

Patrick McCullough, Joseph Kosteck, Adam Rainey

# Table of Contents

Table of Contents 1

3. Introduction 3

4. Detailed System Requirements 7

5. Detailed Project Description 9

5.1 System Theory of Operation 9

5.2 System Block Diagram 9

5.3 Detailed Design/Operation of the Mobile Application Subsystem 10

5.4 Detailed Design/Operation of the WiFi Module Subsystem 19

5.5 Detailed Design/Operation of the Physical Build Subsystem 25

6. System Integration Testing 34

6.1 Subsystem Integration Testing 34

6.2 Demonstration of Overall System Requirements 35

7. User Manual 37

7.1 Installation 37

7.2 Setup 37

7.3 Is My Product Working? 41

7.4 Troubleshooting 42

8. To-Market Design Changes 43

9. Conclusions 44

10. Appendices 45

10.1 Power Board 45

10.2 Microboard 47

10.3 Overall Hardware Design 48

10.4 Mobile Application Code 51

10.5 Ardunio ESP Code 58

# 3. Introduction

When we first contemplated ideas for our Senior Design project the only prerequisites that we had to keep in mind was that our device had to utilize the internet of things (IOT). We would be using an ESP8266 Wi-Fi module to enable the improvement of a certain device used in day-to-day life. During the time in which we were tasked with deciding on a project the campus was in the midst an energy competition in order to see which dorm could reduce its power consumption the most. In order to reduce energy usage there were posters throughout campus that advocated for the turning off of electronics and lights when they were not in use. Many of us would like to save energy for reasons pertaining to the environment and the wallet. This waste of energy, which is coined ‘vampire energy,’ amounts to roughly $19 billion per year. In fact, the International Energy Agency (IEA) estimates that on average up to 10% of a household’s utility bill consists of vampire energy. The sheer amount of wasted energy posses as both an economic and environmental problem that people have proposed a plethora of solutions to. Since there was naturally a problem we figured that we might as well try to find some way to fix it.

When approaching the question of why people do not unplug more of their electronics or turn off more devices the answer usually involves convenience and forgetfulness. Often people are in a hurry when they leave their home at the beginning of the day and do not have time to even think about unplugging electronic devices. With this in mind we realized that the greatest benefit of the internet of things was the added convenience that it allowed for the remote monitorization and storage of data, which could be applied to the problem at hand. Additionally we accounted for the fact that many households plug many high-energy usage devices into the same power strip or surge protector. At this point the solution was self-evident in that we could attach the internet of things to a surge protector and allow consumers to control a surge protector from their smartphone. This would allow for two advantages over a traditional surge protector, the first being the remote control of a surge protector rather than having to manually switch the surge protector on or off. The second advantage is that one can also set a predetermined time for the surge protector to be turned on or off in a manner in which one could set it and forget it.

In order to completely address the problem of wasted energy we strove to satisfy a few requirements: the ability to monitor power usage, the ability to remotely switch the power on and off, the ability to manually switch power on and off, the ability to set the power on and off at a future time, and the ability to monitor the current status of outlets. Essential to the solution would be the use of a device that has become ubiquitous in the modern world, the smart phone. Today nearly everyone has one, which would allow for a significantly potential usage, as opposed to the requirement of an additional device one would simply need to download and application on their phone; after all, the most important requirement for the solution to the problem would ease of use. In order for someone to be able to remotely control the device at any practical distance we would need to use an MQTT server to connect the mobile application and the ESP8266 Wi-Fi device. Likewise, the ESP device would connect to the server and would exchange messages with the user via the mobile application and server. We would also need a manual override of the mobile application so that if someone needed to, they could push a button on the surge protector to turn it on and off when needed, and thus would not be overly reliant on the mobile application.

While we did set out to satisfy every one of the aforementioned requirements, we fell short in a few due to hardware failures and a lack of programming expertise. When testing the device during our initial implementations we experience a few errors. We burned out two of the Wi-Fi module chips while attempting to fully integrate them. This made it more difficult to have a comprehensive final produce as we had to demonstrate with the ESP8266 device that we had developed our first programs with throughout the semester and was not suppose to be part of our final project design. Unfortunately we also destroyed a few other devices while trying to integrate our final project: both current sensors and a relay. I suppose these are the risks when working with an extremely high voltage, 120 V, and two other voltages, 3.3 V and 5 V. We would like to think that these mistakes are very common but at the end of the day they did remove a certain luster from our final project. For example we were only able to control one of the ports on our final product because we only had one working relay. This took away much of the functionality and convenience of being able to control two ports. One outlet was designed to be connected to devices that the consumer would almost always want to keep plugged in such as alarm clocks and cable boxes, while the other was intended for devices that could be removed from power when not in use such as televisions and game consoles. Likewise, with the two blown current sensors our final product would no longer be able to have the ability to monitor power usage. The fact that we would no longer be allowed to have the user to monitor power usage seems to take away some of the advantages of having a smart surge protector because it no longer gives the user a means of monitoring their power usage and determine the extent of their reduction in power usage. This, perhaps, would make the user less likely to be motivated to use the device to save money and power as it would be more difficult for the user to gauge the extent of their savings. Finally, we were unable to fully implement a calendar function in which the user would be able to set times and dates in which he or she wants to turn the device on or off based on their weekly schedule. If we had been able to do this the user would have been able to have the device turn off automatically when he or she knew there would be no use for devices that were plugged into it, such as during his or her work hours during the work week. An ability such as this would have enabled the user to automatically see energy savings without having to enable them on a daily basis. We were unable to achieve this due to the lack of a calendar function and the sheer complexity of implementing such a function on the fly with the irregular lengths of months, daylight savings, leap years and so on. We were only able to find a way of finding the seconds from a specified date (January 1, 1900 or 1970). We were unable to find a surefire way of determining the given day of the week, the given month, or even given year due to the complications of the Gregorian calendar. Nonetheless we were able to compromise with a timing function in which the user would set a time and date in which her or she wants the device to turn on or off.

In the end we were proud of what we achieved. We had every intention of fulfilling all of the initial requirements but the resources and time at our disposal proved too great in the face of the difficulties that we faced when trying to integrate the subsystems of our final design.

# 4. Detailed System Requirements

Our initial intentions were to meet the goals as mentioned previously: the ability to monitor power usage, the ability to remotely switch the power on and off, the ability to manually switch power on and off, the ability to set the power on and off at a future time, and the ability to monitor the current status of outlets. In order to meet these design specifications from a hardware perspective we need to be able to measure current, read the current state of outlets, switch outlets on and off, and fully maintain power to the Wi-Fi module and the microcontroller. In order to measure current we planned on using current sensors and using the established resistance to convert that value to power. To control the flow of power to a given outlet we planned on using relays. These relays would be able to control the high voltage of 120 V by using the comparatively small input from the microcontroller. Additionally be reading the voltage applied to the relays we would easily be able to determine the active state of each of the outlets. Obviously we would also require wires that would be able to handle the very higher voltage of 120 V. Likewise we would also need to supply housing for the entire systems in addition to the outlets themselves. We planned on building all of these from other materials as opposed to taking an existing surge protector and altering it.

