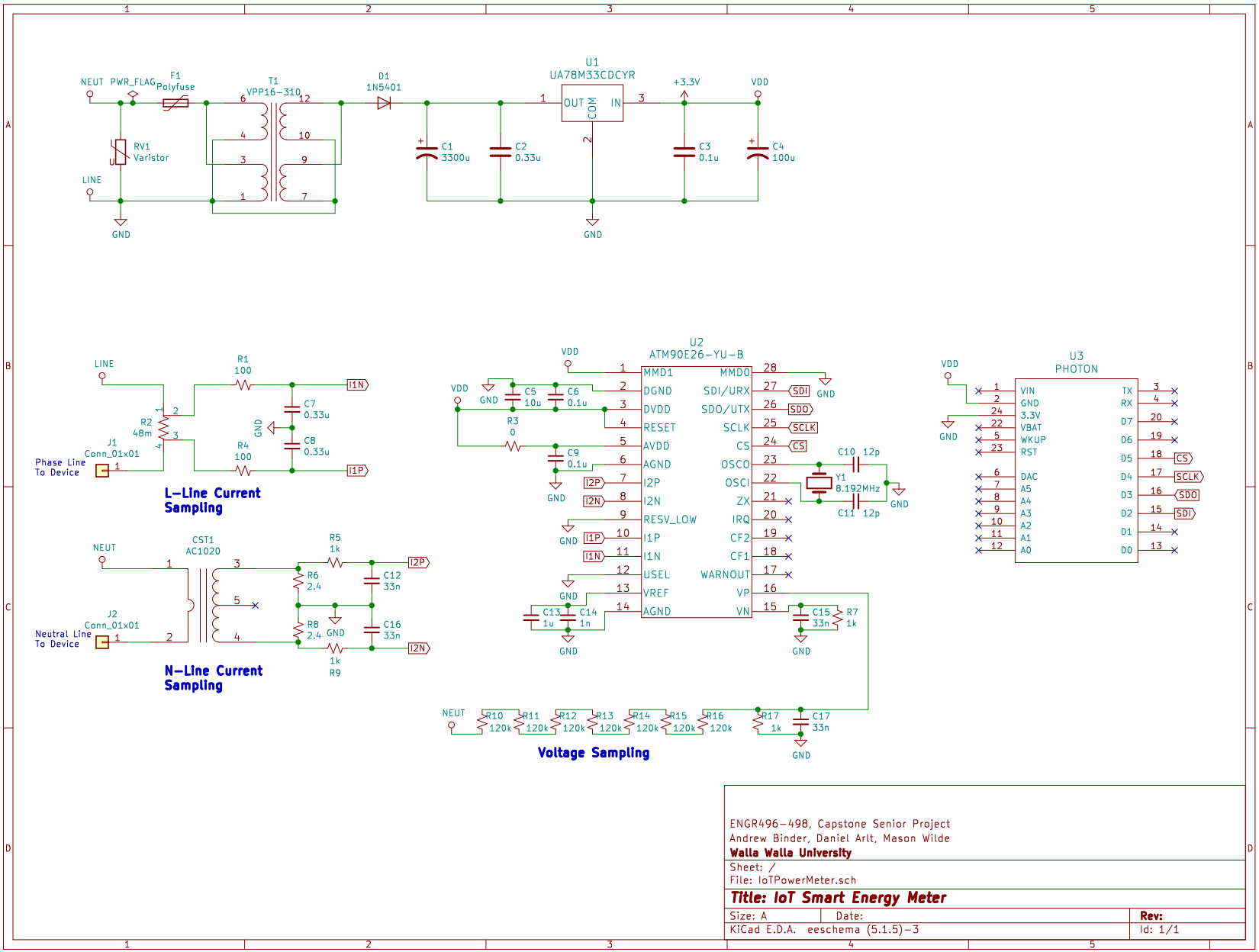
UL. 2020. "PCB Testing and Certification Services — UL - Consumer Technology Services." UL PCB Testing and Certification. Accessed January 14. https://ctech.ul.com/en/services/safety/pcb/.

Appendix

A Hardware

A.1 Design

A.1.1 Project Schematic



ENGR496–498, Capstone Senior Project
Andrew Binder, Daniel Arlt, Mason Wilde
Walla Walla University

