Spring 2023

Smart Power Outlet

Kyrolos Melek
The University of Akron, km237@uakron.edu

Joseph M. Garro
The University of Akron, jmg289@uakron.edu

Ethan G. Frese
The University of Akron, egf14@uakron.edu

Madison Britton
The University of Akron, mb305@uakron.edu

Honors Research Project

Smart Power Outlet (SDP-DT08)

Individual Student Contributions

Madison Britton

The responsibility that I had as the team's Engineering Data Manager included keeping a log of the team progress. I was also responsible for researching and implementing the AC/DC conversion circuitry and the DC/DC conversion circuitry. Along with the AC/DC conversion circuitry, I integrated our relay circuitry. I worked primarily with Ethan Frese to ensure that these subsystems were implemented correctly both individually and in tandem with each other and the rest of the subsystems. Lastly, I created many schematics of our subsystem designs.

Ethan Frese (Graduating with Honors Requirements) or (Honors)

As the team's Hardware Manager, I was primarily responsible for designing and implementing the power measurement subsystem. This included researching which components would fit the necessary specifications as well as ordering them, testing them, and building the subsystem to ensure the subsystem functioned properly. I reviewed Madison's AC/DC conversion circuitry along with the relay circuitry and helped test it with her to ensure it integrated properly with the power measurement subsystem. Additionally, alongside Joseph Garro, I worked heavily on troubleshooting integration between the power measurement subsystem and the outlet communications subsystem.

Joseph Garro (Graduating with Honors Requirements) or (Honors)

I was our team's Project Manager, responsible for organizing meetings, setting project deliverables, and managing my teammates. Additionally, I oversaw the integration of the various hardware and software components of our project. For the project, I was the designer of the embedded processing and communication systems used by our Smart Power Outlet and the Hub, responsible for their design, function, and physical implementation. I was also responsible for the embedded firmware used in the Smart Power Outlet and worked alongside Kyrolos Melek on the Hub's embedded firmware. I researched various communication methods and decided to use XBee as our communication protocol between the Smart Power Outlets and the Hub and was responsible for writing an interface to allow our microcontroller, an ESP32, to communicate with our XBee modules. I was also responsible for the mechanism to structure and process data transmitted between the Smart Power Outlets and Hub. My final major contribution was working on the construction of the power measuring circuitry and the integration of the power measuring circuitry into the Smart Power Outlet with Ethan Frese. Other contributions included the research and design of a circuit that allowed the ESP32 to toggle the relays that we selected, work with our department's Engineering Technicians Max Fightmaster and Erik Rinaldo to acquire various components and assemble PCBs containing surface mount components, and assistance provided across all areas of the design project.

Kyrolos Melek (Graduating with Honors Requirements)

As the software manager on this team, I was responsible for the application development, embedded development for the main hub alongside Joseph Garro, and a backup power measurement solution. I researched various WIFI communication methods (UDP, HTTP), embedded systems from Espressif, and several different frameworks for app development. On the embedded side, I developed an http webserver for communication between the hub and phone

application, a SNTP client for retrieving an epoch time from a SNTP server, an http client that was responsible for posting data to a MongoDB database instance. On the application side, I developed a front and backend in .NET MAUI which employed the use of an MVVMS architecture. On the backend side, I developed an asynchronous http client which contacted the main hub for commands/data, used the C# driver for MongoDB to retrieve power data from the cloud database, and various other infrastructure such as a cloud API that allowed the embedded system to communicate to the database from C++ using HTTP requests. Finally, for the backup power measurement solution I developed timer-based interrupts that would measure the voltage of the output of a voltage and current transformer using the Analog to Digital converter at a frequency of 120 Hz.

Smart Power Outlet

Senior Project Final Report

Design Team 08

Madison Britton

Ethan Frese

Joseph Garro

Kyrolos Melek

Faculty Advisor: Dr. Yan Zhang

4/24/2023

Table of Contents

List of Figures

List of Tables

Abstract (KM)

With an ever-increasing demand on the electrical grid and an increase in electrical consumer goods per household, effective power management and monitoring for home users is a must. The objective of this project is to create a consumer outlet and companion phone application. The power outlet will measure power draw on each socket and report these measurements via an embedded system, wirelessly, to the application. The application will then allow for setting custom power draws at which an individual socket will switch off if exceeded and will display how much a given outlet or individual socket is costing per unit time at current power/electrical rates. The application will also allow for switching on or off any sockets akin to a smart home device and labeling each outlet and socket within the application. The application will enable sending push notifications, if a dangerous or undesirable power draw/current is reached. These features will enable consumers to be more aware of the electricity they are consuming and its associated cost.

Key Features:

- Turn individual sockets on or off through mobile application
- Monitor and view power and energy use per socket and overall household usage
- Enable socket toggling based on conditions such as maximum power draw, cost, or time

1. Problem Statement

1.1. Need (KM)

Modern day electronic devices spend an estimated 1,300 kilowatt-hours while unused (idle) per annum, resulting in an average cost of $210-$440 per year in electric bills according to the NRDC (Natural Resources Defense Council) [2] and research performed by the Sandford Sustainable Systems Labs. Although modern devices may have low-power and sleep modes, there

exists no clear-cut standard as to how much power these devices consume. A more cost-effective solution would allow homeowners to accurately analyze and cut-off high power-consuming idle devices above a certain power draw.

1.2. Objective (KM)

The objective of this project is to create a consumer outlet and companion application. The power outlet will measure power draw on each socket and report these numbers via an embedded system, wirelessly, to the application. The application will then allow for setting custom power draws at which the outlet/socket will switch off if exceeded and will display how much a given outlet/socket is costing per unit time at current power/electrical rates. The application will also allow for switching on or off any sockets akin to a smart home device and labeling each outlet and socket within the app. The application will enable sending push notifications if a dangerous or undesirable power draw/current is reached.

1.3. Background (MB, EF, JG, KM)

Many studies have recognized the significance of limiting and optimizing unnecessary power consumption. One such research project [3] aimed to “minimize energy waste in home environments [by] efficiently managing devices operation modes”. Another research paper [4] identified how the exploding popularity of IoT devices has “contribut[ed] to an increase in phantom standby energy consumption”, suggesting that their design based around low power modes is a must. Furthermore, Wang [4] also proposed the use of low power IoT devices to manage the power consumption of larger devices, finding a 14Wh difference in power consumption when a smart power control device was used in conjunction with a larger appliance compared to none at all. Leveraging this idea, it is apparent that the number of devices plugged into power outlets in a given person’s home results in a large amount of power being wasted each year. The solution is to

construct a smart, whole-home electrical monitoring and receptacle replacement solution that monitors power output per socket and allows for the on or off switching of each socket. The smart power outlet solution requires several electrical and computer related concepts and basic theories such as AC to DC transformation, step-down voltage conversion, mechanical or solid-state switching, embedded system programming, local server hosting, mobile development, and networking.

The process in which each appliance will be measured for its power draw is through a method called “load monitoring” [5]. Load monitoring involves taking in an aggregate (or combined) current signal from an outlet, and when provided with sufficient information about the appliances, can then disaggregate the signal into multiple forms that can then be processed and analyzed. This method allows appliances and their power draw to be measured on an individual level rather than taking the sum of power drawn from a single outlet. One of the challenges of this method is due to a “mix of disparate appliance types and the non-ideal nature of the electrical supply” [5]. A method will need to be found to determine the type of appliance being measured before making its power draw measurements.

AC to DC transformation could be handled with a full-bridge rectifier circuit, using power diodes, and a smoothing capacitor. The full-bridge rectifier consists of four diodes divided into two pairs of two diodes, allowing for opposite polarities of input current to flow, in a single direction, to a load during their respective half-cycles. The filter capacitor allows for smooth and even current flow to the load by storing voltage during rising sinusoidal portions of the input and discharging during falling portions. This AC to DC conversion would be used to power the onboard embedded system. A traditional solution for converting AC power to DC power is using a low frequency transformer and an AC-DC converter. The AC power is stepped down through

the transformer, and the AC-DC converter then converts the AC power to DC power, in other words, the operation of the AC-DC converter is like that of an active rectifier [6].

Voltage step-down could be handled with a buck converter circuit and/or a boost converter. AC-DC step-down conversion can be done by using a buck converter and a buck boost converter in parallel with each other. Placing the buck converter and buck boost converter in parallel with each other converts the AC voltage to a lower DC voltage used for electronic loads [7].

Socket switching could be handled either through a mechanical switching method such as relay or a solid-state method such as a power MOSFET. The on-board embedded system will be responsible for reading in electrical values such as current and power draw and communicating them to an either local or cloud hosted server. These embedded system tasks will be programmed using C and will have to leverage efficient, low-power programming techniques so as not to add significant overhead and cost to the system. Certain techniques such as loop unrolling and function in lining could provide valuable power reductions [8]. A main local server or hub, likely a Raspberry Pi, will receive all communicated values from the distinct outlets and display them through a web application or store/update them for a mobile application to fetch. The mobile application will be cross platform and likely written using a cross platform SDK such as Flutter, Xamarin, or React Native, to avoid the need to develop two separate and native applications for the IOS and Android Platforms. Network communication between the outlets, server hub, and application will be handled using networking protocols such as WIFI with TCP/UDP sockets, or other protocols such as ZigBee. User datagram protocol allows for logical communication amongst different processes across several hosts, however, it does not have the same overhead as transmission control protocol (TCP) since it does not require a four-way handshake, but rather is a connectionless protocol that allows for fast transmission of data [9].

Currently, the proposed idea is not fully in use. However, there are a variety of existing products that implement one or more of the desired features. Examples of such products are the "Kill A Watt" - a device featuring a single power socket that is plugged into an electrical outlet and displays the energy consumption of the device that is plugged into it on an LCD display. Another related product is the "Save A Watt" - another device with a single electrical socket that is plugged into an electrical outlet and allows a consumer to set times of day that the built-in electrical socket will be turned off. Concerning the "smart home" aspect of the proposed idea, smart electrical outlets exist as a replacement for standard electrical outlets. These smart power outlets are capable of being programmed to turn on and off but lack the power monitoring features of the "Kill A Watt". Devices known as "smart plugs" are available that can be plugged into a traditional power outlet and allow a connected device to be automated [10]. A problem with these is that they protrude a significant distance from the power outlet and may not fit in areas where the space between the power outlet and an appliance, piece of furniture, or another obstacle is limited. These devices may also block other power sockets on the power outlet due to their large footprint. Incorporating this feature inside of the smart power outlet allows the proposed idea to work in situations where these smart plugs otherwise will not and avoids the hassle of objects protruding from the power outlet and potentially blocking one of the power outlets in the power socket. Additionally, online calculators exist that allow electrical costs and power consumption to be calculated but rely on the measurements by a consumer's electrical meter rather than the total power consumed per device. This prevents consumers from knowing how much power each of their electrical devices is using, and how much each device is costing them.

As a result, the proposed idea seeks to combine the best features from these similar products into a single product that functions as an all-in-one replacement for them. This allows consumers

to get the most control of their power outlets while allowing them to retain their full functionality as the proposed ideal will not block other power sockets in the power outlet. By monitoring the power used by each smart power outlet, the total power used in one's home can be easily calculated, as well as the power used by each outlet. This allows the user to determine how much electricity each device in their home is using and its associated cost to run. Therefore, by giving the consumer the ability to control when the smart outlet is on, the consumer will be able to turn off outlets when their devices are not in use, saving them money.

The limitations of current designs include the use of wireless communication to actuate the switches connected to each outlet. Wirelessly activating outlets connected to mains power may lead to interfering noise from the 220VAC source. A solution to this could be using a single embedded system at the central device. This central device would ideally receive no actual noise and could be used to safely switch outlets on and off. Using a single embedded system in the central device would also cut down costs by making the technology in the outlets simpler and easier to produce, and instead concentrating all the technology into the central device.

A limitation solved by the proposed product is a way of properly measuring the amount of power consumed and paid for by accounting for energy *generated* by the system. In other words, accounting for any renewable energy sources such as solar panels that might be implemented in some buildings. This way, any building owner can immediately understand how much energy they are paying for by electric companies while also understanding how much money they save by using their own renewable energy sources.

Another limitation seen in current technology is the lack of centralized control and monitoring of systems in products such as the "Kill A Watt." Currently, this product is a simple plug-and-play device used to measure single outlets and their energy consumption. Without

centralized control and monitoring, one would need to purchase multiple “Kill A Watt” products and observe them individually to get a general idea of the building’s consumption. The product seeks to solve that issue by providing a central device that every outlet is connected to in order to properly control and monitor each outlet not only at an individual level but as a whole system.

An additional limitation of the “Kill A Watt” and similar products with a built-in display is that their usefulness is limited to situations where the display can be seen. An electrical outlet located behind an appliance or blocked by a piece of furniture will not allow the consumer to view the power consumption of their devices without moving the obstacle. With the proposed idea’s ability to provide remote monitoring of power consumption, this issue of the energy monitoring statistics not being easily accessible by the consumer is solved.

Being able to recognize the similarities and differences between existing products helps communicate how the proposed idea stands out. As described above, numerous products currently exist that accomplish one or more of the many features of the proposed idea. Some products allow the power computation out of a single power outlet to be measured while others allow a single device to be turned on and off on a timer. These devices operate as an extension of the power outlet, having to be inserted into the power outlet as an intermediary between the power outlet and the electronic device plugged into it. These types of devices have numerous disadvantages as they protrude from the socket, preventing them from fitting in where space is limited and can block the top socket on a power outlet if inserted into the bottom socket.

For example, the “Kill A Watt" product offers the basic feature of reading the load of a product and displaying that for the user, while also providing a display for how much power that appliance consumes over a period of time, allowing for calculations to be made about the estimated cost of running that appliance. The proposed product offers a similar feature but differs in the way

that it is implemented. Instead, this feature will be built into the product and allow the users to wirelessly receive their power consumption statistics on their smart devices.

Additionally, online calculators exist that allow people to calculate the cost of their electricity by inputting their electricity use from their electrical meter and outputting the price that they pay for electricity. This system only tells the user the total cost that they have to pay and is not capable of breaking down the total cost by the devices that contribute the most to it. Other calculators allow users to input the type of device, its power consumption, and the number of hours that it is used per day to determine an estimate of how much it costs to operate the device. However, these calculators are only an estimate and will not reflect devices like computers which can draw more power when under a higher load. The proposed idea aims to be able to identify the total amount of power used by each outlet, allowing the users of the product to know which outlets are contributing the most to their electrical bill. By allowing users to turn their smart power outlets on and off as they desire, it will be possible for the user to turn off their smart power outlets when they are not in use, preventing the devices that are attached to them from sipping power and driving up their electricity bill.

As shown, the proposed idea is a total solution to the problem of wasteful energy consumption in homes today. It takes the best features from a variety of products and combines them into a universal solution that allows users to retain the current functionality of their electric outlets and save money.

A variety of existing and patented technologies exist that are relevant to the proposed idea. The first relevant patent is US Patent US8487634B2- “Smart electrical wire-devices and premises power management system" that outlines a remotely controlled “‘smart’ wire-device” [11]. The device contains a “remotely controllable maker/breaker” that allows the device to pass through or

interrupt the transmission of electricity. This is relevant to the proposed idea as it details a system in which a breaker can be remotely controlled to change whether a device is delivering electricity or not. This patent can be used to identify a safe way in which AC power can be switched on and off in a manner that allows individual power sockets in the smart power outlet to be switched on and off. This patent also details the use of a "management node" that controls the state of the smart wire devices and receives electrical usage data from the smart wire devices [11]. This is similar to the proposed idea that monitors power consumption at each outlet and transmits the recorded data to an application so that the user can view them. Furthermore, their use of a management node to control the states of the smart-wire devices is akin to having an application that allows the user to control the status of each smart electrical outlet.

Another patent that is relevant to the proposed idea is US Patent US8447541B2- "Energy usage monitoring with remote and automatic detection of appliance including graphical user interface" [12]. This patent describes a way of examining the detailed energy usage of a single electrical device that is among a network of many electrical devices from a single point. This energy monitoring device presents a means of detecting energy use from one central location within each electrical device, and it does so in a low-cost, real-time manner [12]. This is relevant to the proposed idea as the goal is to design a smart, whole-home electrical monitoring and receptacle device. This "Energy usage monitoring with remote and automatic detection of appliances including graphical user interface" patent provides useful information on energy monitoring. The device described in this patent can detect when a load is turned on and off, and which load is switched to either setting. Through monitoring this on or off information and a load signature, the described device can produce the detailed energy usage of each electrical device in a given location. With the proposed idea, users have a way to receive a detailed version of their

energy usage per each of their electrical devices, making this patent relevant as a reference for how to provide detailed energy usage information to users. The patented device can display this energy usage information on the monitoring device, on a remote device, or on a local device. Remote devices are described as devices such as a smartphone, tablet, or laptop, and a local device is a device such as a personal computer. A desired feature of the proposed idea is that the user should be able to receive their energy usage information using a remote device, such as a smartphone, via an application on the device. This patent will be useful to reference when discussing how to collect detailed energy usage information for an electrical device and display this energy usage information to a user.

To understand a real-world application of monitoring and managing the performance of appliances, patent US8649987B2 can be referenced. This patent provides a means to monitor and manage more than one appliance at a time by utilizing a central database that includes "expected operating thresholds formed from individual appliance power consumption and run time data, . . . [and] alerts a system user upon variation of said individualized operational data from said expected operating thresholds" [13]. This system can also monitor and manage individual appliance performance by utilizing a gateway system that forms "a data collection and transmission device, . . . [and is] linked to a circuit breaker panel and configured to continuously in real-time obtain individualized operational data corresponding to one or more individual appliances electrically connected to said circuit breaker panel" [13]. This is relevant to the proposed product as it contains features like those previously described. Differences in this patent and our product include being able to toggle outlets from a device such as a cellphone or any other device that can communicate to the local network.

To understand a real-world application of monitoring and managing the performance of appliances, patent US8649987B2 can be referenced. This patent provides a means to monitor and manage more than one appliance at a time by utilizing a central database that includes “expected operating thresholds formed from individual appliance power consumption and run time data, . . . [and] alerts a system user upon variation of said individualized operational data from said expected operating thresholds” [13]. This system can also monitor and manage individual appliance performance by utilizing a gateway system that forms “a data collection and transmission device, . . . [and is] linked to a circuit breaker panel and configured to continuously in real-time obtain individualized operational data corresponding to one or more individual appliances electrically connected to said circuit breaker panel” [13]. This is relevant to the proposed product as it contains features like those previously described. Differences in this patent and our product include being able to toggle outlets from a device such as a cellphone or any other device that can communicate to the local network.

1.4. Marketing Requirements (JG, KM)

1. Unit will fit in a standard wall outlet receptacle
2. Unit will measure Voltage, Current, Power Draw per outlet
3. Unit will allow for remotely switching on and off each individual outlet
4. Unit will consume as little power as possible
5. Communication between units will be encrypted
6. Application will allow for setting custom power limits at each outlet
7. Application will calculate and display each outlet's usage history

2. Engineering Analysis

2.1. Circuits

2.1.1. Power Usage Analysis (MB)

An important function that the smart power outlet must have is the ability to analyze a household's power usage. The goal of this function is to allow the user to see how much power a specific outlet/socket is using, and the financial costs of an outlet's power usage per unit time using the current power/electrical rates. Doing an analysis of the power usage of each outlet is essential in order to determine which outlets may be consuming "phantom power", or to determine which outlets may be using more power than desired by the user.

As a basic definition, "phantom power" is the power that is consumed by devices that are idle. In other words, a device can be turned off, but if it is still plugged into an outlet, it can still consume power. Therefore, the power that is being consumed by a device that is plugged in, even though the device is off, is called "phantom power". The smart power outlet that is being designed here is looking to reduce "phantom power" that is being consumed by idle or off devices that are plugged in. For the smart power outlet to reduce "phantom power", the smart power outlet will first need to do some power usage analysis.

When considering power usage, it is relevant that some key terms be known. The first term that is relevant to know is kilowatt (kW) or watt (W). A watt is the standard unit of power according to the International System of Units and is equivalent to one joule per second. Electrical energy is used at a specific rate, and that rate is specified by the watt. In turn, a kilowatt is simply one-thousand watts, and the term kilowatt is used more often when considering the power used in a household setting. Now that the kilowatt has been defined, the kilowatt-hour can be discussed. The

kilowatt hour (kWh) is used to measure energy, and it is how most household energy costs are calculated [14].

To be able to do a power usage analysis, the power will first need to be measured. Power is calculated using current and voltage, therefore the current and voltage that the device is using must be measured. The process and devices that can be used to measure the current and voltage is described later in this project. Once the measurements of current and voltage are obtained, a simple calculation of multiplying the current with the voltage will result in the power. The values of current and voltage that are found will be processed by a microcontroller, and the microcontroller will determine the value of the power. The microcontroller will then be in charge of determining the power usage over a period of time and determining the financial costs of the power consumption per unit of time based on applicable power/electrical rates. The electrical rate that will be taken into consideration will be user specified, that way a user can specify the electrical rate that applies to them based on factors such as their location or their power company. This user-specified rate, along with the power consumption that is calculated by the microcontroller, will be used by the microcontroller to calculate the cost of electricity being used per kilowatt-hour.

2.1.2. High Voltage Safety Requirements (MB)

There are two electrical codes that are widely followed in the United States. These two electrical codes are the National Electrical Code (NEC) and the National Electrical Safety Code (NESC). The major difference between these two electrical codes is that the NEC applies to electrical systems that are within homes and businesses, while the NESC is applicable to large electrical power systems that supply power to homes and businesses. Using the basic definition given above of what makes these two electrical codes different, it is clear that the NEC will be a more prominent source of guidelines that must be followed for the Smart Power Outlet.

2.1.3. Power Splitter (MB)

A standard duplex receptacle has two plugs, one on the top half of the receptacle and one on the bottom half of the receptacle. Each half of the receptacle has a hot wire terminal and a neutral wire terminal. These two terminals are typically connected by a metal strip called connecting tabs [15]. There are connecting tabs for both the hot terminal and the neutral terminal, and these connecting tabs are what allow for there to only be one hot wire and one neutral wire needed to connect the outlet terminals with the electrical main. In this case, both halves of the receptacle are receiving the power from the same hot wire.

The smart power outlet that is being designed aims to allow for the top half of the receptacle and the bottom half of the receptacle to receive power from different hot wires so that each plug of the receptacle can be controlled. The way that this is done using the standard duplex outlet described above is by removing the connecting tab of the hot terminals [15]. When the connecting tab of the hot terminals is removed, it results in there being two hot terminals, one for the top plug of the receptacle and one for the bottom plug of the receptacle. In turn, each of the hot terminals needs its own hot wire in order to supply both halves of the receptacle with power [15]. The connecting tab of the neutral terminal remains intact; therefore, the two plugs of the receptacle share a neutral [15].

2.2. Electronics

2.2.1. AC-DC Converter (MB)

The conversion of AC voltage to DC voltage is an integral part of the Smart Power Outlet project. This conversion is necessary if the “smart” component of this project is to work. Considering how the outlet will work, the AC voltage will be taken from the electrical main and connected to the outlet. Then, the AC voltage from the electrical main that has been connected to

the outlet will need to be converted into DC voltage in order for the on-board embedded system within the outlet to be powered.

The AC-DC voltage transformation can be done using an AC-to-DC transformer component. An AC-to-DC transformer component is comprised of an AC-to-DC transformer, a full-bridge rectifier, and a smoothing capacitor. An AC-to-DC transformer is made of a number of steel plates that are tightly stacked to each other and epoxied together, and two or more coated-copper wire windings [1]. The coated copper wire windings can have anywhere from just a few turns to thousands of turns, and the number of windings is determined by the desired change in voltage [1]. Introducing a current through a winding of the transformer results in a magnetic field being created, and the magnetic field has poles that form along the axis of the winding. Placing another winding nearby to the first winding, along the same axis, causes a current to be induced by the magnetic field, which causes for a voltage to be induced in the second winding [1]. The losses between these windings can be reduced by introducing a magnetically permeable core between the windings, and this allows for the effects to be greater. Using insulated wire to create the windings means that the windings can be wrapped around each other, and then wrap both windings around the core [1]. This method saves space and is very efficient because several windings can be used to get the desired voltage. The only issue that arises now is that the output is always AC because the magnetic field must change polarity for the magnetic coupling to work. This is done through using AC current that switches at a 60Hz frequency between positive and negative voltages [1]. For electronic circuits to function the stepped-down AC voltage from the transformer needs to be converted to DC. The AC-DC conversion is done using the full-bridge rectifier. The components used in a full-bridge rectifier circuit are power diodes, or MOSFETs, and a smoothing capacitor. In more detail, a full-bridge rectifier circuit consists of four diodes total

that are split into two pairs of two diodes, which allows for the opposite polarities of an input current to flow in a single direction to a load during their respective half-cycles. The smoothing capacitor acts as a filter that results in a smooth and even current flow to the load. The filter capacitor does this by storing voltage during the rising sinusoidal portions of the input and discharging it during the falling sinusoidal portions of the input.

The AC-DC conversion can also be done using a low frequency transformer and an AC-DC converter. This method involves taking the AC power, stepping it down through a transformer, and then using an AC-DC converter to convert the AC power to DC power, where the AC-DC converter operates like an active rectifier [6]. Stepping down voltage from AC to DC is done using a buck converter and a boost converter. The buck converter and the boost converter are placed in parallel with each other in a circuit, and this results in the AC input voltage to be converted to a small DC voltage than can be used for electronic loads [7].

2.2.2. Power Measurement (EF)

To measure the power a device consumes, the current, voltage, and power factor being sent to the device must first be measured by some means. Once these values are found, a microcontroller can either process this data and find a value for the power usage, or receive all the necessary data and send it to a central hub. For this project, the ESP32 microcontroller provides the necessary processing power for this data.

To measure the current, a Hall sensor was first considered. Hall sensors utilize the Hall effect by detecting the presence and magnitude of a magnetic field that is generated by flowing current. The sensor considered, the AC1015, would function as a 1000:1 current transformer. The Hall sensor would be placed after the relay in the circuit and have the mains line wire run through the sensor. The data from this sensor would then be sent to the analog-to-digital converter pin on

the microcontroller. The Hall sensor is optimal as a current sensor as opposed to other types of current sensors because the sensor is not intrusive, as the setup simply involves inserting a wire through the center of the sensor and out the other side, and it provides readings that are about as accurate as the resolution of the microcontroller's analog-to-digital converter. The current transformer cannot be used on its own in the circuit, however. A burden resistor is required to cause a voltage drop in the circuit, allowing the microcontroller to handle the voltage coming from the transformer. Additionally, two dividing resistors of the same value are used to get the 2.5V reference voltage from the microcontroller, and a 10uF capacitor provides stability to the bias voltage.

To measure the voltage, multiple options can be considered. The first option considered is a simple HCPL3700, which is a voltage/current threshold detection optocoupler. That is, the device can sense voltages once it reaches a certain threshold. However, this device may not be as accurate as we would like, as it is mainly used for detecting low voltages and for monitoring relay contacts. The second option considered is a ZMPT101B voltage sensor. The ZMPT101B is an all-inclusive AC voltage sensor module. The device includes a transformer (technically, the actual ZMPT101B component) that takes a high voltage AC as an input with a limiting resistor and outputs a 1:1 AC signal with a sampling resistor in parallel. Using the analog output pin included on the board, the signal can be measured by the ESP32 microcontroller. By default, the pin on an ESP32 board will map analog input voltages between 0 and 3.3V into integer values between 0 and 4095 with a resolution of 0.8mV per unit. The ZMPT101B wiring is seen below in Figure 1 [16].

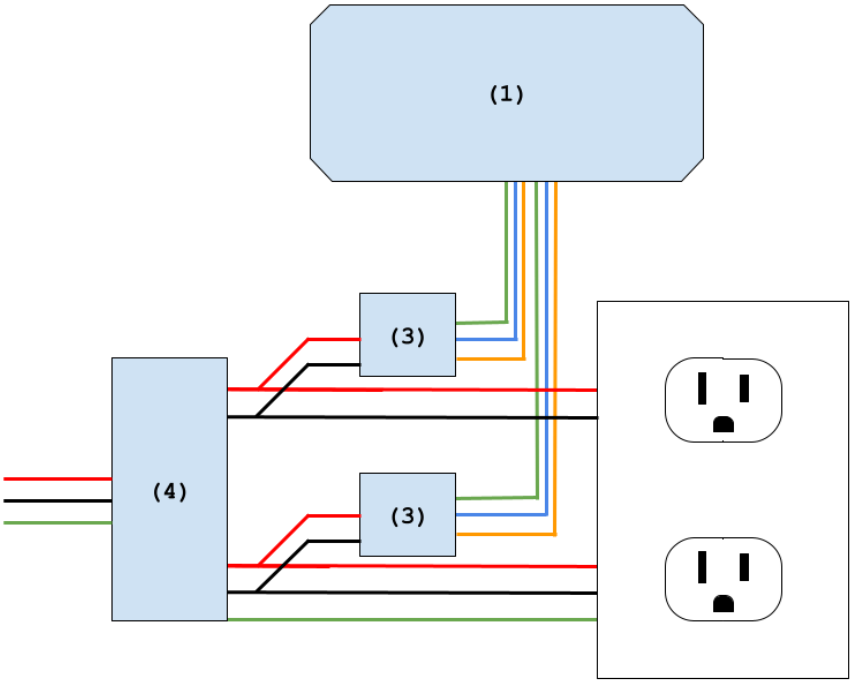

Figure 1: Example of voltage measurement setup

In the above figure, (1) represents a microcontroller that takes measurements from the voltage transformer, (3) represents the ZMPT101B integrated circuits, and (4) represents the terminal blocks that split the power going into each socket.

Measuring the power factor is necessary for this project to give accurate power readings, as devices sometimes consume power at non-unity rates. That is, some loads on a system may be capacitive, such as televisions, and some loads on a system may be inductive, such as laptop chargers. As a result, measuring the power factor in the Smart Power Outlet module is necessary to get accurate readings of *real power* as opposed to *apparent power* for more complex loads like laptop chargers. To achieve this, the waveforms of the current and voltage of the system must be measured. By looking at the phase difference between the two waveforms, the power factor can be determined.

During experimentation, several issues were discovered with using the methods above. First, the bulkiness of a Hall sensor greatly adds to the space required for the project's outlet subsystem. Additionally, using a Hall sensor would require additional circuitry to make sure its output could be read by the desired microcontroller without risk of damaging the microcontroller. For power factor readings, the ESP32 microcontroller did not prove to be powerful enough to provide accurate power factor readings based on the voltage and current waveforms. That is, the waveforms were sampled at a rate that was barely above the Nyquist rate of the 60Hz power signal, sitting at about 122Hz for the sample rate. The ESP32 could not sample at a higher rate because the same microcontroller was responsible for measuring two receptacles per outlet as well as processing that measurement data and transmitting it through an XBee antenna. With both the current measurement and power factor measurement needing to be changed, the voltage measurement also had to be changed.

To achieve all these changes, all-inclusive energy measurement chips were investigated and eventually implemented. In the case of this project, the Analog Devices ADE9153A Energy Metering IC was used. The ADE9153A provides inputs for two current channels and one voltage channel, and outputs for either SPI or UART communication to be sent to a microcontroller for data processing and display. For this project, just one current channel and one voltage channel was used.

The primary current channel of the ADE9153A functions by utilizing a shunt resistor and measuring the voltage difference across two pins on either side of the shunt resistor. The signal goes through a PGA to amplify the input signal from a small value to an input range that goes from -1V to 1V. This signal is then processed through a sigma-delta modulator with respect to the chip's reference voltage of 1.25V. After this, the signal passes through a variety of filters to clean up the

signal as well as a phase compensator. The final product is used in zero-crossing detection, total active and reactive power calculations, and RMS and VA calculations, as well as a couple other helpful features in the chip. The final value can be read off the chip using SPI communications.

The voltage channel of the ADE9153A measures data by measuring the voltage difference across two pins and sends it through a sigma-delta modulator with respect to the chip's reference voltage, similar to the current channel's datapath. This signal has an input range of 0.3V to 1.3V, with the "zero" voltage being represented by 0.8V. The signal is cleaned up the same way as the current channel's signal and is then output for the same uses as the current channel, with the addition of THD calculations. Like the current, this voltage can be read at any time from the chip using SPI communications.

2.3. Signal Processing

2.3.1. Electrical Noise Factor (EF)

In using any system that transmits and receives both power and data, the noise factor of the system must be considered. In this portion of the report, sources of electrical noise will be investigated to determine if they will pose any significant threat to the integrity of the system. Electrical noise can be transmitted into a signal cable in four ways: Radio Frequency Interference (also known as RFI), Electrostatic Coupling (also known as Capacitive Coupling), Inductive Coupling, and Conducted Noise (also known as Galvanic Coupling) [24].

In the case of wired communications, electrostatic coupling is by far the greatest threat of electrical noise. This can be reduced by shielding the signal wires, especially for signal wires that are in close proximity with power wires. However, because the signal won't be sent often, and not at an ultra-high frequency, with most of the raw data processed at the outlet level, this noise should not be an issue.

For this project's implementation, it was very rare for electrical noise to appear that could not be solved by simply following proper grounding procedures and keeping signal wires far enough away from power wires. Additionally, the chosen form of wireless communication from outlet to hub, Zigbee, works much better through walls and other obstacles than other forms of wireless communication such as Bluetooth, as will be discussed in the next sections.

2.4. Communications (JG)

Communication methods are vital to the Smart Power Outlet System as they define how various components can communicate with each other. It is important to have knowledge of various low-level communication methods used by sensors such as I2C, Serial, and UART so that devices within each embedded system can communicate. Additionally, high-level communication methods such as Wi-Fi, Ethernet over AC, and ZigBee can define how each major part of the Smart Power Outlet System will communicate with each other and the user. Hence, appropriate communication methods must be selected so that all devices composing the Smart Power Outlet System can communicate efficiently and effectively.

2.4.1. Low-Level Serial Communication Methods (KM)

Low-level communication methods, in the scope of this project, will be used to communicate between sensors and microcontrollers within the outlet modules. Serial communication methods allow for data exchange between two or more processors that do not have a shared pool of memory. Rather than using several simultaneous signal links per bit of data being sent, as is done in parallel communication, serial communication sequentially transmits data over a single channel [18]. Serial communications, although slower than parallel alternatives, can be implemented simply and perform quick enough for this application. Two important specifications to consider when evaluating serial communication methods are the bit rate and the baud rate [18].

Bit rates identify how many bits per second can be sent across a given channel. Baud rates identify the symbols per second, or the rate at which new symbols are communicated. This baud rate is dependent on how many bits are used to represent a given symbol in a processor or communication scheme.

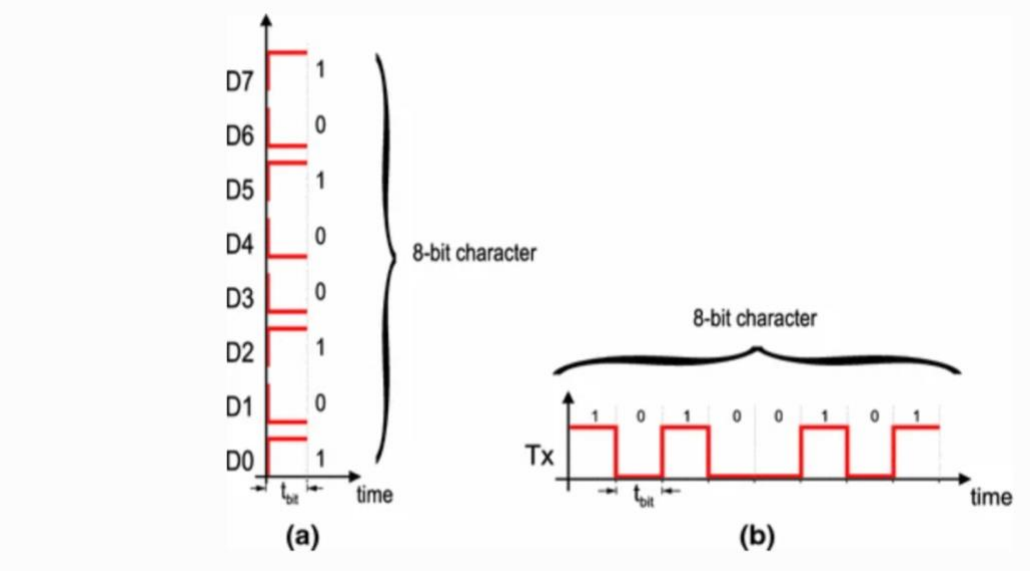

Figure 2: Parallel vs. Serial Communication from [18] Fig. 9.1

Three classifications of serial communication include simplex, half duplex, and full duplex. Simplex serial channels allow for communication in only one direction, having no support for acknowledging received data or the accuracy of such data. Half duplex channels allow for bidirectional communication, however only one direction at a given time. Full duplex serial channels allow for bidirectional, simultaneous communication, meaning two separate links can be used to send or receive data at the same time. A clock signal is used to establish transmission and reception rates. Clocks may be asynchronous or synchronous in nature. Asynchronous clocks involve separate clocks at each end of the transceiver, whereas synchronous ones send their clock

alongside data. These clocks are used to break data down into packets or datagrams, each of which contains a header, body, and footer that are determined by the protocol being used. Headers typically contain information regarding the data such as the length of a message, where footers can contain an end of packet sequence and optionally an error detection or correction field [18].

2.4.1.a I2C (KM)

I2C, or inter-integrated circuit, is a synchronous serial communication method, where two wires are used to share a clock signal and serial data is sent across communication modules [18]. In this communication protocol, there exists a master-slave relationship between the modules, where one or multiple masters can control one or multiple slaves. I2C has speeds that can range from 100kbps to 5Mbps. An I2C message or packet is comprised of a start and stop bit, a slave address, a read or write bit which signifies whether data is being sent or requested, and one or multiple 8-bit data frames separated by acknowledge bits which signify whether a given slave has received data. In, the case of multiple masters, collision detection is implemented by sampling the data line before a master transmits. If the line is low, it is assumed to be in use, otherwise it is idle. I2C has the benefit of being fast, and supporting communication between multiple master/slave devices, however, has the disadvantage of being only half-duplex, meaning data cannot be sent and received simultaneously [18].

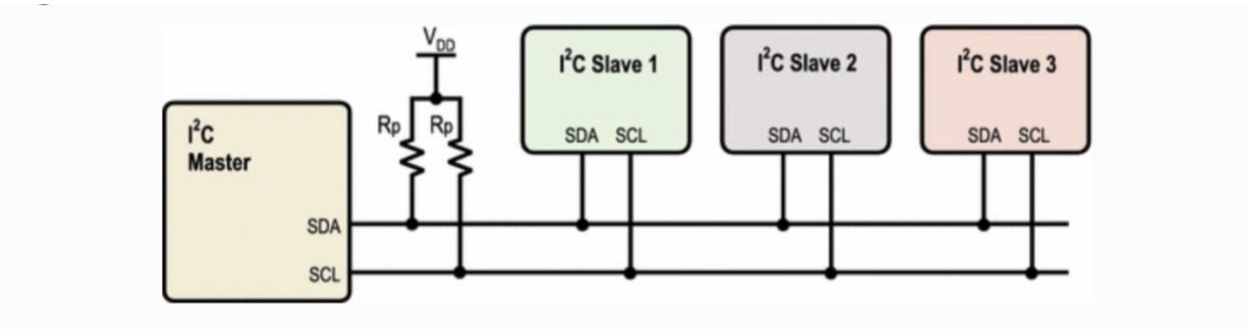

Figure 3: Topology of I^2C bus connection from [18] Fig. 9.23

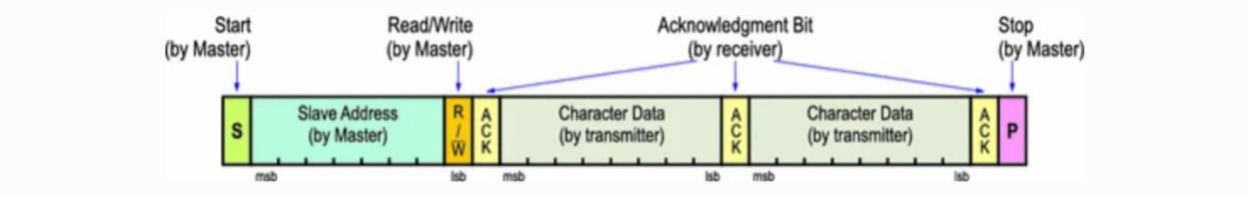

Figure 4: Structure of I^2C message from [18] Fig. 9.26

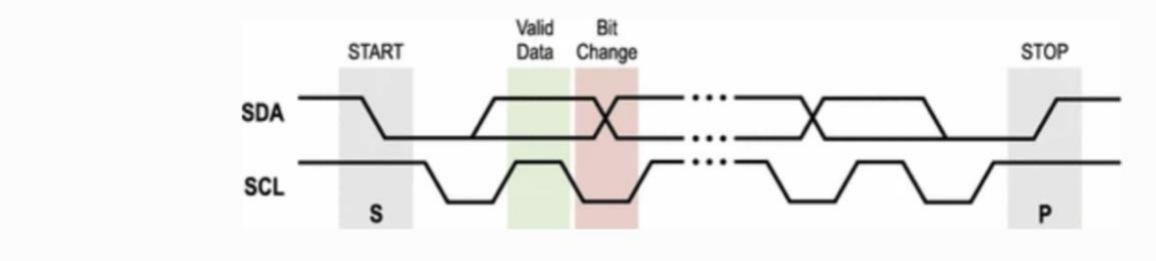

Figure 5: Timing diagram of I^2C signals from [18] Fig. 9.25

2.4.1.b UART (KM)

UART, or universal asynchronous receiver and transmitter, is an optionally full-duplex, asynchronous serial communication interface/protocol [18]. This communication method allows for two devices to communicate without sharing a common clock. They do, however, need to have the same transmission speed or baud rate so the incoming signals can be correctly sampled. UART frames consist of start/stop bits, data bits, and an optional parity bit. Idle states are held at high logic level. Some common parity bit error detection techniques include even parity or odd parity. Even parity sets the parity bit to ensure an even number of ones in each packet where odd parity sets the parity bit to ensure an odd number of ones in each packet. It is important to note that both parity schemes can detect only a single bit flip [18]. Benefits of UART include high transmission range, and simplicity of hardware.

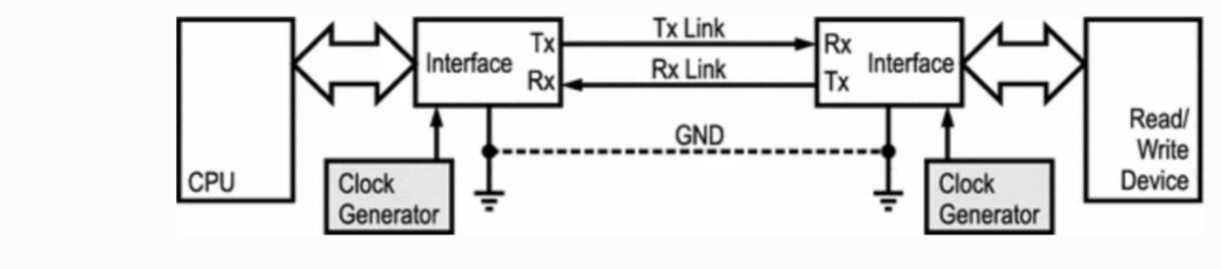

Figure 6: Asynchronous serial channel denoting independent clock source at channel ends from [18] Fig. 9.6

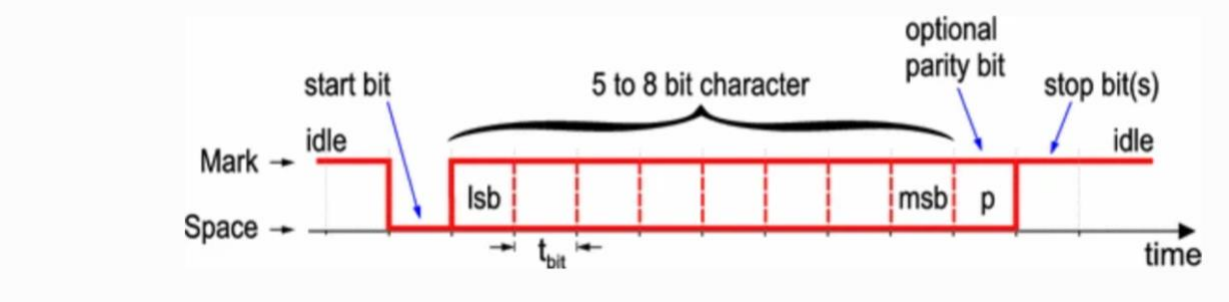

Figure 7: Format of an asynchronous packet from [18] Fig. 9.7

2.4.2. High-Level Communication Methods (JG)

High-level communication methods describe the communication methods that will be used to connect each part of the Smart Power Outlet system with each other and to the user. In the Smart Power Outlet system, high-level communication methods will be used to connect all Smart Power Outlets to the Hub and connect the Hub to the User's control application. As each of these connections has different requirements, it is likely that a different high-level communication method will be used to satisfy each required connection. This section aims to explore various high-level communication methods and identify why they are the best solutions to use in the Smart Power Outlet system.

2.4.2.a. Ethernet-over-AC (JG)

Ethernet-over-AC, commonly referred to by the brand name “Powerline”, is a communication standard specified by the Institute of Electrical and Electronics Engineers’ (IEEE) 1901 Standard. Ethernet-Over-AC specifies a way in which existing electrical wiring can be used to transmit an Ethernet Signal between connected devices. This approach is ideal for connecting the Smart Power Outlets to the Hub, as they are replacing traditional power outlets and will be connected to a home’s existing electrical wiring. Hence, the implementation of Ethernet-over-AC allows the Smart Power Outlets to be “plug and play”, requiring no new wiring to connect them to the Hub.

Another benefit of Ethernet-over-AC is its connectivity. As Ethernet-over-AC provides a wired connection between the devices, communication should be possible in instances where wireless connections would fail such as between rooms or at opposite ends of a house. A limitation of Ethernet-over-AC in this regard is that the maximum theoretical distance an Ethernet-over-AC signal can reach is 300 meters [19] of electrical wire. The problem is that electrical wiring is not wired in a straight path, instead going up and down walls and across floors and ceilings. Hence, some form of extenders may be required so that Ethernet signals can span an entire household. Additionally, the placement of the Hub will also impact the distance that Ethernet-over-AC signals will reach, as the hub is the start or endpoint for all communications between the Hub and Smart Power Outlets. Placing the hub in a far corner of a house opposed to the center or close to the circuit breaker will limit the coverage provided.

2.4.2.b. ZigBee (JG)

ZigBee is a wireless communication method that conforms to the IEEE’s 802.15.4 wireless radio specification [20]. Networks using ZigBee consist of three types of devices: coordinators,

routers, and end devices. The coordinator is responsible for forming and managing the network, routers manage traffic over a ZigBee network and perform tasks, and end devices perform tasks without being able to route traffic [21]. ZigBee also provides desirable features such as low power consumption, built in encryption, and scalability through a "self-organizing, [and] self-healing mesh" [22] that can connect thousands of devices.

Applied to the Smart Power Outlet system, every Smart Power Outlet would function as a router and the hub would serve as the coordinator. This will allow the Hub to manage the ZigBee network and every Smart Power Outlet to be able to communicate with the Hub. ZigBee modules and integrated circuits (ICs) are also cheap and can be found integrated into some microcontrollers, allowing the networking and processing components of the Smart Power Outlets to be combined. This will lower the cost and complexity of each Smart Power Outlet. Evidently, ZigBee is an optimal choice for the network to connect the Smart Power Outlets to the Hub due to its low power usage and built-in security, as well as its scalability due to utilizing a mesh network allowing every Smart Power Outlet to communicate with the Hub- something not guaranteed by Ethernet-over-AC.

2.4.2.c. Wi-Fi (KM)

Wi-Fi, or wireless Fidelity, is a series of protocols that rely on the IEEE 802.11 standards. The Wi-Fi 802.11 family all use the same CSMA/CA medium access protocol. The 802.11b standards supports data rates up to 11 Mbps, 802.11a and 802.11g up to 54 Mbps, 802.11n up to 450 Mbps, and 802.11ac up to 1300Mbps [23]. Wi-Fi signals on the 2.4Ghz band typically have a range of less than 50 meters. A typical network includes one or more wireless stations and a central base station or wireless access point. Each wireless station or endpoint is assigned a unique 6-byte MAC address [23].

Within the scope of this project, Wi-Fi would likely be used to communicate power statistics and outlet state changes between the central hub and mobile application. With the relatively small size of data packets being sent, Wi-Fi speeds will be more than sufficient to communicate the desired information. The mobile application and hub will be on the user's home network, which will handle either static or dynamic IP assignments, and handle communication through a series of access points, switches, and routers. An example ESP32 microcontroller implements 2.4Ghz Wi-Fi protocols 802.11b/g/n with a bit rate of up to 150 Mbps, which is sufficient for communication between the smart outlet hub and mobile application.

2.5. Electromechanics

2.5.1. Relays (EF)

To be able to toggle power flowing to each receptacle in the Smart Outlet, electronic relays are necessary. These relays would have an input of 120VAC and another input for a control signal, and an output of 120VAC that would go to the receptacle. The control signal would be sent from the microcontroller present in the Smart Outlet and would be directly responsible for closing and opening the relay.

Relays function by using a small current to control a larger pass-through current [27]. There are several types of relays that serve different purposes and possess different limits. Electromechanical relays (EMRs), for example, use contacts that are opened or closed by a magnetic force. Solid-state relays (SSRs), on the other hand, have no contacts and switching is completely electronic. SSRs are generally faster than EMRs because no physical parts move, and less voltage is required to toggle them on or off. However, the entire relay must be replaced if it becomes defective. EMRs have replaceable contacts and are generally much cheaper [28]. In this

section, both types of relays will be investigated to determine what would work best for this project.

Electromechanical relays have three subtypes of relays. General purpose relays are operated with AC or DC current, and up to 230V, and can control currents ranging from 2A-30A. Machine control relays are operated by a magnetic coil and are typically used for heavy-duty or industrial appliances. Reed relays are small, fast-operating, and compact switches with one normally open contact [28].

Solid-state relays have four subtypes of relays. Zero-switching relays turn on when the control voltage is applied and the voltage of the load is close to zero. These relays turn off when the control voltage is removed and the current in the load is close to zero. Instant ON relays turn on the load immediately when the pickup voltage is present. This allows the load to be turned on at any point in its up and down wave. Peak switching relays turn on the load when the control voltage is present, and the voltage of the load is at its peak. They turn off when the control voltage is removed and the current in the load is close to zero. Analog switching relays can have any output voltage within the relays rated range rather than set limits like general purpose EMRs have. They turn off when the control voltage is removed and the current in the load is near zero [28].

Besides the type of relay, there are also relays that vary in poles and throws. A SPST (Single Pole Single Throw) relay will have only open and closed positions. A SPDT (Single Pole Double Throw) relay will have a position that powers one circuit and a position that depowers that circuit and powers another. A DPDT (Double Pole Double Throw) relay will control two circuits between two positions [27]. For this project, only a simple SPST relay is necessary. The ratings of the relay should have a 3.3-5VDC control rating to toggle the relay open and closed, and a 120VAC / 15A rating for the power going in and the power going out of the relay.