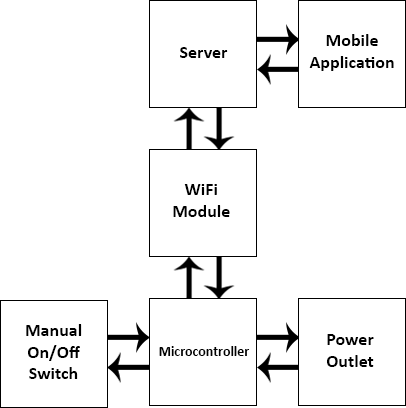
On the user side, with the application, it must be simple and easy to navigate. Initially the user must be able to program the settings for the desired wireless network that the outlet will use. This can achieved by setting the Wi-Fi device as an access point in which the user can log into and select the preferred wireless network as well as its password. This username and password of the network will then be saved by the Wi-Fi device until it can no longer connect to the desired network at which point it will then become an access point again for the user to enter their desired settings again. The main function of the application must to allow the user to turn the outlets on the surge protector on and off in a clear way while also showing the current state of each outlet. In this sense the mobile application must send signals to the outlet and received feedback of the current state of the outlets. In order to allow the user to make informed decisions in turning the surge protector on or off; therefore, he or she must be able to determine the current state of each of the outlets. Likewise, the user must be able to easily switch on or off either one or both of the outlets easily and quickly with immediate feedback of the state of the system. Additionally we must be able to allow the user to easily view the current that each of the outlets on the device is consuming so that they can have an idea of how much energy they are using at any given point, with the aim of giving the user motivation to fully utilize the device and save energy. Finally, in order to make the use of the device as convenient as possible, we must allow for the user to set times in which the device will automatically turn on or off. By doing this the device can be synchronized with the user’s schedule and can be further utilized.

# 5. Detailed Project Description

## 5.1 System Theory of Operation

From the mobile application standpoint it must send and receive messages from the server. The messages to the server are to tell the WiFi module to turn all the outlets on or off, one of them on or off, requesting the status of one of the outlets, requesting the current reading for one of the outlets, setting a time and date for a future switch for both of the outlets, and setting the future on or off value.

## 5.2 System Block Diagram



The overall system is divided into three distinct subsystems. The first subsystem consists of the mobile application, the primary way in which the user interacts with the product. The first subsystem is connected to the second subsystem through the MQTT server. The second subsystem consists entirely as the WiFi module. The second subsystem communicates to the first subsystem through the server. The third subsystem consists of the purely hardware part of our system. It includes the microcontroller, power outlet, and the manual on/off switch.

## 5.3 Detailed Design/Operation of the Mobile Application Subsystem

Subsystem requirements:

* Ability to wirelessly connect to server
* Use of MQTT IoT protocol
* Ability to function on an is mobile device
* Ability to publish information to a specific topic
* Ability to subscribe to a topic and receive information that is published to it.
* Turn on/off physical device
* Set specific times regarding when the device should be turned on or off
* Monitor power consumption out of both outlet boxes

**Login Page Flowchart**

Disconnect from Server

Correct username/password

Incorrect username/password

Login Page

Configure WiFi

Enter username and password

Go to Settings

Back to login page

Server Connection

Connect to Server

Home

Home Page

**Home Page Flowchart**

Login Page

Power On/Off Device Page

Schedule Page

If Pressed

If Pressed

If Pressed

If Pressed

Schedule

Power On/Off

Disconnect

Energy Consumption

Home Page

Energy Consumption Page

\*If no buttons are pressed, user will remain on the Home Page

**Energy Consumption Page Flowchart**

Idle until user presses a button

Display result in energy label

Display result in power label

Display result in right current label

Run calculateEnergy function

Home Page

If “Back” pressed

Subscribe to energy topic

Subscribe to power topic

Subscribe to right relay topic

Subscribe to left relay topic

Run calculatePower function

If “Energy Saved” pressed

If “Power Used” pressed

If “Relay 1 Current” pressed

Energy Consumption Page

If “Relay 2 Current” pressed

Publish “LeftCurrent” to topic “LeftOutlet”

Publish “RightCurrent” to topic “RightOutlet”

Display result in left current label

**Schedule Page Flowchart**

Home Page

If “Back” pressed

Schedule Page

User picks “ON” date

User picks “OFF” date

Variable stored as selectedDate

Variable stored as selectedOffDate

Publish variable to topic “SmartSurge”, publish “OFF” to topic “Schedule”

Publish variable to topic “SmartSurge”, publish “ON” to topic “Schedule”

If nothing pressed

If “Submit” pressed

If “Submit” pressed

Relay 2 Label “ON”

Relay 2 Label “OFF”

All labels “ON”

All labels “OFF”

Publish “LeftState” to topic “LeftOutlet”

Publish “State” to topic “All”

“Current Status of Relay 1”

Power On/Off Device Page

Publish “Off” to topic “All”

**Power On/Off Device Page Flowchart**

“Current Status of SmartSurge”

“Turn Relay 1 OFF”

“Turn Relay 1 ON”

“Current Status of Relay 2”

Publish “LeftOff” to topic “LeftOutlet”

Publish “LeftOn” to topic “LeftOutlet”

Publish “RightState” to topic “RightOutlet”

Relay 1 Label “OFF”

Relay 1 Label “ON”

Update respective “ON/OFF” labels

“Turn Relay 2 OFF”

“Turn Device ON”

“Turn Device OFF”

“Turn Relay 2 ON”

Publish “On” to topic “All”

Publish “RightOn” to topic “RightOutlet”

Publish “RightOff” to topic “RightOutlet”

**Subsystem Requirements:**

The mobile application must do a number of different tasks in order to achieve complete functionality. To start, the application must run on an iOS supported mobile device. Therefore, it was written in a language called Swift 2.0 and compiled in a program called XCode on a MacBook. After the physical layout of the application was created, a number of different software requirements needed to be implemented. The most important piece of software used in our project was the MQTT client, CocoaMQTT. This allowed the application to essentially perform all of its necessary requirements when communicating with the physical device. This includes having the ability to publish information to a specific topic within a host, as well as the ability to subscribe to a topic and receive information that is published to it. Also, the application needed to be able to wirelessly connect to a specific WiFi network, but this is covered in the following subsystem.