Sheet: /
File: IoTPowerMeter.sch

Title: IoT Smart Energy Meter

Size: A	Date:	**Rev:**
KiCad E.D.A. eeschema (5.1.5)–3		Id: 1/1

B Firmware

B.1 Photon

B.1.1 main.ino

```
#include "packetTypes.h"

// Sample rate in seconds
#define SAMPLE_RATE 180

// Uncomment this to get various debug messages output over serial
#define SERIAL_DEBUG

void sendPacket(void* toSend, uint16_t size);
void packetHandler(char* buf);
void takeSample(void);
void TCP_Connect(void);

int16_t endianswap_16_signed(int16_t val);
uint16_t endianswap_16_unsigned(uint16_t val);
uint32_t endianswap_32_unsigned(uint32_t val);

// This disables automatic network connecting and a few other things for more dev control
SYSTEM_MODE(MANUAL);

UDP Udp;
TCPClient client;
int led = D7;

// This address needs to reflect the IP of the TCP server
IPAddress serverIP(192,168,68,123);
// This port needs to reflect the port of the TCP server
int TCPort = 7060;

void setup() {
  WiFi.setListenTimeout(30);
  // Connect to the WiFi network
  WiFi.connect();

  // Set LED for output
  pinMode(led, OUTPUT);

  // Begin Serial and setup SPI1
  Serial.begin(9600);
  SPI1.begin(SPI_MODE_MASTER);

  // Wait until WiFi is connected
  while(WiFi.connecting()){}
  // Connect to the TCP server
  TCP_Connect();
}

void loop() {
  // For keeping track of sample times
  static uint32_t oldTime, newTime = 0;

  // Ensure that the TCP server is connected
  if (!client.connected()) {
    // If the TCP server isn't connected, disable LED and try to connect
    digitalWrite(led, LOW);
    TCP_Connect();
  }
```

```
  else {
    // Read buffer for TCP
    char buf[50];
    // Read from the TCP handler
    int bytesRead = client.read((uint8_t*)buf, sizeof(buf));
    // If we get bytes from the TCP server then handle the packet
    if (bytesRead > 0) {
      packetHandler(buf);
      // Clear buffer
      for (int i = 0; i < bytesRead; i++) {
        buf[i] = '\0';
      }
    }
  }

  // Get new time for sample check
  newTime = Time.now();
  // If we're at or over our sample rate then take a sample
  if ((newTime - oldTime) >= SAMPLE_RATE) {
    oldTime = newTime;
    takeSample();
  }
}


/**
 * Handle the packet provided
 *
 * @param char* buf whose value is the bytes from the TCP read
 * @return None
 */
void packetHandler(char* buf) {
  // Get packet header details
  uint8_t pID = buf[0];
  uint8_t pVersion = buf[1];
  subtype_t pSubtype = (subtype_t)buf[2];

  // Verify that packet is correct version and ID
  if (pID != PACKET_ID || pVersion != CURRENT_VERSION) {
    #ifdef SERIAL_DEBUG
      Serial.println("Error: Incorrect packet ID or version!");
    #endif

    return;
  }
  // Packet is valid, process based on subtype
  else {
    switch (pSubtype) {
      case OPEN_RESPONSE: {   // Open response packet - time
        #ifdef SERIAL_DEBUG
          Serial.println("Open Response receieved.");
        #endif

        // Get time from Open Response packet and set system time
        unsigned long int t;
        t = (buf[3] << 24) | (buf[4] << 16) | (buf[5] << 8) | (buf[6]);
        Time.setTime(t);
      }
      break;

      case STATUS: {
        #ifdef SERIAL_DEBUG
          Serial.println("Status receieved. Building Status Response.");
        #endif

        // Prepare to send Status Response packet
        struct pStatusResponse toSend;
        // SPI transmit/receive buffers
```

```
        char rx[2] = {0,0};
        char tx[2] = {0,0};
        int16_t sStore;
        uint16_t uStore;
        // Read registers and pack
        tx[1] = 74; // Pmean register at 4AH
        SPI1.transfer(tx, rx, sizeof(rx), NULL);
        sStore = (rx[0] << 8) | rx[1];
        toSend.pmean = endianswap_16_signed(sStore);

        #ifdef SERIAL_DEBUG
          Serial.print("Pmean SPI read: ");
          Serial.print(sStore);
        #endif

        rx[0], rx[1] = 0;
        tx[1] = 73; // Urms register at 49H
        SPI1.transfer(tx, rx, sizeof(rx), NULL);
        uStore = (rx[0] << 8) | rx[1];
        toSend.urms = endianswap_16_unsigned(uStore);

        #ifdef SERIAL_DEBUG
          Serial.print("Urms SPI read: ");
          Serial.print(uStore);
        #endif

        rx[0], rx[1] = 0;
        tx[1] = 77; // Powerf register at 4DH
        SPI1.transfer(tx, rx, sizeof(rx), NULL);
        sStore = (rx[0] << 8) | rx[1];
        toSend.powerf = endianswap_16_signed(sStore);

        #ifdef SERIAL_DEBUG
          Serial.print("PowerF SPI read: ");
          Serial.print(sStore);
        #endif

        rx[0], rx[1] = 0;
        tx[1] = 76; // Freq register at 4CH
        SPI1.transfer(tx, rx, sizeof(rx), NULL);
        uStore = (rx[0] << 8) | rx[1];
        toSend.freq = endianswap_16_unsigned(uStore);

        #ifdef SERIAL_DEBUG
          Serial.print("Freq SPI read: ");
          Serial.print(uStore);
        #endif

        // Send fully built Status Response packet
        sendPacket(&toSend, SIZE_STATUS_RESPONSE);
      }
      break;

      default: {
        #ifdef SERIAL_DEBUG
          Serial.println("Unknown Packet");
        #endif
      }
      break;
    }
  }
}


/**
 * Send the provided packet
 *
 * @param void* toSend whose contents are sent out over TCP
```

```
 * @param uint16_t size whose value reflects the size of the provided packet
 * @return None
 */
void sendPacket(void* toSend, uint16_t size) {
  // Make sure we didn't lose TCP connection
  if (client.connected()) {
    // Send packet to TCP server
    client.write((uint8_t*)toSend, size);

    #ifdef SERIAL_DEBUG
      Serial.println("Sent packet!");

      int err = client.getWriteError();
      if (err != 0) {
        Serial.print("Error: ");
        Serial.print(err);
      }
    #endif
  }
  else {
    #ifdef SERIAL_DEBUG
      Serial.println("Failed to send. No connection to TCP server.");
    #endif
  }
}


/**
 * Reads Pmean over SPI and stores the sample with the time in the EEPROM.
 * If a dataDump packet can be filled, then it is filled and sent
 *
 * @param None
 * @return None
 */
void takeSample(void) {
  static int eeAddress = 0;
  static uint16_t sampleCount = 0;

  // SPI transmit/receive buffers
  char rx[2] = {0,0};
  char tx[2] = {0,0};
  int16_t sStore;

  #ifdef SERIAL_DEBUG
    Serial.print("Taking sample: ");
    Serial.println(sampleCount);
  #endif

  tx[1] = 74; // Pmean register at 4AH
  SPI1.transfer(tx, rx, sizeof(rx), NULL);
  sStore = (rx[0] << 8) | rx[1];

  #ifdef SERIAL_DEBUG
    Serial.print("Pmean SPI read: ");
    Serial.println(sStore);
  #endif

  // sStore = 0x0AAA;

  // Get dataChunk to store sample
  struct dataChunk data;
  data.pmean = endianswap_16_signed(sStore);
  data.t = endianswap_32_unsigned(Time.now());

  // Store dataChunk in the EEPROM at the specified address
  EEPROM.put(eeAddress, data);
  // Increment EEPROM address to prevent data overwriting
  eeAddress += sizeof(struct dataChunk);
```

```
  sampleCount++;

  // Check if we have enough samples to send to the server
  if (sampleCount == 20) {
    #ifdef SERIAL_DEBUG
      Serial.println("Building dataDump");
    #endif

    // Reset statics
    eeAddress = 0;
    sampleCount = 0;

    // Get dataDump for storing samples from EEPROM
    struct pDataDump toSend;
    for (int j = 0; j < 20; j++) {
      // Get all the samples and store in packet
      EEPROM.get(eeAddress, toSend.chunk[j]);
      eeAddress += sizeof(struct dataChunk);
    }
    // Append an empty sample for software parsing
    struct dataChunk empty;
    toSend.chunk[21] = empty;
    // Send filled dataDump packet
    sendPacket(&toSend, SIZE_DATA_DUMP);
    // Clear EEPROM and reset the address one more time for future collection
    EEPROM.clear();
    eeAddress = 0;
  }
}


/**
 * Attempt to connect to the TCP server and send an Open Connection packet
 *
 * @param None
 * @return None
 */
void TCP_Connect(void) {
  if (client.connect(serverIP, TCPort)) {
    #ifdef SERIAL_DEBUG
      Serial.println("Connected to TCP server.");
    #endif

    // Enable LED to indicate TCP connection
    digitalWrite(led, HIGH);

    // Get device MAC address
    byte mac[6];
    WiFi.macAddress(mac);

    // Build Open Connection packet
    struct pOpenConnection toSend;
    memcpy(&toSend.mac, &mac, 6);
    // Send filled Open Connection packet
    sendPacket(&toSend, SIZE_OPEN_CONNECTION);
  }
  else {
    #ifdef SERIAL_DEBUG
      Serial.println("TCP connection failed.");
    #endif
  }
}


/**
 * Swap endian of a 16 bit signed integer
 *
 * @param int16_t val whose value will have its endian swapped
```

```
 * @return int16_t swapped whose value reflects the endian swapped result
 */
int16_t endianswap_16_signed(int16_t val) {
  int16_t swapped = ((val >> 8) & 0x00ff) | ((val & 0x00ff) << 8);
  return swapped;
}


/**
 * Swap endian of a 16 bit unsigned integer
 *
 * @param uint16_t val whose value will have its endian swapped
 * @return uint16_t swapped whose value reflects the endian swapped result
 */
uint16_t endianswap_16_unsigned(uint16_t val) {
  uint16_t swapped = ((val >> 8) & 0x00ff) | ((val & 0x00ff) << 8);
  return swapped;
}


/**
 * Swap endian of a 32 bit unsigned integer
 *
 * @param uint32_t val whose value will have its endian swapped
 * @return uint32_t swapped whose value reflects the endian swapped result
 */
uint32_t endianswap_32_unsigned(uint32_t val) {
  uint32_t b0, b1, b2, b3;
  b0 = (val & 0x000000ff) << 24u;
  b1 = (val & 0x0000ff00) << 8u;
  b2 = (val & 0x00ff0000) >> 8u;
  b3 = (val & 0xff000000) >> 24u;

  return (b0 | b1 | b2 | b3);
}
```