2.6. Computer Networks (JG)

Computer networks will play a pivotal role in the function of the Smart Power Outlet as they will define how the outlets are able to communicate power usage to the user and how the user can control the Smart Power Outlets. The design of the computer network connecting the Smart Power Outlets, the Hub, and the user's phone application will influence what data each system is responsible for processing, the volume of the data, and the processing power required to do so. As the design of the Smart Power Outlet is constrained by the size of a standard North American Power Outlet and the Smart Power Outlets are the most prevalent component in our complete power monitoring solution, it is important that the communication method selected for the Smart Power Outlets is enabled with an efficient, inexpensive, and tiny physical solution. Additionally, the role of security will be important as control signals from the Smart Power Outlets to the Hub should be unable to be spoofed and secure from tampering by outside parties. Therefore, it is important that some level of security be incorporated into the final design as well.

2.6.1. Computer Communication Models (JG)

Depending on the communication method selected for each network within the Smart Power Outlet system, the corresponding network model and relevant protocols will vary. The following is an analysis of abstract computer communication models, as well as specific communication models relating to Ethernet-over-AC, ZigBee, and Wi-Fi.

2.6.1.a Abstract Computer Networking Models (JG)

Two common models used to represent computer communication are the Open Systems Interconnect (OSI) Reference Model and the TCP/IP Protocol Model. The OSI model abstracts communication into 7 layers while the TCP/IP model abstracts communication into 5 layers. A

depiction of the OSI model and TCP/IP model can be found below in Figure 8 along with the protocols used by each model at every level.

TCP/IP	OSI Model	Protocols
Application Layer	Application Layer	DNS - DHCP - FTP - HTTPS - LDAP - NTP - POP3 - RTP - RTSP - SSH - SIP - SMTP - Telnet - TFTP
	Presentation Layer	JPEG - MIDI - MPEG - PICT - TIFF
	Session Layer	NetBIOS - NFS - PAP - SCP - SQL - ZIP
Transport Layer	Transport Layer	TCP - UDP
Internet Layer	Network Layer	ICMP - IGMP - IPsec - IPv4 - IPv6 - IPX - RIP
Link Layer	Data Link Layer	ARP - ATM - CDP - FDDI - Frame Relay - HDLC - MPLS - PPP - STP - Token Ring
	Physical Layer	Bluetooth - Ethernet - DSL - ISDN - 802.11 - WiFi

Figure 8: OSI and TCP/IP Communication Models with Associated Protocols from [29]

Anytime that data is sent using either model, it must travel through all layers from top to bottom. Likewise, whenever data is received, it must go through all the layers from bottom to top. Starting at the bottom of [29], the first layer in both models is the Physical layer, which represents the physical connection over which data will be sent and received. The next layer is the Data Link layer, which handles the transfer of data between nodes once in a network [30]. Next is the Network layer, which determines how to move data between various networks so that it can reach the intended destination. The Transport layer is where protocols such as TCP and UDP are implemented so that data can be reliably transmitted and received [31].

Starting at the Session layer is where the OSI and TCP/IP models differ. In the OSI model, the Session layer is responsible for managing the connection, or session, between the two computers that are communication [31]. The Presentation layer, any data is formatted for the Application layer or Session layer to use [31]. Finally, the Application Layer is where the user can

interact with received data or decide to transmit data. However, in the TCP/IP Communication Model, the Session layer and Presentation layer are merged into the Application layer, yielding an application layer with a larger range of responsibilities. The TCP/IP model looks like the following: In the TCP/IP Communication Model, the Session layer and Presentation layer are merged into the Application layer, yielding an application layer with a larger range of responsibilities. Concerning the implementation of computer networks in the Smart Power outlet system, the TCP/IP Communication model can be used for its simplicity. Additionally, the TCP/IP protocol describes protocols that can be used to fulfil the needs of each layer, such as TCP and UDP at the Transport layer and IP at the Network layer [32]. If the previously listed protocols are to be used at their corresponding layers, only the Physical layer, Link layer, and Application layer implementations need to be created. As standard high-level communication models provide a Physical layer and Link layer implementation, these layers will be implemented by the selected communication method. Hence, the Smart Power Outlet system needs to provide a Physical layer solution to specify how data will physically be transferred and an Application Layer that will be able to make sense of data and allow users to interact with it.

Concerning the implementation of computer networks in the Smart Power outlet system, the TCP/IP Communication model can be used for its simplicity. Additionally, the TCP/IP protocol describes protocols that can be used to fulfil the needs of each layer, such as TCP and UDP at the Transport layer and IP at the Network layer [32]. If the previously listed protocols are to be used at their corresponding layers, only the Physical layer, Link layer, and Application layer implementations need to be created. As standard high-level communication models provide a Physical layer and Link layer implementation, these layers will be implemented by the selected communication method. Hence, the Smart Power Outlet system needs to provide a Physical layer

solution to specify how data will physically be transferred and an Application Layer that will be able to make sense of data and allow users to interact with it.

2.6.1.b Ethernet-over-AC (JG)

From a computer network's perspective, Ethernet-over-AC is enabled by receivers, transmitters, and transceivers that modify communication at the Physical layer. In the Physical layers, the IEEE 802.3 standard for Ethernet is implemented over AC wires using signal modulation and filtering. Besides the different medium in the Physical layer, the 802.3 Ethernet Standard is used. As a result, the communication model follows the TCP/IP model, with IPv4 or IPv6 being used at the Network layer and TCP or UDP being used at the Transport layer. As stated previously, the Application layer implementation is left up to the control application.

When using Ethernet-over-AC, standard Ethernet frames are sent. Ethernet frames look like the following:

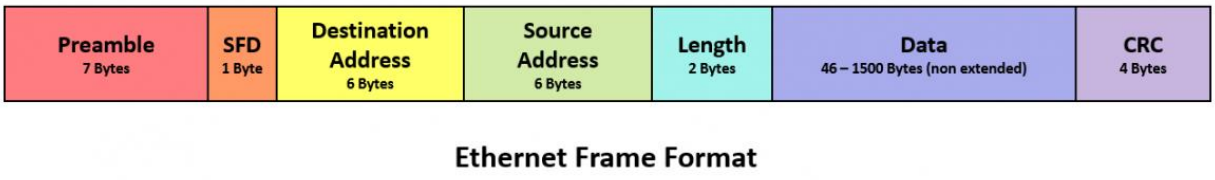

Figure 9: Standard 803.1 Ethernet Frame Format from [31]

As shown in [33], Ethernet frames can vary in size from 72 bytes to 1526 bytes. The largest influence on the size of the frame is the size of the data it is to contain. Should the data exceed 1500 bytes, the data will be split among multiple frames. Once established on an Ethernet network, the destination and source addresses will be used to identify the device sending the Ethernet frame and the device that is supposed to receive it. As there are 6 bytes allocated to the source and destination fields, $2^{(6 \cdot 4)} = 16777216$ possible unique sources and destinations exist. This

satisfies the requirement for scalability as many devices would be able to communicate using ethernet. Additionally, depending on the specific ethernet standard used, varying transmission speeds can be achieved. However, speed is not a crucial requirement to the Smart Power Outlet system as data measurements do not have to be received instantly by the Hub and a short delay when toggling the state of the outlets on and off is acceptable.

As stated previously, Ethernet-over-AC does have some drawbacks. Primarily, Ethernet-over-AC only supports a maximum distance of 300 meters (about 984.25 ft) of wiring, meaning that signals cannot travel far over electrical wires. Additionally, Ethernet-over-AC is very susceptible to noise as discussed previously, which may make the transmission of measurements and control signals difficult.

2.6.1.c ZigBee (JG)

ZigBee communications function differently from Ethernet and Wi-Fi and utilize their own communication model. The ZigBee communication model is as follows:

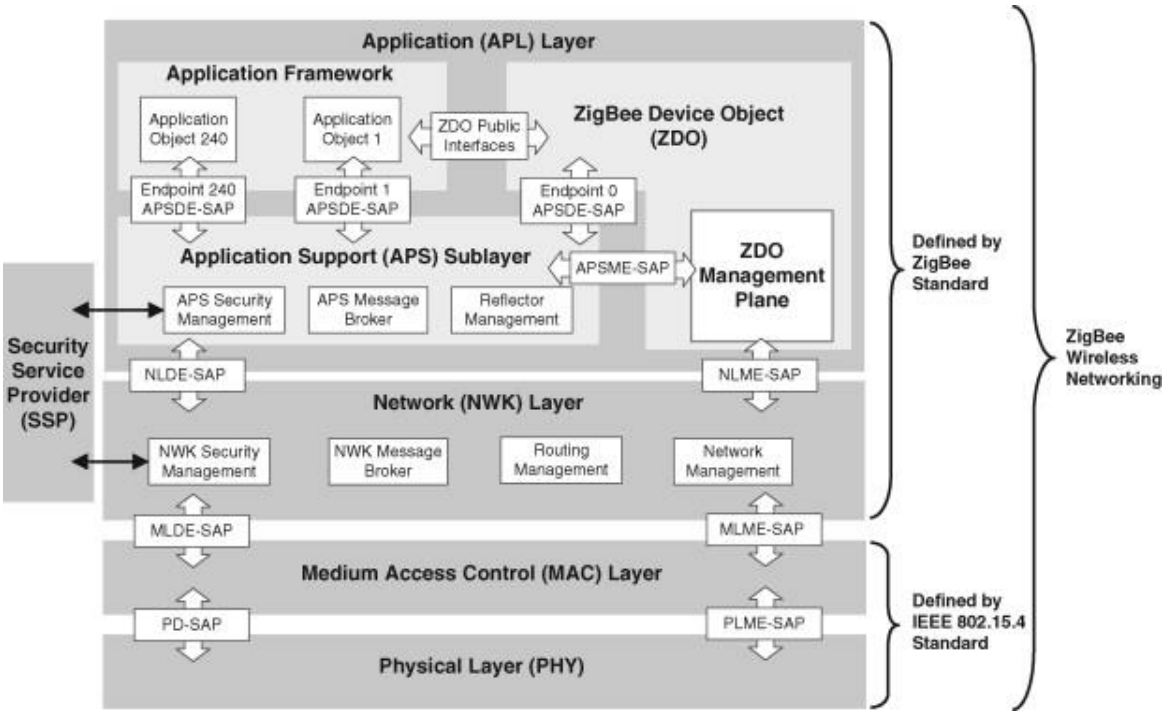

Figure 10: ZigBee Communication Model from [29] Fig 3.1

As shown in [29, Fig 3.1], the physical layer and MAC layer are defined by the IEEE 802.15.4 standard. The ZigBee standard then defines the network layer, application layer, and how ZigBee devices interact over a ZigBee connection.

As stated previously, three types of ZigBee devices exist: a coordinator, a router, and an end device. The coordinator is responsible for creating and managing a ZigBee network. End devices perform tasks, such as using sensors to measure something. Routers can function as an end device but are also capable of propagating communication between devices on the ZigBee network. Otherwise, end devices are incapable of communicating directly with other end devices.

The first step in creating a ZigBee network is establishing a coordinator. The coordinator is responsible for selecting the channel and PAN ID (a unique ZigBee network identifier) that the ZigBee Network will use [34]. Once the ZigBee network has been established, routers and end

devices can join the ZigBee network using the previously established PAN ID. Once on a ZigBee network, connected devices, known as nodes, are given a unique 16-bit address by the coordinator, and allowed to be assigned a 20-character string called a "Node Identifier" to make node identification easy [34].

When creating a ZigBee network, one of three network topologies can be used: star, tree, or mesh. Depictions of each network topology are shown in the following figure:

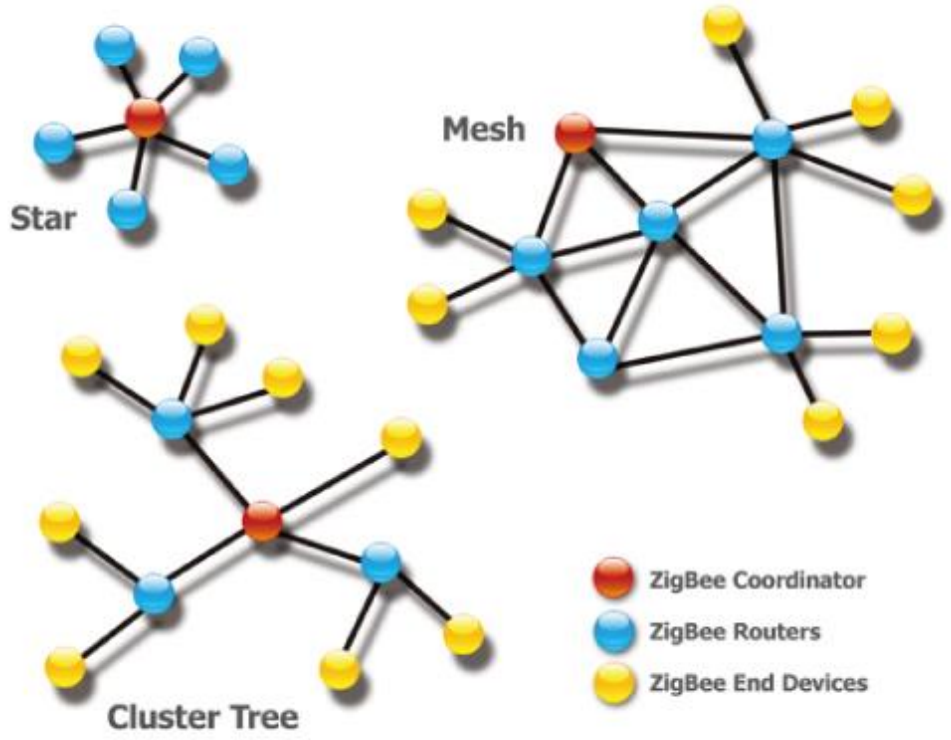

Figure 11: ZigBee Network Topologies from [32]

As shown in [35], a star topology is a pure client-server network architecture. A tree introduces basic forwarding elements where routers are used to direct communications between end devices. Finally, a mesh allows peer-to-peer communication between all nodes, meaning a device should always be able to communicate with another device. When a device is unable to

communicate with another device directly, a router or series of routers will be used to forward messages and achieve communication. Applied to the Smart Power Outlet system, a mesh topology where every Smart Power Outlet functions as a router and the Hub as the coordinator is the best approach. This is due to each Smart Power Outlet being able to forward communication to the Hub from those that are unable to directly communicate with it, so if a Smart Power Outlet is close to another Smart Power Outlet, communication with the Hub should eventually be obtained.

The ZigBee standard states that "[r]aw data throughput rates of 250Kbs can be achieved at 2.4GHz (16 channels), 500kbs at 915 – 921Mhz (27 channels), and 100kbs at 868Mhz (63 channel) ... [and] Transmission distances range from 10 to 100 meters, depending on power output and environmental characteristics" [36]. Hence, ZigBee is not a blazing-fast communication method nor a long-distance one. However, large amounts of data are not being transferred so ZigBee's speeds should be sufficient. The important thing is the range of ZigBee, which should be able to communicate anywhere from 10 meters to 100 meters. Hence, if Smart Power Outlets are placed within every 10 meters, a mesh connecting all Smart Power Outlets to the hub should be able to be formed. Thus, ZigBee can be used in the Smart Power Outlets to provide whole home coverage-something not guaranteed by Ethernet-over-AC.

Regarding the data sent by ZigBee, 127-byte packets containing 68 bytes of data are used. The format of a ZigBee packet is shown below:

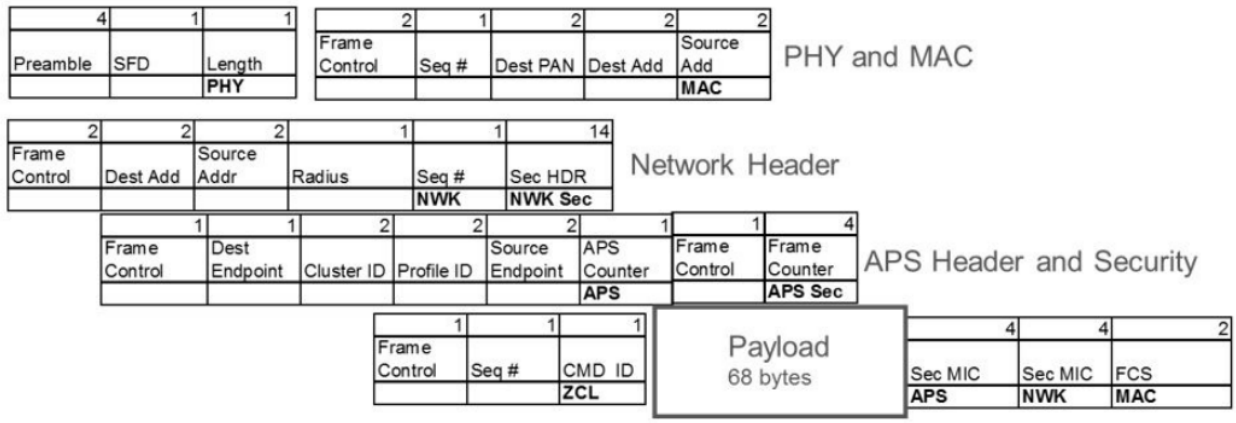

Figure 12: ZigBee Packet Format from [34] Fig. 1.1

As shown in [37], ZigBee packets contain a lot of overhead, but this is necessary to enable the connectivity and security benefits promised by ZigBee. Additionally, the 68-byte payload should be sufficient for transmitting the necessary measurements and control signals between the Smart power Outlets and the Hub. If not, ZigBee supports data fragmentation by which data larger than the 68-byte payload can be broken into multiple packets that will be transmitted.

2.6.1.d Wi-Fi (KM)

A wireless link is used to connect hosts or endpoints to a larger network or to connect devices such as routers, switches, and access points within a network [23]. In traditional networks, a base station or wireless access point is used to communicate between hosts and a router or broader area network. In ad hoc networks, there are no such base stations, and each host must take on the tasks of routing and addressing. Wireless networks such as these can be characterized by how many wireless hops a packet experiences when reaching a host, and whether or not the network infrastructure contains a base station such as an access point. Currently, four types of these networks exist by this categorization, single hop infrastructure-based, single hop infrastructure-less, multi-hop infrastructure-based, and multi-hop infrastructure-less.

Wi-Fi is defined by the IEEE 802.11 standard, which all use the same medium access protocol, carrier sense multiple access with collision avoidance, or CSMA/CA [23]. In this access protocol, device first sense the channel before transmitting to avoid collisions. These standards have the same frame structure for their link-layer frames, the ability to reduce transmission rates to span greater distances and are backwards compatible. These standards operate in the 2.4 -2.485 GHz and the 5.1 -5.8GHz ranges. The 2.4GHz range is unlicensed and Wi-Fi devices must deal with interference from other devices operating in this range. 802.11 devices operating in the 5GHz range suffer from shorter transmission distances at the same power level compared to the 2.4GHz range. As stated previously, the 802.11 data rates range from 11Mbps in 802.11b to in 1300Mbps in 802.11ac. The basic building block in 802.11 architectures is the basic service set, or BSS, which contains one or more wireless hosts and a central base station or AP. Wireless stations or hosts each contain their own globally unique 6-byte mac address. Since many devices and networks can occupy the frequency ranges, wireless stations or hosts need to identify or associate with a single subnet or access point. An 802.11 frame consists of a frame control field, duration field, four address fields, a payload, possibly containing an IP datagram, up to 2312 bytes in size, and a 32-bit CRC [23].

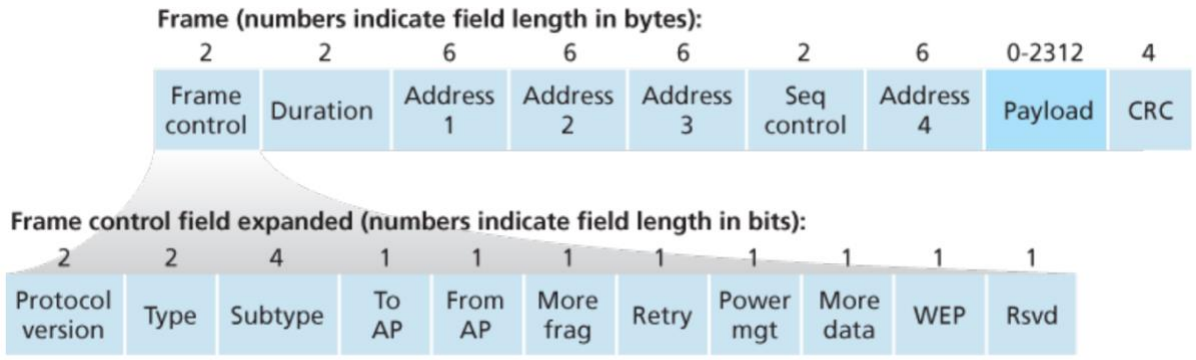

Figure 13: 802.11 Frame Format from [23] Fig 7.13

2.6.2. Network Architecture (JG)

Network architecture defines how the Smart Power Outlets, hub, and control application will be able to communicate with each other over a computer network. Two common network architectures include Peer-to-Peer (P2P) Networking and Client-Server Networking.

2.6.2.a Peer-to-Peer (P2P) Networking (JG)

Peer-to-Peer networking describes a network architecture consisting of multiple devices that are in direct communication with each other. With P2P networking, all devices in the network can send data to any other device and receive data from any other device. A P2P network applied to the Smart Power Outlet system would look like the following:

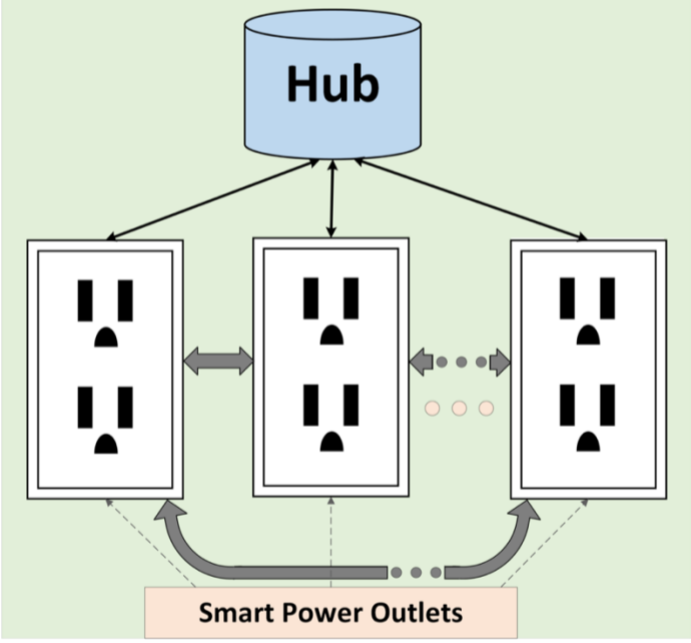

Figure 14: Communication between Smart Power Outlets and Hub Using a P2P Network Architecture

As shown in the above figure, a P2P network architecture involves a lot of unnecessary communication between the Smart Power Outlets as they have no information necessary to share

with each other and the Hub aggregates and processes their data for the user and sends control signals to each outlet. Additionally, as space within each Smart Power Outlet for electronics is limited and the final circuitry should be kept as efficient and lean as possible to keep the cost per unit down, P2P is not the optimal networking architecture.

2.6.2.b Client-Server Networking (JG)

Client-Server networking describes a network architecture consisting of a server that responds to requests from one or many clients. Potential requests from clients include the client asking the server to perform an operation on its behalf, asking the server to retrieve data and forward it to the client, and submitting data to the server for storage. A Client-Server architecture applied to the Smart Power Outlet system would look like the following:

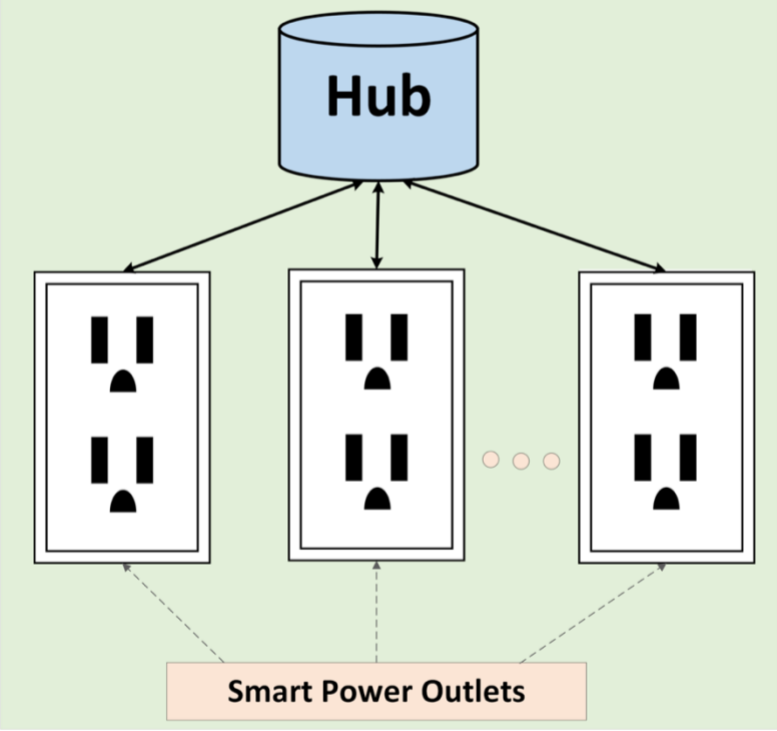

Figure 15: Communication between Smart Power Outlets and Hub Using a Client-Server Network Architecture

Using this architecture, the unnecessary communication between the Smart Power Outlets is removed. Instead, each Smart Power Outlet will only communicate to the hub on the same network. Removing the communication between Smart Power Outlets reduces the amount of traffic on the network and the amount of network data that each Smart Power outlet needs to process, making them more efficient. Another benefit of using this architecture is that each Smart Power Outlet can be programmed as if it is the only Smart Power Outlet connected to the Hub, as the Hub is the only device that each Smart Power Outlet will communicate with. Thus, the code for each Smart Power Outlet should be simple as it only needs to handle sending data to the Hub and receiving data from the hub, and the code can be reused when programming each Smart Power Outlet.

2.6.3. Network Security (JG)

Securing the computer networks used in the Smart Power Outlet system is crucial to ensure that no malicious actors can control the Smart Power Outlets in an individual's home. This could have dire consequences that range from a denial of power to devices to malicious power drain intended to waste electricity. This section aims to explore ways that the Smart Power Outlet system can be secured to help prevent these scenarios.

2.6.3.a Network Segmentation (JG)

The Computer Technology Industry Association (CompTIA), a professional certification body that issues certifications for the Information technology (IT) field, describes network segmentation as “when different parts of a computer network, or network zones, are separated by devices like bridges, switches and routers” [38]. The goal of network segmentation is to apply the “Principle of Least Privilege”, which states that users should only be given the necessary permissions to perform their job [39], to the subnetworks created by the division of the primary

network. As a result, each subnetwork will only have access to the data and necessary privileges that enable the subnet to be efficiently used. Furthermore, communication between the subnetworks can be restricted and monitored to ensure that only appropriate communication takes place, only allowing each subnetwork to communicate with the other subnetworks that are necessary for its function. This is ideal as if a subnetwork is breached, the attacker will only have the privileges granted to the subnetwork, rather than all privileges granted across the entire network, limiting what computer systems they can access and what harm they can cause. Furthermore, their ability to move laterally from subnetwork to subnetwork will be prevented as they can only communicate with connected subnetworks in their previously defined scope.

Applied to the Smart Power Outlet system, network segmentation can be used to divide the entire network of Smart Outlets, the hub, and control application into 2 parts: an internal network consisting of all the Smart Power Outlets that are connected to the hub, and an external network composed of the hub and control application. Graphically, this would look like the following:

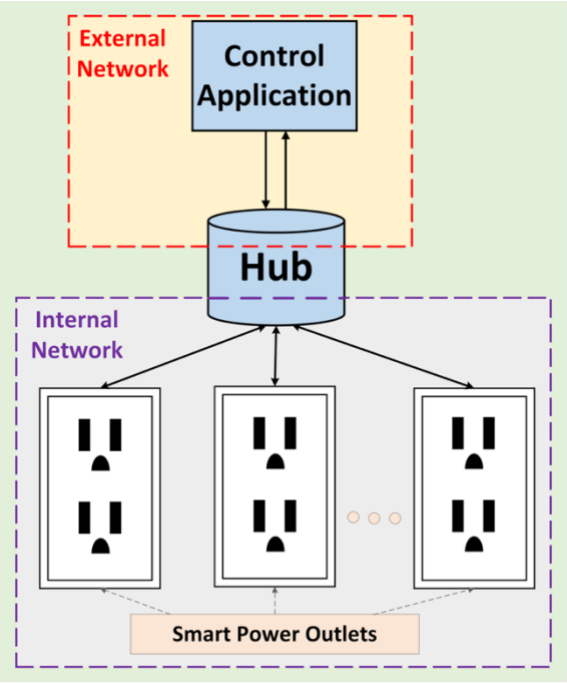

Figure 16: Network Segmentation Applied to the Complete Smart Power Outlet System

Here, the Hub would be responsible for isolating the external and internal networks from each other when processing their communications. Thus, the Hub will have an external-facing communication module for communication with the Control Application and an internal-facing communication module for communication with all connected Smart Power Outlets. The Hub will be responsible for isolating the external and internal communication models from one another, serving as a router for communication between both subnetworks. Finally, by limiting the data that can be exchanged between each subnetwork, the security of the entire network can be guaranteed.

2.6.3.b Unique Device Identifiers and Expected Values (JG)

Stemming from the idea of network separation, additional restrictions can be implemented on a network level to increase security. Primarily, the use of device identifiers or unique identifications numbers assigned to the Hub and each Smart Power Outlet can be used to create a

list of devices allowed to communicate with each other. During a setup process in which the Smart Power Outlets are connected to the Hub for the first time, each Smart Power Outlet could send the Hub its unique identifier and the Hub would respond with its unique identifier. Thus, the Hub will have a list of all connected Smart Power Outlets identifiable by their unique identifiers, and each Smart Power outlet would know the unique identifier that corresponds to the Hub.

As the Hub is the only device that should send information to each Smart Power Outlet, the inclusion of a “SOURCE DEVICE IDENTIFIER” or similar in packets should contain the Hub's identifier. Upon receiving a packet, the Smart Power Outlet will compare this field's value to the value it is expecting. If the identifiers do not match, the packet can be declared invalid and discarded. Similarly, the Hub can cross reference the “SOURCE DEVICE IDENTIFIER” of all incoming packets with its list of unique device identifiers for all connected Smart Power Outlets to determine the validity of incoming packets.

The Hub and Smart Power Outlets can also be programmed to only recognize specific commands present in received packets. If any packet contains an unrecognized instruction, the packet can be discarded. This will restrict the behavior of the Hub and Smart Power Outlets to desired behavior and prevent attacks from executing arbitrary commands on devices in the Smart Power Outlet system.

Using the previously described ideas, both the Hubs and Smart Power Outlets can be programmed to reject packets that do not contain expected values or known unique device identifiers. This strengthens the network as only acceptable communication from known devices will be allowed to occur, preventing the execution of arbitrary commands from unknown devices.

2.6.3.c Cryptography (JG)

The National Institute for Science and Technology defined cryptography as "[t]he discipline that embodies the principles, means, and methods for the transformation of data in order to hide their semantic content, prevent their unauthorized use, or prevent their undetected modification." [40]. Cryptography is based on cryptographic algorithms, which specify the procedures to modify and restore data. Two terms essential to understanding cryptography are encryption and decryption. Encryption refers to the process by which data, known as plaintext, is converted into an illegible representation of itself using the cryptographic algorithm, known as ciphertext. Conversely, decryption corresponds to the process by which ciphertext is reverted into plaintext. Ideally, a cryptographic algorithm will not be feasible for an attacker to break due to computational and time constraints, making the decryption of ciphertext possible only if the key used during its encryption is known. Cryptographic algorithms can be classified based on how keys are applied during encryption to form two categories- asymmetric cryptographic algorithms and symmetric cryptographic algorithms.

2.6.3.c.1 Asymmetric Cryptography (JG)

Asymmetric cryptographic algorithms are cryptographic algorithms that utilize two different keys- a public key and a private key. Each party has their own public key, which does not need to be kept secret, and their own private key, which must be kept secret. Using a combination of the secret private key and known public key as outlined by the selected asymmetric cryptographic algorithm, either party can encrypt data to transmit or decrypt data that they have received. Various asymmetric encryption algorithms exist such as the Rivest-Shamir-Adleman (RSA) algorithm and Elliptic Curve Cryptography (ECC). Additionally, methods for securely swapping keys exist, such as the Diffie-Hellman (DH) Key Exchange algorithm.

Using an asymmetric cryptographic scheme for the Smart Power Outlet system, each device would need to have its own public key and private key. Public keys could be exchanged freely between devices, such as each Smart Power Outlet transmitting its private key to the Hub and vice versa. Each device will need to keep its private key secret. An advantage of this scheme is that every device would be able to uniquely encrypt any data that it sends using the combinations of the other devices public key and its own private key. Any device on the network will not be able to decrypt this content unless it is the intended recipient. Thus, if a Smart Power Outlet's private key were to be stolen, an attacker could only modify and intercept the communications coming to and from the Smart power Outlet. If the Hub's private key were to be stolen, false control signals could be sent to each Smart Power Outlet and the statistics received from each Smart Power Outlet could be read. A disadvantage is that it may be computationally expensive to generate public key private key pairs for each device.

2.6.3.c.2 Symmetric Cryptography (JG)

Symmetric cryptographic algorithms are cryptographic algorithms that utilize a single shared key between all parties. When encrypting and decrypting a message, the same key must be supplied. Consequently, any party that knows the key used to encrypt a message will be able to decrypt it successfully. Examples of symmetric encryption algorithms include the Data Encryption Standard (DES) algorithm, the Advanced Encryption Standard (AES) algorithm, and Blowfish.

Applied to the Smart Power Outlet System, a symmetric cryptographic algorithm would require a single key to be shared across all devices. Using this scheme, all devices on the internal network used for communication by the Smart Power Outlet System would be able to decrypt any message sent across it. While it is simpler to implement a symmetric cryptographic than an asymmetric cryptographic algorithm due to the shared key, symmetric key algorithms provide far

less security. If the key is stolen from any Smart Power Outlet or the Hub, an attacker would be able to decrypt any messages sent across the Smart Power Outlet's network. Thus, a key stolen from any device can compromise communication across the entire network if a symmetric encryption algorithm is used.

2.7. Embedded Systems (KM)

An embedded system, regardless of its specific function, can be broadly broken down into a set of hardware components, including a CPU or microcontroller, and firmware that drives the hardware. Whereas microprocessors contain general purpose central processing units with all other components such as memory, I/O, and busses implemented externally, microcontrollers typically contain a less complex CPU with internally implemented program and data memory, several peripheral interfaces, and the busses needed to communicate between each of these [18]. Embedded systems fall under the latter of these two categories, where the components embedded in the microcontroller allow for the development of complete applications or products. Examples of I/O hardware typically included in an embedded system or microcontroller include communication ports, whether they be serial or parallel, sensors and actuators, and analog to digital or digital to analog data converters. The software or firmware included in an embedded system are typically organized into an operating system and/or application routine. Tasks, or requests are typically sent via interrupts or registers to a kernel in charge of resource management, inter-task communication, and task management. The kernel schedules and reserves resources, such as I/O, for incoming tasks based on their priority level [18].

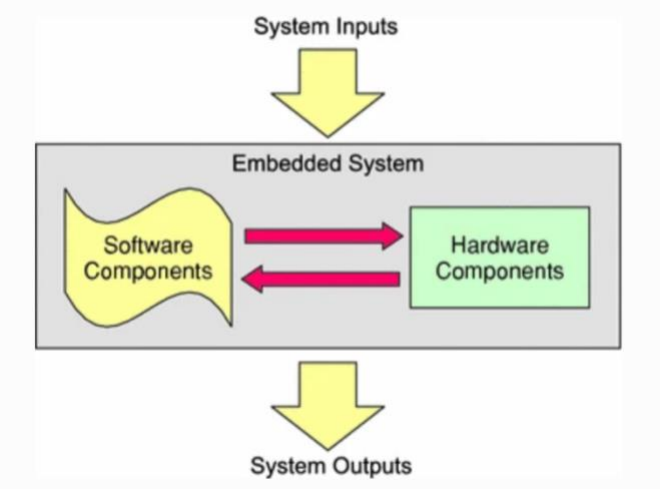

Figure 17: General Embedded System Overview from [18]

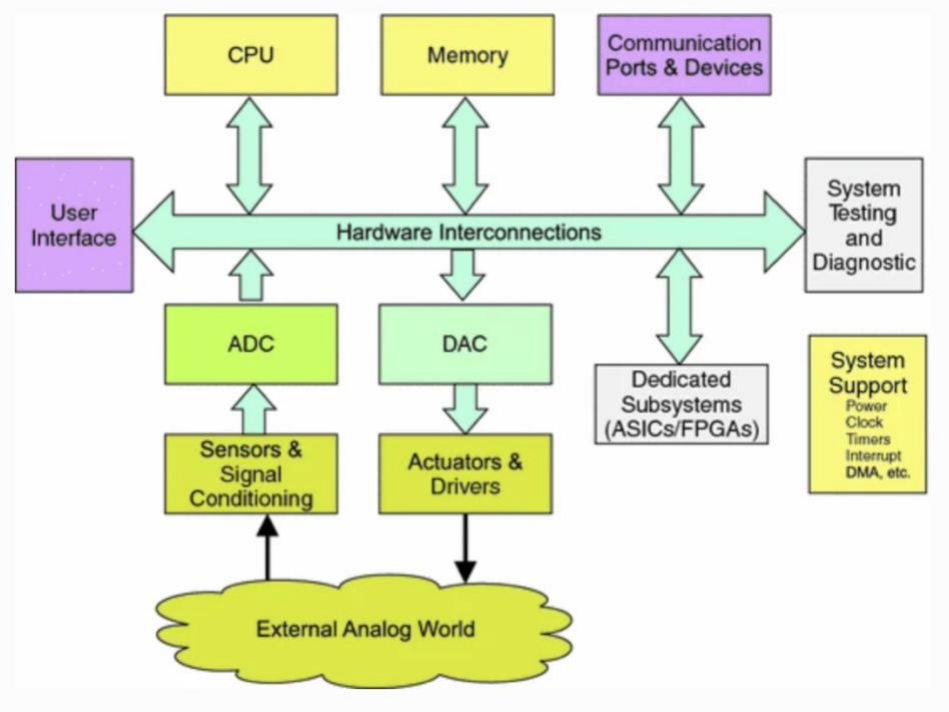

Figure 18: General Embedded System Hardware Overview from [18]

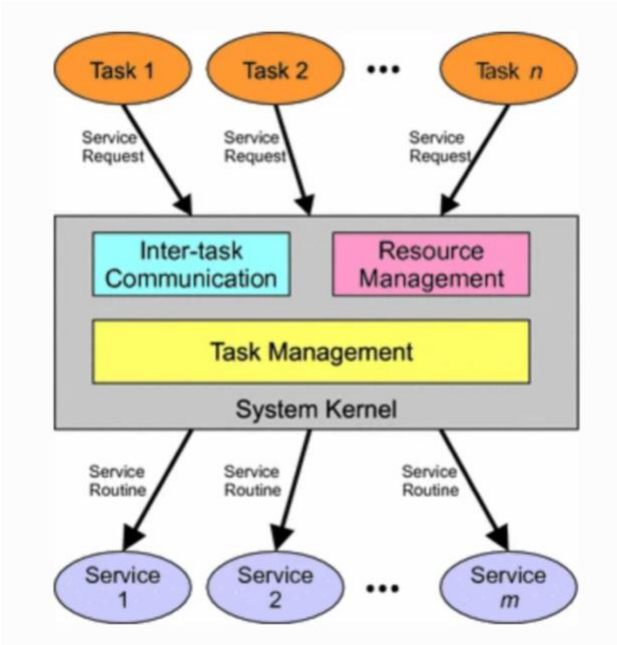

Figure 19: General Embedded System Software Overview from [18]

Embedded systems can generally be classified based on two criteria, performance, and functional requirements. When classifying based on performance, there are three classes of embedded systems, small scale, medium scale, and sophisticated or complex embedded systems. Small scale embedded systems are typically designed with an 8 or 16-bit micro-controller, can be battery powered, use few resources and processing power [41]. Medium scale embedded systems are designed with 16-bit or 32-bit micro-controllers, are faster than small scale systems, and are typically developed with a compiler, debugger, and simulator. Complex embedded systems are designed with 32 or 64-bit micro-controllers and are used to solve large-scale projects. When classifying based on functional requirements, there are four classes of embedded systems, real-time embedded systems, standalone embedded systems, networked embedded systems, and mobile embedded systems. Real-time embedded systems are used in time critical situations, such as the medical and defense sectors, where an output is expected within a given time of an input. These

real-time systems can be further split into soft and hard real time systems. Soft real-time systems are used in cases where it is permissible for a time constraint to be missed, whereas hard RT systems strictly follow time constraints. Stand-alone embedded systems can operate without a host and can produce output independently of a larger system. Network embedded systems rely on wired or wireless communication to produce output. Mobile embedded systems are small, portable, and efficient devices [41].

There are two embedded systems contained within this project. One contained within the smart outlet unit itself, and one contained within the hub controller. The outlet embedded system is responsible for several tasks. These include communication with the hub, measuring power at the receptacles by reading sensor data, toggling individual receptacles on or off, and maintaining receptacle states (on or off) as per user specified conditions. This outlet will not perform complex operations, does not need to be battery powered, and it is important to have development tools such as a compiler, debugger, and simulator. Thus, this embedded system falls under the medium scale embedded system classification. This embedded system also needs to be able to communicate wired or wirelessly therefore it falls under the network embedded system classification. Since this embedded system does not fully perform its functionality without the hub and smart phone application, it cannot be considered standalone. Its operations are also not critically time sensitive, therefore it also does not fall under the real-time embedded system classification. The hub embedded system is responsible for forwarding data to and from the outlet embedded system and the mobile phone or application. Like the outlet embedded system, it also falls under the medium and networked embedded systems classifications for the same reasons as the outlet embedded system.

3. Engineering Requirements Specification (EF, KM, JG)

Table 1: Engineering Requirements Specification

Marketing Requirements	Engineering Requirements	Justification
2	The Smart Power Outlet Module must be able to measure the voltage and current supplied to connected devices with at least 95% accuracy	Getting accurate readings for power consumption is critical for the product to be useful to consumers
4	Each Smart Power Outlet Module must operate at a maximum internal power consumption of 1 Amp AC	A low power draw for the device ensures the data gained by the consumer isn't at a net financial loss
1, 2, 3, 4, 5, 6, 7	The Smart Power Outlet System must be scalable across an entire household, with all Smart Power Outlet Modules able to communicate wirelessly to a single Hub	Scalability will allow lower costs per device and no need for excessive electronics inside the Smart Power Outlet. It also means only one Hub would be needed, lowering costs and complexity substantially
3,	The Smart Power Outlet Module must have two individually selectable outlet receptacles	This will allow the consumer more direct control over multiple devices plugged into one outlet rather than more generalized data from a single outlet
1	Each receptacle in the Smart Power Outlet Module must deliver a clean 120V, 14A, 60Hz AC signal when enabled	The Smart Power Outlet must not impede the performance of devices plugged into it by providing an incompatible power signal
2, 7	The Smart Power Outlet Module will send energy (kWh) totals to the Hub.	The consumer cannot receive the data if the Smart Power Outlet cannot send the measured data to the Hub first.
2, 7	The application must allow users to view current and past outlet energy consumption/cost statistics per Smart Power Outlet Module receptacle and for the entire household	The consumer can better manage their power usage and determine how much they have saved if they can view past statistics and compare it to present statistics
5	Each Smart Power Outlet Module must be able to communicate securely (AES 128-bit encryption) with the Hub and other Smart Power Outlets Modules using encryption	Security between outlets is important so the entire household isn't at risk of being compromised by potential attackers
5	Communication between the Smart Power Outlets and the Hub must utilize an offline Local Area Network (LAN), separate from the user's home network	Security between outlets is further enhanced by utilizing a system that is not directly accessible to unwanted users
2, 7	The Smart Power Outlet Module will optionally send live instantaneous power consumption (Watts) if requested by the mobile application.	This enables the user to see how much instantaneous power a connected device is using if they so request. This information is offered only upon request to avoid high network traffic associated with sending instantaneous power from the outlet to the phone
Marketing Requirements 1. Unit will fit in a standard wall outlet receptacle 2. Unit will measure Voltage, Current, Power Draw per outlet		

3. Unit will allow for remotely switching on and off each individual outlet
4. Unit will consume as little power as possible
5. Communication between units will be secure
6. Application will allow for setting custom power limits at each outlet
7. Application will calculate and display each outlet's usage history

4. Engineering Standards Specification (KM, EF)

Table 2:Engineering Standards

Safety	NEC
Communication	IEEE 802.15.4 (ZigBee), IEEE 802.11 (WIFI), UART, I2C
Data Formats	JSON
Design Methods	MVVM
Programming Languages	C/C++/, C#
Connector Standards	ICE 60320 (NEMA 5-15P)

5. Accepted Technical Design

5.1. Hardware Design

5.1.1. Proposed Hardware Design (KM, JG, EF)

The design for the smart power outlet system contains three major subsystems. The first subsystem being the physical outlet at which data is collected and transmitted and power is controlled. The second being the main hub which facilitates communication between the outlets and phone application. The following figure provides a top-level overview of the inputs and outputs to the overall system.

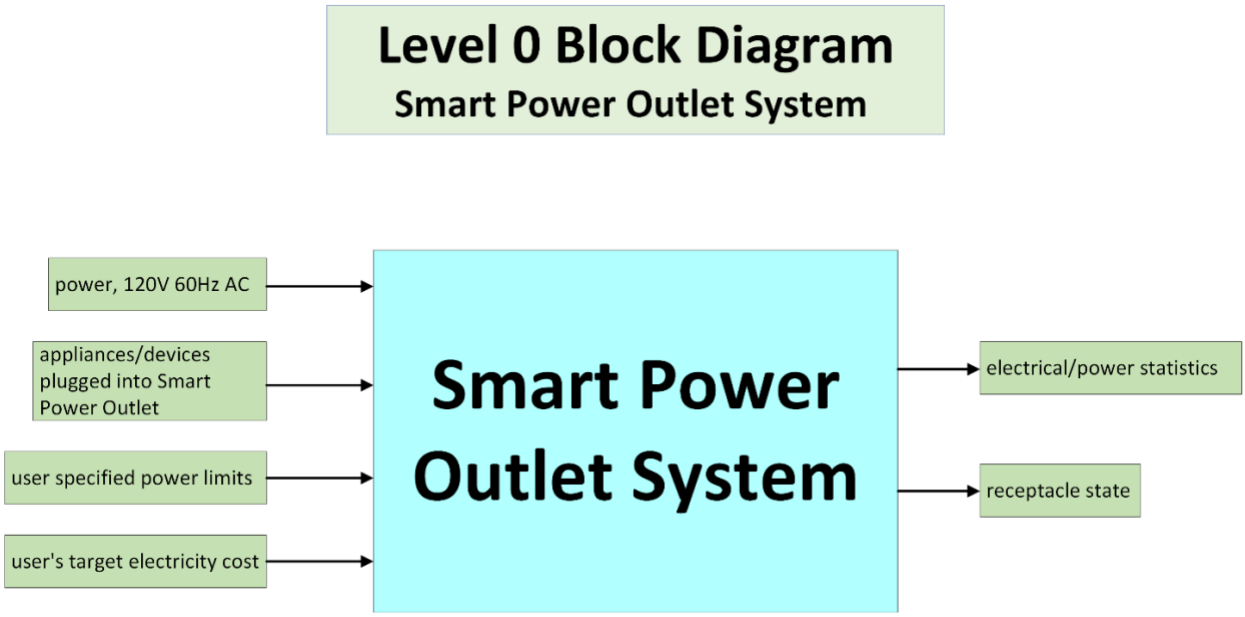

Figure 20: Level 0 System Block Diagram

Table 3: Level 0 System Functional Requirements

Module	**Smart Power Outlet System**
Designer	Madison Britton, Ethan Frese, Joseph Garro, Kyrolos Melek
Inputs	- Power: 120 volts AC rms, 60 Hz - Appliances/Devices - User specified power limits - User target electricity cost
Output	- Electrical/power Statistics - Receptacle State
Functionality	The Smart Power Outlet will control the outlet status and allow the user to view outlet statistics

The above table and figure describe the overall behavior of the smart outlet system. The inputs to the system include power, outlet connected devices, and user specified power and cost targets. The outputs produced by the system include power statistics, cost analytics, and an outlets status or state (on/off). The goal of the overall system is to provide granular control over outlet

power consumption while providing useful statistics regarding power consumption and cost. The device will meet user specified power and cost targets. Finally, the device and correlated statistics will be controlled and reported to a companion application.

As discussed above, the smart outlet system can be broken down into three major subsystems which include the physical outlet, the main hub, and the phone application. These three components and the integration between them can be seen in the below level one block diagram and accompanying table.

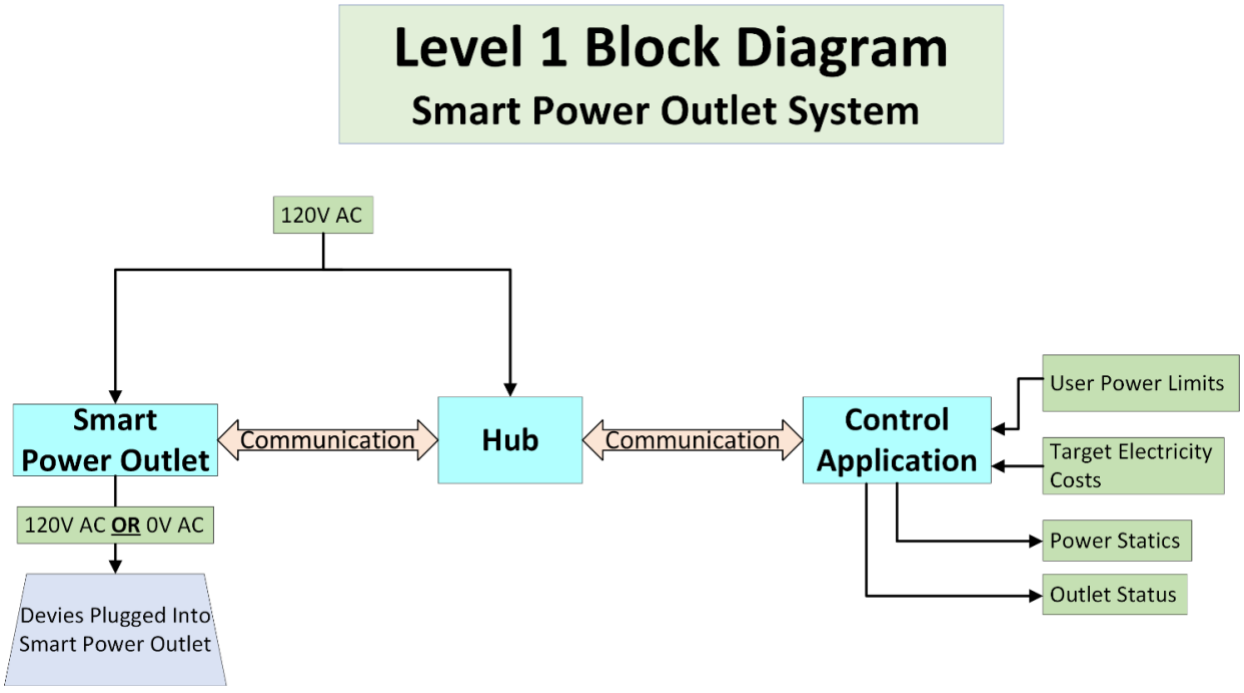

Figure 21: Level 1 System Block Diagram

Table 4: Level 1 System Functional Requirements

Module	**Smart Outlet**
Designer	Madison Britton, Ethan Frese, Joseph Garro, Kyrolos Melek
Inputs	- Power: 120 volts AC rms, 60 Hz - Devices plugged into Smart Outlet - Control signals from the hub
Output	- Power/no power to connected devices - Measurements for hub
Functionality	The Smart Outlet will measure and report power usage and toggle outlet status correspondingly
Module	**Hub**
Designer	Madison Britton, Ethan Frese, Joseph Garro, Kyrolos Melek
Inputs	- Power: 120 volts AC rms, 60 Hz - Measurements from Smart Outlets - Commands from Phone Application
Output	- Control signals to Smart Outlets - Statistics to phone
Functionality	The Hub will forward user input from application to Smart Outlet and forward power measurements from Smart Outlet to application
Module	**Control Application**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- User Power Limits - Target Electricity Costs - Power usage of outlet connected devices
Output	- Power Statistics
Functionality	The Control Application will take user input (Power/Cost Targets and outlet status) and display power usage and cost analytics

The above level one block diagram shows the integration of the three main subsystems in the smart outlet system. The first and leftmost component, the smart outlet, takes as its inputs

power at 120V AC rms 60Hz, control signals forwarded to it from the hub, and outlet connected devices. For outputs, it provides power to connected devices and power measurements which are sent to the main hub. Overall, the Smart Outlet receives control signals from the hub. The control signals dictate whether the Smart Outlet is supplying power to the connected devices. The Smart Outlet reports the power consumption of powered devices to the hub.