When both the physical layout and software implementation were complete, the application had three main requirements in order to make our project functional. The user needed the ability to turn on/off the physical device wirelessly through the mobile application. This was achieved by asking the user to press a button, publishing a specific message to a specific topic pertaining to the power status of the desired relay, and then updating the power status label that the user sees on the application. Also, the user needed to be able to set specific times regarding when the device should be turned on or off. This was achieved through the same type of routine as the first requirement. The user picks a specific date and time, presses the submit button on the application, and the message is sent via the MQTT client to a specific topic. The board reads this message and stores it in its local memory, so the user is able to close out of the application without losing the desired on/off time. Lastly, the application needed to be able to monitor power consumption of both outlet boxes. Once again, using the MQTT client, messages were published to individual topics asking for specific current values, and the board published the results back to the application. These results were then displayed to the user in the static label fields next to each button on the “Energy Consumption” page. Below are the five flowcharts for the mobile application – each one displays the functionality of a specific page within the application to increase clarity.

The reason why an iOS application was chosen over an Android application was simply because all of the group members owned iPhones, so it was much easier to share the application with different group members due to its compatibility with iPhones. Also, the group member in charge of building the application owned a MacBook, which significantly facilitated his coding experience and made the possibility of writing an Android application unfeasible.

Software Description:

All of the code written for the mobile application is in Swift 2.0. The majority of its functions involve publishing information to a specific topic within a client and subscribing to specific topics in order to receive information. The skeleton code that was used was from the CocoaMQTT client installed as a ‘Pod’ within the application, which is essentially a dependency manager to make scaling the project easier. Please see the Appendix for the complete listing of the SmartSurge application’s code.

**Communication Protocols Flowchart:**

User Publishes to a topic

User enters correct SSID/Password

Host server

Application

App subscribes to topic that comms board publishes to

Comms Board on Physical Device

MQTT Client

User disconnects from server

Sends board info to host w/ topic

User publishes to a topic

Server sends comms board published info to app-subscribed topic

User publishes to a topic

Comms board sends info corresponding to topic

**Subsystem Testing:**

An MQTT client was downloaded for all of the application testing. The client was called Mosquitto, and it was run through a terminal window on one of the group members’ laptops. The following figure shows the specific client subscribed to the topic “SmartSurge”.

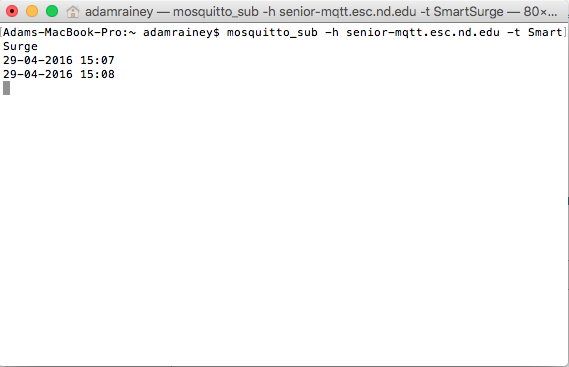


Figure 1: Mosquitto subscribed to topic ‘SmartSurge’

As seen in the figure, the message “29-04-2016 15:07” was published to this topic. From this, our group can conclude that the application is functioning correctly since it successfully published the desired message. A similar test was performed for all topics that were published/subscribed to in order to determine the application’s full functionality.

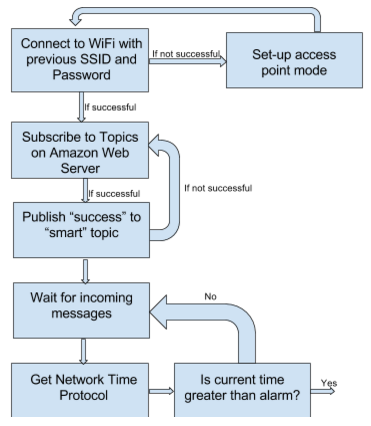
The most important tests that were conducted involved publishing information directly to a page on the application. Figuring out how to publish to a topic was the trivial part, so constructing the code that successfully allowed messages to go to the application took a few steps. First, a message was published to a topic within a specific host. After a number of trial runs within the code, it was determined that the ‘message receive’ function needed to go inside a function within the CocoaMQTT framework code. The necessary commands were added under the ‘didReceiveMessage’ function in the MQTT delegate, and this code was tested using the mobile application. The published message was connected to its corresponding label in the application, so whenever the comms board published a message to a certain topic, the application would read this message and then display it as a label. The application development went very smoothly after that point, since all of the necessary commands were known and it was simply a matter of where to put other commands and what they should do.

## 5.4 Detailed Design/Operation of the WiFi Module Subsystem

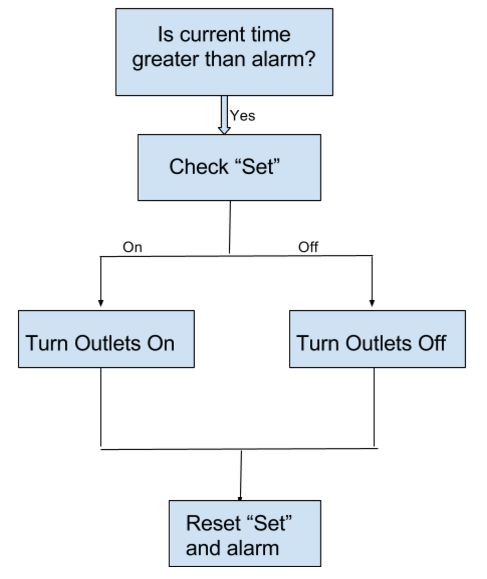
Subsystem requirements:

* Must connect to local WiFi network with saved SSID and password
  + If not, set-up an access point to acquire SSID and password
* Connect and subscribe to relevant topics on amazon server
* Communicate successful connection to server
* Must actively wait for incoming messages
* Must be able to get current time in network time protocol
* Read the current value of each outlet on demand
* Turn the outlets on or off on demand
* Determine the current state of each outlet on demand
* Publish the state and current draw of each outlet to the server
* Store and compare future time value to the current time

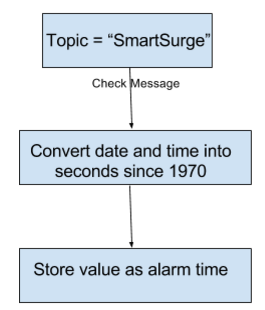
**Initial Setup Flowchart:**



The first step is the program attempts to connect to the previous WiFi settings. If this SSID and password does not work then the device will go into access point mode. In access point mode the user is able to connect with their phone or computer to the device and set their desired SSID and WiFi settings. Once the device successfully connects to the wireless network it attempts to subscribe. If it does not successfully connect to the server it will attempt to keep trying, once connected it will publish “success” to the “smart” topic. Once connected to the server it will subscribe to the relevant topics: “RightOutlet”, “LeftOutlet”, “All”, “SmartSurge”, and “Schedule”. The program will then check for current messages and get the current time via network time protocol. Then the program will compare this time to see if an scheduled time has passed. If a scheduled time has not passed then it will continue this loop of checking messages. If a scheduled time has pass then the program will enter the following procedure.