B.1.2 packetTypes.h

```
#ifndef PHOTON_PACKETTYPES_H
#define PHOTON_PACKETTYPES_H

// Packet sizes for sending routine
#define SIZE_OPEN_CONNECTION 9
#define SIZE_STATUS_RESPONSE 11
#define SIZE_DATA_DUMP 129

#define CURRENT_VERSION 0
#define PACKET_ID 0xBA

typedef enum {
  OPEN_CONNECTION,
  OPEN_RESPONSE,
  STATUS,
  STATUS_RESPONSE,
  DATA_DUMP,
  UNKNOWN
} subtype_t;

struct __attribute__((__packed__)) pHeader {
  uint8_t ID      = PACKET_ID;
  uint8_t version   = CURRENT_VERSION;
};

struct __attribute__((__packed__)) pOpenConnection {
  struct pHeader head;
  subtype_t subtype = OPEN_CONNECTION;
```

```
  uint8_t mac[6];
};

struct __attribute__((__packed__)) pStatusResponse {
  struct pHeader head;
  subtype_t subtype = STATUS_RESPONSE;
  int16_t pmean;
  uint16_t urms;
  int16_t powerf;
  uint16_t freq;
};

struct __attribute__((__packed__)) dataChunk {
  int16_t pmean = 0;
  uint32_t t = 0;
};

struct __attribute__((__packed__)) pDataDump {
  struct pHeader head;
  subtype_t subtype = DATA_DUMP;
  struct dataChunk chunk[21];
};

#endif //PHOTON_PACKETTYPES_H
```

C Software

C.1 client (Front-End)

C.1.1 /client/src/App.js

```
import React from 'react';
import './App.css';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import Navigation from './components/Navigation';
import Status from './components/Status';
import History from './components/History'
import Settings from './components/Settings';
import { withRouter } from "react-router";


const NavigationWithRouter = withRouter(Navigation);

// Main call for website
class App extends React.Component {
 componentDidMount() {
  document.title = "Power Monitor"
 }

 render() {
  return (
    <div className="App">
     <BrowserRouter>
      <div>
       <NavigationWithRouter />
       <Switch>
        <Route path="/" component={Status} exact/>
        <Route path="/status" component={Status}/>
        <Route path="/history" component={History}/>
        <Route path="/settings" component={Settings}/>
        <Route component={Status}/>
       </Switch>
      </div>
     </BrowserRouter>
    </div>
  );
 }
}

export default App;
```

C.1.2 /client/src/config.json

```
{
 "API": {
  "HOST": "192.168.1.218",
  "PORT": "9000"
 }
}
```

C.1.3 /client/src/components/Home.js

```
import React from 'react';
import logo from "../ScaryJD.png";
```

```
import Button from 'react-bootstrap/Button'

//unused module, was used for testing
class home extends React.Component {
  constructor(props) {
    super(props);
    this.state = { apiResponse : [] };
    this.sayHello = this.callAPI.bind(this);
  }

  callAPI(header) {
    fetch("http://192.168.1.218:9000/testTCP", {
      headers: {
        data: header
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ apiResponse: res }));
  }

  componentDidUpdate() {
    console.log(this.state.apiResponse);
  }


  render() {
    return (
      <div>
        <h1>Home</h1>
        <hr/>
        <p>Home page body content</p>
        <img src={logo} className="App-logo" alt="logo"/>
        <hr className='mt-5'/>
          <Button
            onClick={() => this.callAPI('0')}>Send '0'</Button>
          {' '}
        <Button
          onClick={() => this.callAPI('2')}>Send '2'</Button>

          <h3 className="App-intro">API Result: {JSON.stringify(this.state.apiResponse)}</h3
    ↪ >

      </div>
    );
  };
};

export default home;
```

C.1.4 /client/src/components/Navigation.js

```
import React from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
import Navbar from 'react-bootstrap/Navbar'
import Nav from 'react-bootstrap/Nav'
import logo from "../lightning.svg";

// Navigation bar
const Navigation = props => {
  const { location } = props;
  return (
    <Navbar bg="dark" variant="dark" expand="lg">
      {/* Logo Setup */}
      <Navbar.Brand>
        <img
          alt=""
```

```
          src={logo}
          width="30"
          height="30"
          className="d-inline-block align-top"
        />{' '}
        Power Monitor
      </Navbar.Brand>

      {/* Handles routing to appropriate pages */}
      <Navbar.Toggle aria-controls="basic-navbar-nav"/>
      <Navbar.Collapse id="basic-navbar-nav">
        <Nav className="ml-auto" activeKey={location.pathname}>
          <Nav.Link href="/">Home</Nav.Link>
          {/*<Nav.Link href="/status">Status</Nav.Link>*/}
          <Nav.Link href="/history">History</Nav.Link>
          <Nav.Link href="/settings">Settings</Nav.Link>
        </Nav>
      </Navbar.Collapse>
    </Navbar>
  );
};

export default Navigation;
```

C.1.5 /client/src/components/Status.js

```
import React from 'react';
import Button from 'react-bootstrap/Button'
import Container from 'react-bootstrap/Container'
import Row from 'react-bootstrap/Row'
import Col from 'react-bootstrap/Col'
import ListGroup from 'react-bootstrap/ListGroup'
import Popover from 'react-bootstrap/Popover'
import OverlayTrigger from 'react-bootstrap/OverlayTrigger'
import Table from 'react-bootstrap/Table'
import config from '../config.json'

// Main component for status/home page
class Status extends React.Component {
  constructor(props) {
    super(props);
    this.state = { apiResponse : [] };
  }

  // Call to backend to get current devices
  getDeviceList() {
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/getDevices")
      .then(res => res.json())
      .then(res => this.setState({ apiResponse: res }))
  }

  // On page load, get the current devices
  componentDidMount() {
    this.getDeviceList();
  }

  // Create page with ListGroup for devices
  render() {
    return (
      <div>
        <h1>Device Status</h1>
        <hr/>
        <Container>
          <ListGroup className="list-group">
            {this.state.apiResponse.map(device => (
              // Pass devices properties to each individual entry
```

```
              <DeviceEntry
                key={device.DeviceID}
                device={device}
              />))
            }
          </ListGroup>
        </Container>
        <hr/>
      </div>
    );
  }
}

// Generate UI component for a single device in the list
class DeviceEntry extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      deviceID : this.props.device.DeviceID,
      deviceChecked : false,
      available : false,
      data : [],
    }
  }

  // On page load, check to see if device is connected
  componentDidMount() {
    this.CheckAvailable();
  }

  // Cancel auto-refresh when page closes
  componentWillUnmount() {
    clearInterval(this.interval);
  }

  // If state becomes available, ping device and start auto-refresh of data
  componentDidUpdate(prevProps, prevState, snapshot) {
    if (prevState.available !== this.state.available) {
      if (this.state.available) {
        this.PingDevice();
        this.interval = setInterval(() => (this.CheckAvailable()), 3000);
        this.interval = setInterval(() => (this.PingDevice()), 3000);
      } else {
        clearInterval(this.interval);
      }
    }
  }

  // Back-end call to see if device is online
  CheckAvailable() {
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/TCPServer", {
      headers: {
        'action': 'checkavailable',
        'device': this.state.deviceID
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ available: res }))
      .then(() => this.setState({deviceChecked : true}));
  }

  // Backend call to get detailed data from device
  PingDevice() {
    if (this.state.available) {
      fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/TCPServer", {
        headers: {
          'action': 'pingdevice',
          'device': this.state.deviceID
```

```
        }
      })
        .then(res => res.json())
        .then(res => this.setState({ data: res }))
        .then(() => this.setState({deviceChecked : true}));
    }
  }

  render() {
    return (
      <ListGroup.Item key={this.state.DeviceID}>
        <Row>
          <Col className="text-center">
            <h5 className="align-bottom">{this.props.device.FriendlyName}</h5>
          </Col>
          <Col className="text-center">
            <OverlayTrigger
              key={this.state.DeviceID}
              delay={{ show: 250, hide: 400 }}
              placement='right'
              overlay={
                <Popover id={`popover-positioned-right`}>
                  <Popover.Title as="h3">Device Details</Popover.Title>
                  <Popover.Content>
                    <Table borderless xs hover>
                      <tbody>
                      <tr>
                        <th>Mean Power:</th>
                        <th>{this.state.data.PMean} W</th>
                      </tr>

                      <tr>
                        <th>Volts:</th>
                        <th>{this.state.data.Urms} Vrms</th>
                      </tr>

                      <tr>
                        <th><>Frequency:</></th>
                        <th>{this.state.data.Frequency} Hz</th>
                      </tr>

                      <tr>
                        <th><>Power Factor:</></th>
                        <th>{this.state.data.PowerFactor}</th>
                      </tr>
                      </tbody>
                    </Table>
                    This data updates every 3 seconds.
                  </Popover.Content>
                </Popover>
              }
            >
              <Button
                variant={this.state.available ? 'outline-primary' : 'outline-danger'}
                disabled={!this.state.available}
              >
                {(this.state.data.PMean === undefined) ? (this.state.available) ?
                  'Loading...' : 'N/A' : parseInt(this.state.data.PMean) + ' W'}
              </Button>
            </OverlayTrigger>
          </Col>
          <Col className="text-center">
            <Button
              variant={(this.state.deviceChecked) ? (this.state.available) ?
                'success' : 'outline-danger' : 'warning'}>
              {(this.state.deviceChecked) ? (this.state.available) ?
                'Online' : 'Offline' : 'Loading'}
            </Button>
```

```
          </Col>
        </Row>
      </ListGroup.Item>
    );
  }
}

export default Status;
```