The next block in the above level one block diagram, the main hub, takes as its inputs, power from the wall at 120V AC rms at 60Hz, measurements from the Smart Outlet, and control signals/user input from the phone application. It produces, as its outputs, forwarded power measurements to the application and forwarded control signals or user input from the phone application to the Smart Outlet. Overall, the hub receives data from connected Smart Outlets. The hub sends control signals to Smart Outlets to control the supply of power. The hub reports data to the Control Application for viewing. The hub also accepts commands from the Control Application that are used to guide smart Outlet behavior.

Finally, the last module depicted in the above level one block diagram is the phone application. This module takes, as its inputs, user input including specified power and cost limits, and outlet status changes, as well as power usage measurements forwarded from the hub. It produces power usage and costs and displays them to the user. Overall, the control application is used to receive measured data from the outlets and display them in a legible manner to the user. The Control Application is also used to control the behavior of the Smart outlets by allowing users to set power limits per outlet and their desired electricity bill cost.

The following level two block diagram and accompanying table provide a more in-depth view of the main hub subsystem, which is responsible for data forwarding between the several

connected outlets and phone application. The modules within this subsystem include an AC/DC converter, two isolated communication models, and an embedded system.

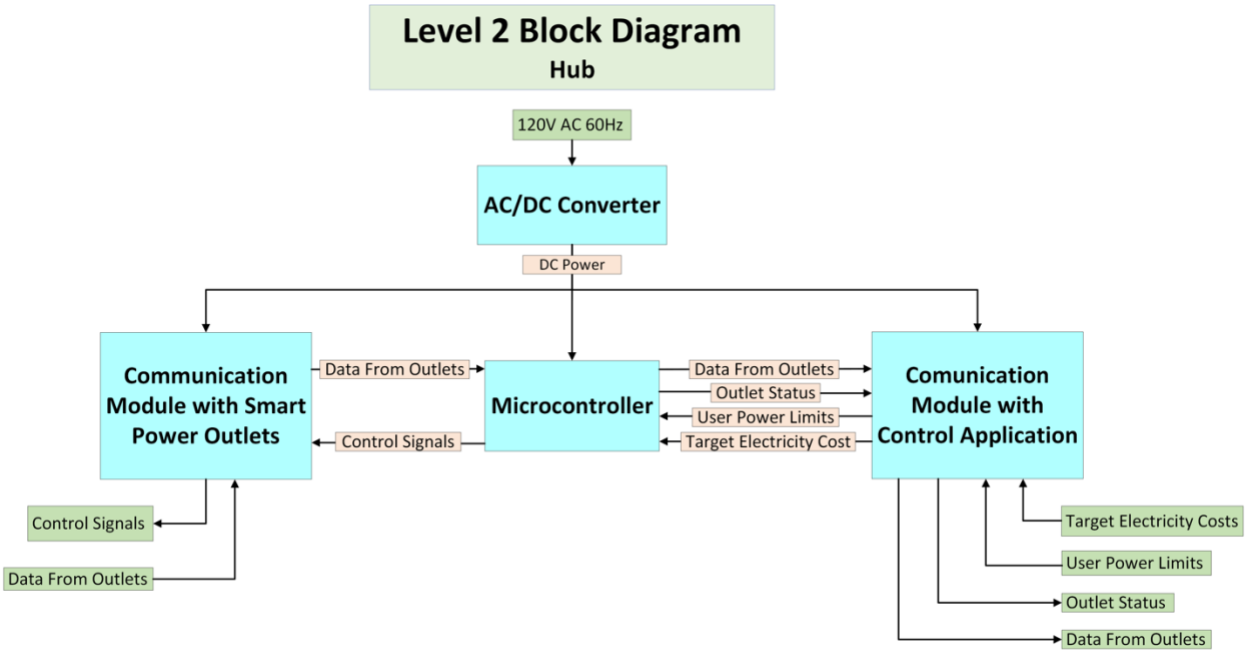

Figure 22:Level 2 Hub Block Diagram

Table 5: Level 2 Hub Functional Requirements

Module	**AC/DC Converter**
Designer	Madison Britton
Inputs	- Power: 120 volts AC rms, 60 Hz
Output	- 5V DC to Outlet Communication Module - 5V DC to Hub Microcontroller - 5V DC to Phone Communication Module
Functionality	The AC/DC converter converts 120V 60HZ AC signal to 3 DC voltages to power other electronics within the Hub.
Module	**Microcontroller**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - Data from Outlets - User's Power Limits - User's Target Electricity Cost
Output	- Control Signals to Outlets - Data from Outlets - Outlet Status
Functionality	The Microcontroller forwards power measurements to application and forwards user input (power/cost targets, and outlet states) to the Smart Outlets
Module	**Communication Module for Smart Outlets**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - Data from Smart Outlets - Control Signals to Smart Outlets
Output	- Data from Smart Outlets - Control Signals to Smart Outlets

Functionality	The Communication Module for Smart Outlets handles communication to/from Smart Outlets

Module	**Communication Module for Phone App**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - Data from Smart Outlets - Smart outlet Status - User Power Limits - User Target Electricity cost
Output	- User Power Limits - User Target Electricity cost
Functionality	The Communication Module for the Phone App handles communication to/from phone application

In the above hub level 2 block diagram, the microcontroller is responsible for controlling the behavior of connected Smart Outlets by processing input from the Smart Outlets and the user via the Control Application. The microcontroller will process the states of Smart Outlets and power statistics received from the Smart Outlets for viewing in the Control Application. Received measurements are used to control the behavior of connected Smart Outlets with comparisons to user supplied power limits and target electricity costs to provide automated control over connected Smart Outlets.

The module that communicates with Smart Outlets is responsible for receiving power statistics from outlets and forwarding them to the microcontroller for processing. Control signals produced by the microcontroller are sent to the Smart Outlets using this module as well. This

module needs to be directly compatible with the communication module found in the Smart Outlets.

The module that communicates with control application is responsible for accepting user power limits and target electricity costs to guide the behavior of the Smart Outlets. This module will forward these user inputs to the microprocessor. Additionally, power statistics from the Smart Outlets and the status of the Smart Outlets which are processed by the hub's microcontroller be sent to the control application using this module.

The following level two block diagram and accompanying table provide a more in-depth view of the main Smart Outlet subsystem. Which is responsible for calculating or measuring power of connected devices and toggling outlets states based on stored user inputs. The modules within this subsystem include an AC/DC converter, measuring and control circuitry, and a communication module.

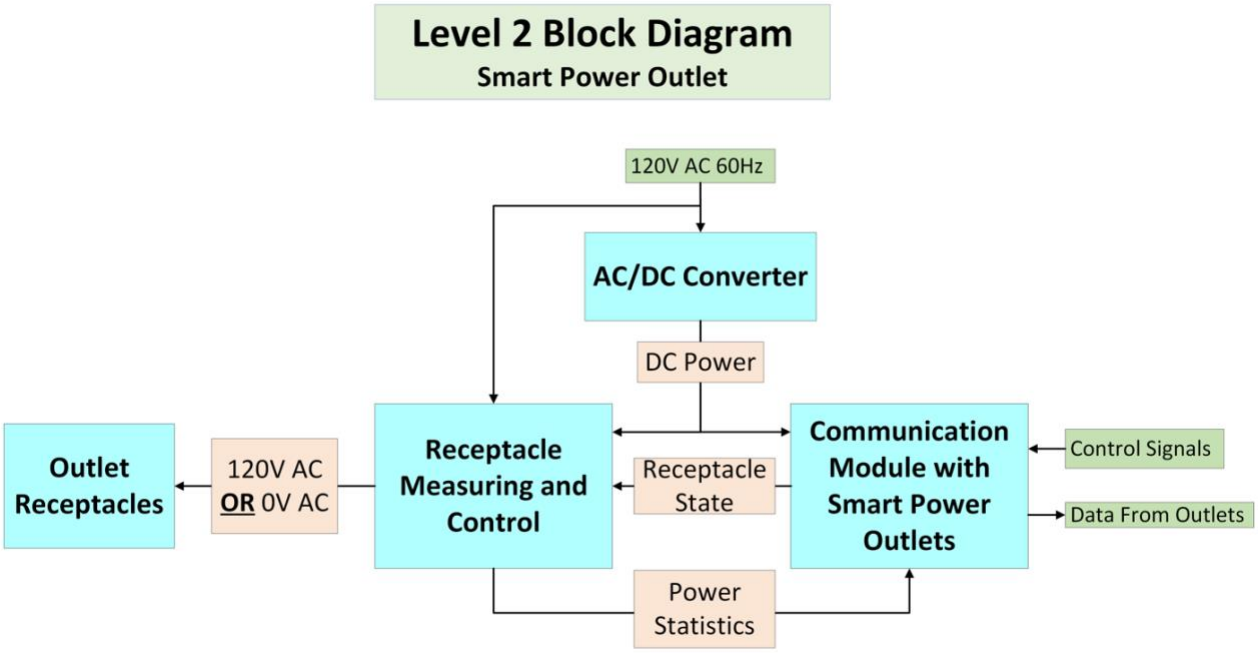

Figure 23:Level 2 Smart Power Outlet Block Diagram

Table 6: The Level 2 Smart Power Outlet Functional Requirements

Module	**AC/DC Converter**
Designer	Madison Britton
Inputs	- Power: 120 volts AC rms, 60 Hz
Output	- 5V DC to Hub Communication Module - 5V DC to Receptacle Measuring and Control Module
Functionality	- Converts a 120V 60HZ AC signal into 2 DC voltages
Module	**Communication Module for Hub**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - Data from Smart Outlets - Control Signals From Hub
Output	- Power Statistics - Receptacle State
Functionality	Handles communication to/from Hub

Module	**Receptacle Measuring and Control**
Designer	Ethan Frese, Madison Britton, Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - 120V AC 60HZ - Receptacle State
Output	- Power Statistics - 120V AC 60HZ
Functionality	Measures power of outlet connected devices

Module	**Outlet Receptacles**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ
Output	- 120V AC 60HZ **OR** - 0V AC
Functionality	Toggles power supply on/off per user input

In the above level two block diagram for the smart outlet, the AC/DC converter module takes in a 120V 60HZ AC power signal and converts it into two DC voltages to power other electronics systems within the Smart Outlet.

The receptacle measuring and control module is responsible for measuring the power draw of each receptacle and controlling the state of each receptacle in the Smart Outlet. If the receptacle is supposed to be on, 120V AC power will be passed through to the appropriate Smart Outlet receptacles. If a receptacle is supposed to be off, it will not output any power.

The module that communicates with the hub receives power statistics from the receptacle measuring and control system in the Smart Outlet and forwards them to the Hub for processing.

This module also receives control signals from the Hub that are sent to the receptacle measuring and control system to control the state of the Smart outlet's receptacles.

The receptacles in the Smart Outlet will be independently controlled by the Hub. If the receptacle is supposed to output power, it will output 120V AC 60HZ. If the receptacle is supposed to be off, it will output 0V AC.

The following level three block diagram takes a deeper look at the modules involved in the power measuring and control circuity found in the Smart Outlet subsystem. The modules included in this block diagram include top and bottom receptacle switches, top and bottom power measuring circuitry, an AC power splitter, and a microcontroller.

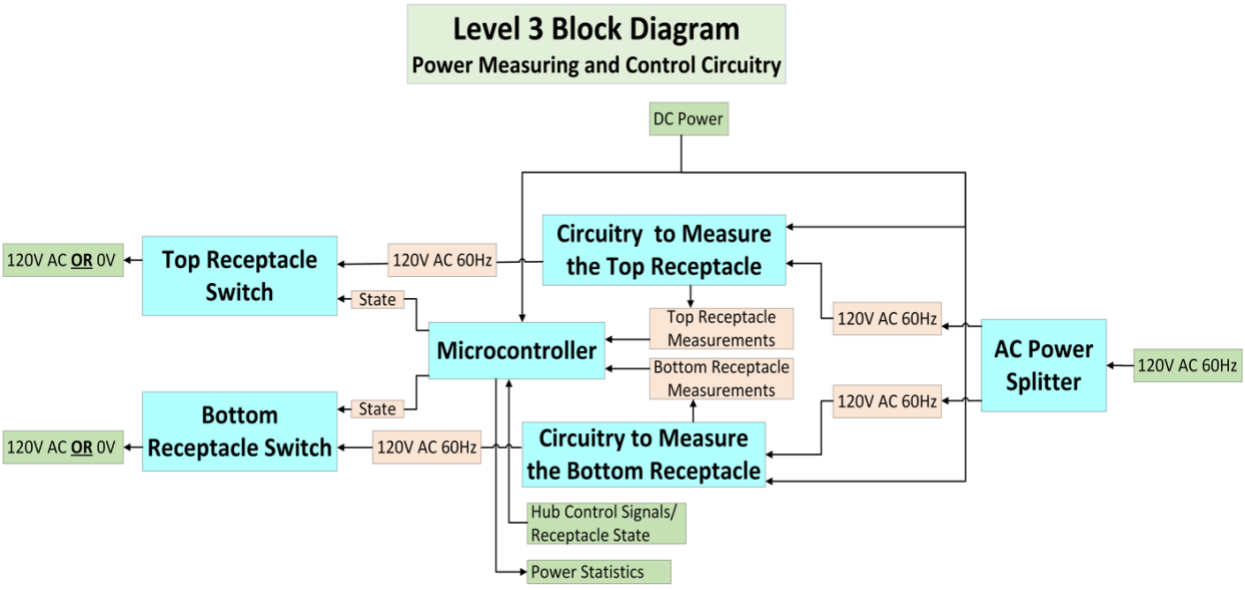

Figure 24: Level 3 Smart Power Outlet Measuring and Control Circuitry Block Diagram

Table 7: The Level 3 Smart Power Outlet Measuring and Control Circuitry Functional Requirements

Module	**AC Power Splitter**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ
Output	- 2 Wires with 120V AC 60HZ Power
Functionality	The AC Power Splitter takes a single 120V AV 60HZ input and splits it into 2 120V AC 60HZ inputs
Module	**Circuitry to Measure the Top Receptacle**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ
Output	- Power Consumption of Top Receptacle
Functionality	Measures power consumption of top receptacle in Smart Outlet and outputs measurements to Microcontroller
Module	**Circuitry to Measure the Bottom Receptacle**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ
Output	- Power Consumption of Bottom Receptacle
Functionality	Measures power consumption of bottom receptacle in Smart Outlet and outputs measurements to Microcontroller
Module	**Top Receptacle Switch**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ - Receptacle State
Output	- 120V AC 60HZ **OR** - 0V AC
Functionality	Opens and closes per user input

Module	**Bottom Receptacle Switch**
Designer	Ethan Frese, Madison Britton
Inputs	- 120V AC 60HZ - Receptacle State
Output	- 120V AC 60HZ **OR** - 0V AC
Functionality	Opens and closes per user input
Module	**Microcontroller**
Designer	Joseph Garro, Kyrolos Melek
Inputs	- 5V DC - Top Receptacle Measurements - Bottom Receptacle Measurements - Hub Control Signals
Output	- Top Receptacle State - Bottom Receptacle State - Power Statistics
Functionality	Reads power measurement sensors/circuitry, receives control signals from communication module, forwards power measurements to communication module

In the above level 3 block diagram, the AC Power Splitter takes a single 120V AV 60HZ input and splits it into 2 120V AC 60HZ inputs. The measuring circuitry for both the bottom and top receptacle will measure the power consumption of the top/bottom receptacle in the Smart Outlet. The measured value will be output to a microcontroller in the Smart outlet for processing. The switches for the top and bottom receptacles will open and close in response to the user input

to allow the receptacles to supply or not supply power. When open, the top receptacle will output 0V AC, and when closed, the top receptacle will output 120V AC 60HZ.

The microcontroller will be used to process the measurements for power consumption of the top and bottom receptacles, as well as to control the state of each receptacle. Control signals received from the hub will be used to control the state of each receptacle. The processed power statistics will be forwarded from the microcontroller to the communication module in the Smart Outlet for transmission to the hub.

5.1.2. Realization of Proposed Hardware Design

5.1.2.a Smart Power Outlet (JG)

To implement the Smart Power Outlet, four key subsystems are used: a motherboard containing the ESP32 and XBee communication module, measuring circuitry to measure the power consumption of an attached load, an AC/DC converter to power the Smart Power Outlet, and the relays used to control the state of the Smart Power Outlet's receptacles. The following sections describe each component of the Smart Power Outlet:

5.1.2.a.1 Motherboard (JG)

The motherboard for the Smart Power Outlet utilizes a surface mount ESP32-S3-WROOM-1U-N16R8 microprocessor and a surface mount XB3-24Z8UM-J XBee module for communication. The motherboard features two separate JST-PHD headers to connect the AC/DC subsystem and the measuring circuitry subsystem. This modular design allows the motherboard to be swapped independently of the AC/DC conversion system or measuring circuitry (or vice versa), and work to be done on all 3 components independently. Additionally, should one subsystem break, this design allows it to be replaced without rebuilding the entire Smart Power Outlet system.

The schematic for the motherboard, the PCB layout corresponding to it, and a 3D render of the PCB are shown in the following three figures:

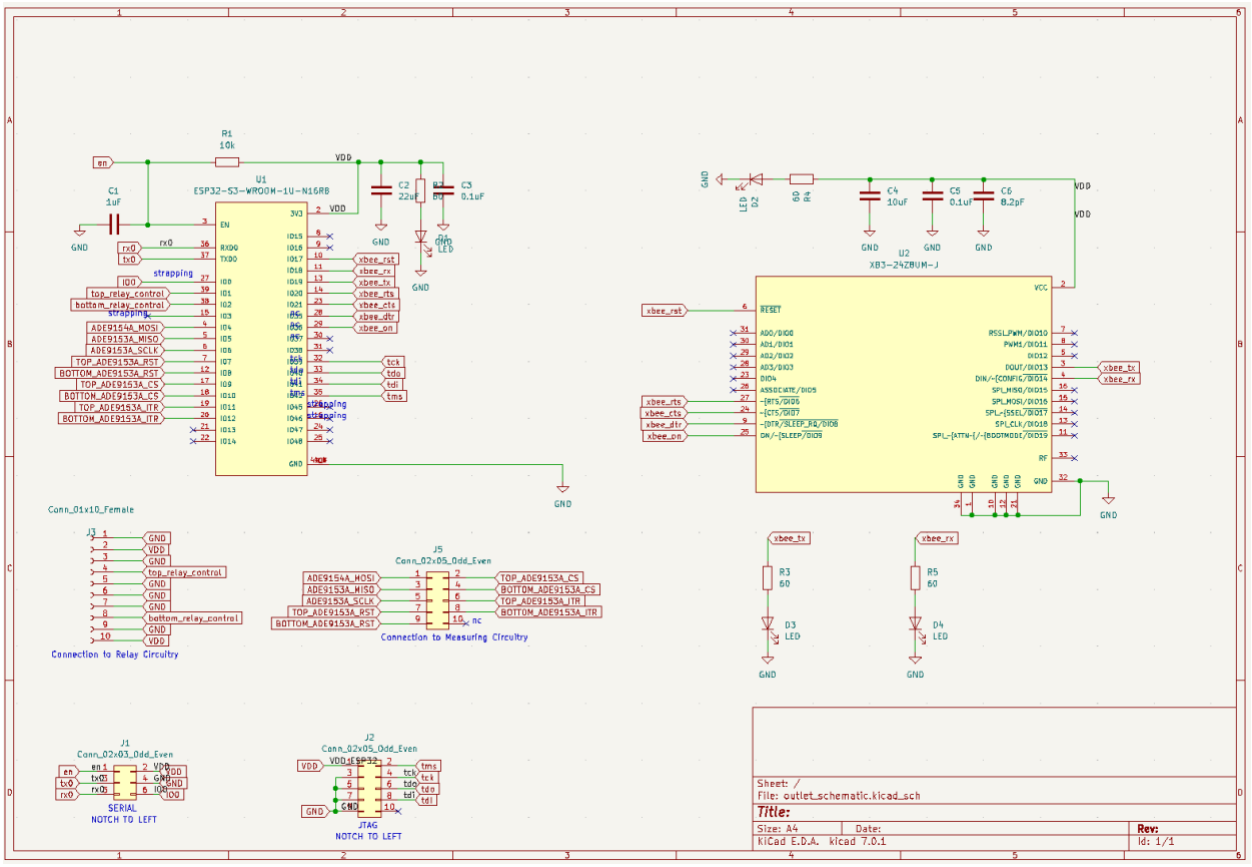

Figure 25:Smart Power Outlet Motherboard Schematic

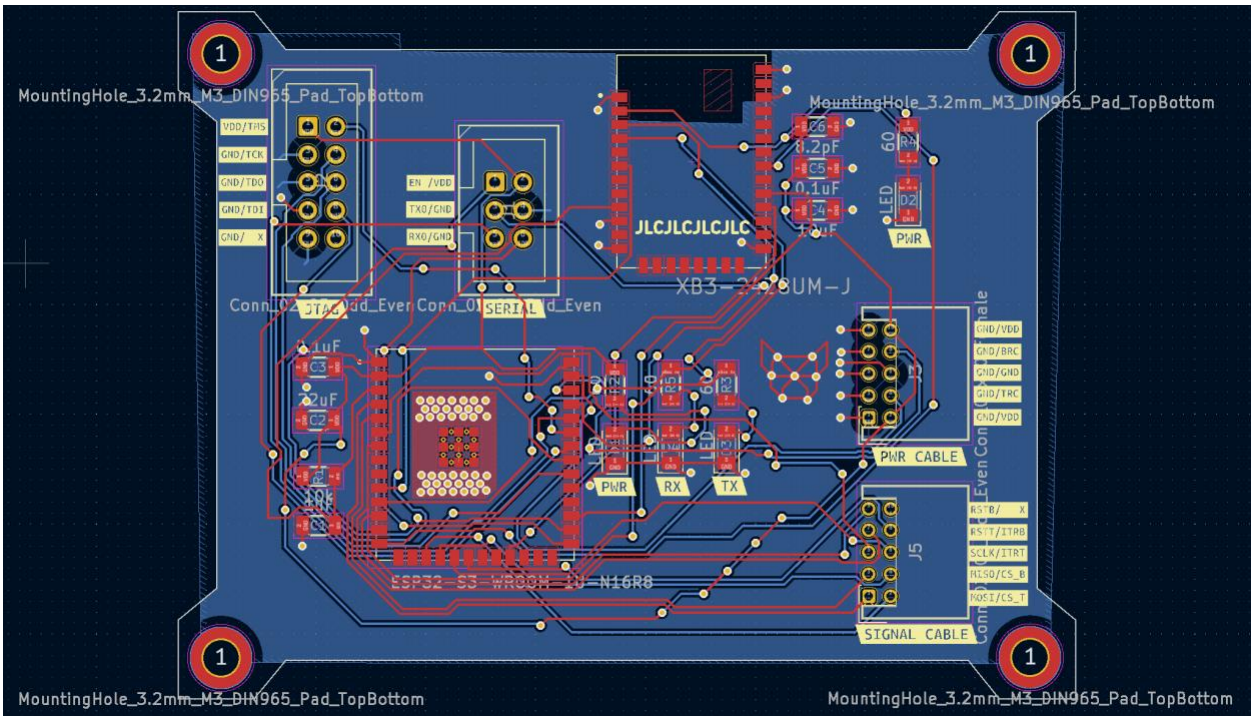

Figure 26: Smart Power Outlet Motherboard PCB Layout

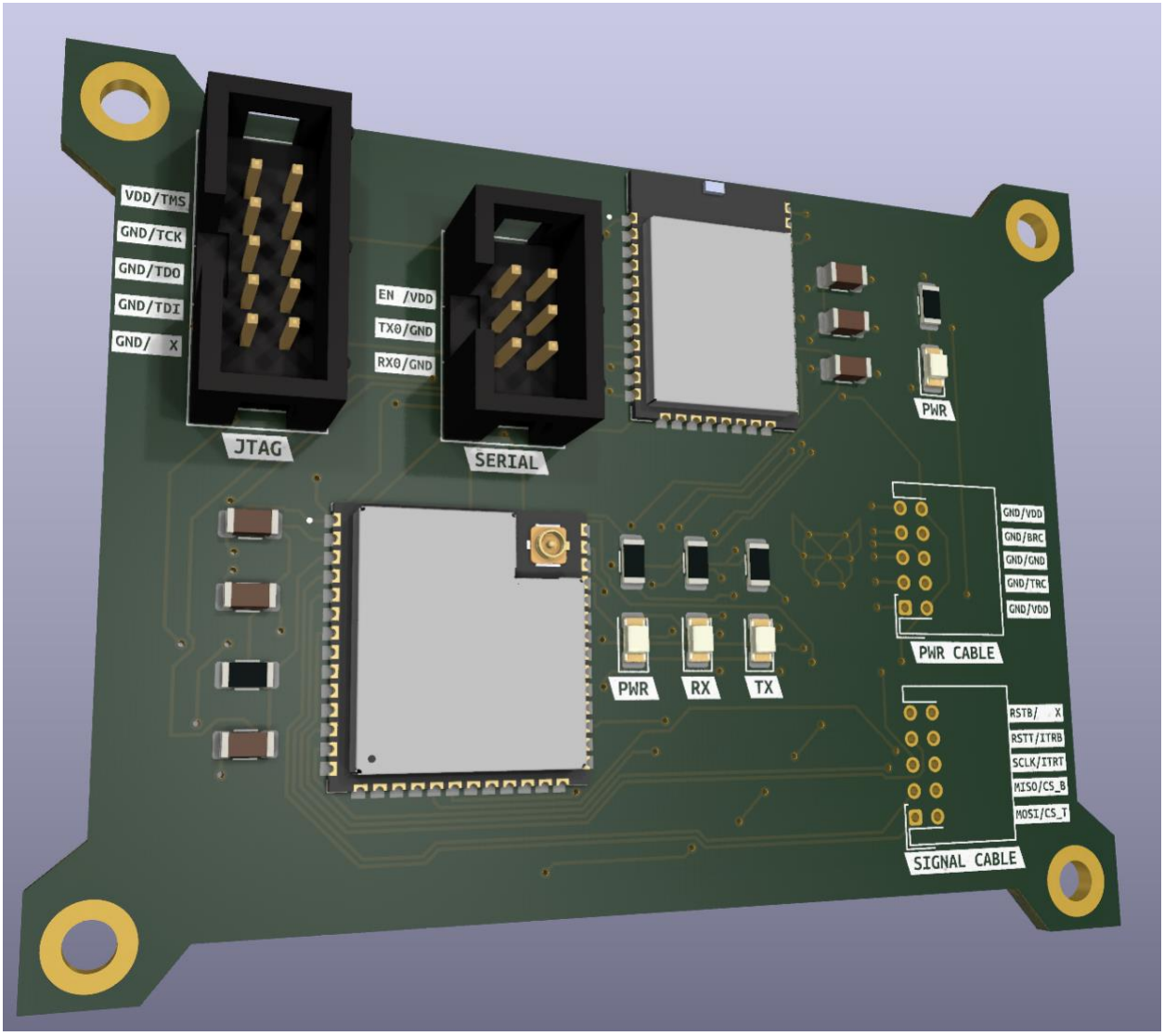

Figure 27: Smart Power Outlet Motherboard 3D Render

As shown in the schematic, multiple debug LEDs are included to show whether the ESP32 and XBee module are powered, and whether data is being transferred or received. Additional headers are provided to enable JTAG debugging and serial flashing using the ESP-PROG- a serial programmer and JTAG Debugger for ESP32 microcontrollers.

The PCB layout shows the design of the motherboard, showing the location of each component and the connections between them. Present in the layout are the ESP32, XBee, supporting components for each device, and spots for each header. The traces connecting the components are visible, as is the large rear copper pour that serves as the ground for this PCB. The final thing to note on the PCB is the inclusion of four 3.2mm M3 mounting holes so that the PCB can be mounted easily.

Finally, the render shown in the final image is a depiction of what the board was expected to look like once assembled.

5.1.2.a.2 Measuring Circuitry (EF)

The proposed hardware design for the Smart Power Outlet's measuring circuitry is outlined using the below schematic.

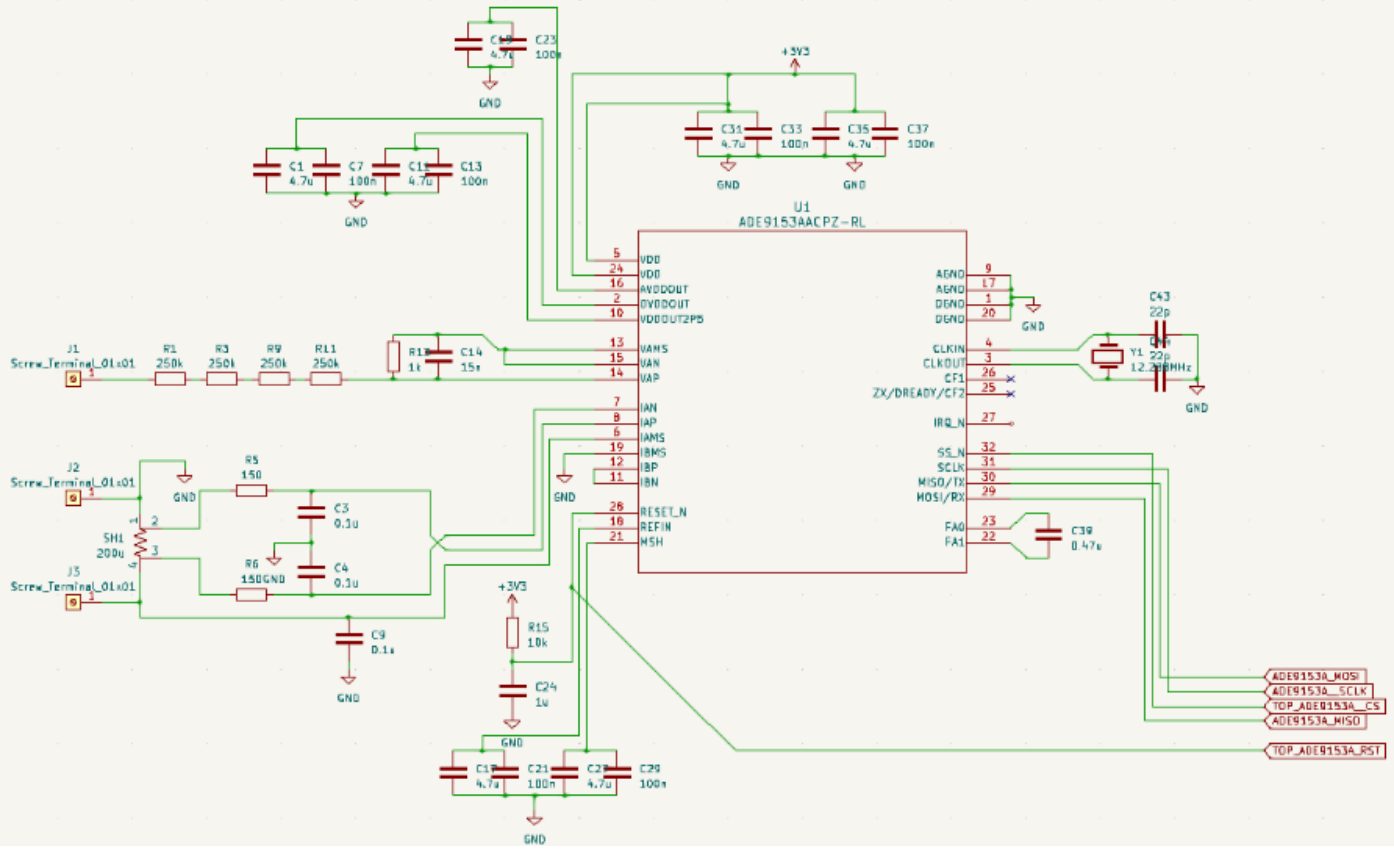

Figure 28: ADE9153A Schematic

At its core, the measuring circuitry utilizes the Analog Devices ADE9153A chip to perform all voltage and current measurements. The voltage measurements are taken from the voltage difference between the VAP and VAN pins of the ADE9153A chip, which has a voltage divider circuit before it to step down the 120VAC to suitable levels. A screw terminal in the circuit breaks off into the neutral line that goes to the outlet. The current measurements are taken from the voltage difference between the IAP and IAN pins of the ADE9153A chip, using a known resistance coming from a shunt resistor setup. The additional 150-ohm resistors are for antialiasing purposes. The

VAMS and IAMS pins are used for the *mSure* calibration feature that comes with the ADE9153A chip. The chip takes a 3.3V source as power and uses up to 15mA during operation.

5.1.2.a.3 AC-DC Conversion Circuitry (MB)

The proposed hardware design for the AC-DC conversion circuitry is shown in the below schematic.

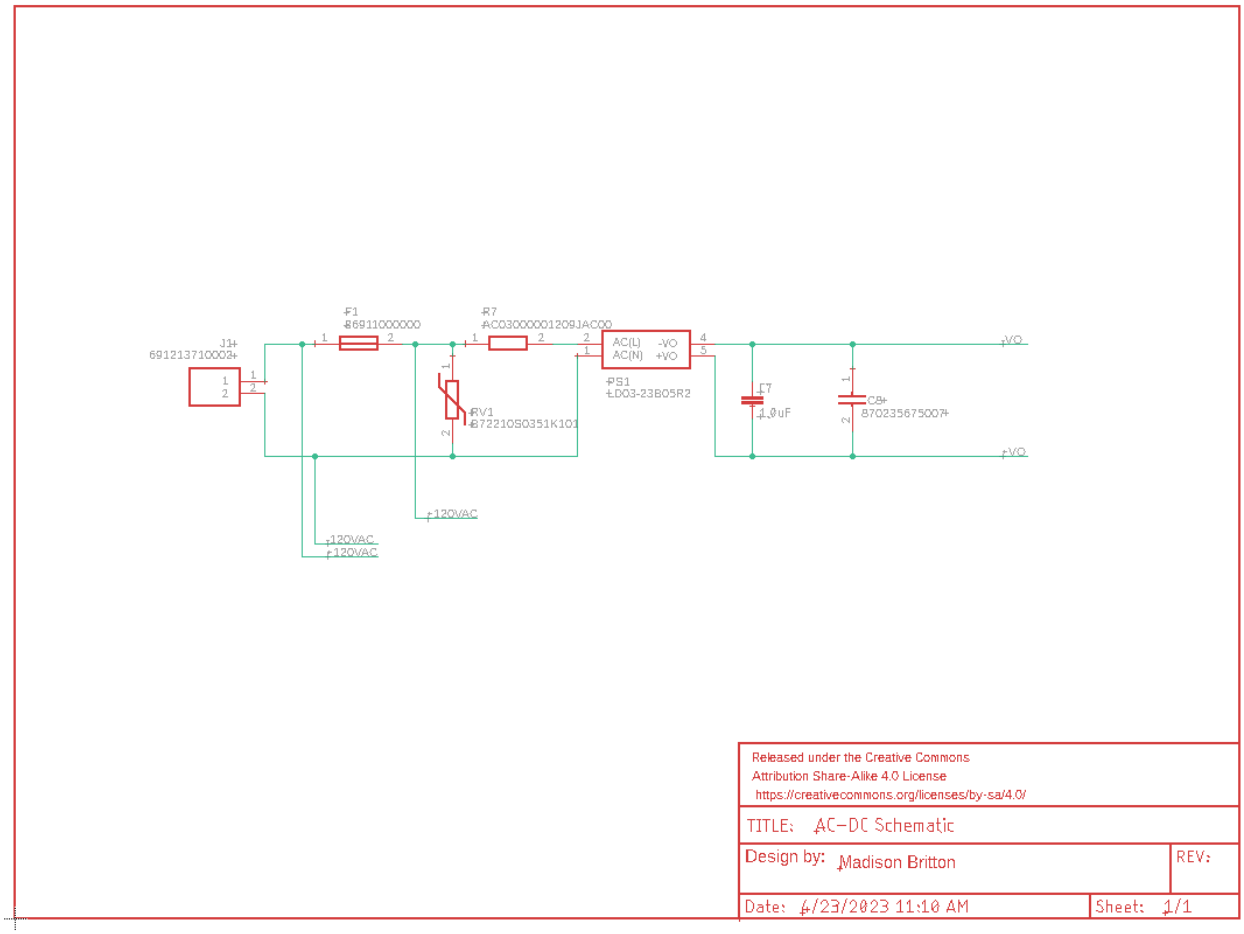

Figure 29: AC-DC Conversion Circuit Schematic

The central component of the AC-DC conversion circuit is the LD03 -23BO5R2, which is an AC-DC converter component that converts 120VAC into 5VDC. The AC-DC conversion circuit implements some other components, mostly for safety and for cleaning up noise. The first

component that is encountered when reading the schematic from left to right is a screw terminal, which allows for 120VAC to go into the circuit from some external point such as the wall. Next, there is a 1A/300V slow-blow fuse that protects the circuit from taking on too much current. After the fuse is a S10K350, which is a varistor that protects the circuit from high voltage surges. The next component in the circuit is a 12Ω/3W wire-wound resistor that is used to limit the current flow in the circuit. Then, after the AC-DC converter that was already explained, the next component is a 1uF ceramic capacitor used for filtering high-frequency noise. Lastly, there is a 150uF electrolytic capacitor that is used to reduce the voltage ripple. The input of this circuit is 120VAC and the output of this circuit is 5VDC.

5.1.2.a.4 Relay Circuitry (MB, EF, JG)

The circuit displayed in the below schematic depicts the relay circuit.

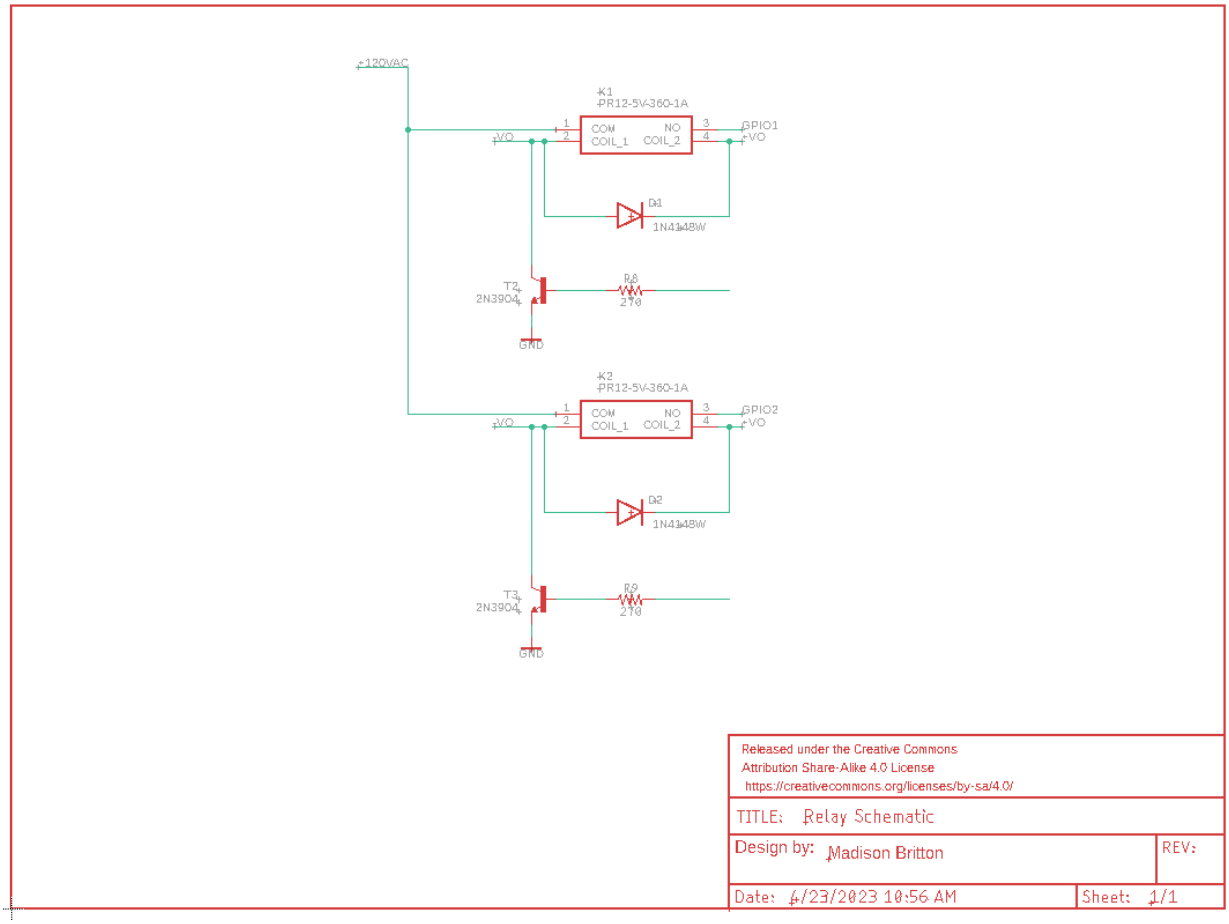

Figure 30: Relay Circuit Schematic

The relay circuit functions as a means to deactivate and reactivate the Smart Power Outlet's receptacles as needed. The relay itself takes a signal from the ESP32 device using a GPIO pin that will determine if the relay will stay open, cutting off power to the circuit, or close, providing power to the circuit. The relay will only affect power going to the sockets of the outlet, as power still needs to be provided to the ESP32 microcontroller in order for the module to function correctly.

The chosen relay has a minimum switching load of 5V DC at 100mA, which is much larger than the 20mA to 40mA source current of the ESP32's GPIO pins. Thus, an alternative circuit needs to be constructed to control the relay. To achieve this, a 2N3904 NPN BJT is used as a switch. When the relay is closed, it has a resistance of 80Ω which makes the BJT's collector current

equal to 5V/80Ω = 62.5mA. Setting the target collector current to 110mA, it can be determined that the BJT needs to have a gain greater 5V * $\frac{110mA}{20mA}$, or a gain of at least 27.5. According to the 2N3904's data sheet, a 2N3904 operating at 25C with a collector-emitter voltage of 5V has a gain of approximately 70, which satisfies the required gain. Finally, a diode is placed across the terminals of the relay's coil in a reverse direction to protect GPIO pin of the ESP32 connected to the relay circuit against sudden spikes in voltage when the relay is switched.

5.1.2.a.5 DC-DC Conversion Circuitry (MB)

The DC-DC conversion circuit is depicted in the below schematic.

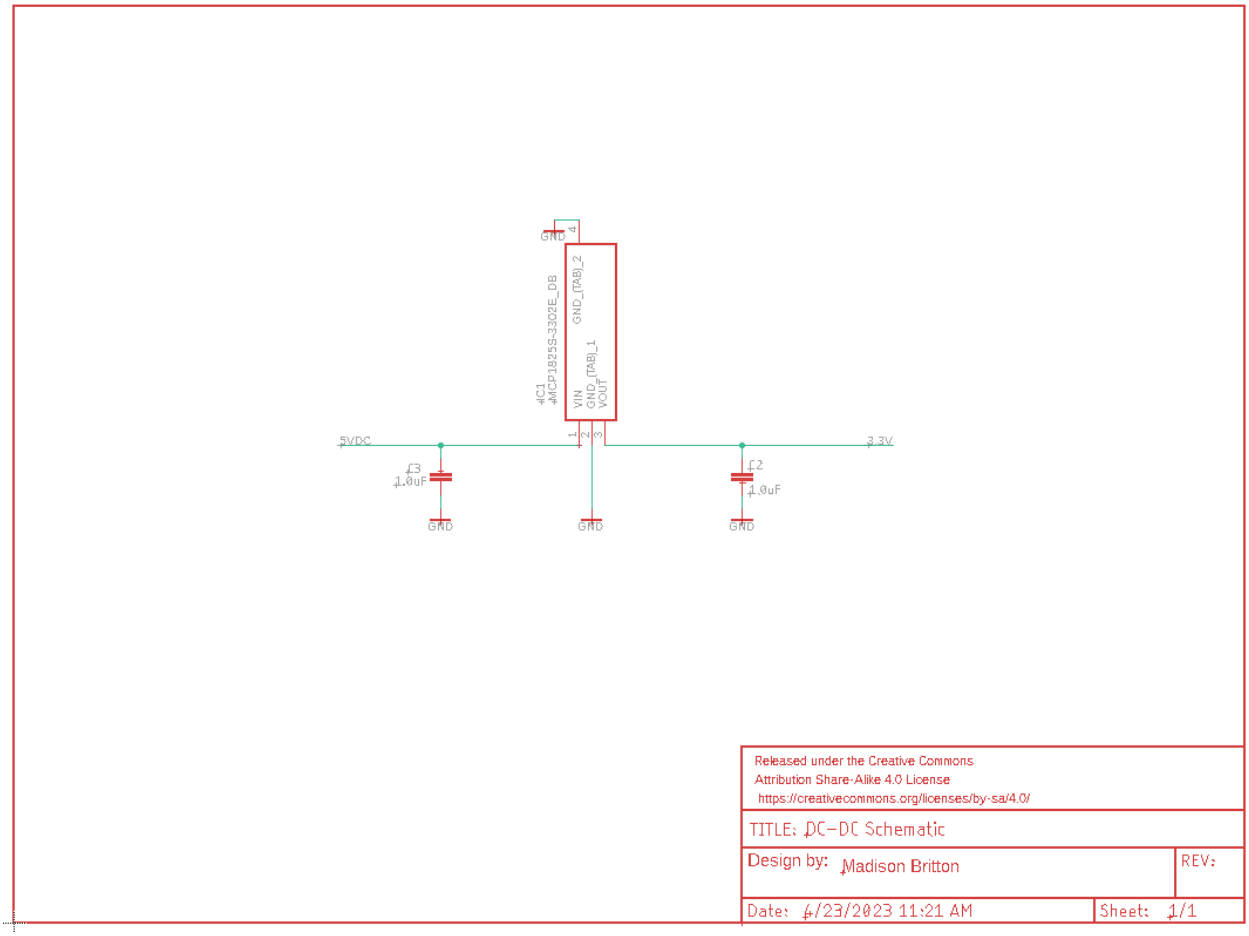

Figure 31: DC-DC Conversion Circuit Schematic

The main component of the DC-DC conversion circuit is the MCP1825S-3302E/DB, which is a low-dropout (LDO) linear regulator that provides high current and low output voltages. This LDO chip is what converts the 5VDC that comes from the AC-DC conversion circuit to the 3.3VDC that we need to power our ESP32. The DC-DC conversion circuit is also uses two 1uF capacitors that act as decoupling capacitors. The function of this DC-DC conversion circuit is to convert 5VDC that comes from the AC-DC conversion circuit and convert it to 3.3VDC which will power the ESP32 modules.

5.1.2.b Hub (JG)

Like the Smart Power Outlet, the Hub utilizes a surface mount ESP32-S3-WROOM-1U-N16R8 microprocessor and a surface mount XB3-24Z8UM-J XBee module for communication. The Hub is a much simpler system though, as it is self-contained and does not measure anything, instead acting as the intermediary between multiple Smart Power Outlets and the control application. To power the Hub, a USB C connector containing only power pins is utilized in conjunction with an LDO to generate a 3.3V DC voltage source. The use of USB C allows the Ac/DC Conversion to take place outside of the Hub in a wall brick, simplifying its internals. USB C was selected over other USB types or connectors, such as barrel jacks, as it is a modern standard and the reversible plug is convenient. Finally, as the Hub is not enclosed in a wall like the Smart Power Outlets, it was designed to be placed in an enclosure. For this, a black aluminum rectangular enclosure was purchased to provide an aesthetically appealing design.