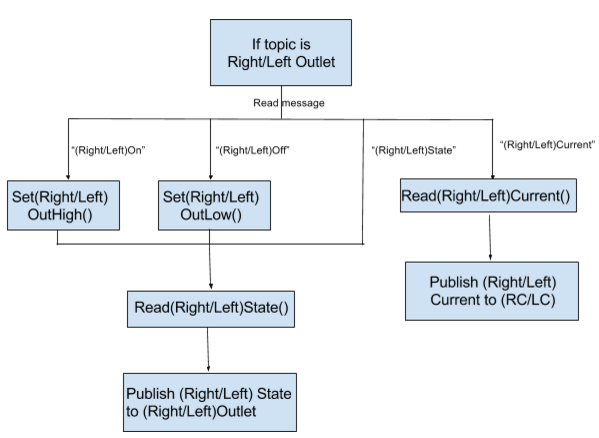


Essentially the program then checks Set to see if the alarm is for the outlets to be turned on or off. The program then either turns the outlets on or off, depending on this command, and resets “Set” and the alarm time. “Set” is reset by setting it equal to “NotSet” and the alarm time is reset by making it equal to zero. The process of setting the schedule value is shown in the process below.



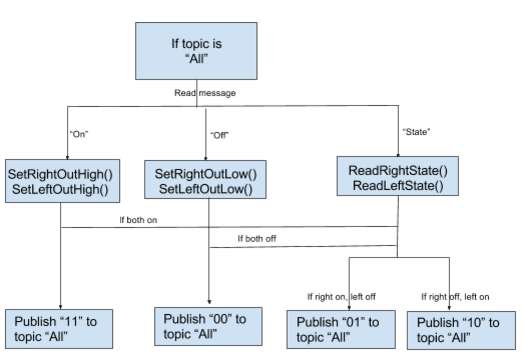
If the topic is “SmartSurge” then the program will read the message as a DD-MM-YYY hh:mm date time format, in military time. The program then converts this time to seconds since 1970 in order to make it compatible with the network time protocol time. Finally, this value is stored as the alarm time so that it can be compared to the current time.

**Right/Left Outlet Topics Flowchart:**

****

If the topic is either “RightOutlet” or “LeftOutlet” then the program will read the message and initiate the next step. If the message is “RightOn” or “LeftOn” then the program will initiate the function “SetRightOutHigh()” or “SetLeftOutHigh()” and turn the right or left outlet on, respectively. If the message is “SetRightOutLow” or “SetLeftOutLow” then the program will turn the right or left outlet off, respectively. In both cases the program will then read the state of either the right or left outlet, depending on the previous message. With this message the program will then publish the current state of the status to the corresponding topic, either RightOutlet or LeftOutlet. If the initial message is “RightState” or “LeftState” the program will jump immediately to reading the state of either the right or left outlet and publish it to the corresponding topic. If the initial message is “RightCurrent” or “LeftCurrent” then the program will initial the function ReadRightCurrent or ReadLeftCurrent and publish the corresponding value to the topic RC or LC, respectively.

**All Topic Flowchart:**

****

If the topic is all then the resulting messages that the program will respond to are “On”, “Off”, and “State”. If the message is “On” then the program will set both outlets high and publish “11” to “All”, to confirm that both the right and left outlets are on. Likewise, if the message is “Off” then the progam will set both outlets to off and send “00” to “All”, confirming that both outlets are off. If “State” is the message then the program will read the states of both outlets. The program will then send a message to “All”, 11 if both are on, 00 if both are off, 01 if the right outlet is on and the left outlet is off, and 10 if the oppossite is true.

**Subsystem Testing:**

In order to test the functionality of this subsystem a few important steps were required. The first step was to initially connect to WiFi which we were able to accomplish in January with a scripted program. Then we were able to use the ESP device as an access point so that we could change the WiFi settings on the device if they were not working through the use of a smartphone. This program automatically saves the entered settings and tries to connect to them on startup, only if a successful connection is not established does the device resort to establishing itself as an access point. To test that the device could receive and send messages we used an MQTT client to view messages that the ESP device had published once it had succesfully connected to the server. Once a connection had been established the MQTT client published messages to the server and provoked the ESP device to publish its own message back to the server. Once we had established that these fundamental aspects of the subsystem worked we were able to integrate it with the two other subsystems.

## 5.5 Detailed Design/Operation of the Physical Build Subsystem

System Requirements

The physical design subsystem contains all the hardware components of the device as well as local functions to achieve the desired effect from the user input. The physical devices interfaces with the other subsystems through software on the ESP12 microcontroller. The first requirement of this subsystem is to provide power to the overall device, including the microcontroller and the outlets. In Figure 5.5.3 below, one can see that 120 Vac power is directly connected to the outlets and the Power Board. This provides the normal voltage used for everyday devices and allows voltage to be regulated on the Power Board for use with the microcontroller. On the Power Board in Figure 5.5.2, a VSK-S3-5U AC-DC Power Supply is used to output a +5 Vdc voltage from the 120 Vac input. This model of power supply was used because it provides continuous power, is quite efficient, and a compact board mount design. This +5 Vdc voltage is then sent to the Micro Board, as shown in Figure 5.5.3. Here, a TPS61201 is used to convert power from +5 Vdc to +3.3 Vdc, as shown in Figure 5.5.1. Similarly, this converter was used because it is quite efficient and can source enough current to supply to microcontroller.

Next, the physical design must be able to control toggle power in the outlets via a user input from the mobile application or the pushbutton switch. This is accomplished by applying a +5 Vdc to each relay (G3NE-210T) to turn the outlets on and 0 Vdc to turn the outlets off. This brand of relays were chosen because they are solid-state relays, meaning they have near-zero power consumption and are rated for up to 10 amps. Because there are two relays, the outlets can be controlled independently of each other. To provide this voltage to toggle the relays, the microcontroller outputs a digital high (+3.3 Vdc) or low (0 Vdc) from GPIO 16 or 13, as shown in Figure 5.5.1. In order to reach the +5 Vdc operating level, an op-amp (TLV272) is used to boost the voltage to the 5 Vdc level. A non-inverting amplifier configuration with a gain of 2 is used. The signal clips at the op-amp supply voltage of +5 Vdc, providing the needed voltage to control the op-amps. The pushbutton can also toggle to the relays by triggering a falling edge interrupt on GPIO12. GPIO12 has a pull-up resistor it, so that when the pushbutton is pressed, it will go from high-to-low. Below are the function to toggle power on/off for each relay, as well as the set up for the pushbutton interrupt.