C.1.6 /client/src/components/History.js

```
import React from 'react';
import { Line } from 'react-chartjs-2';
import Container from 'react-bootstrap/Container'
import Dropdown from 'react-bootstrap/Dropdown'
import DatePicker from "react-datepicker"
import Button from 'react-bootstrap/Button'
import ButtonGroup from 'react-bootstrap/ButtonGroup'
import ButtonToolbar from 'react-bootstrap/ButtonToolbar'
import InputGroup from 'react-bootstrap/InputGroup'
import config from '../config.json'

// Import distributed CSS
import "react-datepicker/dist/react-datepicker.css"
// Custom CSS Import
import './react-datepicker.css'

// Main function to handle history searching
class history extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      graphData : [],
      devices : [],
      friendlyName: [{"FriendlyName": "undefined"}],
      startDate: new Date(),
      endDate: new Date(),
      minTime: new Date()};

    // Configure datetime extreme values
    this.state.startDate.setDate(this.state.startDate.getDate() - 1);
    this.state.minTime.setHours(0);
    this.state.minTime.setMinutes(0);
    this.state.minTime.setTime(0);

    this.handleClick = this.handleClick.bind(this);
  }

  // On page load, get device list
  componentDidMount() {
    this.callDevices();
  }

  // Back-end call to get friendly name of device for the graph
  getFriendlyName(device) {
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/getFriendlyName", {
      headers: {
        'device': device
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ friendlyName: res }));
  }

  // Back-end call to get actual graph data
  getGraphData(devices) {
```

```
    console.log(this.state.startDate.toJSON());
    console.log(this.state.startDate.toLocaleDateString());
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/testAPI", {
      headers: {
        'startdate': this.state.startDate.toISOString(),
        'enddate': this.state.endDate.toISOString(),
        'devices': devices
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ graphData: res }));
  }

  // Back-end call to get list of devices
  callDevices() {
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/getDevices")
      .then(res => res.json())
      .then(res => this.setState({ devices: res }));
  }

  // Handle press of dropdown item
  ToggleSearch(DeviceID) {
    let newstate = this.state;

    // determine if a device is selected
    for (const device of newstate.devices) {
      device.selected = device.DeviceID === DeviceID;
    }

    this.getFriendlyName(DeviceID);

    let activeDevices = [];
    // Push selected device to activeDevices array
    for (const device of this.state.devices) {
      if (device.selected === true) {
        activeDevices.push(device.DeviceID);
      }
    }

    // Get graph data for selected devices
    this.getGraphData(activeDevices);

    this.setState({newstate});
  }

  // Handle Search Button Press
  handleClick(e) {
    e.preventDefault();
    let testDevices = [];

    for (const device of this.state.devices) {
      if (device.selected === true) {
        testDevices.push(device.DeviceID);
      }
    }

    this.getGraphData(testDevices);
  }

  handleStartChange = date => {
    this.setState({
      startDate: date
    });
  };

  handleEndChange = date => {
    this.setState({
      endDate: date
```

```
    });
  };

  render() {
    // Chart configuration
    let response = this.state.graphData;

    let labels = response.map(function(e) {
      return e.MeasurementTime;
    });

    let data = response.map(function(e) {
      return e.WattValue;
    });

    let chart = {
      labels: labels,
      datasets: [
        {
          fill: true,
          lineTension: 0.1,
          backgroundColor: 'rgba(0,133,255,0.56)',
          borderColor: '#007bff',
          borderCapStyle: 'butt',
          borderDash: [],
          borderDashOffset: 0.0,
          borderJoinStyle: 'miter',
          pointBorderColor: '#007bff',
          pointBackgroundColor: '#fff',
          pointBorderWidth: 1,
          pointHoverRadius: 5,
          pointHoverBackgroundColor: '#007bff',
          pointHoverBorderColor: 'rgba(220,220,220,1)',
          pointHoverBorderWidth: 2,
          pointRadius: 1,
          pointHitRadius: 10,
          data: data
        }
      ],
    };

    const options = {
      legend: {
        display: false,
        position: 'bottom'
      },
      scales: {
        xAxes: [{
          // stacked: true,
          ticks: {
            min: 0,
            maxTicksLimit: 20
          },
          type: 'time',
          time: {
            // parser: 'MM/DD/YYYY HH:mm',
            tooltipFormat: 'll hh:mm A',
            unit: 'hour',
            unitStepSize: 0.5,
            displayFormats: {
              'day': 'MM/DD/YYYY',
              'hour': 'hh:mm A'
            }
          },
        }],
        yAxes: [{
          ticks: {
            beginAtZero: true,
```

```
            callback: function(value) {
              return value + " W";
            }
          }
        }],

      }
    };

    // If a device has been searched, enable and assign a value to the label
    if (this.state.friendlyName[0].FriendlyName !== 'undefined') {
      chart.datasets[0].label = this.state.friendlyName[0].FriendlyName;
      options.legend.display = true;
    }

    return (
      <div>
        <h1>Usage History</h1>
        <hr/>
        <ButtonToolbar style={{ display: 'flex', alignItems: 'center', justifyContent: '
    ↪ center' }}>
          <CalendarSearch startDate = { this.state.startDate }
                  endDate = { this.state.endDate }
                  minTime = { this.state.minTime }
                  handleStartChange = {e => this.handleStartChange(e) }
                  handleEndChange = {e => this.handleEndChange(e) }
          />

          <Dropdown className='ml-3' as={ButtonGroup}>
            <Button variant="light" onClick={this.handleClick}>Search</Button>
            <Dropdown.Toggle split variant="light" id="dropdown-split-basic" />
            <Dropdown.Menu>
              {this.state.devices.map(device =>
                <Dropdown.Item
                  active = { device.selected }
                  key = { device.DeviceID }
                  onClick = { () => { this.ToggleSearch(device.DeviceID) } }
                >
                  { device.FriendlyName }
                </Dropdown.Item>
              )}
            </Dropdown.Menu>
          </Dropdown>
        </ButtonToolbar>
        <Container>
          <Line data={chart} options={options}/>
        </Container>
        <hr/>
      </div>
    );
  };
}

// Handle calendar dropdowns for date range searches
class CalendarSearch extends React.Component {
  render() {
    return (
      <>
        <ButtonGroup className="mr-lg-3 ">

          <InputGroup.Prepend>
            <InputGroup.Text id="btnGroupAddon">Start: </InputGroup.Text>
          </InputGroup.Prepend>
          <DatePicker
            selected={this.props.startDate}
            onChange={date => this.props.handleStartChange(date)}
            selectsStart
            startDate={this.props.startDate}
```

```
            endDate={this.props.endDate}
            maxDate={this.props.endDate}
            minTime={this.props.minTime}
            maxTime={this.props.endDate}
            dateFormat="MMMM d, yyyy h:mm aa"
          />
        </ButtonGroup>
        <ButtonGroup className="my-3 ">
          <InputGroup.Prepend>
            <InputGroup.Text id="btnGroupAddon">End:</InputGroup.Text>
          </InputGroup.Prepend>
          <DatePicker
            selected={this.props.endDate}
            onChange={date => this.props.handleEndChange(date)}
            selectsEnd
            startDate={this.props.startDate}
            endDate={this.props.endDate}
            minDate={this.props.startDate}
            maxDate={new Date()}
            maxTime={new Date()}
            minTime={this.props.startDate}
            dateFormat="MMMM d, yyyy h:mm aa"
          />
        </ButtonGroup>
      </>
    );
  }
}

export default history;
```