5.1.2.b.1 Hub Motherboard (JG)

The Hub’s motherboard is like the Smart power Outlet’s, although it does not contain headers for connections to AC/DC circuitry or measuring circuitry. Instead, the PCB contains the USB C port and a 5V to 3.3V LDO to step down the voltage and provide a consistent output

current. The schematic for the Hub's motherboard, the PCB layout corresponding to it, and a 3D render of the PCB are shown in the following three figures:

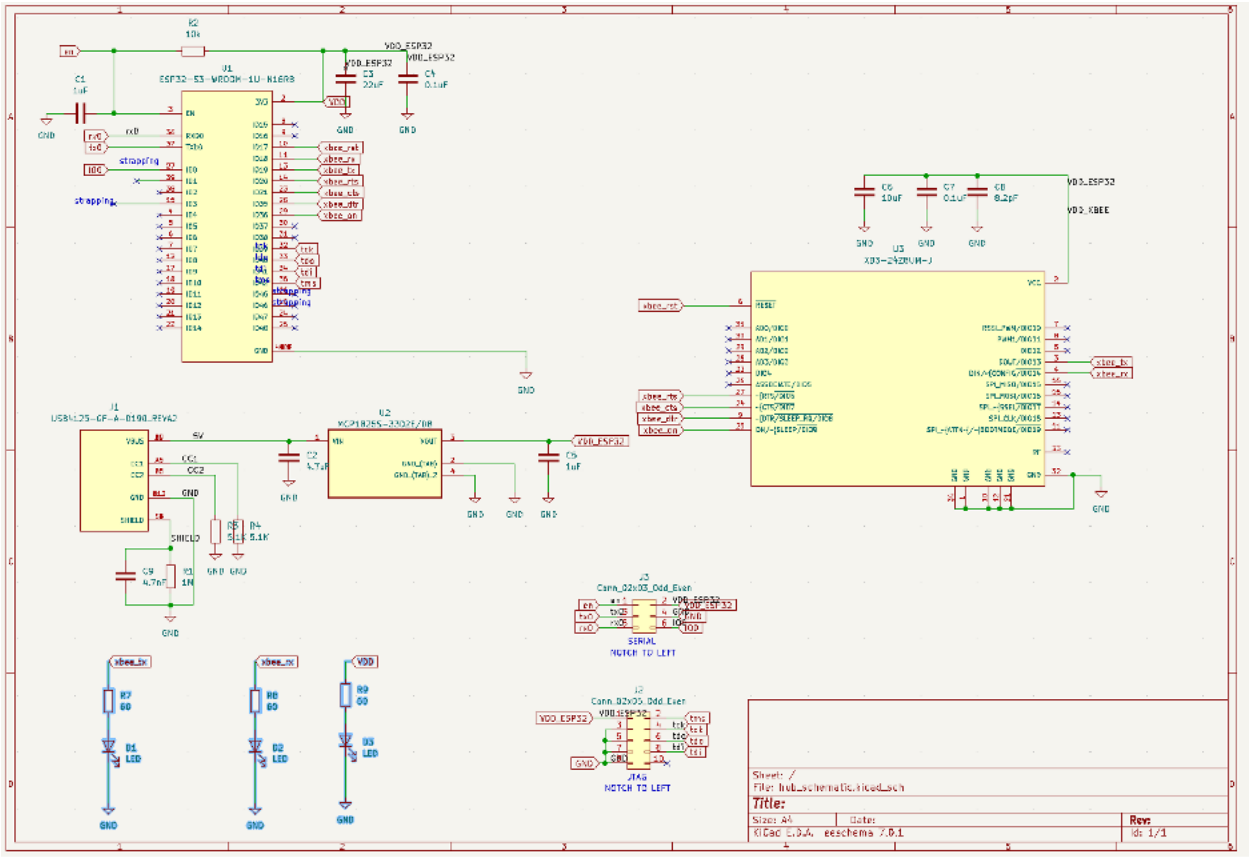

Figure 32: Hub Motherboard Schematic

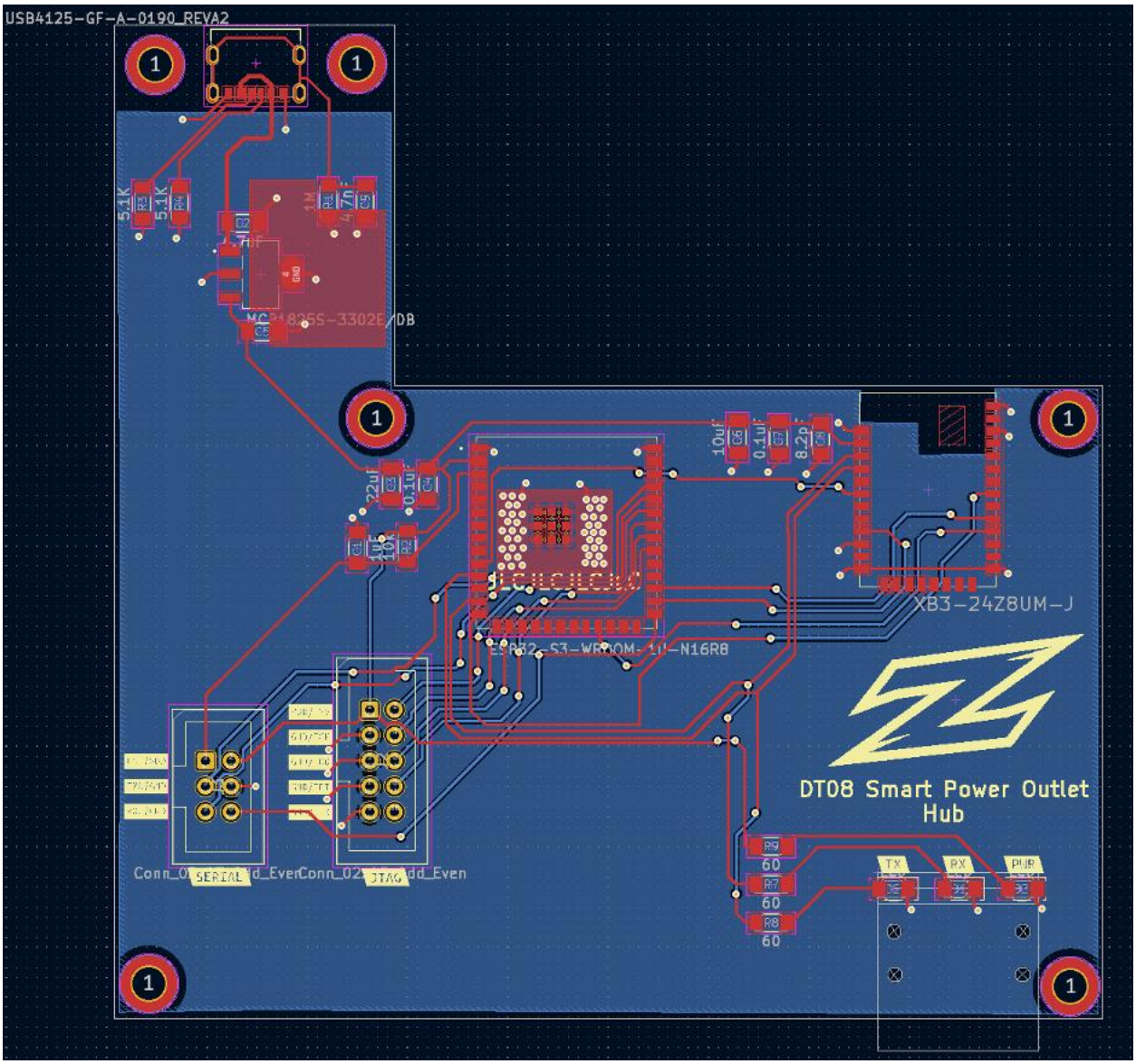

Figure 33: Hub Motherboard PCB Layout

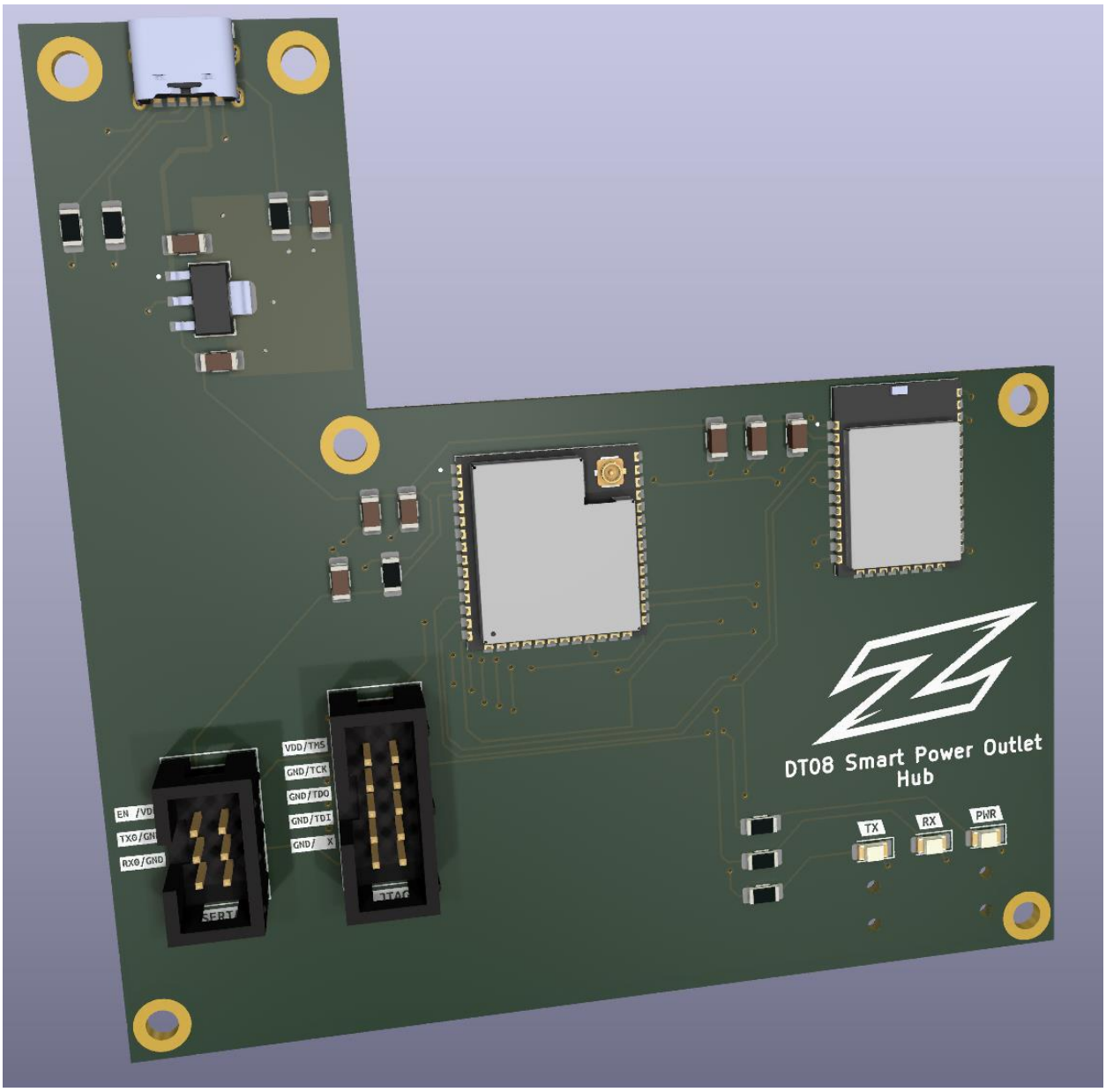

Figure 34: 3D Render of Hub Motherboard

The schematic illustrates the connections between the ESP32, XBee, and LDO used to power the Hub. As with the Smart Power Outlet's motherboard, LEDS are used as status indicators to indicate that the ESP32 and XBee have power, and that data is being transmitted.

Looking at the PCB layout, the PCB is larger than the Smart Power Outlet's and has an "L" shape. The shape is to allow external antennas to be mounted to the enclosure and connect to the ESP32 and XBee using u.FL to SMA adapters, leaving space for each adapter's cable to rest and hopefully avoid interference. Also shown on the schematic are four tiny holes on the bottom right corner of the motherboard, which are used as mounting points for a light pipe. The light pipe is used to direct the light given by the three upward facing LEDs forward so that the status indicators for power and data transmissions are always visible and useful. Finally, the 3D render provides an approximate idea of what the Hub motherboard will look like once assembled.

The final assembly of the Hub in its metal enclosure can be seen below:

Figure 35: Assembled Hub

5.2. Software Design:

5.2.1. Proposed Software Design (JG)

As discussed above, the proposed hardware design consists of three components: the Smart Power Outlets, the Hub, and the Control Application. Each of the three components will be running unique software designed to allow it to complete its assigned tasks and communicate with other

components as necessary. To aid in understanding the responsibility of each component, the following figure shows a level zero software block diagram for the Smart Power Outlet system:

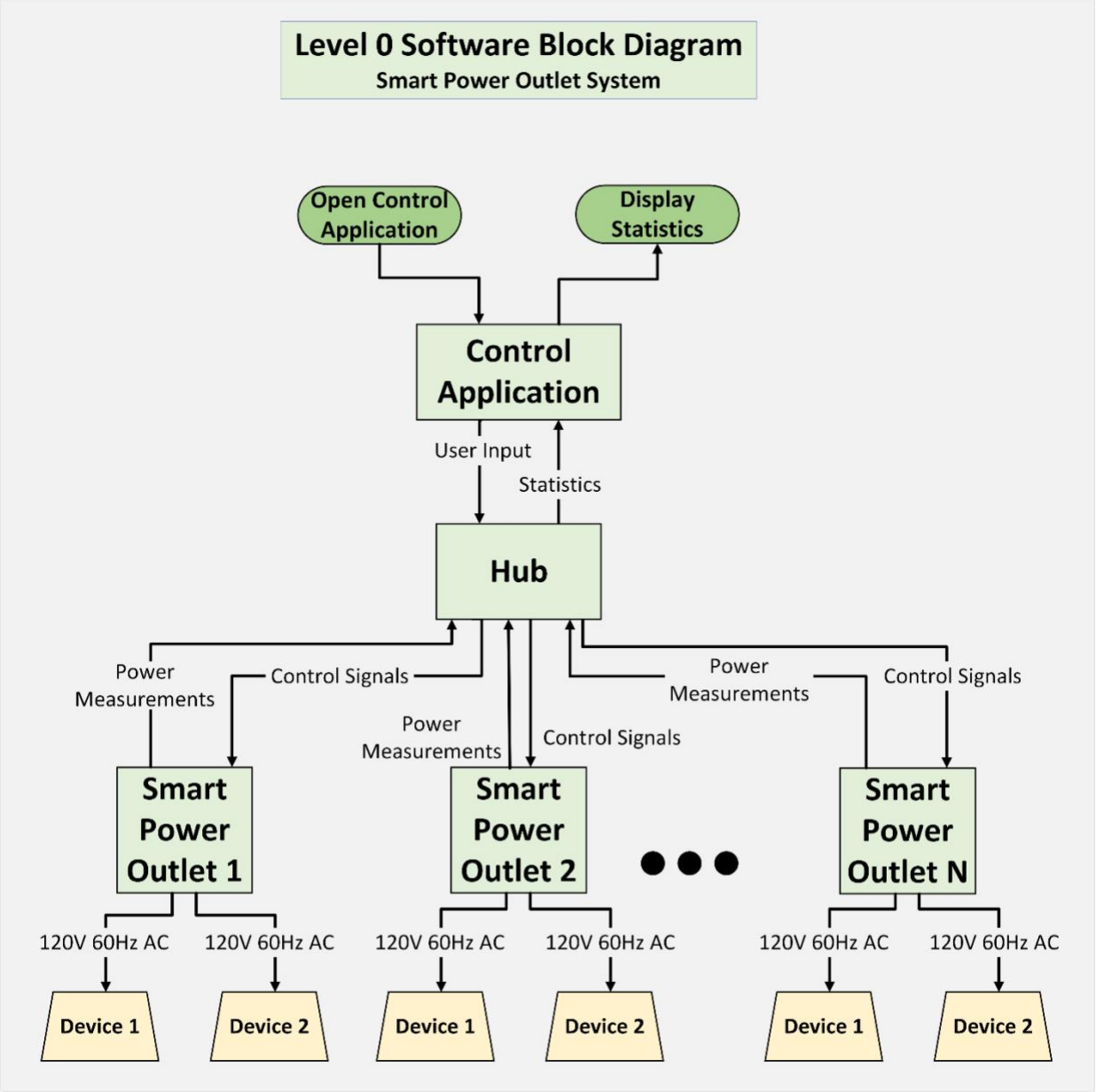

Figure 36 Level 0 Software Block Diagram for the Smart Power Outlet System

As shown in the level 0 software block diagram above, the Control Application will accept input from the user and forward it to the Hub. The Hub serves to connect all the Smart Power Outlets to the Control Application and will receive, process, and store all their power measurements. The Hub will allow the user to retrieve measurements taken by the Smart Power

Outlets and their associated power consumption statistics. The hub also forwards user input to the appropriate Smart Power Outlet in the form of control signals, which are used to control the behavior of a specified Smart Power Outlet. Thus, the Hub acts as a middleman between the two producers of data in the Smart Power Outlet System: The Control application and the Smart Power Outlets. The level 1 software block diagram shown below further breaks down the inner workings of each component:

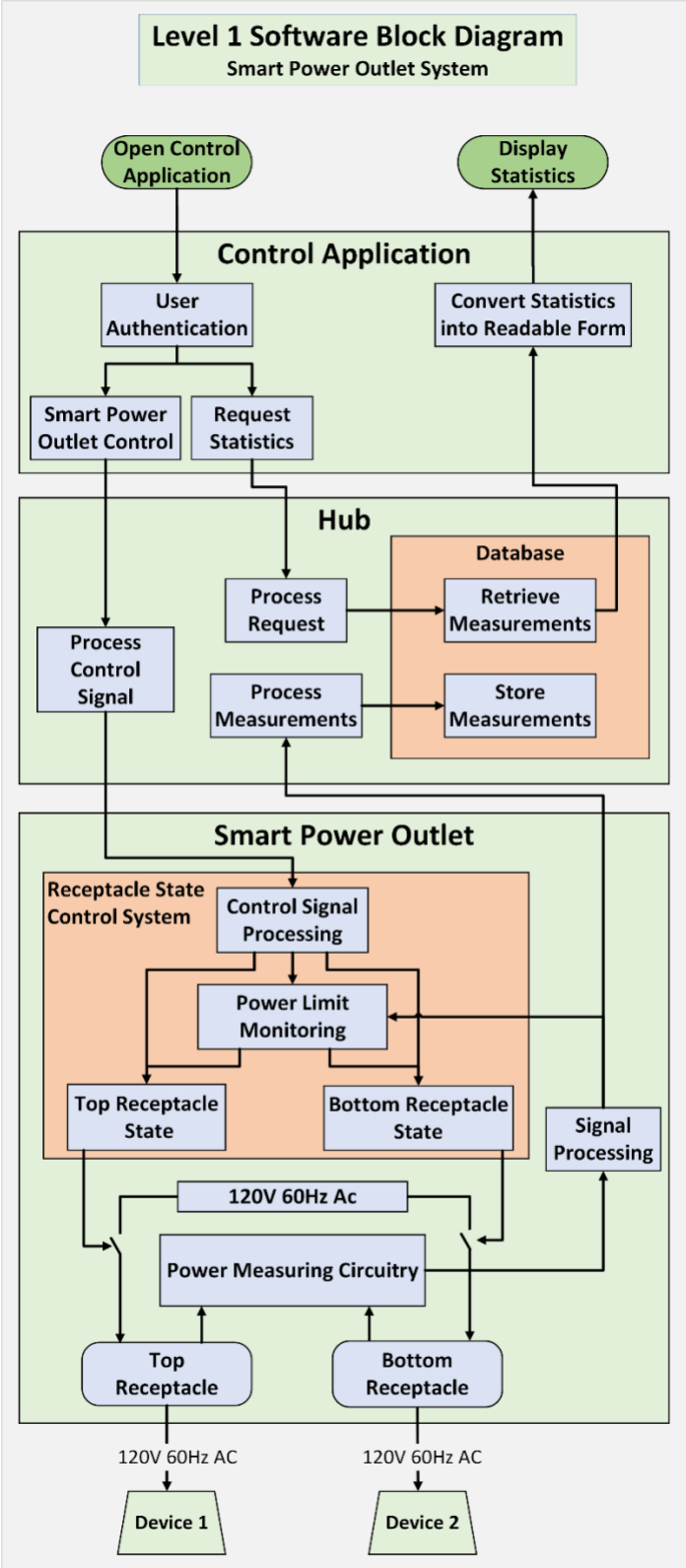

Figure 37 Level 1 Software Block Diagram for the Smart Power Outlet System

As shown in the level 1 software block diagram, the control application is responsible for handling user input and formatting statistics into the user's desired form. The control application allows the user to control connected Smart Power Outlets and to view statistics. Hence, the Control Application sends control signals and requests for data to the Hub. As the Hub is responsible for being the intermediary between the Control Application and Smart Power Outlets, the Hub handles requests from both. The Hub will process input from the Control Application to form control signals for Smart power Outlets and to retrieve desired information from its database. The Hub will also process measurements that it receives from the Smart Power Outlets and store them in its database. Finally, the Smart Power Outlets are responsible for interpreting the control signals that are received from the Hub and measuring the power consumption of the devices plugged into the Smart Power Outlet. The receptacles in each Smart Power Outlet can be toggled on and off as desired, so they have a state: on or off. If the total power consumption of a device exceeds a limit that the user sets, the receptacle supplying the power to it should be disabled. Therefore, the recorded measurements will be compared to this limit so that the state of the receptacles can be automatically controlled. Finally, the measurements need to be sent to the hub for storage as described previously. To aid in understanding the behavior of the Smart Power Outlet system, the following figure shows key hardware elements and the flow of data within the Smart Power Outlet system:

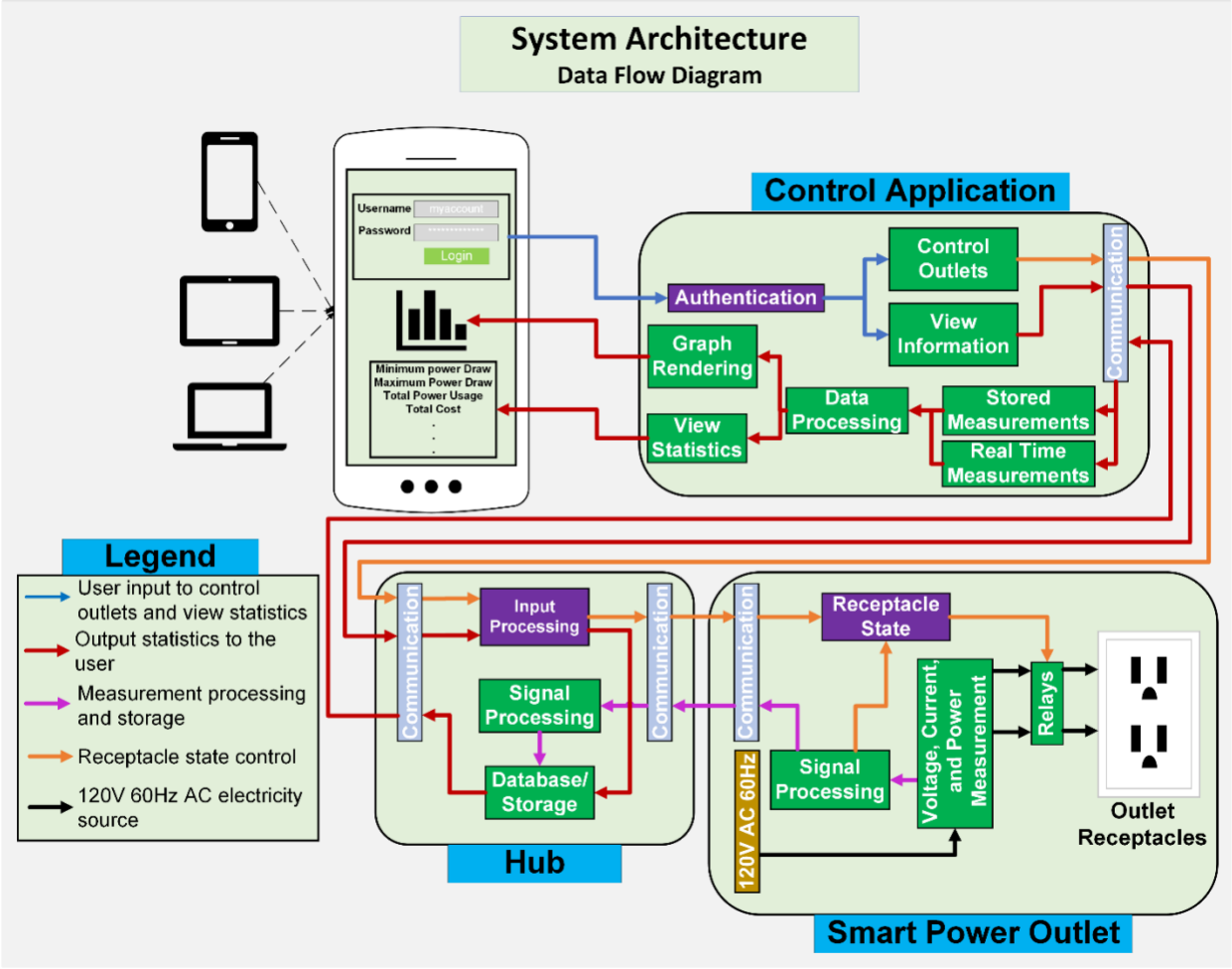

Figure 38: Flow of Data Within the Smart Power Outlet System

By following the line associated with each operation shown in the legend, the components in the Smart power Outlet system that are involved can be seen. Additionally, the operations that will be performed to enable the desired operation are shown. From the Data Flow Diagram, it is shown how the components in the Smart Power Outlet system are linked and depend on each other to function.

The following flowcharts describe the behavior of software in each major component in the Smart Power Outlet system with respect to their responsibilities outlined above. In the following flowcharts, dark green ellipses represent input and output to the Smart Power Outlet

system. Light blue ellipses with black bars represent operations involving communication with other components in the Smart Power Outlet system. Finally, light green shapes represent internal operations and decisions performed by each component. Additionally, components in the Smart Power Outlet system are intended to always remain on and operational. Hence, each component's Software Flowchart contains an initial "power on" start block followed software that loops indefinitely.

The first component is the Smart Power Outlets, which are responsible for measuring information regarding the power consumption of the devices plugged into their receptacles. As discussed previously, these measurements include things such as current, voltage, instantaneous power, average power, and potentially power factor depending on the measuring device used. The other input to the Smart Power Outlets is control signals, which are used to set limits within the Smart Power Outlet and Control the state of its receptacles. The output of the Smart Power Outlets is then the recorded measurements, which are sent to the Hub for further processing and storage. Below is the flow chart and associated Functional Requirement Table showing the software's behavior in the Smart Power Outlet to accomplish this.

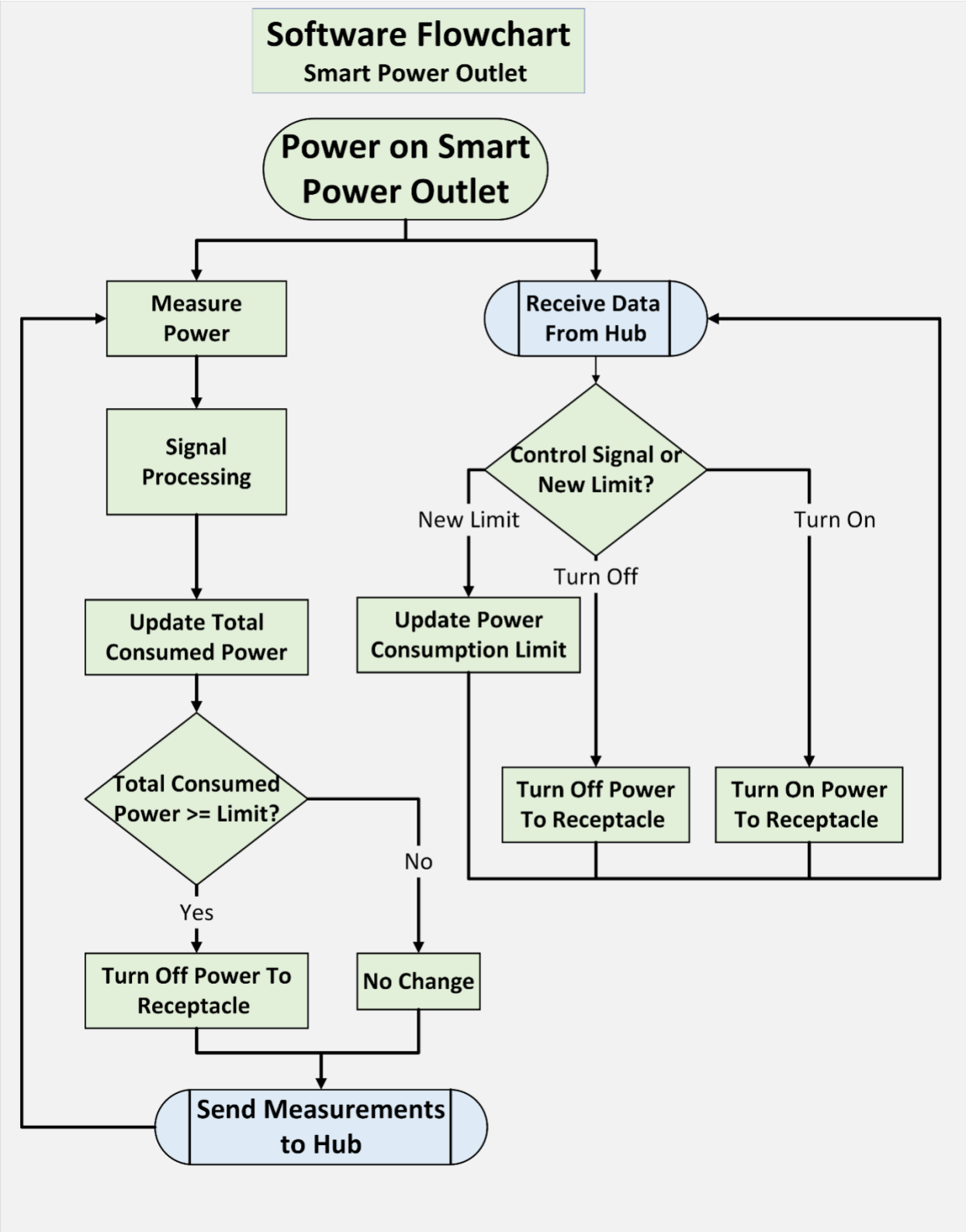

Figure 39: Smart Power Outlet Software Behavior Model

Table 8: Level 0 Smart Power Outlet Software Functional Requirements

Module	**Outlet Software**
Designer	Kyrolos Melek, Joseph Garro
Inputs	- Power Measurements - Control Signals from the Hub
Outputs	- Communication Signal (Power Measurements) - Power delivered to each Smart Power Outlet receptacle
Functionality	Provides a method of measuring power and communicating that data back to a main hub and toggles receptacle state (On or Off), based on user input

As shown in Figure 27 and Table 8, the Smart Power Outlet needs to do two tasks to function as intended: collect and forward power measurements to the Hub and listen for control signals coming from the Hub. Control signals can be sent from the Hub at any time and power measurements must be collected at regular intervals, so both tasks must be perpetually running. To enable this, a microprocessor equipped with multiprocessing or multiple threads will be used so that both tasks can run concurrently.

When collecting measurements, the Smart Power Outlet needs to first poll the measuring circuitry connected to each receptacle. Some signal processing is needed to convert the analog data into digital data and remove any noise. Once the appropriate signal processing methods are decided on (as they are dependent on the noise and accuracy of the measuring circuitry used), this block can be expanded into a level 1 flowchart to further explain the processing that will take place. After this step, an additional two tasks are performed. First, the average power will be added to a running sum and compared to a user-defined power limit. If this power limit is exceeded, the Smart Power

Outlet will disable the receptacle associated with the greedy device. Second, the collected measurements will be forwarded to the Hub for further processing and storage.

While measurements are being collected, the Smart Power Outlet is also listening for control signals sent by the Hub. Depending on the type of control signal received, the state of a receptacle in the Smart Power Outlet may be changed or a new total power limit for each receptacle may be set. By recognizing only these control signals and encrypting them when they are sent between the Smart Power Outlets and the Hub, malicious control of the Smart Power Outlets is prevented.

The next major component in the Smart Power Outlet system is the Hub. The Hub is responsible for managing all the connected Smart Power Outlets. The Hub receives measurements from all Smart Power Outlets and provides an interface for the user to control them via input from the Control Application. Additionally, the Hub is also responsible for further processing the measurements and storing them for later retrieval. The flowchart shown below in Figure 28 and the functional requirements found in Table 9 show the flow of data in the Hub and its responsibilities, respectively.

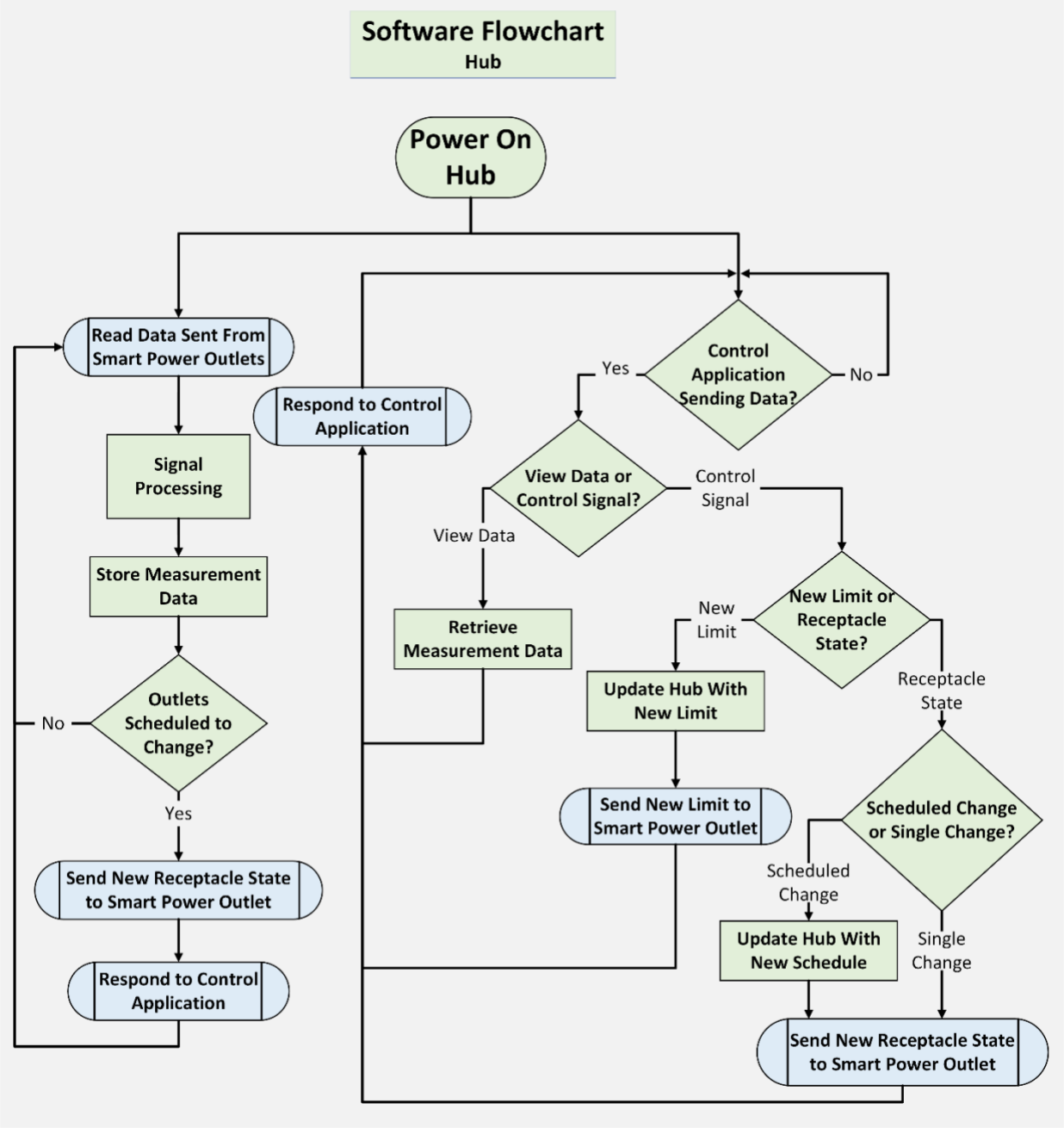

Figure 40: Hub Software Behavior Model

Table 9: The Level 1 Hub Software Functional Requirements

Module	**Hub**
Designer	Kyrolos Melek, Joseph Garro
Inputs	- Outlet/Receptacle State from Control Application - Schedules from Control Application - Power Limits from Control Application - Cost Limits from Control Application - Power Measurements from Smart Power Outlets - Outlet/Receptacle State from Smart Power Outlets
Outputs	- Measurements to User (User Data) - Control Signals to Smart Power Outlets (State and Power Limits)
Functionality	Manages all connected Smart Power Outlets autonomously and with user input via the control application and can process and store measurements

Just as with the Smart Power Outlets, the Hub is doing two tasks concurrently: receiving measurements from all connected Smart Power Outlets and listening for input from the Control Application. Consequently, another microprocessor with multiprocessing or multiple threads is required here. When measurements are received, the Hub will further process them and then store them for later retrieval. Another feature of the Smart Power Outlets is that they can be programmed to be on and off according to set schedules (for example, this Smart Power Outlet should always be on from 8:00 AM to 12:00 PM Monday through Friday). This requires the Hub to check to see if any Smart power Outlets are programmed to have their states changed at any given time and to send the appropriate control signal if so.

The other task delegated to the Hub is to listen for user input from the Control Application. User input can take one of four forms: requests for data, Smart Power Outlet power limits,

instantaneous receptacle state changes, and scheduled receptacle state changes. The code for the Hub that handles user input needs to be able to accept these four inputs and control the Smart Power Outlets accordingly. Also, the Control Application must be updated to recognize any changes that take place.

The final major component in the Smart Power Outlet system is the Control Application. The Control Application is used to provide an interface to the hub and Smart Power Outlets, allowing the user to control their behavior and view power consumption information. The idea here is that the Control Application can be used to program the Hub for autonomous management of the Smart Power Outlets also used for real time configuration and control. The Control Application also allows the user to view live data and past data regarding power usage reported by the Smart Power Outlets. The flowchart shown below in Figure 29 and the functional requirements found in Table 10 summarize the flow of data in the Control Application and its responsibilities, respectively.

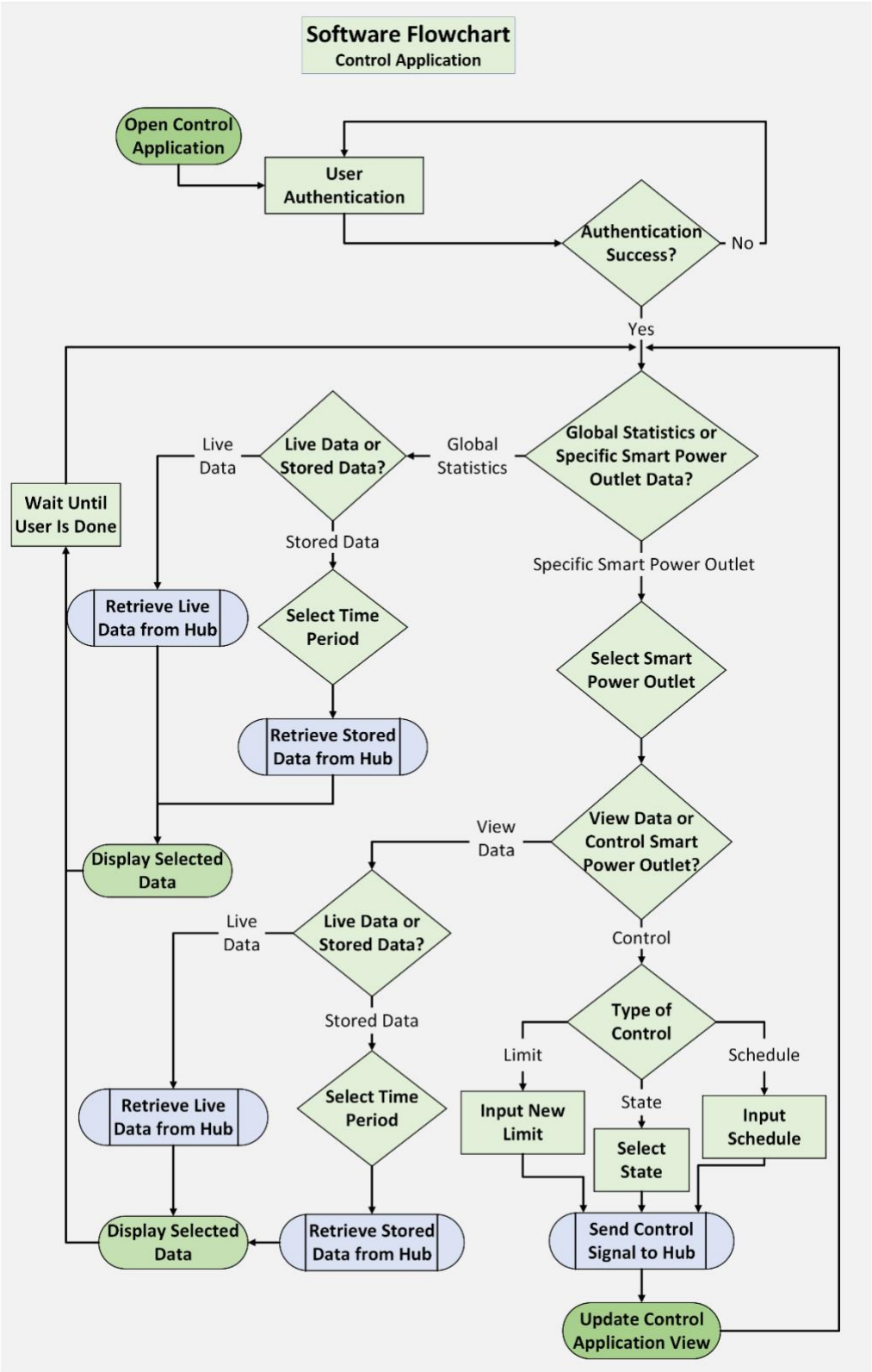

Figure 41: Application Software Behavior Model

Table 10: The Level 0 Control Application Functional Requirements

Module	**Control Application**
Designer	Kyrolos Melek, Joseph Garro
Inputs	- Outlet/Receptacle State - Power Limits - Cost Limits - Schedules - Power Measurements from Hub
Outputs	- Display Measurements to User (User Data) - Control Signals to Smart Power Outlets (State and Power Limits)
Functionality	Displays outlet data, toggles Outlets, inputs Cost and Power Limits

As shown above in Figure 29 and table 10, the Control Application provides an interface to the Hub that allows users to control the Smart Power Outlets and view the data that they have collected. Once logged in to the Control Application, the User can choose to view global statistics or view a particular Smart Power Outlet. When viewing global statistics, either live data or past data can be viewed. When a particular Smart Power Outlet is chosen, the user can view the data collected by it or configure it with a new power limit, a new schedule, or change its current state.

The Control Application is going to be accessible by smart devices such as smartphones, tabled, and computers and interface wirelessly with the hub. Following the idea of network segmentation discussed previously, the connection between the Control Application and the Hub is the only way that a user can control the Smart Power Outlets. This helps to protect the Smart Power Outlets from abuse as only those with access to the Control Application can change their behavior.

5.2.2. Realization of Proposed Software Design (JG)

To implement the proposed software design, the above data flow diagrams and software flowcharts are used to identify the responsibilities of each component in the Smart Power Outlet system. Code must be written to handle the data that each system produces and receives, in addition to code that allows the ESP32 microcontroller to communicate with the XBee modules used in the design. The following sections describe the software [that must be] written for each component and their necessary subsystems to enable the Smart Power Outlet system to function.

5.2.2.a Smart Power Outlet Software (JG)

At the core of each Smart Power Outlet is an ESP32 microcontroller responsible for controlling the entire component. Additional functionality is provided using other components capable of communication with the ESP32, such as the XBee modules to enable a Zigbee connection and the sensors that measure power consumption of connected devices. To program the ESP32 microcontroller, Espresif's IoT Development Framework (ESP-IDF)[1] is used. ESP-IDF is a framework used to write programs that run on the ESP32 consisting of prewritten libraries that control features on the ESP32 such as its GPIO pins, serial communication, and wireless communication. ESP-IDF is available in two flavors- an Arduino-compatible version and a C version. The Arduino version is simpler than the C version, but the C version is used as it provides more control than the Arduino based version due to its lower-level nature. An advantage to using ESP-IDF is that it is an official software release by Espressif, meaning that the provided libraries are guaranteed to work if they are implemented correctly in the code using them. The final tool

[1] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html

used to write code for the ESP32 is PlatformIO[2]- an IDE and Visual Studio Code extension designed to enhance the development of programs for embedded systems. PlatformIO supports ESP-IDF, has many debugging features, and can build images for the ESP32 and flash them, eliminating the need for other tools and simplifying the development process.

As stated in section 5.2.1. and shown in Figure 27, the Smart Power Outlets are designed to have two simultaneous tasks running- one that is measuring and calculating the current power consumption and one listening for commands from the Hub. The ESP32 contains a dual-core Xtensa 32-bit LX6 microprocessor whose multiprocessing capabilities are unlocked using the "freertos/FreeRTOS.h" header file included in ESP-IDF. Expressif describes their implementation of FreeRTOS as "ESP-IDF FreeRTOS... a modified version of Vanilla FreeRTOS v10.4.3... [that can] utilize the dual core SMP capabilities of ESP SoCs" [42]. Hence, ESP-IDF FreeRTOS can bring the qualities that make a real time operating system such as Vanilla FreeRTOS desirable to their multicore ESP32 microcontrollers.

The ESP-IDF FreeRTOS documentation lists the entry point of ESP-IDF FreeRTOS as the "user defined void app_main(void) function" [43], meaning no other modifications are required to enable ESP-IDF FreeRTOS. Once enabled, functions can be assigned to one of the ESP32's cores using the "freertos/task.h" header file and one of the variations of the "xTaskCreate()" functions. For the Smart Power Outlets, two main functions would have to be created- the function that handles the power measurements and calculations and the function that listens for commands from the Hub. Each function can then be assigned to its own core so that they can run concurrently. By omitting an end from both functions, such as by placing their bodies within an indefinite loop (I.E.

[2] https://docs.platformio.org/en/latest/what-is-platformio.html

while(1){… }), the functions will never end and run forever. A potential danger of this approach is how much memory, storage, and other resources the functions will use, as they can potentially crash due to overusing memory or starve each other from resources. Hence, careful coding will be required to avoid memory leaks and ensure a sharing of resources.

As stated previously, the Smart Power Outlet software has two main tasks: collecting energy measurements and listening for commands from the Hub. Data will be exchanged between the Smart Power Outlets and the Hub using JSON to serialize data into a known format, allowing for easy serialization and deserialization of data at any point in the system. The serialized JSON data is transmitted as the RF data of XBee Transmit Request frames, which will be discussed in detail later. Any data sent or received is expected to follow the following format:

{“data”: {*Relevant Serialized Data*}, “op” : *NUMBER}*

Here, the “data” key contained a nested JSON object with all relevant data, while the “op” key denotes a number that corresponds to the operation to be performed. The Smart Power Outlets and the Hub will know what operation is requested based on the value of the “op” field and will use the data corresponding to the “data” key to complete it. Many functions are used to implement this functionality. The first function, parseFrame(), analyzes incoming communications and extracts relevant information. If the type of incoming frame is the response to a Transmit Request, this function calls performOutletAction() to act based on the received data, otherwise it breaks. The function performOutletAction() acts on the received frame and uses its “op” key to determine the action to proceed. Actions are implemented as callable functions within performOutletAction(), making it easy to add additional functionality in the future.