Commands to Toggle Power:

#define leftRELAY 16

#define rightRELAY 13

void setLeftOutHigh()

{

digitalWrite(leftRELAY, HIGH);

}

void setRightOutHigh()

{

digitalWrite(rightRELAY, HIGH);

}

void setLeftOutLow()

{

digitalWrite(leftRELAY, LOW);

}

void setRightOutLow()

{

digitalWrite(rightRELAY, LOW);

}

Commands to enable interrupts:

//Must be included in the setup statement

attachInterrupt(12, intrChangeOut, FALLING);    //Set interrupt to trigger on falling edge

//Interrupt routine for the push button

void intrChangeOut() {                //Interrupt toggle on Pin 12

  int r1 = digitalRead(leftRELAY);    //Obtain High/Low status of relays

  int r2 = digitalRead(rightRELAY);

  if (r1 & r2 == 0) {                 //If both relays are off, turn power on

    digitalWrite(leftRELAY, HIGH);

    digitalWrite(rightRELAY, HIGH);

  }

  else {                              //Else, turn both relays off

    digitalWrite(rightRELAY, LOW);

    digitalWrite(rightRELAY, LOW);

  }

  delayMicroseconds(100000);                        //Slight delay for bounce

}

In addition to the above, the Physical Design must also be able to automatically turn power on and off via a schedule inputted by the user. Although this functionality was not fully realized due to limitations on scheduling in the mobile application, local functions were written to intake various Epoch (seconds since January 1st, 1970) times with instructions to turn the relays on/off and store these times in the microcontroller’s EEPROM memory. These functions are listed below:

Commands to store/check schedule times:

EEPROM.begin(512); //Must be included in the setup function

void EEPROMWritelong(int address, unsigned long value)

{

//Decomposition from a long to 4 bytes by using bitshift.

//One = Most significant -> Four = Least significant byte

byte four = (value & 0xFF);

byte three = ((value >> 8) & 0xFF);

byte two = ((value >> 16) & 0xFF);

byte one = ((value >> 24) & 0xFF);

//Write the 4 bytes into the eeprom memory.

EEPROM.write(address, four);

EEPROM.write(address + 1, three);

EEPROM.write(address + 2, two);

EEPROM.write(address + 3, one);

}

unsigned long EEPROMReadlong(int address)

{

//Read the 4 bytes from the eeprom memory.

unsigned long four = EEPROM.read(address);

unsigned long three = EEPROM.read(address + 1);

unsigned long two = EEPROM.read(address + 2);

unsigned long one = EEPROM.read(address + 3);

//Return the recomposed long by using bitshift.

return ((four << 0) & 0xFF) + ((three << 8) & 0xFFFF) + ((two << 16) & 0xFFFFFF) + ((one << 24) & 0xFFFFFFFF);

}

void setCalender(String onoffTimeStr, int\* scheduleMemoryAddressPNT, byte relayState) {

unsigned long onoffTime = onoffTimeStr.toInt();

int scheduleMemoryAddress = \*scheduleMemoryAddressPNT;

if (scheduleMemoryAddress >= 250){

scheduleMemoryAddress = 150;

}

EEPROMWritelong(scheduleMemoryAddress, onoffTime);

EEPROM.write(scheduleMemoryAddress + 4, relayState);

EEPROM.commit();

\*scheduleMemoryAddressPNT = scheduleMemoryAddress + 5;

}

void checkCalender(unsigned long NTPtime) {

byte relayState;

for (int i = 150; i <= 250; i = i + 5) {

unsigned long onoffTime = EEPROMReadlong(i);

unsigned long comp = abs(onoffTime - NTPtime);

if (comp <= 120) {

relayState = EEPROM.read(i + 4);

switch (relayState) {

case 0: //Set Both outputs low

setLeftOutLow();

setRightOutLow();

break;

case 1: //Set left outlet low, right high

setLeftOutLow();

setRightOutHigh();

break;

case 2: //Set left outlet high, right low

setLeftOutHigh();

setRightOutLow();

break;

case 3: //Set both outlets high

setLeftOutHigh();

setRightOutHigh();

break;

}

}

}

}

Furthermore, the device must be able to read data from current sensors in order to estimate the amount of power the user is consuming. Current is fed through the sensors (ACS759) as shown on the Power Board schematic in Figure 5.5.2. These sensors were chosen because they operate at +3.3 Vdc with very low power consumption with a high degree of accuracy. The +3.3 Vdc supply voltage received from the Micro Board, as shown in Figure YY. The output from each current sensor is fed to an external analog-to-digital converter (ADS1015) as shown in Figure 5.5.2 on the Power Board. This external ADC was utilized the ESP12 lacks multiple internal ADCs and can only operate at a maximum of +1 Vdc, while the ACS759 has a nominal output of +1.65 Vdc. This ADC communicated with the microcontroller via I2C, as shown in Figures 5.5.1 and 5.5.3 below. Because the current through these sensors is alternating, software is utilized to sample the current an ample amount to achieve a true maximum. This can then be converted the IRMS value, which can be sent to the mobile application to be estimate energy consumption. Additionally, commands were written to store historical current consumption every 5 minutes in EEPROM, although these functions were unable to be implemented in the finished product. The functions to measure current and store/read that data are some below. Note that an external Adafruit ADS1015 library was utilized to accomplish I2C interfacing with the ADC.

Commands to read, store, and measure current:

//Define pins by their function

#define SCL 4

#define SDA 5

#define leftRELAY 16

#define rightRELAY 13

Adafruit\_ADS1015 ads1015

//These must be included in the setup statement

Wire.begin(SDA, SCL);         //Initialize I2C on SDA & SCL pins

ads1015.begin();              //Initialize ads1015 operation

ads1015.setGain(GAIN\_TWO);    //Set ads gain to 2, 1 mV/bit

//Read the current through the left outlets

float readLeftCurrent(int samples)  //Input is number of samples

{

  float adc0;         //Declare Variables

  float data\_corr;

  float data\_raw = 0;;

  int i;

  // For Loop measuring the current for the specified number of samples

  for (i = 0; i <= samples - 1; i = i + 1) {

    adc0 = abs(ads1015.readADC\_SingleEnded(0)); //adc1 is number from right outlet

    data\_raw = max(adc0, data\_raw);             //Ensure maximum value is found

  }

  data\_corr = (data\_raw - 1650) / 26.4 / 1.41421; //Gain = 2, 1 mV/bit, 26.4 mV/A, sqrt(2) for RMS value

  return data\_corr;

}

//Read the current through the right outlets

float readRightCurrent(int samples) //Input is number of samples

{

  float adc1;           //Declare Variables

  float data\_corr;

  float data\_raw = 0;;

  int i;

  // For Loop measuing the current for the specified number of samples

  for (i = 0; i <= samples - 1; i = i + 1) {

    adc1 = abs(ads1015.readADC\_SingleEnded(1));   //adc1 is number from right outlet

    data\_raw = max(adc1, data\_raw);               //Ensure maximum value is found

  }

  data\_corr = (data\_raw - 1650) / 26.4 / 1.41421; //Gain = 2, 1 mV/bit, 26.4 mV/A, sqrt(2) for RMS value

  return data\_corr;