C.1.7 /client/src/components/Settings.js

```
import React from 'react';
import Button from 'react-bootstrap/Button'
import ButtonGroup from 'react-bootstrap/ButtonGroup'
import Dropdown from 'react-bootstrap/Dropdown'
import Container from 'react-bootstrap/Container'
import Row from 'react-bootstrap/Row'
import Col from 'react-bootstrap/Col'
import Form from 'react-bootstrap/Form'
import ListGroup from 'react-bootstrap/ListGroup'
import Modal from 'react-bootstrap/Modal'
import config from '../config.json'

// Main Module definition
class Settings extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      dbDeviceState : [],
      changes: false,
      isLoading: false};
    this.UpdateTable = this.UpdateTable.bind(this);
    this.modifyTable = this.modifyTable.bind(this);
  }

  // When page loads, request list of all devices
  componentDidMount() {
    this.getDevices();
  }

  // On page update, check of the state of the db has changes re: active devices
  componentDidUpdate(prevProps, prevState, snapshot) {
    if (this.state.dbDeviceState !== prevState.dbDeviceState) {
      this.createNewDeviceState();
```

```
    }
  }

  // Make HTTP request to back-end db, update state
  getDevices() {
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/getDevices", {
      headers: {
        'active': 'all',
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ dbDeviceState: res }));
  }

  // Make change to active state of devices when save changes button is pressed
  modifyTable(e) {
    // Disable save changes button
    this.setState( {changes: false });

    // Logic to search for changed devices and save the changes
    for (const device of this.state.dbDeviceState) {
      if (device.newActive !== device.selected) {
        fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/manageDevices", {
          headers: {
            'action': 'changeactive',
            'deviceid': device.DeviceID,
            'active': device.newActive
          }
        })
          .then(res => res.json())
          .then(res => this.setState({result: res}))
          .then((res) => {
            console.log(res);
            // After device successfully toggles, update the table
            this.UpdateTable(e);
          });
      }
    }
  }

  // Creates some additional fields from db response containing theme variants and modified
    ↪ tag
  createNewDeviceState() {
    // Make copy of state
    let newstate = this.state;

    // Setup every device from the database response
    for (const device of newstate.dbDeviceState) {
      device.active = Boolean(device.active);
      if (!device.hasOwnProperty('modified') || !device.modified) {
        device.modified = false;
        device.newActive = device.active;
        if (device.active) {
          device.variant = 'success'
        } else {
          device.variant = 'secondary'
        }
      }
    }

    // Update state of instance
    this.setState({newstate});
  }

  // Check if save changes button should be enabled
  determineChanges() {
    let newstate = this.state;
    newstate.changes = false;
```

```
    for (const device of newstate.dbDeviceState) {
      if (device.selected !== device.newActive) {
        newstate.changes = true;
      }
    }

    // Rerun state setup
    this.createNewDeviceState();
    this.setState({newstate});
  }

  // Handle toggling devices theme variant and new state
  handleClick(deviceID) {
    // Find device in table
    for (const device of this.state.dbDeviceState) {
      // Once it's found, toggle the newActive and modified fields
      if (device.DeviceID === deviceID) {
        device.newActive = !device.newActive;
        device.modified = !device.modified;

        // Update variants
        if (device.newActive !== device.selected) {
          device.modified = true;
          device.variant = 'warning'
        } else {
          device.modified = false;
        }
      }
    }

    // Handle additional theme and state changes
    this.determineChanges();
  }

  // Reload the table from scratch from the database
  UpdateTable(e) {
    e.preventDefault();
    this.getDevices();
  }

  render() {
    return (
      <div>
        <h1>System Settings</h1>
        <hr/>
        <h3>Manage Current Devices</h3>
        <Container>
          <ListGroup className="list-group">
            <TableOutput
              UpdateTable={this.UpdateTable}
              data={this.state.dbDeviceState}
              handleClick={deviceID => this.handleClick(deviceID)}
            />
            <ListGroup.Item>
              <Col className="text-center">
                <Button
                  block
                  onClick={this.modifyTable}
                  disabled={!this.state.changes}
                  variant='primary'
                >
                  Save Changes
                </Button>
              </Col>
            </ListGroup.Item>
          </ListGroup>
          <h3 className="mt-5">New Device</h3>
```

```
          <ListGroup className="list-group">
            <ListGroup.Item >
              <AddDevice UpdateTable={this.UpdateTable}/>
            </ListGroup.Item>
          </ListGroup>
        </Container>
      </div>
    );
  }
}

// Output Table of database results, combined with user modification components
class TableOutput extends React.Component {
  render() {
    return (
      this.props.data.map(listitem => (
        <ListGroup.Item key={listitem.DeviceID}>
          <Row>
            <Col className="text-center">
              <h5>{listitem.FriendlyName}</h5>
            </Col>
            <Col className="text-center">
              <h5>{listitem.DeviceID}</h5>
            </Col>
            <Col className="text-center">
              <Dropdown as={ButtonGroup}>
                <Button
                  variant={listitem.variant}
                  onClick = { () => { this.props.handleClick(listitem.DeviceID) } }
                >
                  {/* Logic to determine button label */}
                  {(listitem.modified) ? (listitem.newActive) ? 'Enable' : 'Disable' :
                    (listitem.active) ? 'Enabled' : 'Disabled'}
                </Button>

                <Dropdown.Toggle
                  split
                  variant={listitem.variant}
                  id="dropdown-split-basic"
                />

                <Dropdown.Menu>
                  {/* Pass appropriate methods and names to delete device */}
                  <DeleteDevice
                    UpdateTable={e => this.props.UpdateTable(e)}
                    FriendlyName={listitem.FriendlyName}
                    DeviceID={listitem.DeviceID}
                  />
                </Dropdown.Menu>
              </Dropdown>
            </Col>
          </Row>
        </ListGroup.Item>
      ))
    );
  }
}

// Handle delete buttons, we want a delete device confirmation
class DeleteDevice extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      show : false,
      setShow : false,
      isLoading : false,
      result : []};
```

```
    this.Delete = this.Delete.bind(this);
  }

  // Handle state for popup of Modal
  handleClose = () => this.setState({show: false});
  handleShow = () => this.setState({show: true});

  // Back-end call to remove device from device table
  Delete(e, device) {
    e.preventDefault();

    // Disable button after first click
    this.setState( {isLoading: true });

    // Make Request to backend
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/manageDevices", {
      headers: {
        'action': 'delete',
        'device': device
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ result: res }))
      .then(() => {
        // On successful response, close popup and update parent's table
        this.handleClose();
        this.props.UpdateTable(e);
      });
  }

  render() {
    return (
      <>
        <Dropdown.Item onClick={this.handleShow}>
          Delete
        </Dropdown.Item>

        <Modal
          show={this.state.show}
          onHide={this.handleClose}
          aria-labelledby="contained-modal-title-vcenter"
          centered
        >
          <Modal.Header closeButton>
            <Modal.Title>Confirm Delete</Modal.Title>
          </Modal.Header>
          <Modal.Body>
            Are you sure you want to delete <b>{this.props.FriendlyName}</b>?
            <br/>
            This action will <i>not</i> remove data associated with device <b>{this.props.
  ↪ DeviceID}</b>.
          </Modal.Body>
          <Modal.Footer>
            <Button
              variant="secondary"
              onClick={this.handleClose}
            >
              Cancel
            </Button>
            <Button
              variant="danger"
              disabled={this.state.isLoading}
              onClick={e => this.Delete(e, this.props.DeviceID)}
            >
              {this.state.isLoading ? 'Deleting...' : 'Delete'}
            </Button>
          </Modal.Footer>
        </Modal>
```

```
      </>
    )
  }
}

// Component to handle adding a new device
class AddDevice extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      validated : false,
      deviceName : '',
      deviceID : '',
      result : []
    };
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  // Backend call to add device to database
  addDevice(e, deviceID, deviceName) {
    this.setState( {isLoading: true });
    fetch("http://" + config.API.HOST + ":" + config.API.PORT + "/manageDevices", {
      headers: {
        'action': 'insert',
        'deviceid': deviceID,
        'devicename': deviceName,
      }
    })
      .then(res => res.json())
      .then(res => this.setState({ result: res }))
      .then(() => {
        // Update parent's table after response
        this.props.UpdateTable(e);
      });
  }

  // Handle submission button press
  handleSubmit(e) {
    e.preventDefault();
    e.stopPropagation();
    this.setValidated();

    // Make sure fields aren't empty, then make request and reset form
    if(this.state.deviceID !== '' && this.state.deviceName !== '') {
      this.addDevice(e, this.state.deviceID, this.state.deviceName);
      document.getElementById("newDevice").reset();
      this.setState({validated: false})
    }

  };

  // Keep temp variables updated with text box values
  onChange = (e) => {
    this.setState({ [e.target.name]: e.target.value });
  };

  // Set form validation
  setValidated = () => this.setState({validated: true});

  render() {
    return (
      <Form noValidate id="newDevice" validated={this.state.validated} onSubmit={this.
    ↪ handleSubmit}>
        <Form.Row>
          <Form.Group as={Col} md="4" controlId="validationCustom01">
            {/*<Form.Label>Device Name</Form.Label>*/}
            <Form.Control
              required
```

```
              size="lg"
              type="text"
              name="deviceName"
              onChange={e => this.onChange(e)}
              placeholder="Device Name"
            />
            <Form.Control.Feedback type="valid">Looks good!</Form.Control.Feedback>
            <Form.Control.Feedback type="invalid">
              Please provide a valid device name.
            </Form.Control.Feedback>
          </Form.Group>
          <Form.Group as={Col} md="4" controlId="validationCustom02">
            {/*<Form.Label>Device ID</Form.Label>*/}
            <Form.Control
              required
              size="lg"
              type="text"
              name="deviceID"
              onChange={e => this.onChange(e)}
              placeholder="Device ID"
            />
            <Form.Control.Feedback type="valid">Looks good!</Form.Control.Feedback>
            <Form.Control.Feedback type="invalid">
              Please provide a valid device ID.
            </Form.Control.Feedback>
          </Form.Group>
          <Form.Group as={Col} md="4">
            <Button size="lg" type="submit">Add Device</Button>
          </Form.Group>
        </Form.Row>
      </Form>
    );
  }
}

export default Settings;
```