The second responsibility of the Smart Power Outlet is to obtain energy measurements for the power consumed by a connected load. In this case, the Smart Power Outlet samples two Analog Devices ADE9153A energy metering ICs; one for the top receptacle of the Smart Power Outlet and one for the bottom receptacle. The ADE9153As obtain energy measurements independently of the Smart Power Outlet, requiring a serial interface between them and the ESP32 to be established to start them and read their measurements. The code to implement this is dscussed later in section 5.2.2.f. The complete code for the Smart Power Outlet, exclusng the utilized classes discussed in other sections, is found below:

main.cpp:

```
/* Joseph Garro
// jmg289@uakron.edu
// 4/22/2023
//
// firmware for esp32 and xbee based smart power outlet
// resources used:
   https://github.com/theElementZero/ESP32-UART-interrupt-handling/blob/master/uart_interrupt.c
   https://github.com/analogdevicesinc/arduino/tree/master/Arduino%20Uno%20R3/libraries/ADE9153A
   https://github.com/nlohmann/json
//---------------------------------*/

// additional classes for functionality
#include "xbee_api.hpp"
#include "json.hpp"

// ADE9153A Code
#include "../include/ADE9153A.h"
#include "../include/ADE9153AClass.h"

// esp32 classes
#include "stdio.h"                          // standard io
#include "driver/gpio.h"                    // esp GPIO pin control
#include "driver/uart.h"                    // esp UART driver
#include "driver/spi_master.h"              // esp SPI driver
#include "driver/spi_common.h"
```

```
#include "freertos/FreeRTOS.h"                    // freeRTOS for multitasking
#include "freertos/task.h"                        // create and schedudle tasks
#include "freertos/queue.h"
#include "esp_log.h"
#include "esp_task_wdt.h"
#include "Arduino.h"

// c++ classes
#include <iostream>
#include <queue>
#include <ctime>
#include <time.h>
//-----------------------------------------------------------------------------------------------
// link json class
using json = nlohmann::json;

//-------------------------------//
//     global definitions      //
//-------------------------------//

#define BUFFER_SIZE (1024 * 4)                    // hold 4096 bytes in each buffer
const int MEASUREMENT_MULTIPLIER = 10000;         // multiplier to set sig figs in outlet measurements

//-------------------------------//
//     ADE9153A definitions     //
//-------------------------------//

ADE9153AClass meters;                             // Control Dual ADE9153As
bool INITALIZED;                                  // both ADE9153A initlaized

// structs to hold measurement data
struct EnergyRegs energyValsTop;
struct PowerRegs powerValsTop;
struct RMSRegs rmsValsTop;
struct PQRegs pqValsTop;
struct EnergyRegs energyValsBottom;
struct PowerRegs powerValsBottom;
struct RMSRegs rmsValsBottom;
struct PQRegs pqValsBottom;
struct AcalRegs acalTop;
struct AcalRegs acalBottom;

//-------------------------------//
//          queues           //
```

```
//------------------------------//

static QueueHandle_t xbee_queue;                    // queue to handle xbee events
// queue for pointers to incoming XBEE frames
std::queue<std::vector<uint8_t>> xbee_incoming;      // hold incoming XBEE frames
// queue for pointers to outgoing XBEE frames
std::queue<std::vector<uint8_t>> xbee_outgoing;      // hold outgoing xbee frames
// queues to store power measurement information
std::queue<json> collected_measurements;            // hold collected power measurements

//------------------------------//
//         semaphores         //
//------------------------------//

SemaphoreHandle_t measurements_lock = NULL;          // semaphore for measurement queue
SemaphoreHandle_t outgoing_lock = NULL;              // sempahore for outgoing xbee queue
SemaphoreHandle_t incoming_lock = NULL;              // semaphore for incoming data

//------------------------------//
//     outlet information     //
//------------------------------//

// global receptacle state
static int RECEPTACLE_STATE = 0;
// global maximum instantaneous power draw; shut off outlet if exceeded
static float MAX_INSTANTANEOUS_POWER_DRAW = -1;      // none by default
// global sustatined maximum power draw; shut off if exceeded
static float MAX_SUSTAINED_POWER_DRAW = -1;          // none by default
// global period for sustained power draw
static int SUSTAINED_POWER_DRAW_PERIOD = -1;         // none by default

//------------------------------//
//    GPIO pin definitions    //
//------------------------------//

// XBEE UART PINS
#define XBEE_UART (UART_NUM_2)                       // uart2 to communicate between xbee and esp32
#define XBEE_UART_RX (GPIO_NUM_19)                   // uart2 TX
#define XBEE_UART_TX (GPIO_NUM_18)                   // uart2 RX

// ADE9153A SPI Pins
#define SPI_SPEED 1000000                            // SPI speed
#define ADE9153A_MOSI (GPIO_NUM_4)                   // MOSI line => pin 29 on both ADE9153As
#define ADE9153A_MISO (GPIO_NUM_5)                   // MISO line => pin 30 on both ADE9153As
```

```
#define ADE9153A_SCLK (GPIO_NUM_6)                  // SCLK line => pin 31 on both ADE9153As
#define TOP_RESET (GPIO_NUM_7)                     // rst line => pin 28 on ADE9153A #1
#define BOTTOM_RESET (GPIO_NUM_8)                   // rst line => pin 28 on ADE9153A #2
#define TOP_CS (GPIO_NUM_9)                       // chipselect line for top ADE9153A => pin 32 on ADE9153A #1
#define BOTTOM_CS (GPIO_NUM_10)                     // chipselect line for bottom ADE9153A => pin 32 on ADE9153A #2
#define DUMMY_CS  (GPIO_NUM_37)                     // dummy CS line

// receptacle control pins
#define TOP_CONTROL (GPIO_NUM_1)
#define BOTTOM_CONTROL (GPIO_NUM_2)

//------------------------------//
//        logging tags        //
//------------------------------//

static const char *XBEE_TAG = "xbee uart";
static const char *RECEPTACLE_TAG = "receptacle state";
static const char *MAX_INSTANTANEOUS_POWER_DRAW_TAG = "set max instantanous power draw";
static const char *MAX_SUSTAINED_POWER_DRAW_TAG = "set max sustained power draw";
static const char *SET_SYSTEM_TIME = "set system time";
static const char *PARSE_FRAME = "parse xbee frame";
static const char *SETUP = "setup";
static const char *OUTLET = "ADE9153A Setup";

//------------------------------//
//     function definitions     //
//------------------------------//

// configure UART connection to xbee module
static void xbee_uart_init(void){
   ESP_LOGI(XBEE_TAG, "configuring xbee uart connection");
   // UART configuration settings
   const uart_config_t xbee_uart_config = {
      .baud_rate = 115200,
      .data_bits = UART_DATA_8_BITS,
      .parity = UART_PARITY_DISABLE,
      .stop_bits = UART_STOP_BITS_1,
      .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
      .rx_flow_ctrl_thresh = 122,
      .source_clk = UART_SCLK_APB};

   // install UART driver
    ESP_ERROR_CHECK(uart_driver_install(XBEE_UART, BUFFER_SIZE * 4, BUFFER_SIZE * 4, 1024, &xbee_queue, 0));        // install
UART driver on pins connected to xbee, buffer of 2048 bytes, event queue enabled
```

```
   ESP_ERROR_CHECK(uart_param_config(XBEE_UART, &xbee_uart_config));                                    // write xbee_uart_config to xbee
UART
            ESP_ERROR_CHECK(uart_set_pin(XBEE_UART,      XBEE_UART_TX,      XBEE_UART_RX,      UART_PIN_NO_CHANGE,
UART_PIN_NO_CHANGE));   // assign TX and RX pins to xbee UART
};

// UART event handler for XBEE
//                                synthesized                                from                                https://github.com/espressif/esp-
idf/blob/49551cc48cb3cdd5563059028749616de313f0ec/examples/peripherals/uart/uart_events/main/uart_events_example_main.c
static void xbee_uart_event_task(void *pvParameters){
   uart_event_t xbee_event;                                          // hold UART event
   uint8_t* dtmp = (uint8_t*) malloc(BUFFER_SIZE);                                  // temporary buffer
   for(;;){
      // activate when a UART event is detected
      if(xQueueReceive(xbee_queue, (void*)&xbee_event, (TickType_t)portMAX_DELAY)){
         size_t eventsize = xbee_event.size;
         bzero(dtmp, BUFFER_SIZE);
         ESP_LOGI(XBEE_TAG, "uart[%d] event:", XBEE_UART);                          // zero buffer
         switch(xbee_event.type){                                          // handle different UART events
            // read incoming UART data
            case UART_DATA:
               ESP_LOGI(XBEE_TAG, "[UART DATA]: %d", eventsize);
               uart_read_bytes(XBEE_UART, dtmp, eventsize, portMAX_DELAY);          // write data to dtmp
               xSemaphoreTake(incoming_lock, portMAX_DELAY);                      // lock incoming frame queue
               xbee_incoming.push(std::vector<uint8_t>(dtmp, dtmp + eventsize));       // push frame to incoming xbee queue
               xSemaphoreGive(incoming_lock);                                  // release incoming frame queue
               break;

            // HW FIFO overflow detected
            case UART_FIFO_OVF:
               ESP_LOGI(XBEE_TAG, "hw fifo overflow");
               uart_flush_input(XBEE_UART);
               xQueueReset(xbee_queue);
               break;

            // UART ring buffer full
            case UART_BUFFER_FULL:
               ESP_LOGI(XBEE_TAG, "ring buffer full");
               uart_flush_input(XBEE_UART);
               xQueueReset(xbee_queue);
               break;

            // UART RX break detected
            case UART_BREAK:
```

```
                ESP_LOGI(XBEE_TAG, "uart rx break");
                break;

            // UART parity check error
            case UART_PARITY_ERR:
                ESP_LOGI(XBEE_TAG, "uart parity error");
                break;

            // UART frame error
            case UART_FRAME_ERR:
                ESP_LOGI(XBEE_TAG, "uart frame error");
                break;

            default:
                ESP_LOGI(XBEE_TAG, "xbee UART event: %d", xbee_event.type);
                break;
        }
      }
   }
   free(dtmp);
   dtmp = NULL;
   vTaskDelete(NULL);
};

// toggle receptacle states based on control packet
void toggleReceptacles(json json_data){
   // make sure value exists
   if(!json_data["value"].is_null() && !json_data["value"].is_array() && json_data["value"].is_number()){
      switch(json_data["value"].get<int>()){
         // top off bottom off
         case 0:
            gpio_set_level(TOP_CONTROL, 0);
            gpio_set_level(BOTTOM_CONTROL, 0);
            RECEPTACLE_STATE = 0;
            ESP_LOGI(RECEPTACLE_TAG, "top off, bottom off");
            break;

         // top off bottom on
         case 1:
            gpio_set_level(TOP_CONTROL, 0);
            gpio_set_level(BOTTOM_CONTROL, 1);
            RECEPTACLE_STATE = 1;
            ESP_LOGI(RECEPTACLE_TAG, "top off, bottom on");
            break;
```

```
// top on bottom off
case 2:
    gpio_set_level(TOP_CONTROL, 1);
    gpio_set_level(BOTTOM_CONTROL, 0);
    RECEPTACLE_STATE = 2;
    ESP_LOGI(RECEPTACLE_TAG, "top on, bottom off");
    break;

// top on bottom on
case 3:
    gpio_set_level(TOP_CONTROL, 1);
    gpio_set_level(BOTTOM_CONTROL, 1);
    RECEPTACLE_STATE = 3;
    ESP_LOGI(RECEPTACLE_TAG, "top on, bottom on");
    break;

// top on only
case 4:
    gpio_set_level(TOP_CONTROL, 1);
    RECEPTACLE_STATE = 4 & RECEPTACLE_STATE;
    ESP_LOGI(RECEPTACLE_TAG, "top on");
    break;

// top off only
case 5:
    gpio_set_level(TOP_CONTROL, 0);
    RECEPTACLE_STATE = 5 & RECEPTACLE_STATE;
    ESP_LOGI(RECEPTACLE_TAG, "top off");
    break;

// bottom on only
case 6:
    gpio_set_level(BOTTOM_CONTROL, 1);
    RECEPTACLE_STATE = 6 & RECEPTACLE_STATE;
    ESP_LOGI(RECEPTACLE_TAG, "bottom on");
    break;

// bototm off only
case 7:
    gpio_set_level(BOTTOM_CONTROL, 0);
    RECEPTACLE_STATE = 7 & RECEPTACLE_STATE;
    ESP_LOGI(RECEPTACLE_TAG, "bottom off");
    break;
```

```
            // invalid
            default:
                ESP_LOGI(RECEPTACLE_TAG, "%d is an invalid value", json_data["value"].get<int>());
                break;
        }
    } // otherwise error
    else
        ESP_LOGI(RECEPTACLE_TAG, "error setting receptacle state");
}

// set maximum instantaneous power
void setMaximumInstantaneousPowerDraw(json json_data){
    // make sure value exists and is valid
    if(!json_data["value"].is_null() && !json_data["value"].is_array() && json_data.at("value").is_number()){
        // update maximum instaneous power
        MAX_INSTANTANEOUS_POWER_DRAW = json_data["value"].get<float>();
                ESP_LOGI(MAX_INSTANTANEOUS_POWER_DRAW_TAG, "set maximum instantanoues power draw to %f",
MAX_INSTANTANEOUS_POWER_DRAW);
    }
    // otherwise error
    else
        ESP_LOGI(MAX_INSTANTANEOUS_POWER_DRAW_TAG, "error setting maximum instantaneous power");
}

// set maximum power over a sustained period
void setMaximumSustainedPowerDraw(json json_data){
    // make sure value exists and is valid
    if((!json_data["value"].is_null() && !json_data["range"].is_null()) && (!json_data["value"].is_array() && !json_data["range"].is_array()) &&
(json_data.at("value").is_number() && json_data.at("range").is_number())){
        // update maximum instaneous power
        MAX_SUSTAINED_POWER_DRAW = json_data["value"].get<float>();
                    ESP_LOGI(MAX_SUSTAINED_POWER_DRAW_TAG, "set maximum sustained power draw to %f",
MAX_SUSTAINED_POWER_DRAW);
        SUSTAINED_POWER_DRAW_PERIOD = json_data["range"].get<int>();
                ESP_LOGI(MAX_SUSTAINED_POWER_DRAW_TAG, "set maximum sustained power draw period to %d days",
SUSTAINED_POWER_DRAW_PERIOD);
    }
    // otherwise error
    else
        ESP_LOGI(MAX_SUSTAINED_POWER_DRAW_TAG, "error setting maximum sustained power");
}

// set system time using time recieved from hub
```

```
void setSystemTime(json json_data){
 // make sure value exists and is valid
        if((!json_data["s"].is_null() && !json_data["us"].is_null() && !json_data["tz"].is_null()) && (!json_data["s"].is_array() && !json_data["us"].is_array() && !json_data["tz"].is_array()) && (json_data.at("s").is_number() && json_data.at("us").is_number() && json_data["tz"].is_string())){
      // update timezone
      setenv("TZ", (json_data["tz"].get<std::string>()).c_str(), 1);
      tzset();
      // set time
      timeval newtime;
      newtime.tv_sec = json_data["s"].get<int>();
      newtime.tv_usec = json_data["us"].get<int>();
      settimeofday(&newtime, NULL);
      ESP_LOGI(SET_SYSTEM_TIME, "set system time");
   }
   // otherwise error
   else
      ESP_LOGI(SET_SYSTEM_TIME, "error setting system time");
}

// perform an action in the smart power outlet based on the contents of recieved frame
void performOutletAction(json json_payload){
   if(!json_payload["op"].is_null() && json_payload["op"].is_number()){
      //  get type of JSON
      int json_type = json_payload["op"].get<int>();
      // get data from json packet
      json json_data = json_payload["data"];
      switch(json_type){
          // outlet will not recieve measurement packets, ignore type == 0
          // handle changes in receptacle state
          case 1:
             toggleReceptacles(json_data);
             break;

          //handle setting maximum instaneous power draw
          case 2:
             setMaximumInstantaneousPowerDraw(json_data);
             break;

          // handle maximum sustained power draw
          case 3:
             setMaximumSustainedPowerDraw(json_data);
             break;
```

```
            // handle time set
            case 4:
                setSystemTime(json_data);
                break;
            default:
                break;
        };
    };
};

//-------------------------------------------------------
// sample data
//float test_v0 = 120.690;
//float test_i0 = 14.589;

float test_rp0 = 140.346;
float test_pf0 = 0.9999;

float test_rp1 = 5.524;
float test_pf1 = 0.8890;

unsigned char source = 1;

unsigned char type = 0;

// sample ADE9153A every second
static void sampleADE9153A(void* pvParameters){
    for(;;){
        if(INITALIZED == true){
            time_t currTime;
            time(&currTime);
            // read top power values
            meters.ReadPowerRegs(TOP_CS, &powerValsTop);
            // read bottom power values
            meters.ReadPowerRegs(BOTTOM_CS, &powerValsBottom);
            meters.ReadRMSRegs(TOP_CS, &rmsValsTop);
            meters.ReadRMSRegs(BOTTOM_CS, &rmsValsBottom);

            std::cout << "TOP VOLTAGE: " << rmsValsTop.VoltageRMSValue / 1000 << std::endl;
            std::cout << "TOP CURRENT: " << rmsValsTop.CurrentRMSValue / 1000 << std::endl;

            std::cout << "BOTTOM VOLTAGE: " << rmsValsBottom.VoltageRMSValue / 1000 << std::endl;
            std::cout << "BOTTOM CURRENT: " << rmsValsBottom.CurrentRMSValue / 1000 << std::endl;
```

```
        std::cout << "TOP POWER: " << powerValsTop.ActivePowerValue / 1000 << std::endl;
        std::cout << "BOTTOM POWER: " << powerValsBottom.ActivePowerValue / 1000 << std::endl;


        // add to collected measurements
        xSemaphoreTake(measurements_lock, portMAX_DELAY);                    // take measurements semaphore
        collected_measurements.push(json{
          {"s", currTime},
          {"t", {
            {"p", int(MEASUREMENT_MULTIPLIER*powerValsTop.ActivePowerValue)},
            {"f", int(MEASUREMENT_MULTIPLIER*test_pf1)}}},
          {"b",{
            {"p", int(MEASUREMENT_MULTIPLIER*powerValsBottom.ActivePowerValue)},
            {"f", int(MEASUREMENT_MULTIPLIER*test_pf1)}}}
        });
        xSemaphoreGive(measurements_lock);
        vTaskDelay(600/portTICK_PERIOD_MS);
      }
   vTaskDelay(1000/portTICK_PERIOD_MS);
   }
}

void formRoute(json json_object){
   json frame_payload = (json_object)["FRAME DATA"];
   json frame_overhead = (json_object)["FRAME OVERHEAD"];
   std::cout << frame_overhead.dump() << std::endl;
   std::cout << frame_payload.dump() << std::endl;
   // act if data and it is an array
   if (!frame_payload["DATA"].is_null() && frame_payload["DATA"].is_array()){
      std::cout << "MAKE SOURCE ROUTE" << std::endl;
   }
}

/*
 * xbee frames arrive in the following format
      {
        "FRAME TYPE", X,                    -- type of xbee frame, IE at command response, transmit request, etc
        "FRAME OVERHEAD", {                    -- data relevant to xbee protocol, IE frame ID, destination, etc
          XXX, XXX
          .
          .
          .
        },
        "FRAME DATA", {                    -- data to perform outlwt interactions with
```

```
            "data"{                         -- data to act on, necessary data for operations will be found here in the expected key-value pairs
               "value", X,
               .
               .
            }
            "op", x,                        -- operation, IE toggle receptacles, set power limit, etc
         }
      }
*/

// determine action to take based on recieved XBEE frame
static void parseFrame(void *pvParameters){
   bool set = false;                                    // bool for if semaphore is stil set after loop
   for(;;){
      // work on existing xbee frames
      xSemaphoreTake(incoming_lock, portMAX_DELAY);                     // take incoming semaphore
      set = true;                                       // semaphore taken
      while(!xbee_incoming.empty()){
         // get oldest xbee frame
         std::vector<uint8_t> xbee_frame = xbee_incoming.front();
         // remove xbee frame from queue
         xbee_incoming.pop();
         xSemaphoreGive(incoming_lock);                         // release once front popped
         set = false;                                   // semaphore released
         // json with all information about xbee frame
         json j = readFrame(xbee_frame.data());
         //std::cout << "JSON Frame: " << j.dump() << std::endl;
         // handle error cases
         // unrecognized frame
         if(j == -1){
            ESP_LOGI(PARSE_FRAME, "recieved an unrecognized frame");
         }
         // invalid frame
         else if(j == -2){
            ESP_LOGI(PARSE_FRAME, "recieved an invalid frame");
         }
         // otherwise do something
         else{
            // get type of frame
            uint8_t frameType = j["FRAME TYPE"].get<int>();
            // should I change these to dynamiclly allocated vars????
            json frame_payload = j["FRAME DATA"];
            json frame_overhead = j["FRAME OVERHEAD"];
            // switch based on type of frame
```

```
            switch(frameType){
               // rx response                    -- transmit request will include data -> means this is an outlet action
               case 0x90:
                  performOutletAction(frame_payload);
                  break;

               // explicit rx response           -- transmit request will include data -> means this is an outlet action
               case 0x91:
                  performOutletAction(frame_payload);
                  break;

               // handle route record indicator
               case 0xA1:
                  formRoute(j);
                  break;
               // all other cases                -- other operations deal with XBee behavior -> do not need ESP32's attention
               default:
                  break;
            };
         }
         vTaskDelay(1);
      }
      if(set == true)                                    // release if not unset
         xSemaphoreGive(incoming_lock);                  // release semaphore lock
   vTaskDelay(100);
   }
};

// send xbee frames stored in queue
static void sendFrame(void *pvParameters){
   bool set = false;                                     // bool for if semaphore is set
   for(;;){
      // work on existing xbee frames
      xSemaphoreTake(outgoing_lock, portMAX_DELAY);      // take incoming semaphore
      set = true;                                        // semaphore taken
      while(!xbee_outgoing.empty()){
         // get oldest xbee frame
         std::vector<uint8_t> xbee_frame = xbee_outgoing.front();
         // remove xbee frame from queue
         xbee_outgoing.pop();
         xSemaphoreGive(outgoing_lock);                  // release semaphore
         set = false;                                    // semaphore released
         uart_write_bytes(XBEE_UART, xbee_frame.data(), xbee_frame.size());
         vTaskDelay(1);
```

```
        }
        if(set == true)                                         // release if not unset
            xSemaphoreGive(outgoing_lock);                              // release semaphore
        vTaskDelay(100);
    }
};

// test function to print current time
static void printTime(void *pvParameters){
    for(;;){
        struct tm *loctime;
        time_t currtime;
        currtime = time(NULL);
        loctime = localtime(&currtime);
        std::cout << asctime(loctime) << std::endl;
        vTaskDelay(1000/portTICK_PERIOD_MS);
    };
};

// get time from hub
void getTime(){
    // frame to ask hub for time
    std::vector<uint8_t> getTime = {0x7E, 0x00, 0x18, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x7B,
0x22, 0x6F, 0x70, 0x22, 0x3A, 0x31, 0x30, 0x34, 0x7D, 0x05};

    // json to store hub response
    json hubResponse = {};
    // bufffer for response
    uint8_t* dtmp = (uint8_t*) malloc(BUFFER_SIZE);
    bzero(dtmp, BUFFER_SIZE);
    int length = 0;
    // get current time
    struct tm *loctime;                                         // tm
    time_t currtime;
    currtime = time(NULL);                                          // get time since epoch
    loctime = localtime(&currtime);                                 // set local time
    uart_flush(XBEE_UART);
    while(loctime->tm_year < 123){                                  // get years since epoch
        ESP_LOGI(SETUP, "time not valid");
        uart_write_bytes(XBEE_UART, getTime.data(), getTime.size());                // send time request to hub
        vTaskDelay(5000/portTICK_PERIOD_MS);                                    // wait 5 seconds for response
        ESP_ERROR_CHECK(uart_get_buffered_data_len(XBEE_UART, (size_t*)&length));
        if(length > 0){
            //std::cout << "LENGTH- " << length << std::endl;
```

```
        length = uart_read_bytes(XBEE_UART, dtmp, length, 100);                    // read in bytes
        uart_flush(XBEE_UART);
        hubResponse = readFrame(dtmp);
        // check response here
        //std::cout << hubResponse.dump() << std::endl;
        if(hubResponse == -2 || hubResponse == -1){
          ESP_LOGI(SETUP, "invalid time request from hub");
        }
            else if (hubResponse["FRAME DATA"]["data"].contains("s") && hubResponse["FRAME DATA"]["data"].contains("tz") &&
hubResponse["FRAME DATA"]["data"].contains("us")){

          setSystemTime(hubResponse["FRAME DATA"]["data"]);
          currtime = time(NULL);                                          // get time since epoch
          loctime = localtime(&currtime);
        }
        else{
          ESP_LOGI(SETUP, "recieved malformed data from hub");
        }
      }
    }
    ESP_LOGI(SETUP, "valid time set");
}

// clear queues before continuing
// thank you David Rodríguez - dribeas : https://stackoverflow.com/questions/709146/how-do-i-clear-the-stdqueue-efficiently
void clearXbeeQueues(std::queue<std::vector<uint8_t>> &old_queue){
  std::queue<std::vector<uint8_t>> empty_queue;
  std::swap(old_queue, empty_queue);
}

// form frames containing outlet measurements
static void formMeasurementFrames(void *pvParameters){
    json frame;
    bool set = false;                                                 // bool for if semaphore is set
    for(;;){
      xSemaphoreTake(measurements_lock, portMAX_DELAY);                          // take semaphore
      set = true;                                                   // semaphore taken
      // form frame while not empty
      while(!collected_measurements.empty()){
        frame = json{                                                // create JSON
          {"op", 101},                                              // hub OP for measurements = 101
          {"data", collected_measurements.front()}
        };
        collected_measurements.pop();                                    // removed first element
```

```
            xSemaphoreGive(measurements_lock);                              // release semaphore
            set = false;                                                    // semaphore released
            xSemaphoreTake(outgoing_lock, portMAX_DELAY);                   // take semaphore
            //std::cout << frame.dump() << std::endl;
             xbee_outgoing.push(formTXFrame(frame.dump(), 0, 0, NULL, NULL));          // push measurement frame to outgoing XBEE frame
queue
            xSemaphoreGive(outgoing_lock);                                  // release semaphore
            vTaskDelay(1);
        }
        if(set == true)                                                     // release if not unset
            xSemaphoreGive(measurements_lock);                              // release semaphore
        vTaskDelay(100);
    }
}

static void printHeap(void* pvParameter){
    for(;;){
        std::cout << "Free Heap Size: " << int(esp_get_free_heap_size()) << std::endl;
        std::cout << "Largest block : " << int(heap_caps_get_largest_free_block(MALLOC_CAP_8BIT)) << std::endl;
        vTaskDelay(10000/portTICK_PERIOD_MS);
    }
}

// main function
extern "C" void app_main() {
    //esp_log_level_set(XBEE_TAG, ESP_LOG_VERBOSE);                         //uncomment to have more verbose logging

    // enable receptacle control pins
    gpio_reset_pin(TOP_CONTROL);
    gpio_reset_pin(BOTTOM_CONTROL);
    gpio_set_direction(TOP_CONTROL, GPIO_MODE_OUTPUT);
    gpio_set_direction(BOTTOM_CONTROL, GPIO_MODE_OUTPUT);
    gpio_set_level(TOP_CONTROL, 1);                                         // enabled by default
    gpio_set_level(BOTTOM_CONTROL, 1);                                      // enabled by default
    // make sure xbee queues are empty
    clearXbeeQueues(xbee_incoming);
    clearXbeeQueues(xbee_outgoing);

    // initalize xbee UART connection
    xbee_uart_init();
    vTaskDelay(5000/portTICK_PERIOD_MS);
    uart_flush_input(XBEE_UART);                                            // flush xbee buffers after initalization
    uart_flush(XBEE_UART);
```

```
  // set system time
   getTime();
   uart_flush_input(XBEE_UART);                                    // flush xbee buffers after initalization
   uart_flush(XBEE_UART);
   xQueueReset(xbee_queue);                                        // flush xbee queue and buffers after initalization
   vTaskDelay(1000/portTICK_PERIOD_MS);


   // initalize both ADE9153As
    INITALIZED = meters.SPI_Init_Bus(SPI_SPEED, ADE9153A_SCLK, ADE9153A_MISO, ADE9153A_MOSI, TOP_CS, BOTTOM_CS,
DUMMY_CS, TOP_CS, BOTTOM_CS);
   if(INITALIZED == false)
      ESP_LOGI(OUTLET, "Failed to initalize 1 or both ADE9153A");
   else
      ESP_LOGI(OUTLET, "Both ADE9153A Initalized");

   // calibrate ADE9153As
   std::cout << "CALIBRATE TOP ADE9153A" << std::endl;
   meters.StartAcal_AINormal(TOP_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.StopAcal(TOP_CS);
   meters.StartAcal_AV(TOP_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.StopAcal(TOP_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.ReadAcalRegs(TOP_CS, &acalTop);
   long Igain_top = acalTop.AICC;
   long Vgain_top = acalTop.AVCC ;// / 13411.05) - 1) * 134217728;
   meters.SPI_Write_32(TOP_CS, REG_AIGAIN, Igain_top);
   meters.SPI_Write_32(TOP_CS, REG_AVGAIN, Vgain_top);

   std::cout << "CALIBRATE BOTTOM ADE9153A" << std::endl;
   meters.StartAcal_AINormal(BOTTOM_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.StopAcal(BOTTOM_CS);
   meters.StartAcal_AV(BOTTOM_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.StopAcal(BOTTOM_CS);
   vTaskDelay(200/portTICK_PERIOD_MS);
   meters.ReadAcalRegs(BOTTOM_CS, &acalBottom);

   long Igain_bottom = acalBottom.AICC;
   long Vgain_bottom = acalBottom.AVCC;
   meters.SPI_Write_32(BOTTOM_CS, REG_AIGAIN, Igain_bottom);
```

```
  meters.SPI_Write_32(BOTTOM_CS, REG_AVGAIN, Vgain_bottom);
  vTaskDelay(1000/portTICK_PERIOD_MS);

  // semaphore initialization
  measurements_lock = xSemaphoreCreateBinary();
  xSemaphoreGive(measurements_lock);                    // release measurements_lock
  incoming_lock = xSemaphoreCreateBinary();
  xSemaphoreGive(incoming_lock);                        // release incoming lock
  outgoing_lock = xSemaphoreCreateBinary();
  xSemaphoreGive(outgoing_lock);                        // release outgoing lock

  // begin multitasking
  xTaskCreatePinnedToCore(xbee_uart_event_task, "handle xbee", 8*2048, NULL, 12, NULL, 0);
  xTaskCreatePinnedToCore(parseFrame, "parse incoming frames", 2*32768, NULL, 13, NULL, 1);
  xTaskCreatePinnedToCore(sendFrame, "send frames in xbee queue", 32768, NULL, 20, NULL, 1);
  //xTaskCreatePinnedToCore(printTime, "print time", 16384, NULL, 13, NULL, 1);
  xTaskCreatePinnedToCore(sampleADE9153A, "sample ADE9153AS", 8*2048, NULL, 20, NULL, 1);
  xTaskCreatePinnedToCore(formMeasurementFrames, "form measurements", 16384, NULL, 20, NULL, 1);
  //xTaskCreatePinnedToCore(printHeap, "print heap", 8092, NULL, 20, NULL, 1);
}
```

5.2.2.b Hub Software (KM)

Like the Smart Outlet Module discussed above, the Smart Outlet Hub runs on an ESP32 microprocessor and is developed on the ESP-IDF framework, using the Visual Studio Code extension, Platform IO. The hub is responsible for receiving and sending messages over WIFI to the Control Application and receiving and sending messages over Zigbee to the Smart Outlet Modules. The ZigBee interface used both for the Hub and Outlet Module is discussed in section 5.2.2.f. The Hub operates in STA mode as a Wi-Fi station and connects to the user's home network given an SSID and password. In STA mode, the hub receives an IP address from the DHCP server on the user's home router. The Hub contacts a SNTP server on bootup so that it may be seeded with the current time and date in the form of an epoch time. Upon first communication with the hub, each outlet is then seeded with this time which the hub received from an SNTP server. An API in the form of a HTTP Web Server is hosted on the Hub. Several endpoints are setup on this

webserver, allowing the mobile application to perform HTTP GET requests to retrieve data and/or issue commands to the Hub, which then get forwarded to a corresponding outlet over ZigBee, if necessary. In addition, the Hub also receives instantaneous power data, over ZigBee, from both receptacles in every outlet of the user's home. This data is then aggregated into energy per minute values, formatted as JSON, and posted to a MongoDB database instance. The full WIFI connection, HTTP Server, HTTP Client, SNTP Server, and main code files can be found below.

WIFISetup.h

```
#pragma once

#include "esp_wifi.h"

#ifdef __cplusplus
extern "C"
{
#endif

  void event_handler(void *arg, esp_event_base_t event_base,
                     int32_t event_id, void *event_data);

  void wifi_init_sta(void);

#ifdef __cplusplus
}
#endif
```

WIFISetup.c

```
#include <freertos/FreeRTOS.h>
#include <freertos/event_groups.h>
#include "esp_wifi.h"
#include "esp_log.h"

static const char *TAG = "WIFI Setup";

static EventGroupHandle_t wifi_event_group;

#define SSID "iPhone 12 Pro"
#define PASS "SeniorDesignTeam08"
// #define SSID "WhiteSky-Standard"
// #define PASS "9fg6cj5k"

/* FreeRTOS event group to signal when we are connected*/
static EventGroupHandle_t wifi_event_group;
/* The event group allows multiple bits for each event, but we only care about two events:
 * - we are connected to the AP with an IP
 * - we failed to connect after the maximum amount of retries */
#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT BIT1
#define MAXIMUM_RETRY 10
static int current_retry_num = 0;
```

```
#define TOP_GPIO GPIO_NUM_32
#define BOTTOM_GPIO GPIO_NUM_33

void event_handler(void *arg, esp_event_base_t event_base,
                int32_t event_id, void *event_data)
{

  if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START)
  {
    esp_wifi_connect();
  }
  else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_DISCONNECTED)
  {
    if (current_retry_num < MAXIMUM_RETRY)
    {
      esp_wifi_connect();
      current_retry_num++;
      ESP_LOGI(TAG, "retry to connect to the AP");
    }
    else
    {
      xEventGroupSetBits(wifi_event_group, WIFI_FAIL_BIT);
    }
    ESP_LOGI(TAG, "connect to the AP fail");
  }
  else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP)
  {
    ip_event_got_ip_t *event = (ip_event_got_ip_t *)event_data;
    ESP_LOGI(TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));
    current_retry_num = 0;
    xEventGroupSetBits(wifi_event_group, WIFI_CONNECTED_BIT);
  }
}

void wifi_init_sta(void)
{
  wifi_event_group = xEventGroupCreate();

  ESP_ERROR_CHECK(esp_netif_init());

  ESP_ERROR_CHECK(esp_event_loop_create_default());
  esp_netif_create_default_wifi_sta();

  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK(esp_wifi_init(&cfg));

  esp_event_handler_instance_t instance_any_id;
  esp_event_handler_instance_t instance_got_ip;
  ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                                      ESP_EVENT_ANY_ID,
                                      &event_handler,
                                      NULL,
                                      &instance_any_id));
  ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
                                      IP_EVENT_STA_GOT_IP,
                                      &event_handler,
                                      NULL,
                                      &instance_got_ip));

  wifi_config_t wifi_config = {
     .sta = {
         .ssid = SSID,
         .password = PASS,
         /* Authmode threshold resets to WPA2 as default if password matches WPA2 standards (pasword len => 8).
          * If you want to connect the device to deprecated WEP/WPA networks, Please set the threshold value
          * to WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK and set the password with length and format matching to
          * WIFI_AUTH_WEP/WIFI_AUTH_WPA_PSK standards.
          */
```

```
        .threshold.authmode = WIFI_AUTH_WPA2_PSK,
    },
};
ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config));
ESP_ERROR_CHECK(esp_wifi_start());

ESP_LOGI(TAG, "wifi_init_sta finished.");

/* Waiting until either the connection is established (WIFI_CONNECTED_BIT) or connection failed for the maximum
 * number of re-tries (WIFI_FAIL_BIT). The bits are set by event_handler() (see above) */
EventBits_t bits = xEventGroupWaitBits(wifi_event_group,
                                       WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
                                       pdFALSE,
                                       pdFALSE,
                                       portMAX_DELAY);

/* xEventGroupWaitBits() returns the bits before the call returned, hence we can test which event actually
 * happened. */
if (bits & WIFI_CONNECTED_BIT)
{
  ESP_LOGI(TAG, "connected to ap SSID:%s password:%s",
           SSID, PASS);
}
else if (bits & WIFI_FAIL_BIT)
{
  ESP_LOGI(TAG, "Failed to connect to SSID:%s, password:%s",
           SSID, PASS);
}
else
{
  ESP_LOGE(TAG, "UNEXPECTED EVENT");
}
}
```

timeSetup.hpp

```
#pragma once
#include "json.hpp"
using json = nlohmann::json;

 void obtain_time();
 void returnTime(json frame_overhead_);
```

timeSetup.cpp

```
#include <esp_sntp.h>
#include <esp_log.h>
#include "timeSetup.hpp"
#include "json.hpp"
#include <queue>
#include "xbee_api.hpp"
#include <map>
#include <tuple>
#include <iostream>

// tag for time library
static const char *GET_SNTP_TIME = "get time";

extern std::queue<std::vector<uint8_t>> xbee_outgoing;
extern std::map<uint64_t, uint16_t> outletZigbeeAddresses;
// extern std::map<uint64_t, long int> outletNumberMeasurements;

void time_sync_notification_cb(struct timeval *tv)
{
 ESP_LOGI(GET_SNTP_TIME, "Notification of a time synchronization event");
```

```
  time_t now;
  struct tm *timeinfo;

  time(&now);
  // Set timezone to Eastern Standard Time
  setenv("TZ", "EST5EDT,M3.2.0,M11.1.0", 1);
  tzset();
  timeinfo = localtime(&now);
  printf("Current local time and date: %s", asctime(timeinfo));
}

static void initialize_sntp(void)
{
  ESP_LOGI(GET_SNTP_TIME, "Initializing SNTP");
  sntp_setoperatingmode(SNTP_OPMODE_POLL);
  sntp_setservername(0, "pool.ntp.org");
  sntp_set_time_sync_notification_cb(time_sync_notification_cb);
  sntp_init();
}

void obtain_time()
{
  initialize_sntp();

  // wait for time to be set
  time_t now = 0;
  struct tm timeinfo = {};
  int retry = 0;
  const int retry_count = 10;
  while (sntp_get_sync_status() == SNTP_SYNC_STATUS_RESET && ++retry < retry_count)
  {
    ESP_LOGI(GET_SNTP_TIME, "Waiting for system time to be set... (%d/%d)", retry, retry_count);
    vTaskDelay(2000 / portTICK_PERIOD_MS);
  }
  time(&now);
  localtime_r(&now, &timeinfo);
}

// return current time
void returnTime(json frame_overhead_)
{
  // get return info
  uint64_t destAddr = frame_overhead_["DST64"].get<uint64_t>();
  uint32_t destAddrLowOrder = frame_overhead_["DST16"].get<uint32_t>();

  if (!outletZigbeeAddresses.count(destAddr))
    outletZigbeeAddresses[destAddr] = destAddrLowOrder;

  // // initalize outlet measurement counter
  // outletNumberMeasurements[destAddr] = 0;

  // get current time
  time_t now;
  time(&now);
  std::string stringTime = std::to_string(now);

  // json object to hold time information
  json j = {
      {"op", 4},
      {"data",
       {{"s", now}, {"us", 0}, {"tz", "EST+5EDT,M3.2.0/2,M11.1.0/2"}}}};

  // get string representation of json object
  std::string json_bytes = j.dump();
  std::cout << json_bytes << std::endl;
  // add frame to outgoing xbee queue
  std::vector<uint8_t> frame = formTXFrame(json_bytes, destAddr, destAddrLowOrder, NULL, NULL);
  xbee_outgoing.push(frame);
```

```
};
```

HTTPServer.hpp

```
#pragma once

#include <esp_http_server.h>

  httpd_handle_t start_webserver(void);
  void stop_webserver(httpd_handle_t server);
```

HTTPServer.cpp

```
#include <esp_http_server.h>
#include <esp_log.h>
#include <time.h>
#include <map>
#include <vector>
#include <xbee_api.hpp>
#include <driver/uart.h>
#include <queue>
#include <tuple>
#include <iostream>
#include <sstream>
#include <cstdint>

static const char *TAG = "HTTP Server";

extern std::map<uint64_t, uint16_t> outletZigbeeAddresses;
extern std::queue<std::vector<uint8_t>> xbee_outgoing;
struct powerData
{
  float bP;
  float bPF;
  float tP;
  float tPF;
};
extern std::map<uint64_t, std::pair<uint64_t, powerData>> outletPowerDataSeconds;

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t top_on_handler(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        ESP_LOGI(TAG, "Found URL query parameter => %s", param);
        json j = {
            {"op", 1},
```

```
          {"data", {{"value", 4}}}};
      std::string message = j.dump();
      std::string stringAddress = param;
      uint64_t outletAddr;
      std::istringstream iss(param);
      iss >> outletAddr;
      std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);
      xbee_outgoing.push(messageUART);
      const char resp[] = "Top Outlet Turned On";
      httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
      return ESP_OK;
     }
     else
     {
      std::cout << "Result is: " << result << std::endl;
      return ESP_FAIL;
     }
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t top_off_handler(httpd_req_t *req)
{
  size_t buf_len;

  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        json j = {
            {"op", 1},
            {"data", {{"value", 5}}}};

        std::string message = j.dump();
        std::string stringAddress = param;
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;
        std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);

        xbee_outgoing.push(messageUART);

        /* Send a simple response */
        const char resp[] = "Top Outlet Turned Off";
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
```

```
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t bottom_on_handler(httpd_req_t *req)
{
  size_t buf_len;

  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        json j = {
            {"op", 1},
            {"data", {{"value", 6}}}};

        std::string message = j.dump();
        std::string stringAddress = param;
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;
        std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);

        xbee_outgoing.push(messageUART);

        /* Send a simple response */
        const char resp[] = "Bottom Outlet Turned On";
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t bottom_off_handler(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
```

```
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        json j = {
            {"op", 1},
            {"data", {{"value", 7}}}};
        std::string message = j.dump();

        std::string stringAddress = param;
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;
        std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);

        xbee_outgoing.push(messageUART);

        /* Send a simple response */
        const char resp[] = "Bottom Outlet Turned Off";
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t both_on_handler(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        json j = {
            {"op", 1},
            {"data", {{"value", 3}}}};

        std::string message = j.dump();

        std::string stringAddress = param;
        uint64_t outletAddr;
```

```
      std::istringstream iss(param);
      iss >> outletAddr;
      std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);

      xbee_outgoing.push(messageUART);

      /* Send a simple response */
      const char resp[] = "Both on";
      httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
      return ESP_OK;
     }
     else
     {
      std::cout << "Result is: " << result << std::endl;
      return ESP_FAIL;
     }
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t both_off_handler(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        json j = {
            {"op", 1},
            {"data", {{"value", 0}}}};

        std::string message = j.dump();
        std::string stringAddress = param;
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;
        std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);
        xbee_outgoing.push(messageUART);
        /* Send a simple response */
        const char resp[] = "Both Off";
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
    }
    free(buf);
  }
  return ESP_FAIL;
```

```
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t both_power_handler(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128];
      esp_err_t result;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK)
      {
        std::string stringAddress = param;
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;
        std::pair<uint64_t, powerData> firstMeasurement = outletPowerDataSeconds[outletAddr];

        json j = {
            {"time", firstMeasurement.first},
            {"bottomPower", firstMeasurement.second.bP},
            {"topPower", firstMeasurement.second.tP},
            {"bottomPF", firstMeasurement.second.bPF},
            {"topPF", firstMeasurement.second.tPF}};

        std::string httpResponse = j.dump();

        /* Send a simple response */
        // const char resp[] = httpResponse;
        httpd_resp_send(req, httpResponse.c_str(), HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
    }
    free(buf);
  }
  return ESP_FAIL;
}

//  URI Handler functions
/* Our URI handler function to be called during GET /uri request */
esp_err_t all_outlets(httpd_req_t *req)
{
  // json j = outletZigbeeAddresses;

  json message = {};

  for (auto outlet : outletZigbeeAddresses)
  {
    json j = {
        {"Name", ""},
        {"longAddress", outlet.first},
        {"shortAddress", outlet.second}};
```

```
    message += j;
  }

  std::string httpResponse = message.dump();
  // std::cout << "Server: " << httpResponse << std::endl;
  /* Send a simple response */
  // const char resp[] = httpResponse;
  httpd_resp_send(req, httpResponse.c_str(), HTTPD_RESP_USE_STRLEN);
  return ESP_OK;
}

esp_err_t power_limit(httpd_req_t *req)
{
  size_t buf_len;
  /* Read URL query string length and allocate memory for length + 1,
   * extra byte for null termination */
  buf_len = httpd_req_get_url_query_len(req) + 1;
  if (buf_len > 1)
  {
    char *buf = (char *)malloc(buf_len);
    if (httpd_req_get_url_query_str(req, buf, buf_len) == ESP_OK)
    {
      ESP_LOGI(TAG, "Found URL query => %s", buf);
      char param[128*2];
      char powerLim[128];
      char c_duration[128];
      esp_err_t result;
      esp_err_t result2;
      esp_err_t result3;
      result = httpd_query_key_value(buf, "Address", param, sizeof(param));
      result2 = httpd_query_key_value(buf, "PLim", powerLim, sizeof(param));
      result3 = httpd_query_key_value(buf, "Dur", c_duration, sizeof(c_duration));
      free(buf);
      /* Get value of expected key from query string */
      if (result == ESP_OK && result2 == ESP_OK && result3 == ESP_OK)
      {
        ESP_LOGI(TAG, "Found URL query parameter => %s", param);
        ESP_LOGI(TAG, "Found URL query parameter => %s", powerLim);
        ESP_LOGI(TAG, "Found URL query parameter => %s", c_duration);

        float pLim = std::stof(powerLim);
        int dur =std::stoi(c_duration);
        uint64_t outletAddr;
        std::istringstream iss(param);
        iss >> outletAddr;

        // TODO: Format for power limit
        json j = {
            {"op", 3},
            {"data", {{"value", pLim}, {"duration", dur}}}};
        std::string message = j.dump();
        std::cout << "8"<< std::endl;
        std::vector<uint8_t> messageUART = formTXFrame(message, outletAddr, outletZigbeeAddresses[outletAddr], NULL, NULL);
        xbee_outgoing.push(messageUART);
        const char resp[] = "Power Limit Sent";
        httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
        return ESP_OK;
      }
      else
      {
        std::cout << "Result is: " << result << std::endl;
        return ESP_FAIL;
      }
    }
    free(buf);
  }
  return ESP_FAIL;
}
```

```
/* Our URI handler function to be called during POST /uri request */
esp_err_t post_handler(httpd_req_t *req)
{
  /* Destination buffer for content of HTTP POST request.
   * httpd_req_recv() accepts char* only, but content could
   * as well be any binary data (needs type casting).
   * In case of string data, null termination will be absent, and
   * content length would give length of string */
  char content[100];

  /* Truncate if content length larger than the buffer */
  size_t recv_size = req->content_len > sizeof(content) ? sizeof(content) : req->content_len; // MIN(req->content_len, sizeof(content));

  int ret = httpd_req_recv(req, content, recv_size);
  ESP_LOGI(TAG, "Post message: %s", content);
  if (ret <= 0)
  { /* 0 return value indicates connection closed */
    /* Check if timeout occurred */
    if (ret == HTTPD_SOCK_ERR_TIMEOUT)
    {
      /* In case of timeout one can choose to retry calling
       * httpd_req_recv(), but to keep it simple, here we
       * respond with an HTTP 408 (Request Timeout) error */
      httpd_resp_send_408(req);
    }
    /* In case of error, returning ESP_FAIL will
     * ensure that the underlying socket is closed */
    return ESP_FAIL;
  }

  /* Send a simple response */
  const char resp[] = "URI POST Response";
  httpd_resp_send(req, resp, HTTPD_RESP_USE_STRLEN);
  return ESP_OK;
}

/* URI handler structure for top receptacle on */
httpd_uri_t uri_top_on = {
    .uri = "/top/on",
    .method = HTTP_GET,
    .handler = top_on_handler,
    .user_ctx = NULL};

/* URI handler structure for top receptacle off */
httpd_uri_t uri_top_off = {
    .uri = "/top/off",
    .method = HTTP_GET,
    .handler = top_off_handler,
    .user_ctx = NULL};

/* URI handler structure for bottom receptacle on */
httpd_uri_t uri_bottom_on = {
    .uri = "/bottom/on",
    .method = HTTP_GET,
    .handler = bottom_on_handler,
    .user_ctx = NULL};

/* URI handler structure for  bottom receptacle off */
httpd_uri_t uri_bottom_off = {
    .uri = "/bottom/off",
    .method = HTTP_GET,
    .handler = bottom_off_handler,
    .user_ctx = NULL};

/* URI handler structure for  bottom receptacle off */
httpd_uri_t uri_both_on = {
    .uri = "/both/on",
```

```
    .method = HTTP_GET,
    .handler = both_on_handler,
    .user_ctx = NULL};

/* URI handler structure for  bottom receptacle off */
httpd_uri_t uri_both_off = {
    .uri = "/both/off",
    .method = HTTP_GET,
    .handler = both_off_handler,
    .user_ctx = NULL};

/* URI handler structure for top receptacle on */
httpd_uri_t uri_both_power = {
    .uri = "/both/power",
    .method = HTTP_GET,
    .handler = both_power_handler,
    .user_ctx = NULL};

/* URI handler structure for top receptacle on */
httpd_uri_t uri_all_outlets = {
    .uri = "/allOutlets",
    .method = HTTP_GET,
    .handler = all_outlets,
    .user_ctx = NULL};

/* URI handler structure for top receptacle on */
httpd_uri_t uri_pow_Lim = {
    .uri = "/powerLimit",
    .method = HTTP_GET,
    .handler = power_limit,
    .user_ctx = NULL};

/* URI handler structure for POST /uri */
httpd_uri_t uri_post = {
    .uri = "/uri",
    .method = HTTP_POST,
    .handler = post_handler,
    .user_ctx = NULL};

/* Function for starting the webserver */
httpd_handle_t start_webserver(void)
{
  /* Generate default configuration */
  httpd_config_t config = HTTPD_DEFAULT_CONFIG();
  config.core_id = 0;
  config.server_port = 80;
  config.ctrl_port = 5555;

  /* Empty handle to esp_http_server */
  httpd_handle_t server = NULL;

  config.max_uri_handlers = 15;

  /* Start the httpd server */
  if (httpd_start(&server, &config) == ESP_OK)
  {
    /* Register URI handlers */
    httpd_register_uri_handler(server, &uri_top_on);
    httpd_register_uri_handler(server, &uri_top_off);
    httpd_register_uri_handler(server, &uri_bottom_on);
    httpd_register_uri_handler(server, &uri_bottom_off);
    httpd_register_uri_handler(server, &uri_both_on);
    httpd_register_uri_handler(server, &uri_both_off);
    httpd_register_uri_handler(server, &uri_both_power);
    httpd_register_uri_handler(server, &uri_all_outlets);
    httpd_register_uri_handler(server, &uri_pow_Lim);
    // httpd_register_uri_handler(server, &uri_name_outlet);
```

```
    httpd_register_uri_handler(server, &uri_post);
  }
  /* If server failed to start, handle will be NULL */
  return server;
}

/* Function for stopping the webserver */
void stop_webserver(httpd_handle_t server)
{
  if (server)
  {
    /* Stop the httpd server */
    httpd_stop(server);
  }
}
```

Client.hpp

```
#pragma once

// #ifdef __cplusplus
// extern "C"
// {
// #endif
esp_err_t https_with_url(uint64_t address, std::string body);

// #ifdef __cplusplus
// }
// #endif
```