}

unsigned long storeCurrentData(unsigned long timer, int\* currentMemoryAddressPNT) {

timehold = timer;

int currentMemoryAddress = \*currentMemoryAddressPNT;

float rightCur = readRightCurrent();

float leftCur = readLeftCurrent();

unsigned long NTPtime; //NEED TO OBTAIN VALUE OF NTP time

EEPROMWritelong(currentMemoryAddress, NTPtime);

EEPROM.write(currentMemoryAddress + 4, rightCur);

EEPROM.write(currentMemoryAddress + 5, leftCur);

EEPROM.commit();

if (currentMemoryAddress >= 110) {

currentMemoryAddress = 50;

}

else {

currentMemoryAddress = currentMemoryAddress + 6;

}

\*currentMemoryAddressPNT = currentMemoryAddress;

return timehold;

}

void readCurrentData() {

String rightCur, leftCur, measTime;

for (int i = 50; i >= 110; i = i + 6) {

measTime = EEPROMReadlong(i);

rightCur = EEPROM.read(i + 4);

leftCur = EEPROM.read(i + 5);

//SEND THIS TO THE APP

}

}

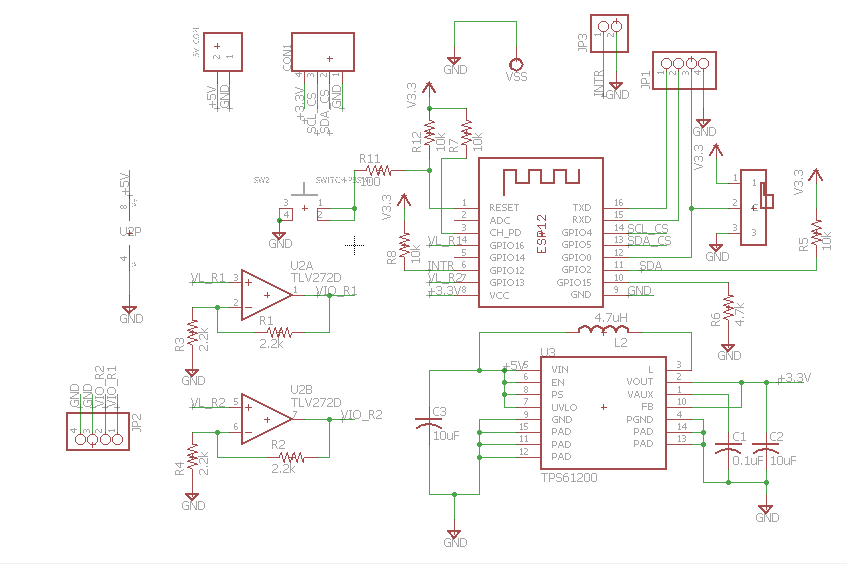


Figure 5.5.1: The Complete Schematic of the Micro Board

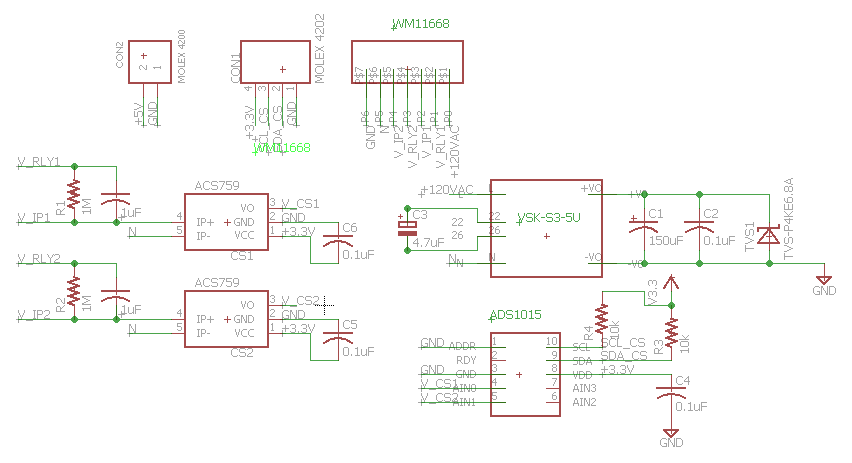


Figure 5.5.2: A Complete Schematic of the Power Board

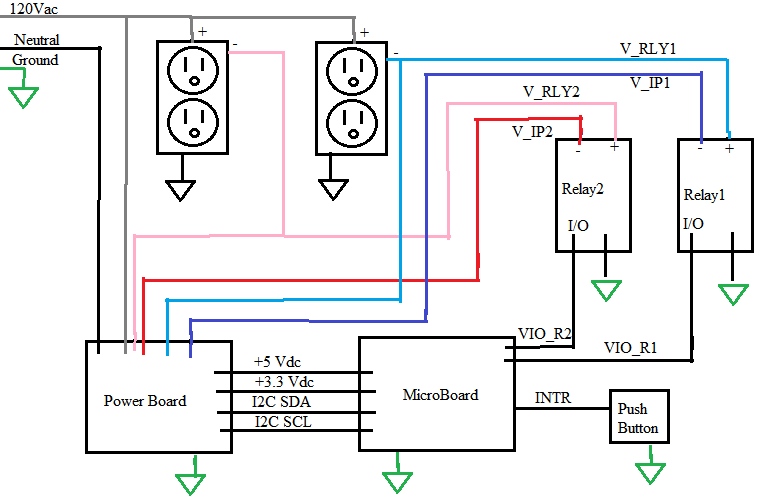


Figure 5.5.3: The Complete Schematic of the Physical Build

Subsystem Testing:

In order to ensure that this subsystem was properly designed, the subsystem was initially prototyped on a breadboard. The +5 Vdc voltage was supplied by an external power supply, while the +3.3 Vdc voltage was supplied by the microcontroller which itself was powered by a computer USB connection. An ADS1015 breakout board was used to replicate a board-mounted device. Using this setup, the local functions were able to be designed and tested. Once construction on our final device began, the Micro Board was the first built. Using an external power supply with +3.3 Vdc, it was first ensured that the microcontroller was connected properly and could be programmed. Next, a +5 Vdc supply was mimicked to ensure that the Micro Board could convert to a +3.3 Vdc. The Power Board was then soldered. After connecting the Power Board to the 120 Vac line voltage, it was ensured that the AC-DC power supply was outputting +5 Vdc before connecting this board to the Micro Board. Finally, with the functions working and ensuring good voltage levels, the entire device was connected.

# 6. System Integration Testing

## 6.1 Subsystem Integration Testing

The first two subsystems to be integrated were the remote Wi-Fi subsystem and the Physical Device subsystem; this was accomplished early in the semester.  Using a MQTT server and the ESP12 breakout board, we were able to verify connection of the Physical Device to the outside world by remotely sending a command to blink an LED. Using an MQTT client we send “On” and “Off” to a subscribed to topic on the server, which would correspond to setting the LED pin to either high or low on the ESP device. After this, we were able to add multiple topics on the MQTT server corresponding to each of the local functions found on the microcontroller. We methodically tested each function and ensured that everything would properly work by setting pins high or low and reading the appropriate values at any given pin. We would then have the ESP device send a proper response to the server to ensure that the values that the ESP device was properly picking up values and that those values were correct.