C.2 api (Back-End)

C.2.1 /api/app.js

```
var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var cors = require("cors");

var indexRouter = require('./routes/index');
var getHistory = require("./routes/getHistory");
var getDevices = require("./routes/getDevices");
var getFriendlyName = require("./routes/getFriendlyName");
var manageDevices = require("./routes/manageDevices");
var TCPServer = require("./routes/TCPServer");


var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(cors());
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use("/testAPI", getHistory);
app.use("/getDevices", getDevices);
app.use("/manageDevices", manageDevices);
app.use("/getFriendlyName", getFriendlyName);
app.use("/TCPServer", TCPServer);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
 next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
 // set locals, only providing error in development
 res.locals.message = err.message;
 res.locals.error = req.app.get('env') === 'development' ? err : {};

 // render the error page
 res.status(err.status || 500);
 res.render('error');
});

module.exports = app;
```

C.2.2 /api/config.json

```
{
  "DATABASE": {
    "host": "localhost",
    "user": "root",
    "password": "lolo"
```

```
  },
  "TCP_SERVER": {
    "HOST": "192.168.1.218",
    "PORT": 7060
  }
}
```

C.2.3 /api/routes/database.js

```
let mysql = require('mysql');
let CONFIG = require('./../config');

// code from stack overflow: oportocala
let pool = mysql.createPool(CONFIG.DATABASE);

exports.conn = {
  query: function () {
    let queryArgs = Array.prototype.slice.call(arguments),
      events = [],
      eventNameIndex = {};

    pool.getConnection(function (err, conn) {
      if (err) {
        if (eventNameIndex.error) {
          eventNameIndex.error();
        }
      }
      if (conn) {
        let q = conn.query.apply(conn, queryArgs);
        q.on('end', function () {
          conn.release();
        });

        events.forEach(function (args) {
          q.on.apply(q, args);
        });
      }
    });

    return {
      on: function (eventName, callback) {
        events.push(Array.prototype.slice.call(arguments));
        eventNameIndex[eventName] = callback;
        return this;
      }
    };
  }
};
```

C.2.4 /api/routes/getDevices.js

```
let express = require('express');
let router = express.Router();
let mysql = require('mysql');

let db = require('./database');

// Returns a JSON object of devices requested
router.get('/', function (req, res) {
  let sql;

  // Check which devices we want, default to active only
  if (req.get('active') === 'all') {
    sql = "SELECT * FROM powermonitoring.devices ORDER BY FriendlyName"
```

```
  } else {
    sql = "SELECT * FROM powermonitoring.devices WHERE active = '1' ORDER BY FriendlyName"
  }


  db.conn.query(mysql.format(sql), function (err, result) {

    // Generate and return HTTP response
    res.send(result);
    console.log(err);
  });
});

module.exports = router;
```

C.2.5 /api/routes/getFriendlyName.js

```
let express = require('express');
let router = express.Router();
let mysql = require('mysql');
let db = require('./database');

// Returns the friendly name of a given device
router.get('/', function (req, res) {
  let sql = "SELECT FriendlyName FROM powermonitoring.devices WHERE DeviceID = ?;";
  let params = [req.get('device')];

  // Generate and run SQL query on DB
  db.conn.query(mysql.format(sql, params), function (err, result) {
    res.send(result);
  });
});

module.exports = router;
```