Client.cpp

```
#include <string.h>
#include <sys/param.h>
#include <stdlib.h>
#include <ctype.h>
#include "esp_log.h"
#include "nvs_flash.h"
#include "esp_event.h"
#include "esp_netif.h"
#include "esp_tls.h"
#if CONFIG_MBEDTLS_CERTIFICATE_BUNDLE
#include "esp_crt_bundle.h"
#endif
#include "iostream"
#if !CONFIG_IDF_TARGET_LINUX
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#endif

#include "esp_http_client.h"

#define MAX_HTTP_RECV_BUFFER 512
#define MAX_HTTP_OUTPUT_BUFFER 2048
static const char *TAG = "HTTP_CLIENT";

// Define client certificate
extern const char howsmyssl_com_root_cert_pem_start[] asm("_binary_howsmyssl_com_root_cert_pem_start");
extern const char howsmyssl_com_root_cert_pem_end[]   asm("_binary_howsmyssl_com_root_cert_pem_end");

esp_err_t _http_event_handler(esp_http_client_event_t *evt)
{
    static char *output_buffer;  // Buffer to store response of http request from event handler
```

```
static int output_len;       // Stores number of bytes read
switch(evt->event_id) {
    case HTTP_EVENT_ERROR:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_ERROR");
        break;
    }
    case HTTP_EVENT_ON_CONNECTED:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_ON_CONNECTED");
        break;
    }
    case HTTP_EVENT_HEADER_SENT:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_HEADER_SENT");
        break;
    }
    case HTTP_EVENT_ON_HEADER:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_ON_HEADER, key=%s, value=%s", evt->header_key, evt->header_value);
        break;
    }
    case HTTP_EVENT_ON_DATA:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_ON_DATA, len=%d", evt->data_len);
        /*
         *  Check for chunked encoding is added as the URL for chunked encoding used in this example returns binary data.
         *  However, event handler can also be used in case chunked encoding is used.
         */
        if (!esp_http_client_is_chunked_response(evt->client)) {
            // If user_data buffer is configured, copy the response into the buffer
            int copy_len = 0;
            if (evt->user_data) {
                copy_len = MIN(evt->data_len, (MAX_HTTP_OUTPUT_BUFFER - output_len));
                if (copy_len) {
                    memcpy(evt->user_data + output_len, evt->data, copy_len);
                }
            } else {
                const int buffer_len = esp_http_client_get_content_length(evt->client);
                if (output_buffer == NULL) {
                    output_buffer = (char *) malloc(buffer_len);
                    output_len = 0;
                    if (output_buffer == NULL) {
                        ESP_LOGE(TAG, "Failed to allocate memory for output buffer");
                        return ESP_FAIL;
                    }
                }
                copy_len = MIN(evt->data_len, (buffer_len - output_len));
                if (copy_len) {
                    memcpy(output_buffer + output_len, evt->data, copy_len);
                }
            }
            output_len += copy_len;
        }

        break;
    }
    case HTTP_EVENT_ON_FINISH:
    {
        ESP_LOGD(TAG, "HTTP_EVENT_ON_FINISH");
        if (output_buffer != NULL) {
            // Response is accumulated in output_buffer. Uncomment the below line to print the accumulated response
            // ESP_LOG_BUFFER_HEX(TAG, output_buffer, output_len);
            free(output_buffer);
            output_buffer = NULL;
        }
        output_len = 0;
        break;
```

```
        }
        case HTTP_EVENT_DISCONNECTED:
        {
            ESP_LOGI(TAG, "HTTP_EVENT_DISCONNECTED");
            int mbedtls_err = 0;
            esp_err_t err = esp_tls_get_and_clear_last_error((esp_tls_error_handle_t)evt->data, &mbedtls_err, NULL);
            if (err != 0) {
                ESP_LOGI(TAG, "Last esp error code: 0x%x", err);
                ESP_LOGI(TAG, "Last mbedtls failure: 0x%x", mbedtls_err);
            }
            if (output_buffer != NULL) {
                free(output_buffer);
                output_buffer = NULL;
            }
            output_len = 0;
            break;
        }
        case HTTP_EVENT_REDIRECT:
        {
            ESP_LOGD(TAG, "HTTP_EVENT_REDIRECT");
            esp_http_client_set_header(evt->client, "From", "user@example.com");
            esp_http_client_set_header(evt->client, "Accept", "text/html");
            esp_http_client_set_redirection(evt->client);
            break;
        }

    }
    return ESP_OK;
}

#if CONFIG_MBEDTLS_CERTIFICATE_BUNDLE
esp_err_t https_with_url(uint64_t address, std::string body)
{
    std::cout << "Entering Post method" << std::endl;
    std::string URL = "https://us-east-1.aws.data.mongodb-api.com/app/sdppoweroutlet-ogswv/endpoint/outlet?Address=" +
std::to_string(address);
    std::cout << "URL is: " << URL << std::endl;
    char local_response_buffer[MAX_HTTP_OUTPUT_BUFFER] = {0};

    esp_http_client_config_t config = {};
    config.url = URL.c_str();//"https://us-east-1.aws.data.mongodb-api.com/app/sdppoweroutlet-ogswv/endpoint/outlet?Address=8";
    config.event_handler = _http_event_handler;
    config.user_data = local_response_buffer;
    config.cert_pem = howsmyssl_com_root_cert_pem_start;
    //config.port = 443;
    // config.is_async = true;
    config.timeout_ms = 10000;
    //config.buffer_size =

    std::cout << "Before Init handle" << std::endl;

    esp_http_client_handle_t client = esp_http_client_init(&config);

    std::cout << "Init handle" << std::endl;

    // POST
    const char *post_data = body.c_str(); //"{\"TopP\":\"6\", \"BottomP\":\"7\", \"EpochTime\":\"1680634492\"}";
    std::cout << "Data to be sent to DB: " << post_data << std::endl;
    //esp_http_client_set_url(client, "https://us-east-1.aws.data.mongodb-api.com/app/sdppoweroutlet-ogswv/endpoint/outlet?Address=6");
    esp_http_client_set_method(client, HTTP_METHOD_POST);
    esp_http_client_set_header(client, "Content-Type", "application/json");
    esp_http_client_set_post_field(client, post_data, strlen(post_data));

    std::cout << "Before perform: " << std::endl;
    esp_err_t err = esp_http_client_perform(client);
    std::cout << "after perform: " << err << std::endl;

    if (err == ESP_OK) {
```

```
        ESP_LOGI(TAG, "HTTP POST Status = %d, content_length = %" PRIu64,
                esp_http_client_get_status_code(client),
                esp_http_client_get_content_length(client));
    } else {
        ESP_LOGE(TAG, "HTTP POST request failed: %s", esp_err_to_name(err));
    }
    esp_http_client_cleanup(client);
    return err;
}
#endif // CONFIG_MBEDTLS_CERTIFICATE_BUNDLE
```

main.cpp

```
// Joseph Garro Kyrolos Melek
// jmg289@uakron.edu
// 3/25/2023
// v1.1
// firmware for esp32 and xbee based smart power outlet
// resources used https://github.com/theElementZero/ESP32-UART-interrupt-handling/blob/master/uart_interrupt.c
//---------------------------------

// additional classes for functionality
#include "xbee_api.hpp"
#include "json.hpp"
#include "../lib/Wifi/WIFISetup.h"
#include "timeSetup.hpp"
#include "HTTPServer.hpp"
#include "esp_http_client.h"
#include "Client.hpp"

// esp32 classes
#include "stdio.h"             // standard io
#include "driver/gpio.h"       // esp GPIO pin control
#include "driver/uart.h"       // esp UART driver
#include "driver/spi_master.h" // esp SPI driver
#include "driver/spi_common.h"
#include "freertos/FreeRTOS.h" // freeRTOS for multitasking
#include "freertos/task.h"     // create and schedudle tasks
#include "freertos/queue.h"
#include "esp_log.h"
#include "esp_task_wdt.h"
#include "esp_sntp.h"
#include "nvs_flash.h"
#include "esp_task_wdt.h"

// c++ classes
#include <iostream>
#include <queue>
#include <ctime>
#include <time.h>
#include <map>
#include <tuple>
#include <algorithm>

// link json class
using json = nlohmann::json;

//---------------------------------
// global definitions
#define BUFFER_SIZE (1024 * 4)             // hold 4096 bytes in each buffer
const int SIGFIGS = 10000;                 // remove decimal point
static QueueHandle_t xbee_queue;           // queue to handle xbee events

// queue for pointers to incoming XBEE frames
std::queue<std::vector<uint8_t>> xbee_incoming;
// queue for pointers to outgoing XBEE frames
std::queue<std::vector<uint8_t>> xbee_outgoing;
```

```
// map to hold outlet Addresses
std::map<uint64_t, uint16_t> outletZigbeeAddresses;

// struct to hold power measurement information
struct powerData
{
   float bP;
   float bPF;
   float tP;
   float tPF;
   int numOfMeasurements = 0;
   uint64_t epochTime;
};

std::map<uint64_t, std::pair<uint64_t, powerData>> outletPowerDataSeconds;   // map to store most recent power measurement for a given
outlet-> live power view

//This will map each outlet to its current minute data, once the minute data is ready to send (i.e. 60 measurements have been aggregated) it will be
sent then replaced by the next minute data to send
std::queue<std::pair<uint64_t, powerData>> outletMinuteEnergyData;
std::map<uint64_t, powerData> outletMinuteEnergyDataAggregation;

// GPIO pin definitions
#define XBEE_UART (UART_NUM_2)     // uart2 to communicate between xbee and esp32
#define XBEE_UART_RX (GPIO_NUM_18) // uart2 TX
#define XBEE_UART_TX (GPIO_NUM_19) // uart2 RX

// logging tags
static const char *XBEE_TAG = "xbee uart";
static const char *PWIC_TAG = "PWIC uart";
static const char *RECEPTACLE_TAG = "receptacle state";
static const char *MAX_INSTANTANEOUS_POWER_DRAW_TAG = "set max instantanous power draw";
static const char *MAX_SUSTAINED_POWER_DRAW_TAG = "set max sustained power draw";
static const char *SET_SYSTEM_TIME = "set system time";
static const char *GET_SNTP_TIME = "get time";
static const char *PARSE_FRAME = "parse xbee frame";
static const char *SETUP = "setup";

static httpd_handle_t server_handle ;
//--------------------------------
// function definitions

// configure UART connection to xbee module
static void xbee_uart_init(void)
{
   ESP_LOGI(XBEE_TAG, "configuring xbee uart connection");
   // UART configuration settings
   const uart_config_t xbee_uart_config = {
      .baud_rate = 115200,
      .data_bits = UART_DATA_8_BITS,
      .parity = UART_PARITY_DISABLE,
      .stop_bits = UART_STOP_BITS_1,
      .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
      .rx_flow_ctrl_thresh = 122,
      .source_clk = UART_SCLK_APB};

   // install UART driver
   ESP_ERROR_CHECK(uart_driver_install(XBEE_UART, BUFFER_SIZE * 8, BUFFER_SIZE * 8, 1024, &xbee_queue, 0));      // install
UART driver on pins connected to xbee, buffer of 2048 bytes, event queue enabled
   ESP_ERROR_CHECK(uart_param_config(XBEE_UART, &xbee_uart_config));                                    // write xbee_uart_config to xbee
UART
   ESP_ERROR_CHECK(uart_set_pin(XBEE_UART, XBEE_UART_TX, XBEE_UART_RX, UART_PIN_NO_CHANGE,
UART_PIN_NO_CHANGE)); // assign TX and RX pins to xbee UART
};

// UART event handler for XBEE
// synthesized from https://github.com/espressif/esp-
idf/blob/49551cc48cb3cdd5563059028749616de313f0ec/examples/peripherals/uart/uart_events/main/uart_events_example_main.c
```

```
static void xbee_uart_event_task(void *pvParameters)
{
    uart_event_t xbee_event;                        // hold UART event
    uint8_t *dtmp = (uint8_t *)malloc(BUFFER_SIZE); // temporary buffer
    for (;;)
    {
        // activate when a UART event is detected
        if (xQueueReceive(xbee_queue, (void *)&xbee_event, (TickType_t)portMAX_DELAY))
        {
            size_t eventsize = xbee_event.size;
            bzero(dtmp, BUFFER_SIZE);
            ESP_LOGI(XBEE_TAG, "uart[%d] event:", XBEE_UART); // zero buffer
            switch (xbee_event.type)
            { // handle different UART events
            // read incoming UART data
            case UART_DATA:
                ESP_LOGI(XBEE_TAG, "[UART DATA]: %d", eventsize);
                uart_read_bytes(XBEE_UART, dtmp, eventsize, portMAX_DELAY); // write data to dtmp

                xbee_incoming.push(std::vector<uint8_t>(dtmp, dtmp + eventsize)); // push frame to incoming xbee queue

                break;

            // HW FIFO overflow detected
            case UART_FIFO_OVF:
                ESP_LOGI(XBEE_TAG, "hw fifo overflow");
                uart_flush_input(XBEE_UART);
                xQueueReset(xbee_queue);
                break;

            // UART ring buffer full
            case UART_BUFFER_FULL:
                ESP_LOGI(XBEE_TAG, "ring buffer full");
                uart_flush_input(XBEE_UART);
                xQueueReset(xbee_queue);
                break;

            // UART RX break detected
            case UART_BREAK:
                ESP_LOGI(XBEE_TAG, "uart rx break");
                break;

            // UART parity check error
            case UART_PARITY_ERR:
                ESP_LOGI(XBEE_TAG, "uart parity error");
                break;

            // UART frame error
            case UART_FRAME_ERR:
                ESP_LOGI(XBEE_TAG, "uart frame error");
                break;

            default:
                ESP_LOGI(XBEE_TAG, "xbee UART event: %d", xbee_event.type);
                break;
            }
        }
    }
    free(dtmp);
    dtmp = NULL;
    vTaskDelete(NULL);
};

// perform an action in the hub based on the contents of recieved frame
void performHubAction(json *json_object)
{
    json frame_payload = (*json_object)["FRAME DATA"];
    json frame_overhead = (*json_object)["FRAME OVERHEAD"];
```

```
if (!frame_payload["op"].is_null() && frame_payload["op"].is_number())
{
   //  get type of JSON
   int frame_type = frame_payload["op"].get<int>();
   // get data from json packet
   switch (frame_type)
   {
   // outlet will not recieve measurement packets, ignore type == 0
   // process measurements
   case 101:
   {
      powerData pd = {};
      json bottom = frame_payload["data"]["b"];
      json top = frame_payload["data"]["t"];

      pd.bP = bottom["p"].get<double>() / SIGFIGS;
      pd.bPF = bottom["f"].get<double>() / SIGFIGS;
      pd.tP = top["p"].get<double>() / SIGFIGS;
      pd.tPF = top["f"].get<double>() / SIGFIGS;
      pd.epochTime = frame_payload["data"]["s"].get<long>();

      uint64_t ZigBAddLong = frame_overhead["DST64"].get<uint64_t>();

      uint32_t destAddrLowOrder = frame_overhead["DST16"].get<uint32_t>();
      if(!outletZigbeeAddresses.contains(ZigBAddLong))
         outletZigbeeAddresses[ZigBAddLong] = destAddrLowOrder;

      outletPowerDataSeconds[ZigBAddLong] = std::make_pair(pd.epochTime, pd);

      if(!outletMinuteEnergyDataAggregation.contains(ZigBAddLong) && (pd.epochTime%60 <= 5))
      {
         std::cout << "adding data to be aggregated" << std::endl;
         outletMinuteEnergyDataAggregation[ZigBAddLong] = pd;
      }
      else if(outletMinuteEnergyDataAggregation.contains(ZigBAddLong))
      {
         std::cout << "aggregating data:" << outletMinuteEnergyDataAggregation[ZigBAddLong].numOfMeasurements << std::endl;
         outletMinuteEnergyDataAggregation[ZigBAddLong].tP += pd.tP;
         outletMinuteEnergyDataAggregation[ZigBAddLong].bP += pd.bP;
         outletMinuteEnergyDataAggregation[ZigBAddLong].numOfMeasurements += 1;
         if(outletMinuteEnergyDataAggregation[ZigBAddLong].numOfMeasurements == 60)
         {
            outletMinuteEnergyDataAggregation[ZigBAddLong].epochTime = pd.epochTime;
            std::cout << "pushing data" << std::endl;
            outletMinuteEnergyData.push(std::pair(ZigBAddLong,outletMinuteEnergyDataAggregation[ZigBAddLong]));
            outletMinuteEnergyDataAggregation[ZigBAddLong] = {};
         }
      }
      break;
   }
   // handle setting maximum instaneous power draw
   case 102:
      // setMaximumInstantaneousPowerDraw(json_data);
      break;

   // handle maximum sustained power draw
   case 103:
      // setMaximumSustainedPowerDraw(json_data);
      break;

   // handle time set
   case 104:
      returnTime(frame_overhead);
      break;
   default:
      break;
   };
};
```

```
};

/*
 * xbee frames arrive in the following format
      {
         "FRAME TYPE", X,                          -- type of xbee frame, IE at command response, transmit request, etc
         "FRAME OVERHEAD", {                           -- data relevant to xbee protocol, IE frame ID, destination, etc
            XXX, XXX
            .
            .
            .
         },
         "FRAME DATA", {                           -- data to perform outlwt interactions with
            "data"{                             -- data to act on, necessary data for operations will be found here in the expected key-value pairs
               "value", X,
               .
               .
            }
            "op", x,                            -- operation, IE toggle receptacles, set power limit, etc
         }
      }
*/
// determine action to take based on recieved XBEE frame
static void parseFrame(void *pvParameters)
{
   for (;;)
   {
      // work on existing xbee frames
      while (!xbee_incoming.empty())
      {
         // get oldest xbee frame
         std::vector<uint8_t> xbee_frame = xbee_incoming.front();
         // remove xbee frame from queue
         xbee_incoming.pop();
         // copy vector to a uint8_t array
         // json with all information about xbee frame
         json j = readFrame(xbee_frame.data());
         //  handle error cases
         //  unrecognized frame
         if (j == -1)
         {
            ESP_LOGI(PARSE_FRAME, "recieved an unrecognized frame");
         }
         // invalid frame
         else if (j == -2)
         {
            ESP_LOGI(PARSE_FRAME, "recieved an invalid frame");
         }
         // otherwise do something
         else
         {
            // get type of frame
            uint8_t frameType = j["FRAME TYPE"].get<int>();
            switch (frameType)
            {
            // rx response                          -- transmit request will include data -> means this is an outlet action
            case 0x90:
               performHubAction(&j);
               break;

            // explicit rx response                    -- transmit request will include data -> means this is an outlet action
            case 0x91:
               performHubAction(&j);
               break;
            // all other cases                      -- other operations deal with XBee behavior -> do not need ESP32's attention
            default:
               break;
            };
```

```
            vTaskDelay(1);
        }
        vTaskDelay(1);
    }
    vTaskDelay(1);
  }
};

// send xbee frames stored in queue
static void sendFrame(void *pvParameters)
{
  for (;;)
  {
    // work on existing xbee frames
    while (!xbee_outgoing.empty())
    {
      // get oldest xbee frame
      std::vector<uint8_t> xbee_frame = xbee_outgoing.front();
      // remove xbee frame from queue
      xbee_outgoing.pop();
      uart_write_bytes(XBEE_UART, xbee_frame.data(), xbee_frame.size());
      vTaskDelay(1);
    }
    vTaskDelay(100);
  }
};

static void printHeap(void *pvParameter)
{
  for (;;)
  {
    std::cout << "Free Heap Size: " << esp_get_free_heap_size() << std::endl;
    std::cout << "Largest block : " << heap_caps_get_largest_free_block(MALLOC_CAP_8BIT) << std::endl;
    vTaskDelay(10000 / portTICK_PERIOD_MS);
  }
}

static void uploadMeasurements(void *pvParameter)
{
  for(;;)
  {
    while(!outletMinuteEnergyData.empty())
    {
      stop_webserver(server_handle);
      std::cout << "About to Post data" << std::endl;
      std::pair<uint64_t, powerData> dataToSend = outletMinuteEnergyData.front();

      uint64_t longAddress = dataToSend.first;

      json j = {
      {"TopP", dataToSend.second.tP },
      {"BottomP", dataToSend.second.bP},
      {"EpochTime", dataToSend.second.epochTime}};
      std::cout << "Posting data" << std::endl;
      vTaskDelay(100);
      esp_err_t err = https_with_url(longAddress,j.dump());
      if(err == ESP_OK)
        outletMinuteEnergyData.pop();
      vTaskDelay(100);
      server_handle = start_webserver();
      std::cout << "after posting data" << std::endl;
    }
    vTaskDelay(100);
  }
}
// main function
extern "C" void app_main()
{
```

```
    std::queue<std::vector<uint8_t>> empty;
    std::swap(xbee_outgoing, empty);

    // Initialize NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND)
    {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    // initalize wifi
    wifi_init_sta();
    // get time from SNTP server
    obtain_time();

    // initalize serial connections
    xbee_uart_init(); // initalize xbee UART connection

    // webserver
    server_handle = start_webserver();
    // json j = {
    //         {"TopP", 8 },
    //         {"BottomP", 9},
    //         {"EpochTime", 1680634492}};

    // https_with_url(55,j.dump());

    // begin multitasking
    xTaskCreate(xbee_uart_event_task, "handle xbee", 8 * 1024, NULL, 12, NULL);
    xTaskCreate(parseFrame, "parse incoming frames", 32768 / 2, NULL, 13, NULL);
    xTaskCreate(sendFrame, "parse incoming frames", 32768 / 2, NULL, 13, NULL);
    xTaskCreate(printHeap, "print heap", 2048, NULL, 20, NULL);
    xTaskCreatePinnedToCore(uploadMeasurements, "Upload measurements", 32768, NULL, 21, NULL, 1);

}
```

5.2.2.c MongoDB Database (KM)

As mentioned above, the Hub posts data sent to it from the outlets to a MongoDB database instance. The format and dashboard of this database can be seen in the below figure.

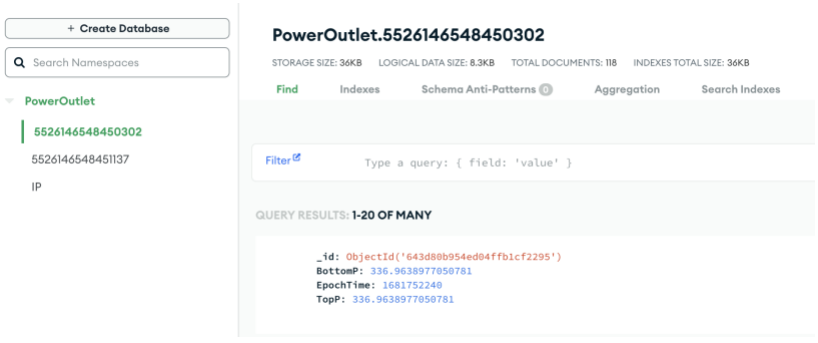

Figure 42: MongoDB Dashboard

As can be seen in the above figure, the database follows the hierarchy structure of Database -> Collection -> Document. The database is the overarching structure which contains all data relevant to the product. Each collection within the database represents an outlet in the system, identified by the outlets 64-bit ZigBee address. Each outlet collection contains several documents which each represent a given minutes' worth of energy consumption. Each document contains a GUID, or object id assigned by MongoDB, a bottom receptacle energy value, a top receptacle energy value, and an epoch time corresponding to the time and date the energy data was measured. The Hub interacts with the database through an HTTPS API endpoint, or webserver, hosted on MongoDB. The Hub uses an HTTPS client to perform HTTPS POST requests on the HTTPS endpoint. This endpoint processes the data sent to it, creates the database if not already created, creates a collection corresponding to each outlet if not already created, and posts the relevant power data to said collection. The control application interacts with the database through the native MongoDB C# driver, from which it can perform all the necessary CRUD operations. The code for the HTTPS endpoint written in JavaScript and hosted on MongoDB can be found below.

```
// This function is the endpoint's request handler.
exports = function({ query, headers, body}, response) {
    const OutletAddress = query;
   var bodyObj = JSON.parse(body.text());
   const result = context.services
   .get("mongodb-atlas")
   .db("PowerOutlet")
   .collection(OutletAddress["Address"])
   .insertOne(bodyObj);
   const contentTypes = headers["Content-Type"];
   return bodyObj;
};
```

5.2.2.d Control Application (KM)

The control application acts as the main entry point for the Smart Power Outlet System. It is here where the user will be able to view and control each outlets power/energy usage and status. The application was built in .NET MAUI using C# and the .NET Framework of tools. The

application follows an easily testable, reusable, and maintainable architecture known as MVVMS or Model, View, View-Model, Services and is completely cross platform with Mac OS, Windows, IOS, and Android. In general, energy data and statistics are retrieved from the cloud hosted MongoDB database instance while instantaneous power data, online outlets, and outlet commands are retrieved/sent directly from/to the hub. The main view or page of the application shows all online outlets, as well as general whole home statistics such as total home cost and energy usage and can be seen in the below figure. It is from this home page that the user can assign names to each outlet, so they are easily recognizable.

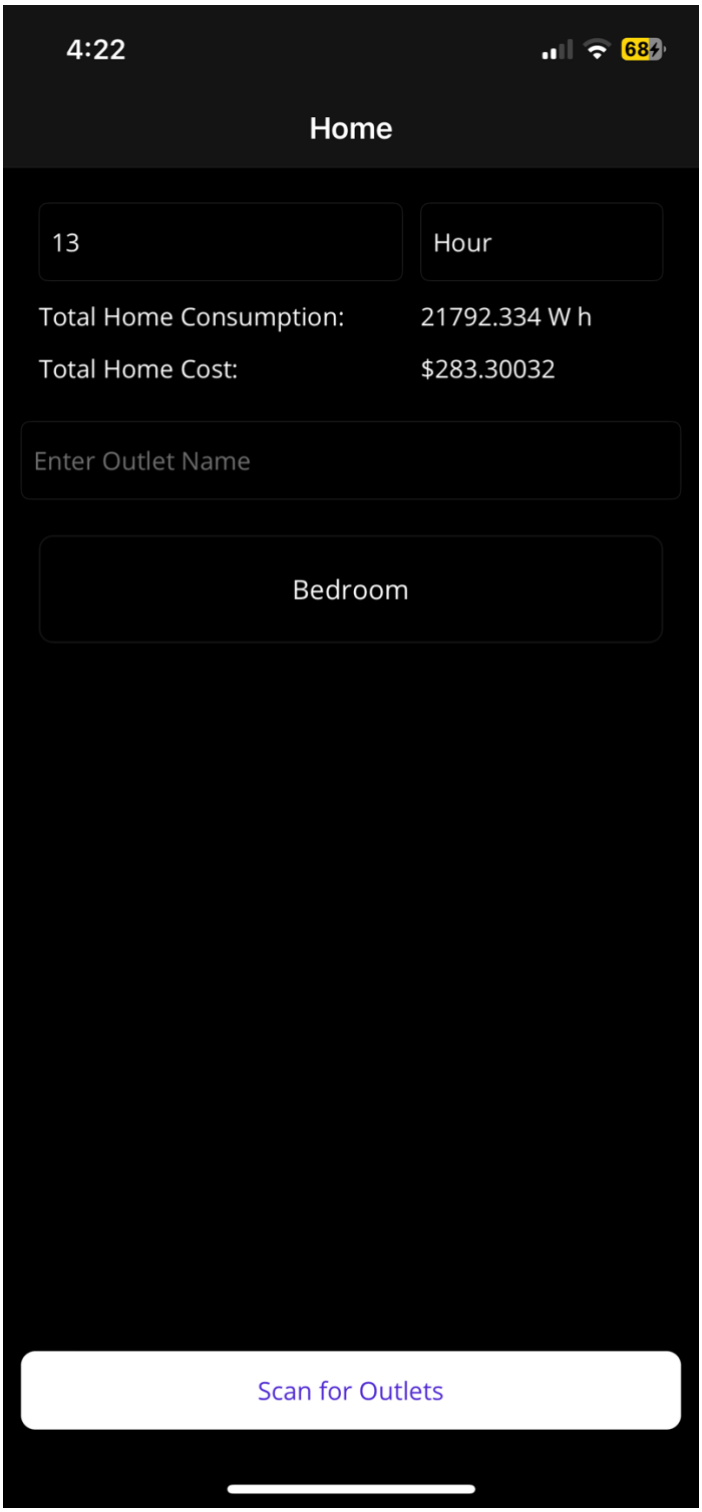

Figure 43: Control Application Home Screen

Upon clicking on a specific outlet, the user will be taken to a details page where they can view all data related to that outlet, in addition to the controls for that outlet. A screenshot of this view can be seen below.

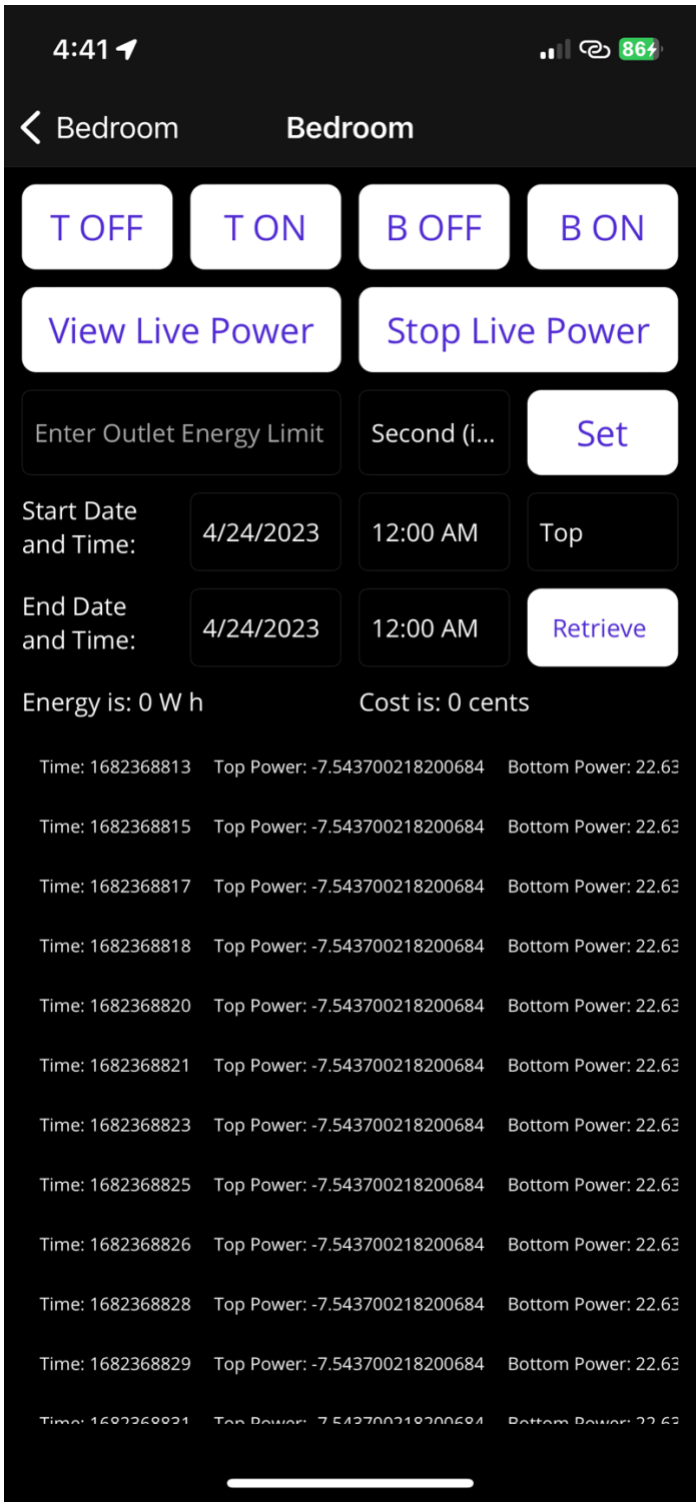

Figure 44: Control Application Details Page

Each view or page is built using ZAML, a markup language like HTML, and has an associated view model where the logic for the view lies. Data such as energy data retrieved from the database, instantaneous power received directly from the hub, and properties that are associated with each outlet are stored in Models, following the MVVM architecture. Finally, services, such as the service used to contact the database and webserver hosted on the hub, also lie in their own files. The full code for each Model, View, View-Model, and Service can be found below.

InstantaneousPowerData.cs

```
using System;
namespace PowerOutlet.Model;
using Newtonsoft.Json;

[JsonObject(MemberSerialization.OptIn)]
public class InstantaneousPowerData
{
        public InstantaneousPowerData()
        {
        }

   [JsonProperty("Bottom PF")]
   public double BottomPF { get; set; }

   [JsonProperty("Bottom Power")]
   public double BottomPower { get; set; }
   public int Time { get; set; }

   [JsonProperty("Top PF")]
   public double TopPF { get; set; }

   [JsonProperty("Top Power")]
   public double TopPower { get; set; }

   //"{\"Bottom PF\":0.8889999985694885,\"Bottom Power\":5.52400016784668,\"Time\":1681756486,\"Top PF\":0.8889999985694885,\"Top
Power\":5.52400016784668}"
}
```

MongoData.cs

```
using System;
namespace PowerOutlet.Model;

using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

[BsonIgnoreExtraElements]
public class MongoData
{

        public MongoData()
```

```
        {
        }

        [BsonId]
        public ObjectId _id { get; set; }

        [BsonElement("BottomP")]
        public float BottomP { get; set; }


    [BsonElement("EpochTime")]
        public UInt64 EpochTime { get; set; }

    [BsonElement("TopP")]
    public float TopP { get; set; }
}
```

OutletProperties.cs

```
using System;
namespace PowerOutlet.Model
{
        public class OutletProperties
        {
                public OutletProperties()
                {
                }


                public string name { get; set; }
                public ulong longAddress { get; set; }
                public ushort shortAddress { get; set; }
    }
}
```

MainPage.xaml

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:viewmodel="clr-namespace:PowerOutlet.ViewModel"
        xmlns:model ="clr-namespace:PowerOutlet.Model"
        x:DataType="viewmodel:MainViewModel"
        Title="{Binding Title}"
        x:Class="PowerOutlet.MainPage">

    <Grid RowDefinitions="Auto, Auto, *, Auto"
        ColumnDefinitions=".75*, .25*"
        Padding="10" RowSpacing="10" ColumnSpacing="10">

        <Grid Grid.Row="0"
            Grid.ColumnSpan="2"
            RowDefinitions="Auto, Auto, Auto"
            ColumnDefinitions=".6*, .4*"
            Padding="10" RowSpacing="10" ColumnSpacing="10">

            <Entry Placeholder="Enter Power Cost per kWHr"
                PlaceholderColor="DarkGray"
                Grid.Row="0"
                Grid.Column="0"
                Grid.ColumnSpan="1"
                Text="{Binding CostPerkWH}"/>

            <Picker Grid.Row="0"
                Grid.Column="2"
```

```
            Title="Select a Time Duration"
            SelectedIndex="{Binding DurationIndex}">
          <Picker.ItemsSource>
            <x:Array Type="{x:Type x:String}">
              <x:String>Hour</x:String>
              <x:String>Day</x:String>
              <x:String>Week</x:String>
              <x:String>Month</x:String>
              <x:String>Year</x:String>
            </x:Array>
          </Picker.ItemsSource>
        </Picker>

        <Label Text="Total Home Consumption:"
            Grid.Row="1"
            Grid.Column="0"/>


        <Label Text="{Binding TotalHomeConsumption,
                StringFormat='{0} W h'}"
            Grid.Row="1"
            Grid.Column="1"/>

        <Label Text="Total Home Cost:"
            Grid.Row="2"
            Grid.Column="0"/>

        <Label Text="{Binding TotalHomeCost,
              StringFormat='${0}'}"
            Grid.Row="2"
            Grid.Column="1"/>
    </Grid>

    <Entry Placeholder="Enter Outlet Name"
        Grid.Row="1"
        Grid.ColumnSpan="2"
        Text="{Binding OutletName}"/>



    <CollectionView Grid.Row="2"
              Grid.ColumnSpan="2"
              ItemsSource="{Binding Outlets}"
              SelectionMode="None">
      <CollectionView.ItemTemplate>
        <DataTemplate x:DataType="model:OutletProperties">
          <SwipeView>
            <SwipeView.RightItems>
              <SwipeItems>
                <SwipeItem Text="Rename"
                      BackgroundColor="Blue"
                      Command="{Binding Source={RelativeSource AncestorType={x:Type viewmodel:MainViewModel}},
Path=RenameCommand}"
                      CommandParameter="{Binding .}"/>
              </SwipeItems>
            </SwipeView.RightItems>

            <Grid Padding="10">
              <Frame>
                <Frame.GestureRecognizers>
                  <TapGestureRecognizer
                    Command="{Binding Source={RelativeSource AncestorType={x:Type viewmodel:MainViewModel}},
Path=TapCommand}"
                    CommandParameter="{Binding .}"/>
                </Frame.GestureRecognizers>

                <Label Text="{Binding name}" FontSize="15" HorizontalTextAlignment="Center"/>
```

```
                    </Frame>
                </Grid>

            </SwipeView>
        </DataTemplate>
    </CollectionView.ItemTemplate>

</CollectionView>

<Button Text="Scan for Outlets"
        Command="{Binding GetOutletsCommand}"
        IsEnabled="{Binding IsNotBusy}"
        Grid.Row="3"
        Grid.ColumnSpan="2"
        >
</Button>

<ActivityIndicator IsVisible="{Binding IsBusy}"
                   IsRunning="{Binding IsBusy}"
                   HorizontalOptions="FillAndExpand"
                   VerticalOptions="CenterAndExpand"
                   Grid.RowSpan="2"
                   Grid.ColumnSpan="2"/>
</Grid>

</ContentPage>
```

DetailsPage.xaml

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PowerOutlet.View.DetailsPage"
             xmlns:viewmodel = "clr-namespace:PowerOutlet.ViewModel"
             xmlns:model = "clr-namespace:PowerOutlet.Model"
             x:DataType="viewmodel:DetailsViewModel"
             Title="{Binding Outlet.name}">

    <Grid RowDefinitions="Auto, Auto,Auto, Auto, Auto, Auto, *"
          ColumnDefinitions=".25*, .25*, .25*, .25*"
          Padding="10" RowSpacing="10" ColumnSpacing="10">

        <Button Grid.Row="0"
                Grid.Column="0"
                Text="T OFF" FontSize="20"
                Command="{Binding TurnTopOffCommand}"/>

        <Button Grid.Row="0"
                Grid.Column="1"
                Text="T ON" FontSize="20"
                Command="{Binding TurnTopOnCommand}"/>

        <Button Grid.Row="0"
                Grid.Column="2"
                Text="B OFF" FontSize="20"
                Command="{Binding TurnBottomOffCommand}"/>

        <Button Grid.Row="0"
                Grid.Column="3"
                Text="B ON" FontSize="20"
                Command="{Binding TurnBottomOnCommand}"/>

        <Button Grid.Row="1"
                Grid.Column = "0"
                Grid.ColumnSpan="2"
                Text="View Live Power" FontSize="20"
                Command="{Binding GetPDAsyncContinouslyCommand}"
                />
```

```
<Button Grid.Row="1"
        Grid.Column="2"
        Grid.ColumnSpan="2"
        Text="Stop Live Power" FontSize="20"
        Command="{Binding StopViewPowerCommand}"
/>

<Entry Placeholder="Enter Outlet Energy Limit"
       PlaceholderColor="DarkGray"
       Grid.Row="2"
       Grid.Column="0"
       Grid.ColumnSpan="2"
       Text="{Binding PowerLimit}"
       />

<Button Grid.Row="2"
        Grid.Column="3"
        Grid.ColumnSpan="1"
        Text="Set" FontSize="20"
        Command="{Binding SendPowerLimitCommand}"/>

<Picker Grid.Row="2"
        Grid.Column="2"
        Grid.ColumnSpan="1"
        Title="Duration"
        SelectedIndex="{Binding PowLimDur}">
        <Picker.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>Second (inst)</x:String>
            <x:String>Minute</x:String>
            <x:String>Hour</x:String>
            <x:String>Day</x:String>
            <x:String>Week</x:String>
            <x:String>Month</x:String>
            <x:String>Year</x:String>
        </x:Array>
        </Picker.ItemsSource>
</Picker>


<DatePicker Grid.Row="3"
            Grid.Column="1"
            MinimumDate="01/01/2023"
            MaximumDate="12/31/2123"
            Date="{Binding BeginDate}" />

<TimePicker Grid.Row="3"
            Grid.Column="2"
            Time="{Binding BeginTime}" />

<Label Text="Start Date and Time:"
       Grid.Row="3"
       Grid.Column="0"/>

<Picker Grid.Row="3"
        Grid.Column="3"
        Grid.ColumnSpan="1"
        Title="Receptacle"
        SelectedIndex="{Binding Receptacle}">
        <Picker.ItemsSource>
        <x:Array Type="{x:Type x:String}">
            <x:String>Top</x:String>
            <x:String>Bottom</x:String>
            <x:String>Both</x:String>
        </x:Array>
        </Picker.ItemsSource>
</Picker>
```

```
    <DatePicker Grid.Row="4"
                Grid.Column="1"
                Grid.ColumnSpan="1"
                MinimumDate="01/01/2023"
                MaximumDate="12/31/2123"
                Date="{Binding EndDate}" />

    <TimePicker Grid.Row="4"
                Grid.Column="2"
                Time="{Binding EndTime}" />

    <Label Text="End Date and Time:"
           Grid.Row="4"
           Grid.Column="0"/>

    <Button Text="Retrieve Data"
            Command="{Binding GetEnergyConsumptionCommand}"
            IsEnabled="{Binding IsNotBusy}"
            Grid.Row="4"
            Grid.Column="3"/>


    <Label Text="{Binding Energy,
                  StringFormat='Energy is: {0} W h'}"
           Grid.Row="5"
           Grid.Column="0"
           Grid.ColumnSpan="2"/>

    <Label Text="{Binding Cost,
           StringFormat='Cost is: {0} cents'}"
           Grid.Row="5"
           Grid.Column="2"
           Grid.ColumnSpan="2"/>

    <CollectionView Grid.Row="6"
                    Grid.ColumnSpan="4"
                    ItemsSource="{Binding IPDList}"
                    SelectionMode="None">

      <CollectionView.ItemTemplate>
        <DataTemplate x:DataType="model:InstantaneousPowerData">
            <HorizontalStackLayout Padding="10">
              <Label FontSize="10">
                <Label.Text>
                  <MultiBinding StringFormat="{}Time: {0}     Top Power: {1}     Bottom Power: {2}">
                    <Binding Path="Time"/>
                    <Binding Path="TopPower"/>
                    <Binding Path="BottomPower"/>
                  </MultiBinding>
                </Label.Text>
              </Label>

            </HorizontalStackLayout>
        </DataTemplate>
      </CollectionView.ItemTemplate>

    </CollectionView>


    </Grid>

</ContentPage>
```

BaseViewModel.cs

```
namespace PowerOutlet.ViewModel;
```

```
public partial class BaseViewModel : ObservableObject
{
   [ObservableProperty]
   [NotifyPropertyChangedFor(nameof(IsNotBusy))]
   bool isBusy;

   [ObservableProperty]
   string title;

   public bool IsNotBusy => !IsBusy;
}
```

MainViewModel.cs

```
using PowerOutlet.Services;
using PowerOutlet.View;

namespace PowerOutlet.ViewModel;
public partial class MainViewModel : BaseViewModel
{
        HubService hubService;

        public ObservableCollection<OutletProperties> Outlets { get; } = new ObservableCollection<OutletProperties>();

        public MainViewModel(HubService hubService)
        {
                Title = "Home";
                this.hubService = hubService;
        }


   [ObservableProperty]
   string outletName;

   [ObservableProperty]
   float costPerkWH;

        [ObservableProperty]
        float totalHomeConsumption;

        [ObservableProperty]
        float totalHomeCost;


        [ObservableProperty]
        int durationIndex;


   partial void OnDurationIndexChanged(int value)
   {
                IsBusy = true;
      if (value != -1)
      {
                        GetHomeUsageAsync(value);
                        IsBusy = false;
      }
   }

   partial void OnCostPerkWHChanged(float value)
   {
      IsBusy = true;
      if (value != -1)
      {
         GetHomeUsageAsync(DurationIndex);
         IsBusy = false;
```

```
    }
}

async Task GetHomeUsageAsync(int option)
        {
                var energy = await hubService.RetrieveAllData(option, Outlets);
                TotalHomeConsumption = energy;
                TotalHomeCost = TotalHomeConsumption / 1000 * CostPerkWH;
        }

[RelayCommand]
        async Task GetOutletsAsync()
        {
                if (IsBusy)
                        return;

                try
                {
                        IsBusy = true;
                        await GetHubIP();
                        var outlets = await hubService.GetOutlets();

                        bool isInCollection = false;
                        foreach (var outlet in outlets)
                        {
                                foreach (var o in Outlets)
                                {
                                        if (outlet.longAddress == o.longAddress)
                                                isInCollection = true;
                                }
                                if (!isInCollection)
                                {
                                        outlet.name = "Name";
                                        Outlets.Add(outlet);
                                        isInCollection = false;
                                }
    }

                }
                catch (Exception ex)
                {
                        Debug.WriteLine(ex);
                        await Shell.Current.DisplayAlert("Error!", $"Unable to get outlets:{ex.Message}", "OK");
                }
                finally
                {
                        IsBusy = false;
    }
        }

        async Task GetHubIP()
        {
                await hubService.GetIPAdd();
        }



        [RelayCommand]
        void Rename(OutletProperties outlet)
        {
                if (string.IsNullOrWhiteSpace(OutletName))
                        return;

    Outlets.Remove(outlet);
    outlet.name = OutletName;
                Console.WriteLine($"Outlet name is:{outlet.name} outlet longAdd is: {outlet.longAddress}");
                Outlets.Add(outlet);
```

```
        }

        [RelayCommand]
        async Task Tap(OutletProperties outlet)
        {
                await Shell.Current.GoToAsync(nameof(DetailsPage),
                        new Dictionary<string, object>
                        {
                                {nameof(OutletProperties), outlet },
                                {nameof(CostPerkWH), CostPerkWH}
                        });

        }
}
```

HubService.cs

```
using System.Net.Http.Json;
using Google.Apis.Auth.OAuth2;
using Google.Apis.Services;
using Google.Apis.Sheets.v4;
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Driver;
using MongoDB.Driver.Core.Configuration;
using Newtonsoft.Json;

using Data = Google.Apis.Sheets.v4.Data;
using JsonSerializer = System.Text.Json.JsonSerializer;

namespace PowerOutlet.Services;
public class HubService
{
   public HubService()
   {
      httpClient = new HttpClient();
   }

   HttpClient httpClient;


   List<OutletProperties> outletList = new List<OutletProperties>();

   private string ipBase = "http://172.20.10.5/";

   InstantaneousPowerData IPD = new InstantaneousPowerData();

   public async Task<List<OutletProperties>> GetOutlets()
   {
      if (outletList?.Count > 0)
         return outletList;

      var url = ipBase + "allOutlets";

      var response = await httpClient.GetAsync(url);

      Console.WriteLine(response.ToString());

      Console.WriteLine(response.Content.ReadAsStringAsync());

      if (response.IsSuccessStatusCode)
      {
         outletList = await response.Content.ReadFromJsonAsync<List<OutletProperties>>();
      }

      return outletList;
   }
```

```
    public async Task<InstantaneousPowerData> GetInstantPD(UInt64 Address)
    {

        var url = ipBase + $"both/power?Address={Address}";

        var response = await httpClient.GetAsync(url);

        var data = response.Content.ToString();

        string responseBody = await response.Content.ReadAsStringAsync();
        Console.WriteLine(response.ToString());

        Console.WriteLine(response.Content.ReadAsStringAsync());

        JsonSerializerOptions _serializerOptions;

        _serializerOptions = new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
            WriteIndented = true
        };

        //var str =
"{\"bottomPF\":0.8889999985694885,\"bottomPower\":5.52400016784668,\"time\":1681756486,\"topPF\":0.8889999985694885,\"topPower\":5.
52400016784668}";

        if (response.IsSuccessStatusCode)
        {
            IPD = JsonSerializer.Deserialize<InstantaneousPowerData>(responseBody, _serializerOptions);
            //IPD = await response.Content.ReadFromJsonAsync<InstantaneousPowerData>();
        }
        Console.WriteLine(IPD.TopPower + IPD.BottomPower + IPD.Time);
        return IPD;
    }


    public async Task TurnTopOn(UInt64 Address)
    {

        var url = ipBase + $"top/on?Address={Address}";

        var response = await httpClient.GetAsync(url);

        Console.WriteLine(response.ToString());

        Console.WriteLine(response.Content.ReadAsStringAsync());

        if (response.IsSuccessStatusCode)
        {

        }

        return;
    }


    public async Task TurnTopOff(UInt64 Address)
    {
        var url = ipBase + $"top/off?Address={Address}";
        var response = await httpClient.GetAsync(url);
        Console.WriteLine(response.ToString());
        Console.WriteLine(response.Content.ReadAsStringAsync());
        if (response.IsSuccessStatusCode)
        {

        }
        return;
    }
```

```
    public async Task TurnBottomOn(UInt64 Address)
    {
        var url = ipBase + $"bottom/on?Address={Address}";
        var response = await httpClient.GetAsync(url);
        Console.WriteLine(response.ToString());
        Console.WriteLine(response.Content.ReadAsStringAsync());
        if (response.IsSuccessStatusCode)
        {

        }
        return;
    }

    public async Task TurnBottomOff(UInt64 Address)
    {
        var url = ipBase + $"bottom/off?Address={Address}";
        var response = await httpClient.GetAsync(url);
        Console.WriteLine(response.ToString());
        Console.WriteLine(response.Content.ReadAsStringAsync());
        if (response.IsSuccessStatusCode)
        {

        }
        return;
    }

    public async Task SetPowLimit(UInt64 Address, string powerLimit, int duration)
    {
        var url = ipBase + $"powerLimit?Address={Address}&PLim={powerLimit}&Dur={duration}";
        var response = await httpClient.GetAsync(url);
        Console.WriteLine(response.ToString());
        Console.WriteLine(response.Content.ReadAsStringAsync());
        if (response.IsSuccessStatusCode)
        {
        }
        return;
    }


    public enum ReceptacleOption
    {
        Top,
        bottom,
        both
    }



    public async Task<float> RetrieveData(UInt64 outletAddress, DateTime begin, DateTime end, ReceptacleOption r)
    {
        DateTime beginUTC = TimeZoneInfo.ConvertTimeToUtc(begin);
        DateTime endUTC = TimeZoneInfo.ConvertTimeToUtc(end);

        MongoClientSettings settings = MongoClientSettings.FromConnectionString("mongodb://kyro1199:lKphLibI1ERv3WRE@ac-e5h2olj-
shard-00-00.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-01.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-
02.1lfqj59.mongodb.net:27017/?ssl=true&replicaSet=atlas-xee5s1-shard-0&authSource=admin&retryWrites=true&w=majority");

        var client = new MongoClient(settings);
        var collection = client.GetDatabase("PowerOutlet").GetCollection<MongoData>(outletAddress.ToString());


        TimeSpan beginEpoch = beginUTC - new DateTime(1970, 1, 1, 0, 0, 0);
        UInt64 secondsSinceEpochBegin = (UInt64)beginEpoch.TotalSeconds;

        TimeSpan endEpoch = endUTC - new DateTime(1970, 1, 1, 0, 0, 0);
        UInt64 secondsSinceEpochEnd = (UInt64)endEpoch.TotalSeconds;
```

```
    var energyStatsWithinTimeFrame = await collection.Find(md => md.EpochTime >= secondsSinceEpochBegin & md.EpochTime <=
secondsSinceEpochEnd).ToListAsync();

    float energySum = 0;

    switch (r)
    {
       //Divide by 60 to convert from watt minutes to watt hours
       case ReceptacleOption.Top:
          foreach (MongoData md in energyStatsWithinTimeFrame)
             energySum += md.TopP / 60;
          break;
       case ReceptacleOption.bottom:
          foreach (MongoData md in energyStatsWithinTimeFrame)
             energySum += md.BottomP / 60;
          break;
       case ReceptacleOption.both:
          foreach (MongoData md in energyStatsWithinTimeFrame)
          {
             energySum += md.TopP / 60;
             energySum += md.BottomP / 60;
          }
          break;
    }
    return energySum;
 }


 public async Task<float> RetrieveAllData(int durationOption, ObservableCollection<OutletProperties> allOutlets)
 {
    DateTime end = DateTime.Now;
    DateTime begin;
    switch (durationOption)
    {
       case 0:
          TimeSpan ts = TimeSpan.FromHours(1);
          begin = end - ts;
          break;
       case 1:
          TimeSpan ts1 = TimeSpan.FromDays(1);
          begin = end - ts1;
          break;
       case 2:
          TimeSpan ts2 = TimeSpan.FromDays(7);
          begin = end - ts2;
          break;
       case 3:
          TimeSpan ts3 = TimeSpan.FromDays(30);
          begin = end - ts3;
          break;
       case 4:
          TimeSpan ts4 = TimeSpan.FromDays(365);
          begin = end - ts4;
          break;
       default:
          begin = DateTime.Now;
          break;
    }

    DateTime endUTC = TimeZoneInfo.ConvertTimeToUtc(end);
    DateTime beginUTC = TimeZoneInfo.ConvertTimeToUtc(begin);

    MongoClientSettings settings = MongoClientSettings.FromConnectionString("mongodb://kyro1199:lKphLibI1ERv3WRE@ac-e5h2olj-
shard-00-00.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-01.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-
02.1lfqj59.mongodb.net:27017/?ssl=true&replicaSet=atlas-xee5s1-shard-0&authSource=admin&retryWrites=true&w=majority");

    var client = new MongoClient(settings);
    var database = client.GetDatabase("PowerOutlet");
```

```
        TimeSpan beginEpoch = beginUTC - new DateTime(1970, 1, 1, 0, 0, 0);
        UInt64 secondsSinceEpochBegin = (UInt64)beginEpoch.TotalSeconds;

        TimeSpan endEpoch = endUTC - new DateTime(1970, 1, 1, 0, 0, 0);
        UInt64 secondsSinceEpochEnd = (UInt64)endEpoch.TotalSeconds;

        float energySum = 0;

        foreach (var outlet in allOutlets)
        {
            var collection = database.GetCollection<MongoData>(outlet.longAddress.ToString());
            var energyStatsWithinTimeFrame = await collection.Find(md => md.EpochTime >= secondsSinceEpochBegin & md.EpochTime <=
secondsSinceEpochEnd).ToListAsync();
            foreach (MongoData md in energyStatsWithinTimeFrame)
            {
                energySum += md.TopP / 60;
                energySum += md.BottomP / 60;
            }
        }
        return energySum;
    }


    [BsonIgnoreExtraElements]
    public class HubProp
    {

        public HubProp()
        {
        }

        [BsonId]
        public ObjectId _id { get; set; }

        [BsonElement("IP")]
        public string IP { get; set; }
    }

    public async Task GetIPAdd()
    {

        MongoClientSettings settings = MongoClientSettings.FromConnectionString("mongodb://kyro1199:lKphLibI1ERv3WRE@ac-e5h2olj-
shard-00-00.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-01.1lfqj59.mongodb.net:27017,ac-e5h2olj-shard-00-
02.1lfqj59.mongodb.net:27017/?ssl=true&replicaSet=atlas-xee5s1-shard-0&authSource=admin&retryWrites=true&w=majority");

        var client = new MongoClient(settings);
        var database = client.GetDatabase("PowerOutlet");

        var collection = database.GetCollection<HubProp>("IP");

        var result = await collection.Find(_ => true).SingleAsync();


        ipBase = result.IP;

    }
}
```

App.xaml

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<Application xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:PowerOutlet"
             x:Class="PowerOutlet.App">
```

```
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Resources/Styles/Colors.xaml" />
                <ResourceDictionary Source="Resources/Styles/Styles.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

AppShell.xaml

```
<?xml version="1.0" encoding="UTF-8" ?>
<Shell
    x:Class="PowerOutlet.AppShell"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:local="clr-namespace:PowerOutlet"
    xmlns:views = "clr-namespace:PowerOutlet.View"
    Shell.FlyoutBehavior="Disabled">

    <ShellContent
        Title="Home"
        ContentTemplate="{DataTemplate local:MainPage}"
        Route="MainPage" />


</Shell>
```

GlobalUsing.cs

```
global using CommunityToolkit.Mvvm.ComponentModel;
global using CommunityToolkit.Mvvm.Input;
global using PowerOutlet.Model;
global using PowerOutlet.ViewModel;
global using System.Collections.ObjectModel;
global using System.ComponentModel;
global using System.Diagnostics;
global using System.Runtime.CompilerServices;
global using System.Text.Json;
```

MauiProgram.cs

```
using Microsoft.Extensions.Logging;
using PowerOutlet.Services;
using PowerOutlet.View;
using PowerOutlet.ViewModel;

namespace PowerOutlet;

public static class MauiProgram
{
	public static MauiApp CreateMauiApp()
	{
		var builder = MauiApp.CreateBuilder();
		builder
			.UseMauiApp<App>()
			.ConfigureFonts(fonts =>
			{
				fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
				fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
			});

#if DEBUG
		builder.Logging.AddDebug();
```

```
#endif

                    builder.Services.AddSingleton<HubService>();

    builder.Services.AddSingleton<MainViewModel>();
    builder.Services.AddSingleton<MainPage>();

    builder.Services.AddTransient<DetailsViewModel>();
    builder.Services.AddTransient<DetailsPage>();



    return builder.Build();
        }
}
```

5.2.2.e Xbee Module (JG)

When initially configuring Xbee modules, Digi-Key's Xbee Configuration and Test Utility (XCTU)[3] program is used. XCTU is a piece software utility that allows a user to configure Xbee modules that are connected to their computer, or remote Xbee modules on the same Zigbee network as the connected Zigbee. XCTU allows users to communicate with the local and remote Xbee modules through a GUI and a terminal that shows sent and received data. To use this utility, it is necessary to have a USB breakout board made for Zigbee modules, such as those by Adafruit, Parallax, SparkFun, or a custom PCB that enables a serial USB connection between a computer and the Xbee module.