The final step in integrating the different subsystems was merging the WiFi and Mobile Application subsystems.  This was accomplished in the second week of April.  Using a specific topic within the MQTT client and the ESP12 breakout board, we were able to publish a message directly from the mobile application to a topic that the ESP12 breakout board was subscribed to and command the board to blink an LED.  To ensure functionality, we also sent commands from the application to turn the breakout board on and off.  Once this connection was established, the two subsystems collaborated to determine which specific topics the application should publish to in order to perform various functions on the microcontroller. To ensure that messages were properly received we also programmed the ESP device to respond to certain commands by publishing its own message to server. This is how we developed the idea of responding to commands to turn outlets on our off by reading the state of the outlet and publishing that state to a corresponding server topic in order to verify that the outlets had been turned on or off.

## 6.2 Demonstration of Overall System Requirements

The testing of the overall system took place after all subsystems were functional.  Initial tests showed that the phone was able to connect to an access point and maintain its connection until either WiFi was turned off or the phone left the network range. In order to test the overall subsystem, a lamp was plugged into one of the outlets in the SmartSurge in the ‘on’ position.  The mobile application was opened, connected to the server, and guided to the ‘Power On/Off’ page.  A user pressed the ‘Turn Relay 1 ON’ button, and one second later the lamp turned on. Furthermore, the current status of relay 1 label in the application successfully updated, which demonstrated application functionality. Likewise, further functionality was demonstrated by requesting the current state of the outlets through the mobile application. We first asked for the current state of the outlets when both of them were switch off, and we received the correct response in the application. We then tried it for all possible outlet state combinations and each time the application successfully updated.

    To test the scheduling functionality, the user sent a specific date and time to the server for the SmartSurge to turn on, and when that time was reached, the physical device did indeed power on without any manual toggle switch. We were able to ensure that the application did not need to be on a specific screen or that the phone needed to be unlocked to accomplish this as the scheduled time was stored on the physical device rather than on the mobile application. We were initially able to achieve accurate current readings of the current through the mobile application. However, as were working on ways to convert these current values into power values we somehow shorted the current sensors. While we were able to establish temporary functionality, unfortunately, we were unable to establish this function permanently.

# 7. User Manual

## 7.1 Installation

To install SmartSurge, simply plug it into any standard wall outlet. To install electrical devices to SmartSurge outlets, simply plug in the device to each outlet. IMPORTANT: SmartSurge is rated for 10 Amps, do not exceed this limit.

## 7.2 Setup

**Login Page:** (Figure 7.2.1)**:** The user will be brought to this login page once clicking on the SmartSurge1.0 application on an iPhone. This is strictly an iOS app, so it does not have Android compatibility. First, the user will have to click the ‘Go to Settings’ button and once on the settings page the user must select WiFi. The user should then see an option labeled “SmartSurge.” The user should select this option and entire “Vampire” as the password to this wireless network. Once connected to this access point the user should go to a browser on their phone. Once there the user can select his or her preferred wireless network and enter the password to said wireless network. After completing this step, a username and password is required to connect to the MQTT server. The username is ‘SmartSurge’ and the password is ‘energy’. Once these fields are completed, click ‘Connect to Server’. The application will not work properly if this information is incorrect, so it is imperative that these are correct. After connecting to the server, the user will press the ‘Home’ button and be brought to the application’s home page.

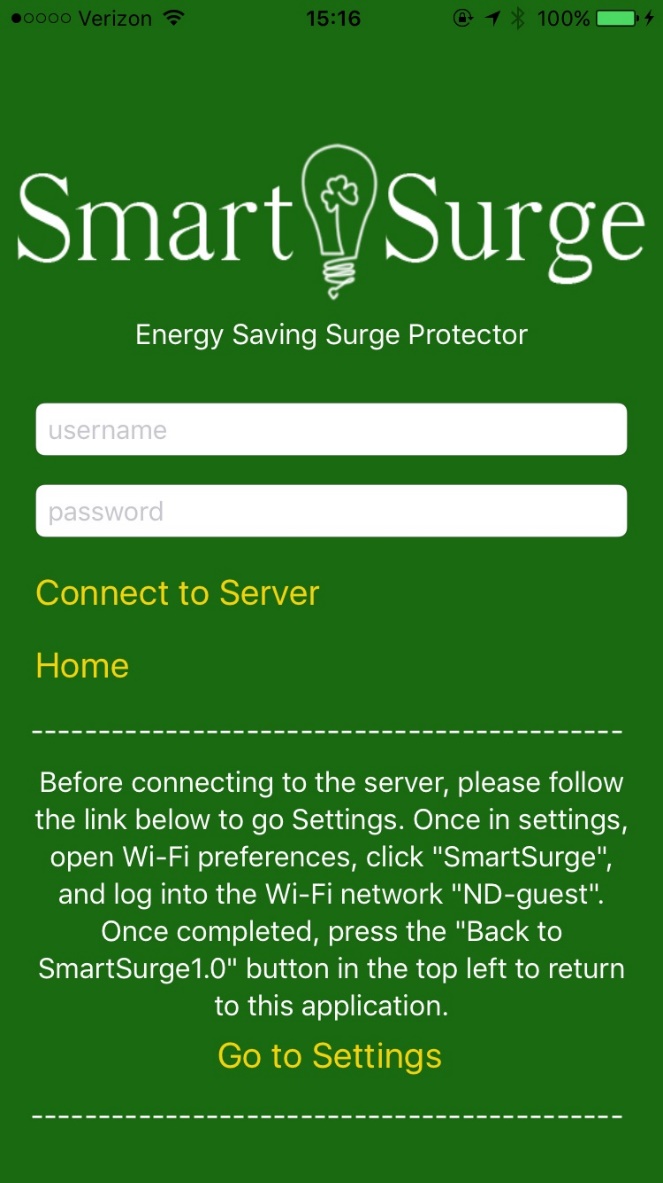


Figure 7.2.1: The Login Page

**Home Page** (Figure 7.2.2)**:** The home page of the SmartSurge app is simply a gateway to other pages in the app. As seen in the figure, the user has four options. They can be connected to the energy consumption page, the schedule page, the power device on/off page, and finally have the option to disconnect from the server and be brought back to the login page.



Figure 7.2.2: The Home Screen

**Energy Consumption Page** (Figure 7.2.3)**:** This is where the user will be able to see all of the relevant data regarding the current values from each power outlet, the power currently being used, and the amount of energy saved compared to a normal surge protector. The user needs to press the button corresponding to any of the four categories in order to receive the live update from the SmartSurge. The values will be displayed on the right side of the table, and they will keep their static values until the user chooses to update them from the page. Pressing the ‘Back’ button will bring the user back to the home page.