C.2.6 /api/routes/getHistory.js

```
let express = require('express');
let router = express.Router();
let mysql = require('mysql');

let db = require('./database');

// Main function for getting graph data
router.get('/', function (req, res) {
  // Get date range headers
  let startDateHdr = new Date(req.get('startdate'));
  let endDateHdr = new Date(req.get('enddate'));

  // Typecasting for SQL datetime format B)
  let startDate = convertUTCDateToLocalDate(startDateHdr).toISOString().substring(0, 10) + '
    ↪  '
            + convertUTCDateToLocalDate(startDateHdr).toISOString().substring(11, 19);
  let endDate = convertUTCDateToLocalDate(endDateHdr).toISOString().substring(0, 10)
            + ' ' + convertUTCDateToLocalDate(endDateHdr).toISOString().substring(11, 19);

  let params = [startDate, endDate, req.get('devices')];

  let sql = "SELECT * FROM powermonitoring.powerusage WHERE MeasurementTime" +
        " BETWEEN CAST(? AS DATETIME)" +
        " AND CAST(? AS DATETIME)" +
        " AND Device = ?;";

  // Make request to DB, using mysql.format for query generation
```

```
  db.conn.query(mysql.format(sql, params), function (err, result) {
    res.send(result);
  });

});

function convertUTCDateToLocalDate(date) {
  return new Date(date.getTime() - date.getTimezoneOffset() * 60 * 1000);
}

module.exports = router;
```

C.2.7 /api/routes/manageDevices.js

```
let express = require('express');
let router = express.Router();
let mysql = require('mysql');

let db = require('./database');

// Handles control of device table in database
router.get('/', function (req, res) {
  let action = req.get('action');
  let sql = '';
  let params = [];

  // Use header to determine what the request wants
  switch (action) {
    case 'delete':
      sql = "DELETE FROM powermonitoring.devices WHERE DeviceID = ?";
      params = req.get('device');
      break;

    case 'insert':
      sql = "INSERT INTO powermonitoring.devices VALUES ( ?, ?, 1)";
      params = [req.get('deviceid'), req.get('devicename')];
      break;

    case 'changeactive':
      sql = "UPDATE powermonitoring.devices SET active = ? WHERE DeviceID = ?";
      let active = +(req.get('active') === 'true');
      params = [ active, req.get('deviceid')];
      break;
    default:
      console.log(action);
  }

  db.conn.query(mysql.format(sql, params), function (err, result) {
    res.send([result, err]);
    console.log(err);
  });
});

module.exports = router;
```

C.2.8 /api/routes/TCPServer.js

```
let express = require('express');
let router = express.Router();
let db = require('./database');
let mysql = require('mysql');
let CONFIG = require('./../config');

// For byte packing
```

```
let Struct = require('struct');

const net = require('net');

// Start TCP Server
const server = net.createServer();

server.listen(CONFIG.TCP_SERVER.PORT, CONFIG.TCP_SERVER.HOST, () => {
  console.log('TCP Server is running on port ' + CONFIG.TCP_SERVER.PORT +'.');
});

// Array to hold connections in
let sockets = [];

// Constants for building packets
const PACKET_ID = 0xBA;
const VERSION_ID = 0;
const SUBTYPE = Object.freeze({
  "OPEN_CONNECTION": 0,
  "OPEN_RESPONSE": 1,
  "STATUS": 2,
  "STATUS_RESPONSE": 3,
  "DATA_DUMP": 4
});

// Handle HTTP requests
router.get('/', function (req, res) {
  // Two options, check available devices and ping device
  switch (req.get('action')) {
    // Checks whether a specified device is connected to the TCP server
    case 'checkavailable':
      if(sockets.length === 0) {
        // No sockets available => the requested device is not available
        res.send(false);
      } else {
        // Check if device exists in current sockets
        res.send(sockets.some(sock => sock.macAddress === req.get('device')));
      }
      break;
    // Pings a specified device to get more detailed usage information
    case 'pingdevice':
      console.log('Ping Device', req.get('device'));
      // Find the socket that holds the device we want, this request is only made when the
    ↪ front end knows
      // the device is available
      let sock = sockets[sockets.findIndex(Socket => Socket.macAddress === req.get('device')
    ↪ )];
      sendStatusResponse(sock);

      sock.setTimeout(5000, function() {
        try {
          // Let the front end know the socket timed-out
          res.send(['Socket Timeout'])
        }
        catch(ERR_HTTP_HEADERS_SENT) {
          // Don't crash trying to timeout
          // console.log('Socket Timeout Already Sent')
        }
      });

      sock.once('data', function (data) {
        // Build an object from the device ping response
        let packet = buildPacketObject(data);
        try {
          // Send HTTP with JSON response object
          res.send(packet);
        }
        catch(ERR_HTTP_HEADERS_SENT) {
```

```
          // Don't crash trying to timeout
          // console.log('Response already sent')
        }
      });
      break;
    default:
      break;
  }
});

// Handle new (unprompted) messages to server
server.on('connection', function(sock) {
  console.log('CONNECTED: ' + sock.remoteAddress + ':' + sock.remotePort);

  sock.setEncoding('hex');

  sock.on('data', function(data) {
    console.log('DATA ' + sock.macAddress + ': ' + data);
    let packet = buildPacketObject(data, sock);
    // Check if this packet is our type
    if (packet.PacketID === 0xba) {
      // Check version type
      if (packet.Version === 0) {
        switch (packet.PacketType) {
          // On new connection from device, add the identifier to the socket and push to
    ↪ sockets array
          case SUBTYPE.OPEN_CONNECTION:
            // Identify and add to sockets list
            sock.macAddress = packet.MACAddress;
            sockets.push(sock);
            // Build and send response packet
            sendOpenConnectionResponse(sock);
            break;
          // This will only happen when requested, which sets up event listeners per
    ↪ requested device/socket
          case SUBTYPE.STATUS_RESPONSE:
            break;
          // Data dump from device, send to database function
          case SUBTYPE.DATA_DUMP:
            sendDataToDatabase(packet);
            break;
          default:
            break;
        }
      }
    }
  });

  // Add a 'close' event handler to this instance of socket
  sock.on('close', function() {
    let index = sockets.findIndex(function(o) {
      return o.remoteAddress === sock.remoteAddress && o.remotePort === sock.remotePort;
    });
    if (index !== -1) sockets.splice(index, 1);
    console.log('CLOSED: ' + sock.remoteAddress + ' ' + sock.remotePort);
  });

  // Log error instead of crashing
  sock.on('error', function(e) {
    console.log('Socket error:', e);
  });
});

// Generates a byte-packed packet to reply to new devices with
// Contains packet ID, version, packet subtype, and current local time in seconds since
    ↪ epoch
function sendOpenConnectionResponse(sock) {
  let date = new Date();
```

```
  let message = Struct()
    .word8('PACKET_ID')
    .word8('VERSION_ID')
    .word8('SUBTYPE')
    .word32Ube('TIME') // big-endian for network!!
    .word8('end');

  // Allocate buffer for message
  message.allocate();
  let buf = message.buffer();

  let messageFields = message.fields;

  // Fill out message fields
  messageFields.PACKET_ID = PACKET_ID;
  messageFields.VERSION_ID = VERSION_ID;
  messageFields.SUBTYPE = SUBTYPE.OPEN_RESPONSE;
  messageFields.TIME = Math.floor(convertUTCDateToLocalDate(date).valueOf()/1000);

  // Send to socket
  sock.write(buf);
}

// Request status from device, sends packet with header and only subtype as a payload
function sendStatusResponse(sock) {
  let message = Struct()
    .word8('PACKET_ID')
    .word8('VERSION_ID')
    .word8('SUBTYPE')
    .word8('end');

  // Allocate buffer for message
  message.allocate();
  let buf = message.buffer();

  let messageFields = message.fields;

  // Fill out message fields
  messageFields.PACKET_ID = PACKET_ID;
  messageFields.VERSION_ID = VERSION_ID;
  messageFields.SUBTYPE = SUBTYPE.STATUS;

  // Write to socket/device
  sock.write(buf);
}

// Parse data dump from device
function sendDataToDatabase(data) {
  // Each sample will be handled with one SQL query
  let sql = "INSERT INTO powermonitoring.powerusage (Device, MeasurementTime, WattValue)
    ↪ VALUES (?, ?, ?);";
  let DeviceID = data.DeviceID;

  // Loop over each sample from data
  data.Samples.forEach(sample => {
    let params = [];

    // Format time to SQL time B)
    let time = (sample.time).toISOString().substring(0, 10) + ' '
      + (sample.time).toISOString().substring(11, 19);

    // Fill out parameters for mysql.format
    params.push(DeviceID);
    params.push(time);
    params.push(sample.power);

    // Send query to database
    db.conn.query(mysql.format(sql, params));
```

```
  })
}

// Parse raw byte-packed data from devices into a more usable JSON object
function buildPacketObject(data, sock) {
  let packet = {};

  // Parse incoming data as string of hex
  packet.PacketID = parseInt("0x" + data.substring(0,2));
  packet.Version = parseInt("0x" + data.substring(2,4));
  packet.PacketType = parseInt("0x" + data.substring(4,6));

  // Handle each packet type differently/appropriately
  switch (packet.PacketType) {
    case SUBTYPE.OPEN_CONNECTION:
      packet.MACAddress = data.substring(6);
      break;
    case SUBTYPE.STATUS_RESPONSE:
      packet.PMean = parseRawData(data.substring(6,10), 'Pmean');
      packet.Urms = parseRawData(data.substring(10,14), 'Urms');
      packet.PowerFactor = parseRawData(data.substring(14,18), 'PowerF');
      packet.Frequency = parseRawData(data.substring(18,22), 'Freq');
      break;
    case SUBTYPE.DATA_DUMP:
      packet.DeviceID = sock.macAddress;
      packet.Samples = [];

      let i = 6;
      while (true) {
        // Loop over data until we see the end: (0 time, 0 Pmean)
        if (parseRawData(data.substring(i, i + 8), 'Time').valueOf() !== 0) {
          let sample = {
            time: parseRawData(data.substring(i, i + 8), 'Time'),
            power: parseRawData(data.substring(i + 8, i + 12), 'Pmean')
          };
          // Push each sample on array
          packet.Samples.push(sample);
          i += 12;
        } else {
          break;
        }
      }
      break;
    default:
      break;
  }

  // Return filled packet
  return packet;
}

// Parse raw byte-packed data from power monitoring IC appropriately
// All registers return 16-bits, signed and unsigned
function parseRawData(hex, reg) {
  switch (reg) {
    case 'Urms':
      // Unsigned XXX.XX
      return parseInt('0x' + hex)/100;
    case 'Pmean':
      // Signed kW, convert to W, treat as signed
      let Pmean = parseInt('0x' + hex, 16);
      if ((Pmean & 0x8000) > 0) {
        Pmean = Pmean - 0x10000;
      }
      return Pmean;
    case 'Freq':
      // Unsigned Hz
      return parseInt('0x' + hex)/100;
```

```
    case 'PowerF':
      // Signed X.XXX, treat as signed data
      let PowerF = parseInt('0x' + hex, 16);
      if ((PowerF & 0x8000) > 0) {
        PowerF = PowerF - 0x10000;
      }
      return PowerF/1000;
    case 'Time':
      return new Date(parseInt('0x' + hex)*1000);
    default:
      return -1;
  }
}


// Convert date to timezone shifted date
function convertUTCDateToLocalDate(date) {
  return new Date(date.getTime() - date.getTimezoneOffset() * 60 * 1000);
}

module.exports = router;
```