When an Xbee module is connected to the computer is discovered by XCTU, it reads all the configuration data stored on the Xbee. For example, the following image shows some of the properties of a connected Xbee device:

[3] https://www.digi.com/products/embedded-systems/digi-xbee/digi-xbee-tools/xctu

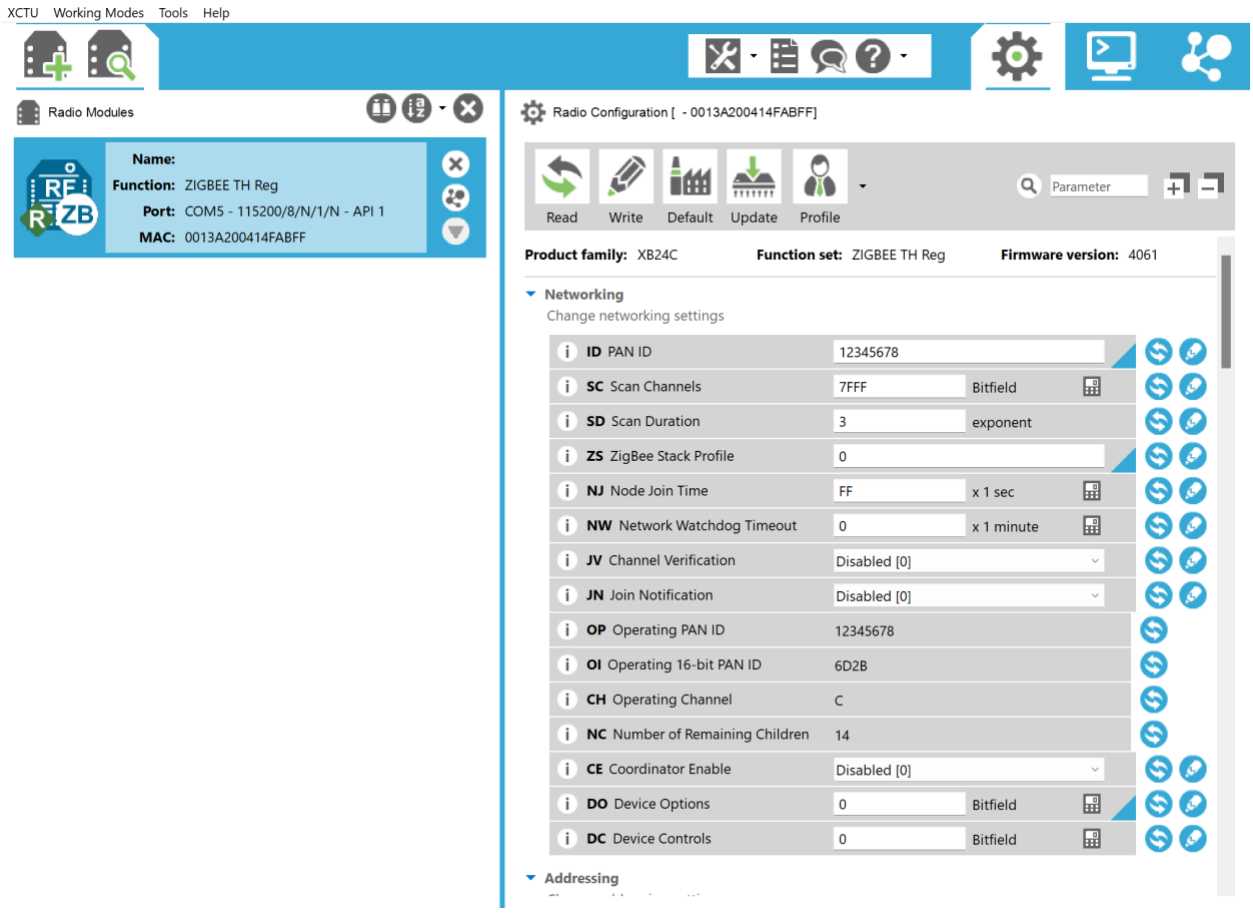

Figure 45: XCTU Menu with an XBee Module Connected to the Computer

In the figure above, some of the visible settings are the Xbee module's PAN ID (ID) and the type of Zigbee device that the Xbee module will operate as (CE)- a coordinator, router, or end device. However, there are a multitude of other options that can be configured such as the baud rate (BD) that the Xbee module will be set at to send and receive data using a serial communication method, the operation of certain pins, and the use of encryption.

As described in section 2.6.1, each Zigbee network requires 1 coordinator that is responsible for managing the Zigbee network. The remaining devices composing the Zigbee network can be routers or end devices, with the difference between the two being the router's

ability to rebroadcast and forward messages that it receives. As a mesh network is the ideal network type to use for the Smart Power Outlet system, the use of routers is desirable as it will allow the routers to retransmit messages so that they can reach their destination. Hence, the Hub will function as the Zigbee network's coordinator and the Smart Power Outlets will function as routers, which will allow them to forward data to the Hub from other Smart Power Outlets that are unable to directly communicate with it.

XCTU can be used to configure an initial Zigbee network. To do this, assign two or more Xbee modules to a common PAN ID and set the first Xbee module as a coordinator (CE = 1) using XCTU. The second Xbee module must be a router or an end device, which is achieved by setting the sleep mode (SM) option to 0 or a value from 1 to 5, respectively. The last step is to set the 64-bit destination addresses of both devices, which is done through the 32-bit high (DH) and 32-bit low (DL) registers. In this case, we want the router to be able to communicate with the coordinator and vice versa. Therefore, the router's DH and DL should be set to '0' to signify that it will communicate with the coordinator. When configuring the DH and DL of the coordinator, a different approach can be used. First, DH and DL can be set to the upper 32-bits and lower 32-bits of the ZigBee router, but this will result in the coordinator only being able to talk to the router. Instead, the 64-bit broadcast address of '0x000000000000FFFF" can be used, which allows the coordinator to broadcast messages that will be accepted by every device in the Zigbee Network. Once these settings are applied and both Xbee modules are powered, XCTU can be used to send data between the XBee modules. The following image is an example of a frame containing the ASCII character phrase "Sent from router to coordinator" (53 65 6e 74 20 66 72 6f 6d 20 72 6f 75 74 65 72 20 74 6f 20 63 6f 6f 72 64 69 6e 61 74 6f 72 as a hexadecimal string) sent between two Xbee modules that were set up using the process described.

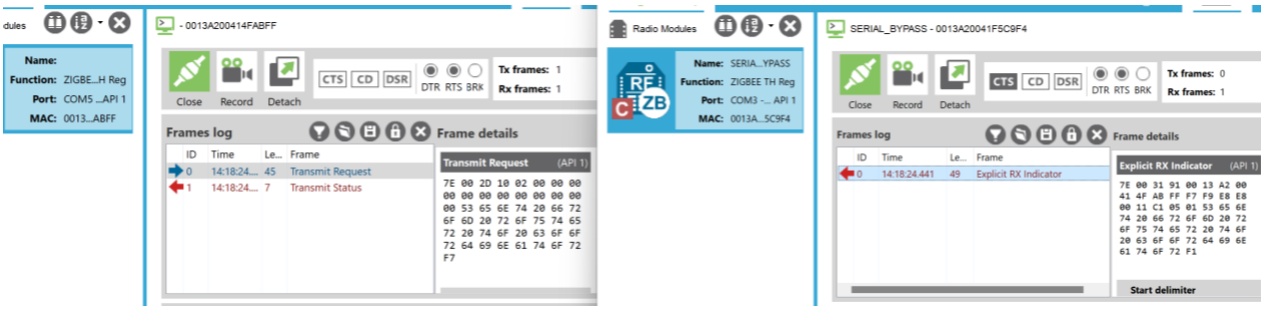

Figure 46: Sample Frame Sent Between Two XBee Modules Connected to XCTU

In the above figure, the indicated phrase was set as the payload of an XBee transmit request frame. Using two instances of XCTU with a different XBee module connected to each, the XCTU's Frame Generator tool was used to create a frame that was sent from the router to the coordinator. From the figure, the data was received in the same second that it was sent, but what was received is slightly different than the source data. The reason for this and why the data was sent in a frame will be discussed shortly. However, the important thing is that the hexadecimal string corresponding to the indicated phrase is present in both frames.

Regardless of the type of Zigbee device, Xbee modules can be programmed to operate in one of 3 ways by changing the value of the API enable (AP) register: transparent mode (AP = 0), API mode (AP = 1), and API mode with escapes (AP = 2). Transparent mode allows the Xbee module to immediately broadcast any data that it receives over a serial connection over the Zigbee network that it is connected to. API Mode requires serial input data to be formatted into to one of many frames that it recognizes to be transmitted. API mode with escapes is near identical but inserts escape characters after some specific characters. A comparison of transparent mode and API mode can be found in the following figure provided by Digi-Key:

Transparent operating mode	API operating mode
When to use: ▪ Conditions for using API mode do not apply.	**When to use:** ▪ Sends wireless data to multiple destinations. ▪ Configures remote XBee devices in the network. ▪ Receives wireless data packets from multiple XBee devices, and the application needs to identify which devices send each packet. ▪ Receives I/O samples from remote XBee devices. ▪ Must support multiple endpoints, clusters, and/or profiles (for Zigbee modules). ▪ Uses Zigbee Device Object (ZDO) services (for Zigbee modules).
Advantages: ▪ Provides a simple interface that makes it easy to get started with XBee devices. ▪ Easy for an application to support; what you send is exactly what other modules get, and vice versa. ▪ Works very well for two-way communication between XBee devices.	**Advantages:** ▪ Can set or read the configuration of remote XBee devices in the network. ▪ Can transmit data to one or multiple destinations; this is much faster than transparent mode where the configuration must be updated to establish a new destination. ▪ Received data includes the sender's address. ▪ Received data includes transmission details and reasons for success or failure. ▪ Several advanced features, such as advanced networking diagnostics, and firmware upgrades.
Disadvantages: ▪ Cannot set or read the configuration of remote XBee devices in the network. ▪ Must first update the configuration to establish a new destination and transmit data. ▪ Cannot identify the source of received data, as it does not include the sender's address. ▪ Received data does not include transmission details or the reasons for success or failure. ▪ Does not offer the advanced features of API mode, including advanced networking diagnostics, and firmware upgrades.	**Disadvantages:** ▪ Interface is more complex; data is structured in packets with a specific format. ▪ More difficult to support; transmissions are structured in packets that need to be parsed (to get data) or created (to transmit data). ▪ Sent data and received data are not identical; received packets include some control data and extra information.

Figure 47 Comparison of XBee Transparent Mode and API Mode from [44]

As shown in the above figure, transparent mode is very to work with as data is received by other XBee modules exactly as it is received. However, transparent mode requires the address of the receiving XBee module to be known, does not indicate the source of data sent to the receiver, and does not allow the configuration of remote XBee modules to be viewed or set. Conversely, API mode is more difficult to implement as it requires data to be formatted into a recognized frame but allows for remote management of XBee modules and provides XBee modules with information about the source of data that they receive. Hence, API mode needs to be used as it will allow the Hub to configure new Smart Power Outlets that a user may purchase and provide useful information about the source of data received by the Hub or Smart Power Outlets.

The XBee API mode supports 21 different frames- 7 used to transmit data and 14 used to receive data. These frames are documented extensively in the "*XBee/XBee-PRO Zigbee RF Module User Guide"* by Digi-Key that is referenced in the appendix. However, in the context of the Smart Power Outlet system, 12 of these frames will not be used. The most important frames are those used to send and receive data, configure local and remote XBee modules, and to check

the status of XBee modules and their transmissions. A table provided by Digi-Key has been modified to list the 9 crucial frames that must be supported and their purpose.

Table 11: Required XBee Frames and Their Purposes (modification of the original table by [45])

Direction	API ID	Frame name	Description
Transmit	**0x08**	AT Command	Queries or sets parameters on the local XBee
	0x10	Transmit Request	Transmits wireless data to the specified destination
	0x17	Remote AT Command Request	Queries or sets parameters on the specified remote XBee module
Receive	**0x88**	AT Command Response	Displays the response to previous AT command frame
	0x8A	Modem Status	Displays event notifications such as reset, association, disassociation, and so on.
	0x8B	Transmit Status	Indicates wireless data transmission success or failure
	0x90	Receive Packet	Sends wirelessly received data out the serial interface (AO = 0)
	0x91	Explicit Rx Indicator	Sends wirelessly received data out the serial interface when explicit mode is enabled (AO 0)
	0x97	Remote AT Command Response	Displays the response to previous remote AT command requests

To analyze the content of a frame, the transmit request (TX) frame used to send user data between XBee modules can be explored. The format of a TX frame is shown below:

Offset	Size	Frame Field	Description
0	8-bit	Start Delimiter	Indicates the start of an API frame.
1	16-bit	Length	Number of bytes between the length and checksum.
3	8-bit	**Frame type**	Transmit Request - **0x10**
4	8-bit	**Frame ID**	Identifies the data frame for the host to correlate with a subsequent response frame. If set to **0**, the device will not emit a response frame.
5	64-bit	**64-bit destination address**	Set to the 64-bit IEEE address of the destination device. Broadcast address is **0x000000000000FFFF**. Zigbee coordinator address is **0x0000000000000000**. When using 16-bit addressing, set this field to **0xFFFFFFFFFFFFFFFF**.
13	16-bit	**16-bit destination address**	Set to the 16-bit network address of the destination device, if known. If transmitting to a 64-bit address, sending a broadcast, or the 16-bit address is unknown, set this field to **0xFFFE**.
15	8-bit	**Broadcast radius**	Sets the maximum number of hops a broadcast transmission can traverse. This parameter is only used for broadcast transmissions. If set to**0**—recommended—the value of **NH**specifies the broadcast radius.
16	8-bit	**Transmit options**	See the Transmit options bit field table below for available options. If set to **0**, the value of **TO** specifies the transmit options.
17-n	variable	**Payload data**	Data to be sent to the destination device. Up to **NP** bytes per packet.
EOF	8-bit	Checksum	0xFF minus the 8-bit sum of bytes from offset 3 to this byte (between length and checksum).

Figure 48: Format of an XBee Transmit Request (TX) Frame by [46]

As shown above, the TX frame is composed of 10 fields whose sizes are expressed in bits. The first field is a byte representing the start delimiter ‘0x7E’, which is present at the beginning of every API frame. The next field is the length of the packet minus 4 bytes, as the overhead of the length, start delimiter, and checksum are overhead and not relevant data to the frame. Next is the

frame ID, which is used to correlate transmissions to responses. Following the frame ID field are 64-bit and 16-bit destination address that decide where the frame will be sent. Various constant values can be used to broadcast frames to all devices on the Zigbee network or to send them to the network's coordinator. Additionally, a known address of a specific XBee module can be used here. Following this are 2 bytes of options related to the transmission. Then, up to 255 bytes of data (or 251 bytes if encryption is enabled) can be sent as the payload of the frame. Finally, a checksum consisting of the sum of the bytes between the length through the end of the payload truncated to the least significant byte and subtracted from the value '0xFF' is added.

In response to a successfully received TX frame, the receiving XBee module will automatically output a receive packet frame or an explicit receive packet indicator frame over its serial output. The output frame will include information about the source of the transmission and the data contained in the incoming packet, with the explicit receive indicator frame containing more information than the standard receive packet frame. Therefore, by parsing the frame output by the XBee module when it receives a frame, it is to extract the information contained within it.

All frames have a different format, with different sizes and required fields relevant to the frame's purpose. However, all frames contain a start delimiter, a length, a frame ID, a checksum, and a similar format. Additionally, each type of sent frame that can be sent has associated frames that are automatically output in response by the receiving XBee module. Due to the highly structured frames, it is easy to extract relevant information if they are valid frames and their format is known.

When operating in API mode, XBee modules will reject any received data if it does not follow the format of an accepted frame. Hence, the data that the ESP32 microcontrollers will send using the XBee modules needs to be placed inside of a proper XBee frame prior to transmission.

Likewise, all commands that will be present in the Control Application will need to be mapped to XBee frames. As the ESP32 microcontrollers do not automatically translate their data into valid XBee frames, an interface needs to be created to allow the ESP32 to use the XBee module.

5.2.2.f ESP32 and XBee Interface (JG)

Digi-Key's XCTU application functions similarly to the desired functionality of the ESP32's XBee interface. When connected to an XBee module via USB, XCTU functions as a GUI based frame creator and transmission tool. Any operation performed using XCTU, such as changing a configuration register value, is converted to the appropriate frame, and sent over USB to the XBee module. Here, the USB breakout board that the XBee module is attached to converts the incoming frame as a serial input. If the data is a local command it is executed on the XBee module, and if it is a remote command or a transmission request, it is sent over the Zigbee network to its destination.

Here, the ESP32'x XBee interface needs to function in a similar manner. However, instead of a GUI based computer application that interacts with the XBee module over USB, it is a low-level program that accepts data and other arguments, creates bytes to represent the appropriate frame based on its input, and sends the frame to the XBee module using a serial connection. As stated previously, the XBee modules are connected to the ESP32 using a serial UART connection operating at a baud rate of 115200 bits per second, and the UART connection is configurable using the ESP-IDF framework.

C++ was used to create the ESP32's XBee interface. Additionally, a C++ JSON class titled 'json'[4] by Niels Lohmann [47] and licensed under the MIT License was imported to aid in the

[4] https://json.nlohmann.me/

processing of incoming frames. Use of Lohmann's JSON class allows data contained in received frames to be nicely serialized and assigned to a corresponding identifier. This will make using the data extracted from a frame simple, as it can be accessed using the field associated with it in the JSON object that it was extracted to. Additionally, logging the data received is also made easier as individual JSON objects can be output to a log file without requiring additional manipulation of the received frame. However, Lohmann's json class is quite large, so it may need to be replaced with a smaller class better suited for an embedded system in the future.

The code written to implement the XBee interface makes extensive use of the uint8_t data type, which is an unsigned 8-bit value. The uint8_t data type allows byte level creation of the frames that will be sent to the XBee. Additionally, the values of each byte can be interpreted as an integer, be easily manipulated using the bitwise operations built into C++, and easily cast to other data types like chars when necessary. Most importantly though, as the XBee module expects bytes, which are represented on computers with an integer value that is 8 bits wide, an array of the uint8_t data type can be sent to the XBee and interpreted properly without any modification.

The ESP32's XBEE interface is designed to support the 9 crucial XBee frames shown in Table 11. Additional frames can be supported by writing a new function to create them and adding a new case that supports their responses to the function that reads received XBee frames. Specific functions must be called to create each transmitted frame, while a single function is used to parse incoming frames. The ESP32's XBee interface is implemented in two files: "xbee_api.hpp" and "xbee_api.cpp". The former of these files contains documentation and function declarations while the latter contains all function definitions. To use the XBee interface, both files must be placed in the "src" directory of the ESP32's project folder and the file "xbee_api.hpp" must be included in the "ain.cpp" file that will be ran on the ESP32.

The following table lists the main functions in the XBee interface and their purpose. The complete code that makes up the interface can be found beneath it.

Table 12: Main Functions in the ESP32's XBee Interface

Function	Responsibility
uint8_t* formATFrame(std::string ATCommand, std::string newvalue = "")	Create an AT frame used to query or set local XBee parameters
uint8_t* formTXFrame(std::string RFData, uint64_t dst_64 = 0x0000000000000000, uint16_t dst_16 = 0x0000, uint8_t bcr = 0x00, uint8_t opt = 0x00);	Create a TX frame used to transmit user data to another Xbee
uint8_t* formATFrame_Remote(std::string ATCommand, uint64_t dst_64, std::string newvalue = "", uint8_t opt = 0x00, uint16_t dst_16 = 0xFFFE);	Create an AT frame used to query or set remote XBee parameters
json* readFrame(uint8_t* frame);	Parse an incoming frame and extract data to a JSON object

xbee_api.hpp:

```
#include <stdint.h>
#include <string.h>
#include <json.hpp>
/* json class license information information
The class is licensed under the MIT License:

Copyright © 2013-2022 Niels Lohmann

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the
©Software©), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the
Software, and to permit
persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ©AS IS©, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT
LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH
THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
```

```
using json = nlohmann::json;

//////////////////////////////////////////////////////
//          TRANSMISSION/FRAME CONSTANTS            //
//////////////////////////////////////////////////////
//static const uint8_t START_DELIMITER = 0x7E;                                          // delimiter at the start of every xbee frame
//static const uint8_t COORDINATOR_ADDRESS_64[] = "0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00";      // coordinator address
//static const uint8_t BROADCAST_ADDRESS_64[] = "0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF";        // PAN broadcast address
//static const uint8_t UNKNOWN_ADDRESS_64[] = "0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF";         // unknown address
//static const uint8_t COORDINATOR_ADDRESS_16[] = "0x00, 0x00";                          // coordinator address
//static const uint8_t BROADCAST_ADDRESS_16[] = "0xFF, 0xFE";                            // PAN broadcast address

//static const std::string AT_COMMAND_LIST = {};

//////////////////////////////////////////////////////
//                  AT COMMANDS                     //
//////////////////////////////////////////////////////



//////////////////////////////////////////////////////
//                  FUNCTIONS                       //
//////////////////////////////////////////////////////

// increment global frame counter
void frameIDIncrement();

// determine if option is hex or ascii
// return size and then option following it in the same array
uint8_t* parseOptions(std::string options);

// calculate xbee frame checksum per xbee documentation
uint8_t calculateCHKSM(uint8_t* framePtr, int frameLength);

//////////////////////////////////////////////////////
//              TRANSMIT DATA FRAMES                //
//////////////////////////////////////////////////////

/* LOCAL AT COMMAND REQUEST
* query or set AT parameter on LOCAL xbee module
*
*     /-----------------------------------------------------------------------------------\
*     |  0 |    1     |     2    |   3 |    4    |   5   |   6   |   7 - N   |  EOF  |
*     |----|----------|----------|-----|---------|-------|-------|-----------|-------|
*     | 0x7E | length MSB | length LSB |  0x08  | Frame ID | CMD MSB | CMD LSB | value (OPT) | CHKSUM |
*     \-----------------------------------------------------------------------------------/
*
* accepts an AT command and an optional new value. If a new value is supplied, it replaces the corresponding
* parameter on the xbee module. If no new value is supplied, the parameter is queried from the xbee module
*/
std::vector<uint8_t>* formATFrame(std::string ATCommand, std::string newvalue = "");

/* TX REQUEST
* transmit data from one xbee module to xbee module whose 64bit and 16bit address is provided
* by default, send message to coordinator
*
*     /-----------------------------------------------------------/ /-----------------------------------------------------------/ /-----------------\
*     |  0 |    1     |     2    |   3 |    4    |  5  \\  12   |    13     |     14    | 15  | 16  |   17  \\   N   | EOF  |
*     |----|----------|----------|-----|---------|-----/ /---------|-----------|-----------|-----|-----|-------/ /--------|------|
*     | 0x7E | length MSB | length LSB |  0x10  | Frame ID | 8BYTE \\ DST ADDR | 2B ADDR MSB | 2B ADDR LSB |  BCR  |  OPT  | RF
DATA \\ RF DATA | CHKSUM |
*     \-----------------------------------------------------------/ /-----------------------------------------------------------/ /-----------------/
*
* constant addresses:
* 64 bit coordinator address          = 0x0000000000000000
* 64 bit broadcast address            = 0x000000000000FFFF
* 16 bit broadcast/unknown address    = 0xFFFE
* 16 bit coordinator address          = 0x0000
```

```
* BCR = maximum # of transmission hops -> default = 0x00 = maximum
* OPT = options -> Default = 0x00
*      0x01 = disable retries + route repair
*      0x02 = enable APS encryption (decreases RF payload by 4 bytes)
*/
std::vector<uint8_t>  formTXFrame(std::string RFData, uint64_t dst_64 = 0x0000000000000000, uint16_t dst_16 = 0x0000, uint8_t bcr = 0x00,
uint8_t opt = 0x00);
std::vector<uint8_t>  formTXFrame(std::vector<uint8_t> *RFData, uint64_t dst_64, uint16_t dst_16, uint8_t bcr, uint8_t opt);

/* REMOTE AT COMMAND SET/REQUEST
* query or set AT parameter on remote xbee module; default is to read the passed parameter
* to read a parameter: supply the AT COMMAND, ADDRESS
* to set a parameter:  supply the AT COMMAND, ADDRESS, VALUE, and set OPT 0x02
*
*      /-----------------------------------------------------/ /-----------------------------------------------------------------------------/ /------------------\
*      |  0 |    1    |    2    |   3 |    4   |  5  \\   12   |    13    |     14    | 15  |   16  |   17  |   18  \\   N   | EOF  |
*      |-----|-----------|-----------|--------|---------|------/ /---------|-------------|-------------|-------|--------|--------|--------/ /---------|--------|
*      | 0x7E | length MSB | length LSB |  0x17  | Frame ID | 8BYTE \ \ DST ADDR | 2B ADDR MSB | 2B ADDR LSB |  OPT  | CMD MSB |
CMD LSB | VAL MSB \ \ VAL LSB | CHKSUM |
*      \-----------------------------------------------------/ /-----------------------------------------------------------------------------/ /------------------/
*
* OPT = options -> Default = 0x00
*      0x01 = disable ACK
*      0x02 = apply change (0 = will not set)
*      0x10 = send securely
*/
std::vector<uint8_t>*  formATFrame_Remote(std::string ATCommand, uint64_t dst_64, std::string newvalue = "", uint8_t opt = 0x00, uint16_t
dst_16 = 0xFFFE);

std::vector<uint8_t> formSourceRoute(uint64_t dst_64, uint16_t dst_16, uint8_t num_addresses, json addresses);

//////////////////////////////////////////////////////
//              RECEIVE DATA FRAMES              //
//////////////////////////////////////////////////////

// interpret received AT frame as json
json parseATCR(uint8_t* frame, uint16_t frameLength);

// interpret received MS frame as json
json parseMS(uint8_t* frame, uint16_t frameLength);

// interpret received TS frame as json
json parseTS(uint8_t* frame, uint16_t frameLength);

// interpret received packet as json
json receivePacket(uint8_t* frame, uint16_t frameLength);

// interpret EXPLICIT RX frame as json
json explicitRX(uint8_t* frame, uint16_t frameLength);

// interpret remote AT frame as json
json remoteAT(uint8_t* frame, uint16_t frameLength);

// create routes
json handleRoutes(uint8_t* frame, uint16_t frameLength);

// interpret frame and return data in JSON form
json readFrame(uint8_t* frame);
```

xbee_api.cpp:

```
#include "xbee_api.hpp"
#include <iostream>
#include <vector>

uint8_t GLOBAL_FRAME_ID = 0x01;

void frameIDIncrement(){
   // avoid ID == 0 to always have a response
   if(GLOBAL_FRAME_ID == 0xFF)
      GLOBAL_FRAME_ID = 0x01;
   else
      GLOBAL_FRAME_ID += 1;
}

uint8_t* parseOptions(std::string options){
   // nullptr if no options
   if(options.empty())
      return nullptr;
   // ptr to parsed value
   uint8_t* parsedOpt = nullptr;
   uint16_t lengthOpt = options.length();
   // get first 2 chars of option
   std::string isHex = options.substr(0, 2);
   // hex if isHex == '0x' || isHex == '0X'
   if((isHex.compare("0x") == 0 || isHex.compare("0X") == 0) && lengthOpt > 2){
      // add leading 0 if odd number of chars
      std::string optVal = "";
      if(lengthOpt % 2 == 1){
         optVal = "0" + options.substr(2, lengthOpt);
      }
      else{
         optVal = options.substr(2, lengthOpt);
         // length is 1 shorter
         lengthOpt -=1;
      }
      //preserve chars in groups of 2
      parsedOpt = new uint8_t[(lengthOpt/2)]();
      // store length in 1st pos
      parsedOpt[0] = lengthOpt/2;
      std::string tmp = "";

      for(int i = 0; i < (lengthOpt/2); i++){
         tmp = optVal[2*i];
         tmp += optVal[(2*i)+1];
         parsedOpt[i+1] = std::stoi(tmp, nullptr, 16);
      }
   }
   // if not hex, convert 1:1
   else{
      parsedOpt = new uint8_t[lengthOpt+1]();
      //store length in first pos
      parsedOpt[0] = lengthOpt;
      for(int i = 0; i < lengthOpt; ++i)
         parsedOpt[i+1] = uint8_t(options[i]);
   }
   return parsedOpt;
}

// calculate xbee frame checksum per xbee documentation
uint8_t calculateCHKSM(uint8_t* framePtr, int frameLength){
   uint8_t sum = 0x00;
   for(int i = 3; i < frameLength -1; ++i){
      sum += framePtr[i];
   }
   return 0xFF - sum;
```

```
}

std::vector<uint8_t>* formATFrame(std::string ATCommand, std::string newvalue){
   // TO DO- Check that ATCommand is valid
   //if()
   // frame has 8 bytes without optional value
   // optional value can be a string or a hex value, denoted with "0x"
   uint16_t frameSize = 8;
   uint8_t *newValCpy = parseOptions(newvalue);
   uint16_t valLen = 0;
   if(newValCpy != nullptr){
      valLen = newValCpy[0];
      // update frame size
      frameSize += valLen;
      }
   // isolate CMD MSB and LSB
   std::string cmdBytes = ATCommand.substr(0, 2);
   // start forming frame
   std::vector<uint8_t>* framePtr = new std::vector<uint8_t>;
   framePtr->push_back(0x7E);
   // length field wants bytes from its end through checksum, so subtract overhead from frame
   int totalLength = frameSize - 4;
   // get MSB of length
   framePtr->push_back((totalLength && 0xFF00) >> 8);
   // get LXB of length
   framePtr->push_back(totalLength);
   // assign AT Command identifier
   framePtr->push_back(0x08);
   // assign + increment running frame ID
   framePtr->push_back(GLOBAL_FRAME_ID);
   frameIDIncrement();
   // assign command MSB and LSB
   framePtr->push_back(uint8_t(cmdBytes[0]));
   framePtr->push_back(uint8_t(cmdBytes[1]));
   // assign opt value
   if(int(valLen) > 0){
      for(int i = 0; i < valLen; ++i){
         framePtr->push_back(newValCpy[i+1]);
      }
   }
   // assign CHKSUM
   framePtr->push_back(calculateCHKSM(framePtr->data(), frameSize));
   return framePtr;
}

std::vector<uint8_t>  formTXFrame(std::string RFData, uint64_t dst_64, uint16_t dst_16, uint8_t bcr, uint8_t opt){
   // 18 bytes of overhead in TX frame
   uint16_t frameSize = 18;
   uint16_t RFMax = 0x100;
   // check for encryption enabled
   if((opt & 2) == 0x02)
      RFMax -=4;
   // get length of RFData
   uint16_t RFSize = RFData.length();
   // only capture up to 256 bytes // throw error if too big
   if(RFSize > RFMax)
      return {0};
   // total frame size
   frameSize += RFSize;

   // begin forming frame
   std::vector<uint8_t> framePtr;// = new std::vector<uint8_t>;
   framePtr.push_back(0x7E);
   // assign sizes
   framePtr.push_back((frameSize-4) >> 8);
   framePtr.push_back(frameSize-4);
   // assign frame type and ID
   framePtr.push_back(0x10);
```

```
    // assign + increment running frame ID
    framePtr.push_back(GLOBAL_FRAME_ID);
    frameIDIncrement();
    // assign 8 byte address
    for(int i = 0; i < 8; ++i){
        framePtr.push_back(dst_64 >> (56-(8*i)));
    }
    //assign 2 byte address
    framePtr.push_back(dst_16 >> 8);
    framePtr.push_back(dst_16);
    // assign BCR and OPT
    framePtr.push_back(bcr);
    framePtr.push_back(opt);
    // assign RF data
    for(int i = 0; i < RFSize; ++i){
        framePtr.push_back(uint8_t(RFData[i]));
    }
    // assign checksum
    framePtr.push_back(calculateCHKSM(framePtr.data(), frameSize));
    return framePtr;
}

std::vector<uint8_t>  formTXFrame(std::vector<uint8_t> *RFData, uint64_t dst_64, uint16_t dst_16, uint8_t bcr, uint8_t opt){
    // 18 bytes of overhead in TX frame
    uint16_t frameSize = 18;
    uint16_t RFMax = 0x100;
    // check for encryption enabled
    if((opt & 2) == 0x02)
        RFMax -=4;
    // get length of RFData
    uint16_t RFSize = RFData->size();
    // only capture up to 256 bytes // throw error if too big
    if(RFSize > RFMax)
        return {0};
    // total frame size
    frameSize += RFSize;

    // begin forming frame
    std::vector<uint8_t> framePtr;// = new std::vector<uint8_t>;
    framePtr.push_back(0x7E);
    // assign sizes
    framePtr.push_back((frameSize-4) >> 8);
    framePtr.push_back((frameSize-4));
    // assign frame type and ID
    framePtr.push_back(0x10);
    // assign + increment running frame ID
    framePtr.push_back(GLOBAL_FRAME_ID);
    frameIDIncrement();
    // assign 8 byte address
    for(int i = 0; i < 8; ++i){
        framePtr.push_back(dst_64 >> (56-(8*i)));
    }
    //assign 2 byte address
    framePtr.push_back(dst_16 >> 8);
    framePtr.push_back(dst_16);
    // assign BCR and OPT
    framePtr.push_back(bcr);
    framePtr.push_back(opt);
    // assign RF data
    for(int i = 0; i < RFSize; ++i){
        framePtr.push_back(RFData->at(i));
    }
    // assign checksum
    framePtr.push_back(calculateCHKSM(framePtr.data(), frameSize));
    return framePtr;
}

std::vector<uint8_t>*  formATFrame_Remote(std::string ATCommand, uint64_t dst_64, std::string newvalue, uint8_t opt, uint16_t dst_16){
```

```
    // 19 bytes without optional value
    uint16_t frameSize = 19;
    // get optional value
    uint8_t *newValCpy = parseOptions(newvalue);
    uint16_t valLen = 0;
    if(newValCpy != nullptr){
        valLen = newValCpy[0];
        // update frame size
        frameSize += valLen;
        }
    // isolate CMD MSB and LSB
    std::string cmdBytes = ATCommand.substr(0, 2);
    // start forming frame
    std::vector<uint8_t>* framePtr = new std::vector<uint8_t>;
    framePtr->push_back(0x7E);
    // length field wants bytes from its end through checksum, so subtract overhead from frame
    int totalLength = frameSize - 4;
    framePtr->push_back((totalLength && 0xFF00) >> 8);
    // get LXB of length
    framePtr->push_back(totalLength);
    // assign AT Command identifier
    framePtr->push_back(0x17);
    // assign + increment running frame ID
    framePtr->push_back(GLOBAL_FRAME_ID);
    frameIDIncrement();
    // assign 8 byte address
    for(int i = 0; i < 8; ++i){
        framePtr->push_back(dst_64 >> (56-(8*i)));
    }
    //assign 2 byte address
    framePtr->push_back(dst_16 >> 8);
    framePtr->push_back(dst_16);
    // assign opt
    framePtr->push_back(opt);
    // assign command MSB and LSB
    framePtr->push_back(uint8_t(cmdBytes[0]));
    framePtr->push_back(uint8_t(cmdBytes[1]));
    if(int(valLen) > 0){
        for(int i = 0; i < valLen; ++i){
            framePtr->push_back(newValCpy[i+1]);
        }
    }
    // assign CHKSUM
    framePtr->push_back(calculateCHKSM(framePtr->data(), frameSize));
    return framePtr;
}

json parseATCR(uint8_t* frame, uint16_t frameLength){
    std::string cmd = "";
    cmd += char(frame[5]);
    cmd += char(frame[6]);
    std::string data = "";
    // length > 5 means there is data
    if(frameLength > 5){
      for(int i=0; i < frameLength-5; ++i)
        data += char(frame[i+8]);
    }
    json tmpData = json::parse(data);
    json tmp = {
        {"DESC", "Local AT Command Response"},
        {"FRAME TYPE", 0x88},
        {"FRAME OVERHEAD", {
            {"FRAME ID", frame[4]},
            {"COMMAND", cmd},
            {"STATUS", frame[7]}}},
        {"FRAME DATA",tmpData}
    };
    return tmp;
```

```
}

json parseMS(uint8_t* frame, uint16_t frameLength){
   json tmp = {
      {"DESC", "MODEM STATUS"},
      {"FRAME TYPE", 0x8A},
      {"FRAME OVERHEAD", {"FRAME ID", frame[3]}},
      {"FRAME DATA", {"STATUS", frame[4]}}
   };
   return tmp;
}

json parseTS(uint8_t* frame, uint16_t frameLength){
   uint32_t dst = 0;
   dst += uint16_t(frame[5]);
   dst += uint16_t(frame[6]);
   json tmp = {
      {"DESC", "Transmission Status"},
      {"FRAME TYPE", 0x8B},
      {"FRAME OVERHEAD", {"FRAME ID", frame[4]}},
      {"FRAME DATA", {
         {"DESTINATION", dst},
         {"TRANSMIT RETRY", frame[7]},
         {"DELIVERY STATUS", frame[8]},
         {"DISCOVERY STATUS", frame[9]}}},
   };
   return tmp;
}

json receivePacket(uint8_t* frame, uint16_t frameLength){
   // get addresses
   uint64_t dst64 = 0;
   for(int i = 0; i <8; ++i)
      dst64 += uint64_t(frame[i+4]) << (56 - (i*8));
   uint16_t dst16 = 0;
   dst16 += uint16_t(frame[12]) << 8;
   dst16 += uint16_t(frame[13]);
   std::string data = "";
   // length > 12 means there is data
   if(frameLength > 12){
     for(int i=0; i < frameLength-12; ++i)
      data += char(frame[i+15]);
   }
   json tmpData = json::parse(data);
   json tmp = {
      {"DESC", "Receive Packet"},
      {"FRAME TYPE", 0x90},
      {"FRAME OVERHEAD", {
         {"FRAME ID", frame[4]},
         {"DST64", dst64},
         {"DST16", dst16},
         {"OPT", frame[14]}}},
      {"FRAME DATA", tmpData}
   };
   return tmp;
}

json explicitRX(uint8_t* frame, uint16_t dataLength){
   // get addresses
   uint64_t dst64 = 0;
   for(int i = 0; i <8; ++i)
      dst64 += uint64_t(frame[i+4]) << (56 - (i*8));
   uint16_t dst16 = 0;
   dst16 += uint16_t(frame[12]) << 8;
   dst16 += uint16_t(frame[13]);
   uint32_t cluster = 0;
   cluster+= uint8_t(frame[16]) << 8;
   cluster+= uint8_t(frame[17]);
```

```
    uint32_t profile = 0;
    profile+= uint8_t(frame[18]) << 8;
    profile+= uint8_t(frame[19]);
    std::string data = "";
    // length > 18 means there is data
    if(dataLength > 18){
      for(int i=0; i < dataLength-18; ++i)
        data += char(frame[i+21]);
    }
    json tmpData = json::parse(data);
    json tmp = {
        {"DESC", "Receive Packet"},
        {"FRAME TYPE", 0x91},
        {"FRAME OVERHEAD", {
            {"FRAME ID", frame[4]},
            {"DST64", dst64},
            {"DST16", dst16},
            {"SRC", frame[14]},
            {"DST", frame[15]},
            {"CLUSTER", cluster},
            {"PROFILE", profile},
            {"OPT", frame[20]}}},
        {"FRAME DATA",tmpData}
    };
    return tmp;
}

json remoteAT(uint8_t* frame, uint16_t frameLength){
    // get addresses
    uint64_t dst64 = 0;
    for(int i = 0; i <8; ++i)
        dst64 += uint64_t(frame[i+5]) << (56 - (i*8));
    uint16_t dst16 = 0;
    dst16 += uint16_t(frame[13]) << 8;
    dst16 += uint16_t(frame[14]);
    std::string cmd = "0x";
    cmd += char(frame[15]);
    cmd += char(frame[16]);
    std::string data = "";
    // length > 12 means there is data
    if(frameLength > 15){
      for(int i=0; i < frameLength-15; ++i)
        data += char(frame[i+18]);
    }
    json tmpData = json::parse(data);
    json tmp = {
        {"DESC", "Receive Packet"},
        {"FRAME TYPE", 0x97},
        {"FRAME OVERHEAD", {
            {"FRAME ID", frame[4]},
            {"DST64", dst64},
            {"DST16", dst16}}},
        {"FRAME DATA",{
            {"DATA", tmpData},
            {"AT CMD", cmd},
            {"STATUS", frame[17]}}},

    };
    return tmp;
}

json handleRoutes(uint8_t* frame, uint16_t frameLength){
    // get addresses
    uint64_t dst64 = 0;
    for(int i = 0; i <8; ++i)
        dst64 += uint64_t(frame[i+4]) << (56 - (i*8));
    uint16_t dst16 = 0;
    dst16 += uint16_t(frame[12]) << 8;
```

```
    dst16 += uint16_t(frame[13]);

    uint8_t num_hops = frame[15];
    json hop_addresses = {};
    // copy addresses
    if(num_hops > 0){
        for(int i = 0; i < num_hops*2; ++i)
            hop_addresses.push_back(frame[16+i]);
    }

    json tmp = {
        {"DESC", "Receive Packet"},
        {"FRAME TYPE", 0xA1},
        {"FRAME OVERHEAD", {
            {"DST64", dst64},
            {"DST16", dst16}}},
        {"FRAME DATA",{
            {"DATA", hop_addresses}}}
        };
    return tmp;
};

json readFrame(uint8_t* frame){
    // return -2 if not valid
    json data = 0;
    if(frame[0] != 0x7E){
        data = -2;
    }
    else{
        // find data size
        uint16_t frameLength = frame[1] + frame[2];
        // read in appropriate data based on frame type
        switch(frame[3]){
            case 0xA1:
                data = handleRoutes(frame, frameLength);
                break;
            // AT command response
            case 0x88:
                data = parseATCR(frame, frameLength);
                break;
            // modem status
            case 0x8A:
                data = parseMS(frame, frameLength);
                break;
            // transmit status
            case 0x8B:
                data = parseTS(frame, frameLength);
                break;
            // receive packet
            case 0x90:
                data = receivePacket(frame, frameLength);
                break;
            // explicit rx
            case 0x91:
                data = explicitRX(frame, frameLength);
                break;
            // remote AT resposne
            case 0x97:
                data = remoteAT(frame, frameLength);
                break;
            // unknown case return -1
            default:
                data = -1;
                break;
        }
    }
    return data;
}
```

An advantage to using C++ is that functions can have optional arguments with assigned default values to use if no argument is provided. Default values are used extensively in the code above and the main functions listed in Table 12 so that only necessary values are be passed. For example, in the first function, formATFrame(), if no value for "newvalue" is provided, the function will create a frame that is used to query a parameter on a local XBee module. If a value for "newvalue" is provided, the resulting frame will try to update the parameter on the local XBee with the provided value. Hence, the same function can be used to read and write parameters on a local XBee module. The formATFrame_Remote() function has a similar operation, but it requires the desired parameter, the 64-bit address of the destination XBee module, a value for "newvalue", and the "opt" field passed a value of "0x02" to set a parameter on a remote XBee module. To view a given parameter, and the 64-bit XBee address are needed.

The formTXFrame() function always requires a string that contains the data that will be sent to be supplied, but by default, will send the frame to the coordinator of the Zigbee network. Providing alternate 64-bit and 16-bit destination addresses will allow the destination of the XBee frame to be changed. Finally, the readFrame() function will parse a provided frame and return a JSON object that contains all pertinent data in the frame. The readFrame function will function regardless of the type of frame that it is provided, as it identifies the type of frame by its unique identifier and uses that value to determine the essential information in the frame.

5.2.2.g ESP32 and ADE9153A Interface (JG)

To communicate with the Analog Devices ADE9153As, a serial interface needs to be established between the ESP32 and each ADE9153A. The ADE9153A supports SPI communication operating at a Baud Rate of 10000000. As 2 ADE9153As are used in each Smat Power Outlet, SPI is a

desirable choice for the serial communication method as its use of a chip select line allows multiple ADE9153As to share the same communication lines (MISO, MOSI, and SCLK), while only needing their own reset and chip select lines. Hence only 7 connections are required to use both ADE9153As. Analog Devices provides an Arduino library used to communicate[5] with the ADE9153A and sample Arduino code utilizing[6] their library. This code was modified to work on the ESP32 by treating their ADE9153A class as an instantiation of an SPI bus, with the pins for each ADE9153A defined within it. By doing this, one ADE9153A object can control both ADE9153As used in the Smart Power Outlet. Here, the provided Arduino code had to be modified and used as attempts to recreate the sampel code's functionality using ESP-IDF did not work, leaving Analog's provided code as the only method of interfacing with the ADE9153As. The modified ADE9153A code is shown below and is a combination of Analog Device's ADE9153A API and ADE9153A example code, with additional changes to allow the code to control two ADE9153As. Additionally, the file "ADE9153A.h" by Analog Devices is required, but it has not been modified so it is not included. The original code provided by Analog Devices can be found at the GitHub links referenced prior.