C.3 db (Database)

C.3.1 /db/create.sql

```
-- MySQL Script generated by MySQL Workbench
-- Wed Apr 29 14:05:20 2020
-- Model: New Model  Version: 1.0
-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,
    ↪ NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';

-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------
-- -----------------------------------------------------
-- Schema powermonitoring
-- -----------------------------------------------------

-- -----------------------------------------------------
-- Schema powermonitoring
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `powermonitoring` DEFAULT CHARACTER SET utf8 ;
USE `powermonitoring` ;

-- -----------------------------------------------------
-- Table `powermonitoring`.`devices`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `powermonitoring`.`devices` (
 `DeviceID` VARCHAR(24) NOT NULL,
 `FriendlyName` VARCHAR(45) NOT NULL,
 `active` TINYINT NULL DEFAULT '1',
 PRIMARY KEY (`DeviceID`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;


-- -----------------------------------------------------
-- Table `powermonitoring`.`powerusage`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `powermonitoring`.`powerusage` (
 `MeasurementTime` DATETIME NOT NULL,
 `Device` VARCHAR(24) NOT NULL,
 `WattValue` SMALLINT NOT NULL,
 PRIMARY KEY (`MeasurementTime`, `Device`))
ENGINE = InnoDB
DEFAULT CHARACTER SET = utf8;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

C.4 Testing Scripts

C.4.1 /testing/main.py

```
import socket
import struct
import ctypes
from enum import Enum

TCP_IP = '192.168.1.218'
TCP_PORT = 7060
BUFFER_SIZE = 1024

PACKET_ID = 0xBA
VERSION_ID = 0

# Subtypes
OPEN_CONNECTION = 0
OPEN_RESPONSE = 1
STATUS_REQUEST = 2
STATUS_RESPONSE = 3
DATA_DUMP = 4


s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))


def main():
  send_open_connection()

  while 1:
    data = s.recv(BUFFER_SIZE)
    print("received data:", data.decode())
    if data.decode()[3] == '2':
      print('received 2')
      send_status_response()
    # if input("Send?\n") == 'Y':
    #   print("Sending")
    #   send_status_response()


  s.close()


def send_open_connection():
  mac_address = [0x3c, 0xe1, 0xa1, 0x9b, 0x6b, 0x48]
  print(type(mac_address))
  # buffer = PACKET_ID + VERSION_ID + "0" + mac_address

  MSG = ctypes.create_string_buffer(9)
  struct.pack_into('!B', MSG, 0, PACKET_ID)
  struct.pack_into('!B', MSG, 1, VERSION_ID)
  struct.pack_into('!B', MSG, 2, OPEN_CONNECTION)
  for i, mac in enumerate(mac_address, start=3):
    struct.pack_into('!B', MSG, i, mac)
  # s.send(buffer.encode())
  s.send(MSG.raw)


def send_status_response():
  MSG = ctypes.create_string_buffer(11)
  struct.pack_into('!B', MSG, 0, PACKET_ID)
  struct.pack_into('!B', MSG, 1, VERSION_ID)
  struct.pack_into('!B', MSG, 2, STATUS_RESPONSE)
  # Mean Power
```

```
    struct.pack_into('!h', MSG, 3, 69)
    # Volts RMS
    struct.pack_into('!B', MSG, 5, 119)
    struct.pack_into('!B', MSG, 6, 123)
    # Power Factor
    struct.pack_into('!B', MSG, 7, 0)
    struct.pack_into('!B', MSG, 8, 98)
    # Frequency
    struct.pack_into('!B', MSG, 9, 60)
    struct.pack_into('!B', MSG, 10, 12)
    s.send(MSG.raw)


if __name__ == '__main__':
    main()
```