ADE9153AClass.h:

```
#ifndef ADE9153AClass_H
#define ADE9153AClass_H
/**************************************************************************
  ADE9153AAPI.h - Library for ADE9153A energy measurement IC with mSure
  autocalibration

  Designed specifically to work with the EV_ADE9153ASHIELDZ board
  ---- http://www.analog.com/ADE9153A

  Created by David Lath for Analog Devices Inc., January 8, 2018
```

[5] https://github.com/analogdevicesinc/arduino/tree/master/Arduino%20Uno%20R3/libraries/ADE9153A

[6] https://github.com/analogdevicesinc/arduino/blob/master/Arduino%20Uno%20R3/examples/ADE9153A_examples/ADE9153AAPI_Test/ADE9153AAPI_Test.ino

```
#pragma once
#include "Arduino.h"
#include "ADE9153A.h"
/**************************************************************************
 Definitions
 **************************************************************************/
/* Configuration Registers */
#define ADE9153A_AI_PGAGAIN 0x000A      /*Signal on IAN, current channel gain=16x*/
#define ADE9153A_CONFIG0 0x00000000     /*Datapath settings at default*/
#define ADE9153A_CONFIG1 0x0300         /*Chip settings at default*/
#define ADE9153A_CONFIG2 0x0C00         /*High-pass filter corner, fc=0.625Hz*/
#define ADE9153A_CONFIG3 0x0000         /*Peak and overcurrent settings*/
#define ADE9153A_ACCMODE 0x0010         /*Energy accumulation modes, Bit 4, 0 for 50Hz, 1 for 60Hz*/
#define ADE9153A_VLEVEL 0x002C11E8      /*Assuming Vnom=1/2 of fullscale*/
#define ADE9153A_ZX_CFG 0x0000          /*ZX low-pass filter select*/
#define ADE9153A_MASK 0x00000100        /*Enable EGYRDY interrupt*/
#define ADE9153A_ACT_NL_LVL 0x000033C8
#define ADE9153A_REACT_NL_LVL 0x000033C8
#define ADE9153A_APP_NL_LVL 0x000033C8
/* Constant Definitions */
#define ADE9153A_RUN_ON 0x0001          /*DSP On*/
#define ADE9153A_COMPMODE 0x0005        /*Initialize for proper operation*/
#define ADE9153A_VDIV_RSMALL 0x03E8     /*Small resistor on board is 1kOhm=0x3E8*/

/* Energy Accumulation Settings */
#define ADE9153A_EP_CFG 0x0009          /*Energy accumulation configuration*/
#define ADE9153A_EGY_TIME 0x0F9F        /*Accumulate energy for 4000 samples*/

/* Temperature Sensor Settings */
#define ADE9153A_TEMP_CFG 0x000C        /*Temperature sensor configuration*/

/* Ideal Calibration Values for ADE9153A Shield Based on Sensor Values */
#define CAL_IRMS_CC    1   // (uA/code)
#define CAL_VRMS_CC    8.84   // (uV/code)
#define CAL_POWER_CC   1   // (uW/code) Applicable for Active, reactive and apparent power
#define CAL_ENERGY_CC  1   // (uWhr/xTHR_HI code)Applicable for Active, reactive and apparent energy
```

```
/**************************************************************************
  Structures and Global Variables
 **************************************************************************/
struct EnergyRegs {
   int32_t ActiveEnergyReg;
   int32_t FundReactiveEnergyReg;
   int32_t ApparentEnergyReg;
   float ActiveEnergyValue;
   float FundReactiveEnergyValue;
   float ApparentEnergyValue;
};

struct PowerRegs {
   int32_t ActivePowerReg;
   float ActivePowerValue;
   int32_t FundReactivePowerReg;
   float FundReactivePowerValue;
   int32_t ApparentPowerReg;
   float ApparentPowerValue;
};

struct RMSRegs {
   int32_t CurrentRMSReg;
   float CurrentRMSValue;
   int32_t VoltageRMSReg;
   float VoltageRMSValue;
};

struct HalfRMSRegs {
   int32_t HalfCurrentRMSReg;
   float HalfCurrentRMSValue;
   int32_t HalfVoltageRMSReg;
   float HalfVoltageRMSValue;
};

struct PQRegs {
   int32_t PowerFactorReg;
   float PowerFactorValue;
   int32_t PeriodReg;
   float FrequencyValue;
   int32_t AngleReg_AV_AI;
   float AngleValue_AV_AI;
};

struct AcalRegs {
   int32_t AcalAICCReg;
   float AICC;
   int32_t AcalAICERTReg;
   int32_t AcalAVCCReg;
   float AVCC;
   int32_t AcalAVCERTReg;
};

struct Temperature {
   uint16_t TemperatureReg;
   float TemperatureVal;
};

class ADE9153AClass
{
public:
   ADE9153AClass(void);
   void SetupADE9153A(uint8_t CS_PIN);

   /* SPI Functions */
   bool SPI_Init_Bus(uint32_t SPI_speed , uint8_t SCLK_Pin, uint8_t MISO_Pin, uint8_t MOSI_Pin, uint8_t chipSelect_Pin_Top, uint8_t
chipSelect_Pin_Bottom, uint8_t dummy_chipSelect_Pin);
   //bool SPI_Init(uint32_t SPI_speed, uint8_t chipSelect_Pin);
```

```
    //bool SPI_Init_bot(uint32_t SPI_speed, uint8_t chipSelect_Pin);
    void SPI_Write_16(uint8_t CS_Pin, uint16_t Address, uint16_t Data );
    void SPI_Write_32(uint8_t CS_Pin, uint16_t Address, uint32_t Data );
    uint16_t SPI_Read_16(uint8_t CS_Pin, uint16_t Address);
    uint32_t SPI_Read_32(uint8_t CS_Pin, uint16_t Address);

    /* ADE9153A Calculated Paramter Read Functions */
    void ReadEnergyRegs(uint8_t CS_Pin, EnergyRegs *Data);
    void ReadPowerRegs(uint8_t CS_Pin, PowerRegs *Data);
    void ReadRMSRegs(uint8_t CS_Pin, RMSRegs *Data);
    void ReadHalfRMSRegs(uint8_t CS_Pin, HalfRMSRegs *Data);
    void ReadPQRegs(uint8_t CS_Pin, PQRegs *Data);
    void ReadAcalRegs(uint8_t CS_Pin, AcalRegs *Data);
    bool StartAcal_AINormal(uint8_t CS_Pin);
    bool StartAcal_AITurbo(uint8_t CS_Pin);
    bool StartAcal_AV(uint8_t CS_Pin);
    void StopAcal(uint8_t CS_Pin);
    void ReadTemperature(uint8_t CS_Pin, Temperature *Data);

private:
    uint8_t _dummy_chipSelect_Pin;
    uint8_t _chipSelect_Pin_top;
    uint8_t _chipSelect_Pin_bottom;
};
#endif
```

ADE9153AClass.cpp:

```
/*****************************************************************************
  ADE9153AAPI.cpp - Library for ADE9153A energy measurement IC with mSure
  autocalibration

  Designed specifically to work with the EV_ADE9153ASHIELDZ board
  ---- http://www.analog.com/ADE9153A

  Created by David Lath for Analog Devices Inc., January 8, 2018

  Copyright (c) 2018, Analog Devices, Inc.  All rights reserved.

  Redistribution and use in source and binary forms, with or without
  modification, are permitted (subject to the limitations in the disclaimer
  below) provided that the following conditions are met:

  * Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.

  * Neither the name of Analog Devices, Inc. nor the names of its
  contributors may be used to endorse or promote products derived from this
  software without specific prior written permission.

  NO EXPRESS OR IMPLIED LICENSES TO ANY PARTY'S PATENT RIGHTS ARE GRANTED
  BY THIS LICENSE.  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
  CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING,
  BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
  FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
  TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
  PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
  LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
  NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
  SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
*************************************************************************/
#include <Arduino.h>
#include <SPI.h>
#include "ADE9153AClass.h"
#include <Wire.h>

SPIClass * hspi = NULL;

 ADE9153AClass::ADE9153AClass()
{

}

/*
Description: Initializes the ADE9153A. The initial settings for registers are defined in ADE9153AAPI.h header file
Input: Register settings in header files
Output:-
*/
void ADE9153AClass::SetupADE9153A(uint8_t CS_PIN)
{
SPI_Write_16(CS_PIN, REG_AI_PGAGAIN,ADE9153A_AI_PGAGAIN);
SPI_Write_32(CS_PIN, REG_CONFIG0,ADE9153A_CONFIG0);
SPI_Write_16(CS_PIN, REG_CONFIG1,ADE9153A_CONFIG1);
SPI_Write_16(CS_PIN, REG_CONFIG2,ADE9153A_CONFIG2);
SPI_Write_16(CS_PIN, REG_CONFIG3,ADE9153A_CONFIG3);
SPI_Write_16(CS_PIN, REG_ACCMODE,ADE9153A_ACCMODE);
SPI_Write_32(CS_PIN, REG_VLEVEL,ADE9153A_VLEVEL);
SPI_Write_16(CS_PIN, REG_ZX_CFG,ADE9153A_ZX_CFG);
SPI_Write_32(CS_PIN, REG_MASK,ADE9153A_MASK);
SPI_Write_32(CS_PIN, REG_ACT_NL_LVL,ADE9153A_ACT_NL_LVL);
SPI_Write_32(CS_PIN, REG_REACT_NL_LVL,ADE9153A_REACT_NL_LVL);
SPI_Write_32(CS_PIN, REG_APP_NL_LVL,ADE9153A_APP_NL_LVL);
SPI_Write_16(CS_PIN, REG_COMPMODE,ADE9153A_COMPMODE);
SPI_Write_32(CS_PIN, REG_VDIV_RSMALL,ADE9153A_VDIV_RSMALL);
SPI_Write_16(CS_PIN, REG_EP_CFG,ADE9153A_EP_CFG);
SPI_Write_16(CS_PIN, REG_EGY_TIME,ADE9153A_EGY_TIME);                    //Energy accumulation ON
SPI_Write_16(CS_PIN, REG_TEMP_CFG,ADE9153A_TEMP_CFG);
}

// initalize SPI bus for 2x ADE9153A
bool ADE9153AClass::SPI_Init_Bus(uint32_t SPI_speed , uint8_t SCLK_Pin, uint8_t MISO_Pin, uint8_t MOSI_Pin, uint8_t
chipSelect_Pin_Top, uint8_t chipSelect_Pin_Bottom, uint8_t dummy_chipSelect_Pin){
hspi = new SPIClass(HSPI);
hspi->begin(SCLK_Pin, MISO_Pin, MOSI_Pin, dummy_chipSelect_Pin);                    // initalize with dummy CS pin, bit bang desired CS pin
// save pin assignments
_chipSelect_Pin_top = chipSelect_Pin_Top;
_chipSelect_Pin_bottom = chipSelect_Pin_Bottom;
_dummy_chipSelect_Pin = dummy_chipSelect_Pin;

// set pins as outputs
pinMode(_dummy_chipSelect_Pin, OUTPUT);
pinMode(_chipSelect_Pin_top, OUTPUT);
pinMode(_chipSelect_Pin_bottom, OUTPUT);

// set outputs high
digitalWrite(_dummy_chipSelect_Pin, HIGH);
digitalWrite(_chipSelect_Pin_top, HIGH);
digitalWrite(_chipSelect_Pin_bottom, HIGH);

//  spi settings for ADE9153a
hspi->beginTransaction(SPISettings(SPI_speed, MSBFIRST, SPI_MODE0));

// bools for setup
bool top_init, bottom_init;

// check top
SPI_Write_16(_chipSelect_Pin_top, REG_RUN, ADE9153A_RUN_ON);
delay(100);
```

```
if(SPI_Read_32(_chipSelect_Pin_top, REG_VERSION_PRODUCT) != 0x0009153A){
top_init = false;
Serial.println("ERROR Initalizing Top ADE9153A");
}

else{
top_init = true;
Serial.println("Top ADE9153A Initalized");
}
/*
// check bottom
SPI_Write_16(_chipSelect_Pin_bottom, REG_RUN, ADE9153A_RUN_ON);
delay(100);
if(SPI_Read_32(_chipSelect_Pin_bottom, REG_VERSION_PRODUCT) != 0x0009153A){
bottom_init = false;
Serial.println("ERROR Initalizing bottom ADE9153A");
}
else{
bottom_init = true;
Serial.println("Bottom ADE9153A Initalized");
}
return (top_init & bottom_init);*/
return top_init;
}



/*
Description: Initializes the arduino SPI port using SPI.h library
Input: SPI speed, chip select pin
Output:-
*/

/*bool ADE9153AClass::SPI_Init(uint32_t SPI_speed , uint8_t chipSelect_Pin)
{
hspi = new SPIClass(HSPI);
hspi->begin(SCLK, MISO, MOSI, CS_PIN); //SCLK, MISO, MOSI, SS
digitalWrite(chipSelect_Pin, HIGH);                     //Set Chip select pin high
pinMode(chipSelect_Pin, OUTPUT);                        //Set Chip select pin as output
hspi->begin();                          //Initiate SPI port
hspi->beginTransaction(SPISettings(SPI_speed,MSBFIRST,SPI_MODE0));              //Setup SPI parameters


SPI_Write_16(REG_RUN,ADE9153A_RUN_ON);
delay(100);
if (SPI_Read_32(REG_VERSION_PRODUCT) != 0x0009153A)
return false;

return true;
}

bool ADE9153AClass::SPI_Init_bot(uint32_t SPI_speed , uint8_t chipSelect_Pin)
{
digitalWrite(chipSelect_Pin, HIGH);                     //Set Chip select pin high
pinMode(chipSelect_Pin, OUTPUT);                        //Set Chip select pin as output
//hspi->beginTransaction(SPISettings(SPI_speed,MSBFIRST,SPI_MODE0));              //Setup SPI parameters

_chipSelect_Pin = chipSelect_Pin;
digitalWrite(chipSelect_Pin, LOW);
SPI_Write_16(REG_RUN,ADE9153A_RUN_ON);
digitalWrite(chipSelect_Pin, HIGH);
delay(100);
digitalWrite(chipSelect_Pin, LOW);
if (SPI_Read_32(REG_VERSION_PRODUCT) != 0x0009153A){
digitalWrite(chipSelect_Pin, HIGH);
return false;
}
```

```
digitalWrite(chipSelect_Pin, LOW);
return true;
}
*/

/*
Description: Writes 16bit data to a 16 bit register.
Input: Register address, data
Output:-
*/

void ADE9153AClass:: SPI_Write_16(uint8_t CS_Pin, uint16_t Address , uint16_t Data )
{
uint16_t temp_address;

digitalWrite(CS_Pin, LOW);
temp_address = ((Address << 4) & 0xFFF0);                //shift address  to align with cmd packet
hspi->transfer16(temp_address);
hspi->transfer16(Data);

digitalWrite(CS_Pin, HIGH);
}

/*
Description: Writes 32bit data to a 32 bit register.
Input: Register address, data
Output:-
*/

void ADE9153AClass:: SPI_Write_32(uint8_t CS_Pin, uint16_t Address , uint32_t Data )
{
uint16_t temp_address;
uint16_t temp_highpacket;
uint16_t temp_lowpacket;

temp_highpacket = (Data & 0xFFFF0000) >> 16;
temp_lowpacket = (Data & 0x0000FFFF);

digitalWrite(CS_Pin, LOW);

temp_address = ((Address << 4) & 0xFFF0);                //shift address  to align with cmd packet
hspi->transfer16(temp_address);
hspi->transfer16(temp_highpacket);
hspi->transfer16(temp_lowpacket);

digitalWrite(CS_Pin, HIGH);
}

/*
Description: Reads 16bit data from register.
Input: Register address
Output: 16 bit data
*/

uint16_t ADE9153AClass:: SPI_Read_16(uint8_t CS_Pin, uint16_t Address)
{
uint16_t temp_address;
uint16_t returnData;

digitalWrite(CS_Pin, LOW);

temp_address = (((Address << 4) & 0xFFF0) + 8);
hspi->transfer16(temp_address);
returnData = hspi->transfer16(0);

digitalWrite(CS_Pin, HIGH);
return returnData;
}
```

```
/*
Description: Reads 32bit data from register.
Input: Register address
Output: 32 bit data
*/

uint32_t ADE9153AClass:: SPI_Read_32(uint8_t CS_Pin, uint16_t Address)
{
uint16_t temp_address;
uint32_t temp_highpacket;
uint16_t temp_lowpacket;
uint32_t returnData;

digitalWrite(CS_Pin, LOW);

temp_address = (((Address << 4) & 0xFFF0) + 8);
hspi->transfer16(temp_address);
temp_highpacket = hspi->transfer16(0);
temp_lowpacket = hspi->transfer16(0);

digitalWrite(CS_Pin, HIGH);

returnData = temp_highpacket << 16;
returnData = returnData + temp_lowpacket;

return returnData;
}

/*
Description: Reads the metrology data from the ADE9153A
Input: Structure name
Output: Respective metrology data
*/

void ADE9153AClass:: ReadEnergyRegs(uint8_t CS_Pin, struct EnergyRegs *Data)
{
int32_t tempReg;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AWATTHR_HI));
Data->ActiveEnergyReg = tempReg;
tempValue = (float)tempReg * CAL_ENERGY_CC / 1000;
Data->ActiveEnergyValue = tempValue;                                    //Energy in mWhr

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AFVARHR_HI));
Data->FundReactiveEnergyReg = tempReg;
tempValue = (float)tempReg * CAL_ENERGY_CC / 1000;
Data->FundReactiveEnergyValue = tempValue;                              //Energy in mVARhr

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AVAHR_HI));
Data->ApparentEnergyReg = tempReg;
tempValue = (float)tempReg * CAL_ENERGY_CC / 1000;
Data->ApparentEnergyValue = tempValue;                                  //Energy in mVAhr
}

void ADE9153AClass:: ReadPowerRegs(uint8_t CS_Pin, struct PowerRegs *Data)
{
int32_t tempReg;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AWATT));
Data->ActivePowerReg = tempReg;
tempValue = (float)tempReg * CAL_POWER_CC / 1000;
Data->ActivePowerValue = tempValue;                                             //Power in mW

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AFVAR));
Data->FundReactivePowerReg = tempReg;
```

```
tempValue = (float)tempReg * CAL_POWER_CC / 1000;
Data->FundReactivePowerValue = tempValue;                                    //Power in mVAR

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AVA));
Data->ApparentPowerReg = tempReg;
tempValue = (float)tempReg * CAL_POWER_CC / 1000;
Data->ApparentPowerValue = tempValue;                                        //Power in mVA
}

void ADE9153AClass:: ReadRMSRegs(uint8_t CS_Pin, struct RMSRegs *Data)
{
uint32_t tempReg;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AIRMS));
Data->CurrentRMSReg = tempReg;
tempValue = (float)tempReg * CAL_IRMS_CC / 1000; //RMS in mA
Data->CurrentRMSValue = tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin,REG_AVRMS));
Data->VoltageRMSReg = tempReg;
tempValue = (float)tempReg * CAL_VRMS_CC / 1000;              //RMS in mV
Data->VoltageRMSValue = tempValue;
}

void ADE9153AClass:: ReadHalfRMSRegs(uint8_t CS_Pin, struct HalfRMSRegs *Data)
{
uint32_t tempReg;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AIRMS_OC));
Data->HalfCurrentRMSReg = tempReg;
tempValue = (float)tempReg * CAL_IRMS_CC / 1000; //Half-RMS in mA
Data->HalfCurrentRMSValue = tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_AVRMS_OC));
Data->HalfVoltageRMSReg = tempReg;
tempValue = (float)tempReg * CAL_VRMS_CC / 1000;              //Half-RMS in mV
Data->HalfVoltageRMSValue = tempValue;
}

void ADE9153AClass:: ReadPQRegs(uint8_t CS_Pin, struct PQRegs *Data)
{
int32_t tempReg;
uint16_t temp;
float mulConstant;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_APF)); //Read PF register
Data->PowerFactorReg = tempReg;
tempValue = (float)tempReg / (float)134217728; //Calculate PF
Data->PowerFactorValue = tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_APERIOD)); //Read PERIOD register
Data->PeriodReg = tempReg;
tempValue = (float)(4000 * 65536) / (float)(tempReg + 1); //Calculate Frequency
Data->FrequencyValue = tempValue;

temp = SPI_Read_16(CS_Pin, REG_ACCMODE); //Read frequency setting register
if((temp & 0x0010) > 0){
mulConstant = 0.02109375;  //multiplier constant for 60Hz system
}else{
mulConstant = 0.017578125; //multiplier constant for 50Hz system
}
tempReg = int16_t (SPI_Read_16(CS_Pin, REG_ANGL_AV_AI)); //Read ANGLE register
Data->AngleReg_AV_AI = tempReg;
tempValue = tempReg * mulConstant;          //Calculate Angle in degrees
Data->AngleValue_AV_AI = tempValue;
```

```
}

void ADE9153AClass:: ReadAcalRegs(uint8_t CS_Pin, struct AcalRegs *Data)
{
uint32_t tempReg;
float tempValue;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_MS_ACAL_AICC)); //Read AICC register
Data->AcalAICCReg = tempReg;
tempValue = (float)tempReg / (float)2048; //Calculate Conversion Constant (CC)
Data->AICC = tempValue;
tempReg = int32_t (SPI_Read_32(CS_Pin, REG_MS_ACAL_AICERT)); //Read AICERT register
Data->AcalAICERTReg = tempReg;

tempReg = int32_t (SPI_Read_32(CS_Pin, REG_MS_ACAL_AVCC)); //Read AVCC register
Data->AcalAVCCReg = tempReg;
tempValue = (float)tempReg / (float)2048; //Calculate Conversion Constant (CC)
Data->AVCC = tempValue;
tempReg = int32_t (SPI_Read_32(CS_Pin, REG_MS_ACAL_AVCERT)); //Read AICERT register
Data->AcalAVCERTReg = tempReg;
}

/*
Description: Start autocalibration on the respective channel
Input: -
Output: Did it start correctly?
*/

bool ADE9153AClass::StartAcal_AINormal(uint8_t CS_Pin)
{
uint32_t ready = 0;
int waitTime = 0;

ready = SPI_Read_32(CS_Pin, REG_MS_STATUS_CURRENT);                                    //Read system ready bit

while((ready & 0x00000001) == 0)
{
if(waitTime > 11)
{
return false;
}
delay(100);
waitTime++;
}
SPI_Write_32(CS_Pin, REG_MS_ACAL_CFG, 0x00000013);
return true;
}

bool ADE9153AClass::StartAcal_AITurbo(uint8_t CS_Pin)
{
uint32_t ready = 0;
int waitTime = 0;

while((ready & 0x00000001) == 0)
{
ready = SPI_Read_32(CS_Pin, REG_MS_STATUS_CURRENT);                //Read system ready bit
if(waitTime  >15)
{
return false;
}
delay(100);
waitTime++;
}
SPI_Write_32(CS_Pin, REG_MS_ACAL_CFG, 0x00000017);
return true;
}

bool ADE9153AClass::StartAcal_AV(uint8_t CS_Pin)
```

```
{
uint32_t ready = 0;
int waitTime = 0;

while((ready & 0x00000001) == 0)
{
ready = SPI_Read_32(CS_Pin, REG_MS_STATUS_CURRENT);                    //Read system ready bit
if(waitTime > 15)
{
return false;
}
delay(100);
waitTime++;
}
SPI_Write_32(CS_Pin, REG_MS_ACAL_CFG, 0x00000043);
return true;
}

void ADE9153AClass::StopAcal(uint8_t CS_Pin)
{
SPI_Write_32(CS_Pin, REG_MS_ACAL_CFG, 0x00000000);
}

/*
Description: Starts a new acquisition cycle. Waits for constant time and returns register value and temperature in Degree Celsius
Input:        Structure name
Output: Register reading and temperature value in Degree Celsius
*/

void ADE9153AClass:: ReadTemperature(uint8_t CS_Pin, struct Temperature * Data)
{
uint32_t trim;
uint16_t gain;
uint16_t offset;
uint16_t tempReg;
float tempValue;

SPI_Write_16(CS_Pin, REG_TEMP_CFG,ADE9153A_TEMP_CFG);//Start temperature acquisition cycle
delay(10); //delay of 2ms. Increase delay if TEMP_TIME is changed

trim = SPI_Read_32(CS_Pin, REG_TEMP_TRIM);
gain = (trim & 0xFFFF);  //Extract 16 LSB
offset = ((trim >> 16) & 0xFFFF); //Extract 16 MSB
tempReg = SPI_Read_16(CS_Pin, REG_TEMP_RSLT);                  //Read Temperature result register
tempValue = ((float)offset / 32.00) - ((float)tempReg * (float)gain/(float)131072);

Data->TemperatureReg = tempReg;
Data->TemperatureVal = tempValue;
}
```

6. Mechanical Sketch (JG)

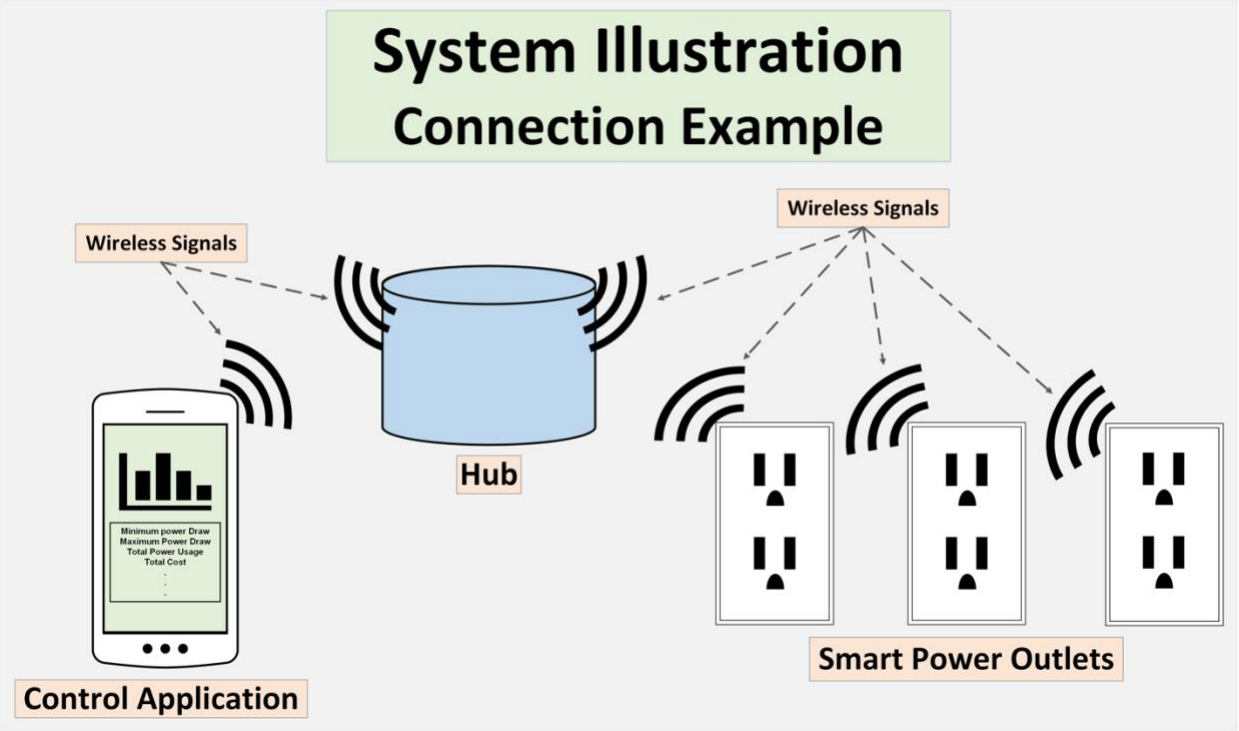

Figure 49: Mechanical Sketch depicting the Smart Power Outlet System

In Figure 43 above, a sketch of what the final connected Smart Power Outlet system will look like is shown. On the left is the Control Application, which is connected to the Hub and used to view measurements, statistics, and control the Smart Power Outlets. In the middle is the Hub, which will be placed in the user's home and communicate with all the Smart Power Outlets. Finally, on the right are the Smart Power Outlets, which will replace the user's current standard duplex power outlets.

7. Team Information (EF)

Table 13: Team Information

Team Names:	Role:	Major:
Madison Britton	Engineering Data Manager	Electrical Engineering
Ethan Frese	Hardware Manager	Electrical Engineering
Joseph Garro	Project Leader	Computer Engineering
Kyrolos Melek	Software Manager	Computer Engineering

8. Parts Lists

8.1. Schematic Parts List (EF)

Table 14: Schematic Parts List

Qty.	Refdes	Part Num.	Description
3	U1	ESP32-DEVKITC-DA	MCU to function as brains of Smart Power Outlets and Hub.
3	U2	XB24CZ7WITB003	Zigbee transceiver module
3	U3	LD03-23B05R2	AC/DC Converter
4	U4	AC1015	Current Sensor
6	U5	PR12-5V-360-1A	General Purpose Relay
10	U6	LSF0102DCTR	Level Converter
1	U6	1212	SMT ADAP 6 PACK 8SOIC/MSOP/TSSOP
3	U2	BOB-08276	BREAKOUT BOARD FOR XBEE MODULE
1	U2	WRL-11812	USB BREAKOUT BOARD FOR XBEE MODULE
10	IC1	CS5480-INZ	CS5480 Power Measurement IC
2		PA0064C	CS5480 Breakout Board
10		4N35	Optocoupler
10		AC03000001209JAC00	Wire-Wound Resistor
10		B72210S0351K101	Varistor
10		SMBJ7.0A	TVS Diode
5		36911000000	Fuse
3	R1	WSK12161L000FEK	Shunt Resistor (3W, 0.001 Ohm, Surface Mount)
3	U7	ADuM3152BRSZ	Digital Isolator, 3750Vrms protection, 7ch SPI, 20SSOP

10	J1	691213710002	2-pos Screw Terminal for PCB
10	J2	B4B-EH-A(LF)(SN)	4-pos JST Connection Header
10		LM3671MF-3.3/NOPB	Buck Switching Regulator IC Fixed 3.3V Output 600mA
10		NRH2412T2R2MNGH	2.2uH Shielded Drum, Wirewound Inductor
6		ADE9153AACPZ	Single Phase Meter IC
5		ADP2504ACPZ-3.3-R7	Buck-Boost switching regulator
5		C2012X5R0J226M085AB	22uF ceramic capacitor
5		C1608JB0J106K080AB	10uF ceramic capacitor
5		NR3015T1R5N	1.5uH wirewound inductor
10		CL31B104KBCNNNC	0.1uF Capacitor
10		CL31B105KBHNNNE	1uF Capacitor
10		CL31A226KAHNNNE	22uF Capacitor
10		CL31A106KAHNFNE	10uF Capacitor
5		C1206C829D5GAC7800	8.2pF Capacitor
2		C1206C472K5RECAUTO	4.7nF Capacitor
4		TNPW12065K10BEEA	5.1k Resistor
2		CRGP1206F1M0	1M Resistor
10		CRGCQ1206J10K	10k Resistor
5		ERA-8AEB3483V	348k Resistor
5		ERA-8AEB623V	62k Resistor
2		ESP32-S3-WROOM-1U-N16R8	Esp32 uC
3		XB3-24Z8UM-J	Xbee 3
2		USB4125-GF-A	USB C Power Header
2		61200621621	2x3 Male IDC Header
2		3020-10-0100-00	2x5 male IDC Header
3		SM04B-SRSS-TB(LF)(SN)	4 Pin JST Header
10		S10B-PHDSS(LF)(SN)	2x5 JST Header
4		LTST-C230AKT	Orange LED
4		LTST-C150KRKT	Red LED
10		LTST-C150KGKT	Green LED
2		W1095K	SMA Antenna
2		CAB.011	u.FL to SMA Adapter
10		PHDR-10VS	JST Connector Plug
100		SPHD-001T-P0.5	JST Crimps
16		CRCW120660R0KNEAIF	60 Resistor
1		5151106F	LIGHT PIPE 3 ELEMENT
18		691213610001	1-Entry Screw Terminal
24		RC1206FR-07249KL	249k SMD Resistor

15		RC1206JR-07150RL	150 SMD Resistor
15		RC1206FR-071KL	1k SMD Resistor
15		RC1206FR-1010KL	10k SMD Resistor
75		CC1206KFX7R0BB104	0.1u SMD Capacitor
15		CC1206KRX7R8BB474	0.47u SMD Capacitor
15		CC1206JRNPO9BN220	22p SMD Capacitor
6		CC1206ZRY5V8BB105	1u SMD Capacitor
6		CC1206KKX7R9BB154	0.15u SMD Capacitor
50		CL31B475KAHNNNE	4.7u SMD Capacitor
5		Y14-5721755A	ADE9153A PCB
5		Y15-5721755A	DC-DC Converter PCB
10		RT1206BRD0750KL	50k SMD Resistor
10		RC1206FR-07280KL	280k SMD Resistor
6		ADE9153AACPZ	Single Phase Meter IC

8.2. Materials Budget List (EF)

Table 15: Materials Budget List

Qty.	Part Num.	Description	Unit Cost	Total Cost
3	ESP32-DEVKITC-DA	MCU to function as brains of Smart Power Outlets and Hub.	15.00	45.00
3	XB24CZ7WITB003	Zigbee transceiver module	37.66	112.98
3	LD03-23B05R2	AC/DC Converter	5.33	15.99
4	AC1015	Current Sensor	5.53	22.12
6	PR12-5V-360-1A	General Purpose Relay	1.03	6.18
10	LSF0102DCTR	Level Converter	0.67	6.65
1	1212	SMT ADAP 6 PACK 8SOIC/MSOP/TSSOP	2.95	2.95
3	BOB-08276	BREAKOUT BOARD FOR XBEE MODULE	3.50	10.50
1	WRL-11812	USB BREAKOUT BOARD FOR XBEE MODULE	27.95	27.95
10	CS5480-INZ	CS5480 Power Measurement IC	4.89	48.90
2	PA0064C	CS5480 Breakout Board	8.79	17.58
10	4N35	Optocoupler	0.40	3.98
10	AC03000001209JAC00	Wire-Wound Resistor	0.62	6.22
10	B72210S0351K101	Varistor	0.41	4.14
10	SMBJ7.0A	TVS Diode	0.32	3.20
5	36911000000	Fuse	0.57	2.86
3	WSK12161L000FEK	Shunt Resistor (3W, 0.001 Ohm, Surface Mount)	0.94	2.82

3	ADuM3152BRSZ	Digital Isolator, 3750Vrms protection, 7ch SPI, 20SSOP	9.18	27.54
10	691213710002	2-pos Screw Terminal for PCB	0.92	9.19
10	B4B-EH-A(LF)(SN)	4-pos JST Connection Header	0.18	1.81
10	LM3671MF-3.3/NOPB	Buck Switching Regulator IC Fixed 3.3V Output 600mA	1.57	15.70
10	NRH2412T2R2MNGH	2.2uH Shielded Drum, Wirewound Inductor	0.20	2.00
6	ADE9153AACPZ	Single Phase Meter IC	10.04	60.24
5	ADP2504ACPZ-3.3-R7	Buck-Boost switching regulator	5.48	27.40
5	C2012X5R0J226M085AB	22uF ceramic capacitor	0.40	2.00
5	C1608JB0J106K080AB	10uF ceramic capacitor	0.33	1.65
5	NR3015T1R5N	1.5uH wirewound inductor	0.36	1.80
10	CL31B104KBCNNNC	0.1uF Capacitor		0.78
10	CL31B105KBHNNNE	1uF Capacitor		2.51
10	CL31A226KAHNNNE	22uF Capacitor		1.6
10	CL31A106KAHNFNE	10uF Capacitor		1.6
5	C1206C829D5GAC7800	8.2pF Capacitor	0.34	1.7
2	C1206C472K5RECAUTO	4.7nF Capacitor	0.26	0.52
4	TNPW12065K10BEEA	5.1k Resistor	0.66	2.64
2	CRGP1206F1M0	1M Resistor	0.33	0.66
10	CRGCQ1206J10K	10k Resistor		0.58
5	ERA-8AEB3483V	348k Resistor	0.66	3.3
5	ERA-8AEB623V	62k Resistor	0.66	3.3
2	ESP32-S3-WROOM-1U-N16R8	Esp32 uC	4.29	8.58
3	XB3-24Z8UM-J	Xbee 3	21.32	63.96
2	USB4125-GF-A	USB C Power Header	0.67	1.34
2	61200621621	2x3 Male IDC Header	0.48	0.96
2	3020-10-0100-00	2x5 male IDC Header	0.61	1.22
3	SM04B-SRSS-TB(LF)(SN)	4 Pin JST Header	0.59	1.77
10	S10B-PHDSS(LF)(SN)	2x5 JST Header		4.53
4	LTST-C230AKT	Orange LED	0.37	1.48
4	LTST-C150KRKT	Red LED	0.35	1.4
10	LTST-C150KGKT	Green LED		1.88
2	W1095K	SMA Antenna	4.03	8.06
2	CAB.011	u.FL to SMA Adapter	3.07	6.14
10	PHDR-10VS	JST Connector Plug		2.96
100	SPHD-001T-P0.5	JST Crimps		3.21
16	CRCW120660R0KNEAIF	60 Resistor	0.66	10.56
1	5151106F	LIGHT PIPE 3 ELEMENT	1.82	1.82
18	691213610001	1-Entry Screw Terminal	0.43	7.67
24	RC1206FR-07249KL	249k SMD Resistor	0.06	1.42
15	RC1206JR-07150RL	150 SMD Resistor	0.05	0.81
15	RC1206FR-071KL	1k SMD Resistor	0.06	0.89

15	RC1206FR-1010KL	10k SMD Resistor	0.06	0.89
75	CC1206KFX7R0BB104	0.1u SMD Capacitor	0.07	4.92
15	CC1206KRX7R8BB474	0.47u SMD Capacitor	0.19	2.88
15	CC1206JRNPO9BN220	22p SMD Capacitor	0.17	2.49
6	CC1206ZRY5V8BB105	1u SMD Capacitor	0.24	1.44
6	CC1206KKX7R9BB154	0.15u SMD Capacitor	0.22	1.32
50	CL31B475KAHNNNE	4.7u SMD Capacitor	0.07	3.66
5	Y14-5721755A	ADE9153A PCB		9.00
5	Y15-5721755A	DC-DC Converter PCB		2.00
10	RT1206BRD0750KL	50k SMD Resistor	0.50	5.00
10	RC1206FR-07280KL	280k SMD Resistor	0.04	0.37
6	ADE9153AACPZ	Single Phase Meter IC	10.04	60.24
			Total	$731.39

9. Project Schedules (EF)

Task Name	Duratio	Start	Finish	Predecess	Resource Names
SDP2 Implementation 2023	**106 days?**	**Mon 1/9/23**	**Mon 4/24/23**		
Revise Gantt Chart	14 days	**Mon 1/9/23**	Sun 1/22/23		Ethan Frese
Implement Project Design	**92 days?**	**Mon 1/9/23**	**Mon 4/10/23**		
Hardware Implementation	**43 days?**	**Mon 1/9/23**	**Tue 2/21/23**		
Breadboard Components	**14 days?**	**Mon 1/9/23**	**Sun 1/22/23**		
Current Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Madison Britton
Voltage Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Madison Britton
Power Factor Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese
Relays	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Joseph Garro,Madison Britton
AC/DC Conversion	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese
Layout and Generate PCB(s)	**14 days?**	**Mon 1/9/23**	**Sun 1/22/23**	**5**	
Current Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Madison Britton
Voltage Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Madison Britton
Power Factor Measurement	14 days	Mon 1/9/23	Sun 1/22/23		Madison Britton,Ethan Frese
Relays	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Joseph Garro,Madison Britton
AC/DC Conversion	14 days	Mon 1/9/23	Sun 1/22/23		Ethan Frese,Madison Britton
Assemble Hardware	**14 days**	**Mon 1/23/23**	**Sun 2/5/23**	**11**	
Outlet Measurement Subsystem	14 days	Mon 1/23/23	Sun 2/5/23		Ethan Frese,Madison Britton
Outlet Communication Subsystem	14 days	Mon 1/23/23	Sun 2/5/23		Kyrolos Melek,Joseph Garro
Hub Subsystem	14 days	Mon 1/23/23	Sun 2/5/23		Joseph Garro,Kyrolos Melek
Test Hardware	**7 days**	**Mon 2/6/23**	**Sun 2/12/23**	**17**	
Outlet Measurement Subsystem	7 days	Mon 2/6/23	Sun 2/12/23		Ethan Frese,Joseph Garro,Kyrolos Melek,Madison Britton
Outlet Communication Subsystem	7 days	Mon 2/6/23	Sun 2/12/23		Kyrolos Melek,Ethan Frese,Joseph Garro,Madison Britton
Hub Subsystem	7 days	Mon 2/6/23	Sun 2/12/23		Madison Britton,Ethan Frese,Joseph Garro,Kyrolos Melek
Revise Hardware	**7 days**	**Mon 2/13/23**	**Sun 2/19/23**	**21**	
Outlet Measurement Subsystem	7 days	Mon 2/13/23	Sun 2/19/23		Ethan Frese

Task Name	Duratio	Start	Finish	Predecess	Resource Names
Outlet Communication Subsystem	7 days	Mon 2/13/23	Sun 2/19/23		Kyrolos Melek,Joseph Garro
Hub Subsystem	7 days	Mon 2/13/23	Sun 2/19/23		Joseph Garro,Kyrolos Melek
MIDTERM: Demonstrate Hardware Subsystems	0 days	Tue 2/21/23	Tue 2/21/23		
Software Implementation	**43 days**	**Mon 1/9/23**	**Tue 2/21/23**		
Develop Software	**28 days**	**Mon 1/9/23**	**Sun 2/5/23**		
Phone Application	28 days	Mon 1/9/23	Sun 2/5/23		Kyrolos Melek
Measurement Program	28 days	Mon 1/9/23	Sun 2/5/23		Kyrolos Melek
Communications	28 days	Mon 1/9/23	Sun 2/5/23		Joseph Garro,Kyrolos Melek
Hub Functionality	28 days	Mon 1/9/23	Sun 2/5/23		Joseph Garro,Kyrolos Melek
Test Software	**28 days**	**Mon 1/9/23**	**Sun 2/5/23**		
Phone Application	28 days	Mon 1/9/23	Sun 2/5/23		Kyrolos Melek
Measurement Program	28 days	Mon 1/9/23	Sun 2/5/23		Kyrolos Melek
Communications	28 days	Mon 1/9/23	Sun 2/5/23		Joseph Garro,Kyrolos Melek
Hub Functionality	28 days	Mon 1/9/23	Sun 2/5/23		Joseph Garro,Kyrolos Melek
Revise Software	**14 days**	**Mon 2/6/23**	**Sun 2/19/23**	**31**	
Phone Application	14 days	Mon 2/6/23	Sun 2/19/23	31	Kyrolos Melek
Measurement Program	14 days	Mon 2/6/23	Sun 2/19/23	31	Kyrolos Melek
Communications	14 days	Mon 2/6/23	Sun 2/19/23	31	Joseph Garro,Kyrolos Melek
Hub Functionality	14 days	Mon 2/6/23	Sun 2/19/23	31	Joseph Garro,Kyrolos Melek
MIDTERM: Demonstrate Software Subsystems	0 days	Tue 2/21/23	Tue 2/21/23		
System Integration	**88 days**	**Mon 1/9/23**	**Thu 4/6/23**		
Assemble Complete System Integration	**14 days**	**Tue 2/21/23**	**Mon 3/6/23**	**46**	
Zigbee Communication	14 days	Tue 2/21/23	Mon 3/6/23	46	Joseph Garro,Kyrolos Melek
Wireless Communication	14 days	Tue 2/21/23	Mon 3/6/23	46	Joseph Garro,Kyrolos Melek
Measurement Sensor to Microcontroller	14 days	Tue 2/21/23	Mon 3/6/23	46	Joseph Garro
Test Complete System Integration	**7 days**	**Tue 3/7/23**	**Mon 3/13/23**	**48**	
Zigbee Communication	7 days	Tue 3/7/23	Mon 3/13/23	48	Joseph Garro,Kyrolos Melek
Wireless Communication	7 days	Tue 3/7/23	Mon 3/13/23	48	Joseph Garro,Kyrolos Melek
Measurement Sensor to Microcontroller	7 days	Tue 3/7/23	Mon 3/13/23	48	Joseph Garro
Revise Complete System Integration	**24 days**	**Tue 3/14/23**	**Thu 4/6/23**	**52**	
Zigbee Communication	24 days	Tue 3/14/23	Thu 4/6/23	52	Joseph Garro,Kyrolos Melek
Wireless Communication	24 days	Tue 3/14/23	Thu 4/6/23	52	Joseph Garro,Kyrolos Melek
Measurement Sensor to Microcontroller	24 days	Tue 3/14/23	Thu 4/6/23	52	Joseph Garro
Preliminary Demonstration of Complete System	4 days	Mon 1/9/23	Thu 1/12/23		
Develop Final Report	**106 days**	**Mon 1/9/23**	**Mon 4/24/23**		
Write Final Report	106 days	Mon 1/9/23	Mon 4/24/23		Ethan Frese,Joseph Garro,Kyrolos Melek,Madison Britton
Submit Final Report	0 days	Mon 4/24/23	Mon 4/24/23	62	Ethan Frese,Joseph Garro,Kyrolos Melek,Madison Britton
Spring Recess	7 days	Mon 3/20/23	Sun 3/26/23		
Project Demonstration and Presentation	0 days	Tue 4/25/23	Tue 4/25/23		

Figure 50: Gantt Chart Team Schedule

10. Conclusions and Recommendations (KM)

After a thorough analysis of the available technologies and existing protocols, there are several recommendations that can be made and conclusions to be drawn. The first recommendation, after an in-depth analysis of both ethernet over ac and ZigBee protocols, is to favor ZigBee when deciding on a networking/communication technology to interface between the Smart Outlets and hub. The tradeoffs of using ethernet over ac include possible significant interference/noise across home wiring, Smart Outlets operating on different circuits than the hub, and thus not being able to communicate with the hub or user application, and a limited distance of 300 meters across home wiring. ZigBee allows for communication across a greater range and allows for easier extensibility, while still isolating the Smart Outlet-Hub Network from the Hub-Phone application network. Additional recommendations include using a medium scale, networked embedded system for both the hub and outlets. Further areas of research may include creating a topographic map or network for the outlet-hub network, allowing hubs that are out of range of the hub to forward their information to a nearby outlet that is in range of the hub.

11. References

[1] P. Delforge, "Home Idle Load: Devices Wasting Huge Amounts of Electricity When Not in Active Use," Natural Resources Defense Council, 14 July 2015. [Online]. Available: https://www.nrdc.org/resources/home-idle-load-devices-wasting-huge-amounts-electricity-when-not-active-use. [Accessed 7 October 2022].

[2] A. Barbato, L. Borsani, A. Capone and S. Melzi, "Home Energy Saving through a User Profiling System Based on Wireless Sensors," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, New York City, Association for Computing Machinery, 2009, pp. 49-54.

[3] W. Wang, Z. Hicks and B. Campbell, "The Standby Energy in Smart Homes: Problems, Progress, & Potential," in *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, New York City, Association for Computing Machinery, 2019, pp. 346-347.

[4] P. Meehan, C. McArdle and S. Daniels, "An Efficient, Scalable Time-Frequency Method for Tracking Energy Usage of Domestic Appliances Using a Two-Step Classification Algorithm," *Energies*, vol. 7, no. 11, pp. 7041-7066, 2014.

[5] T. J. C. Sousa, V. Monteiro, J. S. Martins, M. J. Sepúlveda, A. Lima and J. L. Afonso, "Comparative Analysis of Power Electronics Topologies to Interface dc Homes with the Electrical ac Power Grid," in *International Conference on Smart Energy Systems and Technologies (SEST)*, Porto, 2019.

[6] B. Partheeban and B. J. Justin, "Simulation of AC-DC step up and step down converter," *International Conference on Emerging Trends in Electrical and Computer*, vol. 2, no. 9, pp. 285-290, 2011.

[7] O. Assare and M. Goudarzi, "Opportunities for embedded software power reductions," in *2011 24th Canadian Conference on Electrical and Computer Engineering (CCECE)*, Niagara Falls, IEEE, 2011, pp. 763-766.

[8] M. T. Naing, T. T. Khaing and A. H. Maw, "Evaluation of TCP and UDP Traffic over Software-Defined Networking," in *International Conference on Advanced Information Technologies (ICAIT)*, Yangon, 2019.

[9] A. Colon and A. Moscaritolo, "PCMAG," 8 February 2022. [Online]. Available: https://www.pcmag.com/picks/the-best-smart-plugs-and-power-strips. [Accessed 7 October 2022].

[10] G. Masters and M. Pernia, "Smart electrical wire-devices and premises power management system". United States of America Patent US8487634B2, 25 September 2009.

[11] P. Rada and J. Magnasco, "Energy usage monitoring with remote display and automatic detection of appliance including graphical user interface". United States of America Patent US8447541B2, 21 May 2013.

[12] C. Steenberg and N. V. Duijn, "System and method to monitor and manage performance of appliances". United States of America Patent US8649987B2, 7 May 2008.

[13] E. Griffith, "How to Measure Home Power Usage," PCMag, 6 April 2020. [Online]. Available: https://www.pcmag.com/news/how-to-measure-home-power-usage. [Accessed 7 October 2022].

[14] T. Thiele, "Wiring for Split-Wire or Split-Feed Outlets," The Spruce, 21 November 2021. [Online]. Available: https://www.thespruce.com/what-is-a-split-outlet-1152347. [Accessed 7 October 2022].

[15] "AC to DC Conversion," [Online]. Available: https://www.instructables.com/AC-to-DC-Conversion/.

[16] SurtrTech, "hackster.io," 9 April 2020. [Online]. Available: https://www.hackster.io/SurtrTech/measure-ac-voltage-with-zmpt101b-and-esp8266-12e-24e367. [Accessed 7 October 2022].

[17] H. Latchman, S. Katar, L. I. Yonge and S. Gavette, "HomePlug Green PHY," in *Homeplug AV and IEEE 1901*, Hoboken, John Wiley & Sons, Ltd, 2013, pp. 302-311.

[18] Fluke, "Electrical Noise and Transients," [Online]. Available: https://www.fluke.com/en-us/learn/blog/power-quality/electrical-noise-and-transients. [Accessed 7 October 2022].

[19] FCC, "Interference with Radio, TV and Cordless Telephone Signals," Federal Communications Commission, 28 January 2020. [Online]. Available: https://www.fcc.gov/consumers/guides/interference-radio-tv-and-telephone-signals. [Accessed 7 October 2022].

[20] T. Higgins, "How To Troubleshoot Your Powerline Network," 6 July 2015. [Online]. Available: https://www.smallnetbuilder.com/basics/lanwan-basics/32769-how-to-troubleshoot-your-powerline-network. [Accessed 7 October 2022].

[21] M. Jiménez, R. Palomera and I. Couvertier, Introduction to Embedded Systems, Springer, 2014.

[22] K. Jones, "How Far Can Powerline Adapters Reach?," techreviewer, 28 December 2021. [Online]. Available: https://www.techreviewer.com/tech-answers/how-far-do-powerline-adapters-reach/.

[23] Digi, "Zigbee Wireless Mesh Networking," Digi, [Online]. Available: https://www.digi.com/solutions/by-technology/zigbee-wireless-standard.

[24] R. Felch, "Understanding Zigbee and Wireless Mesh Networking," BLACK HILLS INFORMATION SECURITY, 27 August 2021. [Online]. Available: https://www.blackhillsinfosec.com/understanding-zigbee-and-wireless-mesh-networking/.

[25] Connectivity Standards Alliance, "zigbee," Connectivity Standards Alliance, [Online]. Available: https://csa-iot.org/all-solutions/zigbee/.

[26] J. F. Kurose and K. W. Ross, Computer networking: a top-down approach, Pearson, 2017.

[27] Solectro, "A Beginner's Guide to Using Relay Modules in Arduino Projects," 26 June 2020. [Online]. Available: https://solectroshop.com/en/blog/a-beginners-guide-to-using-relay-modules-in-arduino-projects-n28. [Accessed 7 October 2022].

[28] Galco, "Relays - How Relays Work," [Online]. Available: https://www.galco.com/comp/prod/relay.htm. [Accessed 7 October 2022].

[29] CertBlaster, dti Publishing Corp., [Online]. Available: https://www.certblaster.com/examnotes-for-network-plus-n10-007-1-1-explain-the-purposes-and-uses-of-ports-and-protocols-part-1-of-2/.

[30] R. Pramberger, "Data Link Layer," [Online]. Available: https://osi-model.com/data-link-layer/.

[31] Fortinet, Inc, "OSI Model," Fortinet, Inc, [Online]. Available: https://www.fortinet.com/resources/cyberglossary/osi-model.

[32] Sun Microsystems, Inc, "System Administration Guide, Volume 3," Palo Alto, 2000.

[33] Fluke Networks, "101 Series: Ethernet Back to Basics," Fluke Corporation. , 31 July 2019. [Online]. Available: https://www.flukenetworks.com/blog/cabling-chronicles/101-series-ethernet-back-basics.

[34] libelium, "Node Parameters," Libelium Comunicaciones Distribuidas S.L., [Online]. Available: https://development.libelium.com/zigbee-networking-guide/node-parameters.

[35] G. Mazzi, "Wireless communication protocol: Zigbee," EDALAB s.r.l., 3 May 2021. [Online]. Available: https://edalab.it/en/protocollo-di-comunicazione-wireless-zigbee-2/.

[36] Connectivity Standards Alliance , "Zigbee FAQ," Connectivity Standards Alliance , [Online]. Available: https://csa-iot.org/all-solutions/zigbee/zigbee-faq/.

[37] SILICON LABS, "AN1138: Zigbee Mesh Network".

[38] D. Landsberger, "What Is Network Segmentation and Why It Matters?," CompTIA, 23 October 2020. [Online]. Available: https://www.comptia.org/blog/security-awareness-training-network-segmentation.

[39] M. Gegick and S. Barnum, "Least Privilege," Cybersecurity and Infrastructure Security Agency, 10 May 2013. [Online]. Available: https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege#:~:text=The%20Principle%20of%20Least%20Privilege%20states%20that%20a%20subject%20should,should%20not%20have%20that%20right.

[40] Computer Security Resource Center, "cryptography," National Institute of Standards and Technology, [Online]. Available: https://csrc.nist.gov/glossary/term/cryptography.

[41] S. Jena, "Classification of Embedded Systems," GeeksforGeeks, 21 August 2020. [Online]. Available: https://www.geeksforgeeks.org/classification-of-embedded-systems/. [Accessed 1 October 2022].

[42] ESPRESSIF, "ESP-IDF FreeRTOS (SMP)ℑ," 2022. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/freertos-smp.html.

[43] Espressif, "FreeRTOS," 2022. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html#.

[44] Digi International Inc., "Comparison of transparent and API modes," 28 March 2022. [Online]. Available: https://www.digi.com/resources/documentation/Digidocs/90001942-13/concepts/c_xbee_comparing_at_api_modes.htm?TocPath=How%20XBee%20devices%20work%7CSerial%20communication%7C_____2.

[45] Digi International Inc., "Supported frames," 28 March 2022. [Online]. Available: https://www.digi.com/resources/documentation/Digidocs/90001942-

13/Default.htm#reference/r_supported_frames_zigbee.htm?TocPath=API%2520mode%257C_____3.

[46] Digi International Inc., "XBee/XBee-PRO® S2C Zigbee® RF Module User Guide," 2022.

[47] N. Lohmann, *json*, 2022.

[48] S. Farahani, "ZigBee Wireless Networks and Transceivers," Newnes, 2008.

12. Appendices

ESP32 WROOM-DA Data Sheet -

https://www.espressif.com/sites/default/files/documentation/esp32-wroom-da_datasheet_en.pdf

ESP32 DevKitC Guide

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html

ESP-IDF Programming Guide -

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html

XBEE S2C Pro Data Sheet -

https://www.digi.com/resources/documentation/digidocs/pdfs/90002002.pdf

PlatformIO Documentation - https://docs.platformio.org/en/stable/integration/ide/pioide.html

ESP32

2N3904 Data Sheet -

https://global.oup.com/us/companion.websites/fdscontent/uscompanion/us/pdf/microcircuits/students/bjt/2N3904-rohm.pdf

ADE9153A Data Sheet - https://www.analog.com/media/en/technical-documentation/data-sheets/ade9153a.pdf

ADE9153A Evaluation Kit User Guide - https://www.analog.com/media/en/technical-documentation/user-guides/ev-ade9153ashieldz-ug-1253.pdf

ADE9153A Technical Reference Manual - https://www.analog.com/media/en/technical-documentation/user-guides/ade9153a-technical-reference-manual-ug-1247.pdf

Digi XBee 3 RF Module Hardware Reference Manual -

https://www.digi.com/resources/documentation/digidocs/pdfs/90001543.pdf

ESP32-S3-WROOM-1U Datasheet - https://www.espressif.com/sites/default/files/documentation/esp32-s3-wroom-1_wroom-1u_datasheet_en.pdf

ESP=Prog Documentation - https://espressif-docs.readthedocs-hosted.com/projects/espressif-esp-iot-solution/en/latest/hw-reference/ESP-Prog_guide.html