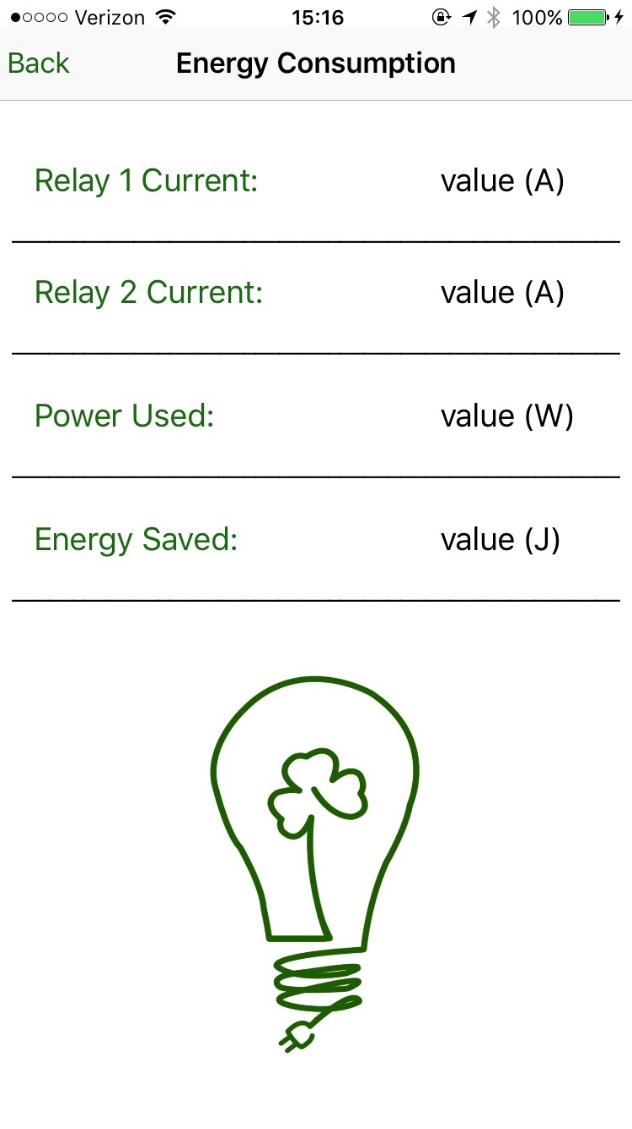


Figure 7.2.3: The Energy Consumption Page

**Schedule Page** (Figure 7.2.4)**:** The user can set a specific date and time to turn the SmartSurge on or off. If the SmartSurge needs to be turned on at, for example, April 29th at 5:30 PM, the user will scroll to this value (as shown in the figure) and press the ‘Submit’ button to send the information to the server. Likewise, if the SmartSurge needs to be turned off at 9:30AM on the same day, the user can scroll to this specific date and time and then press the ‘Submit’ button. The function is clearly explained above each date/time picker to avoid user confusion. Pressing the ‘Back’ button will bring the user back to the home page.

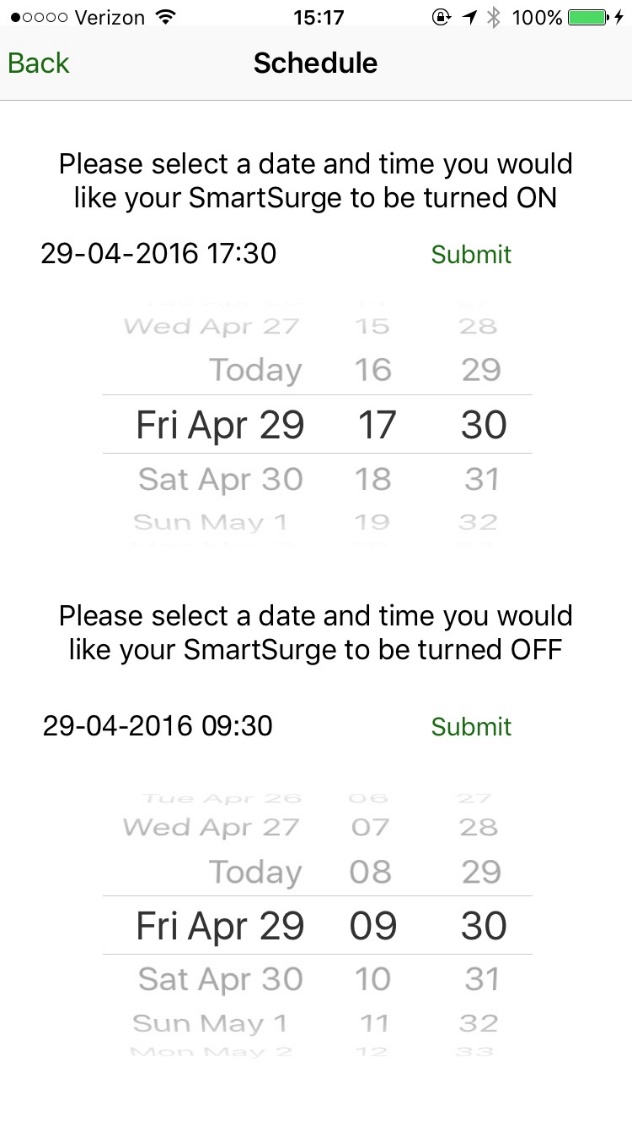


Figure 7.2.4: The Schedule Page

## 7.3 Is My Product Working?

To test if the product is working after configuring all Wi-Fi settings, plug in an electrical device to one of the outlets. Make sure it is easy to see if this device is on or off (e.g. a lamp or alarm clock). Using the mobile app, toggle power on and off this device according to the procedure described in Section 7.3. After verifying operation of this outlet, test the next outlet. If this functionality is working, then the device is operating normally.

## 7.4 Troubleshooting

If one of the above tests does not work and the application appears to be working normally, the issue is probably due to a loose connection. IMPORTANT: Unplug the SmartSurge product. Next, examine the schematics in the Appendix, Sections 10.1-3. Ensure that each line is properly and securely tightened. Make sure the slide switch on the Micro Board is set to hold GPIO0 High. If it is low, the device is in programming mode. To exit programming mode, slide the switch to hold GPIO0 high then plug in the device.

If the device cannot detect your home Wi-Fi network, move the device closer to your wireless router. If the mobile application is not responding, simply closing the app and restarting it will satisfy most issues. If more issues persist, try deleting and reinstalling the application.

# 8. To-Market Design Changes

In order to make the product more practical for mass production and the consumer market we would need to make a few major adjustments to the device. As mentioned before, we had the intention of splitting the device into two outlets so that devices that always need to be connected to power, such as alarm clocks, and devices that should be unplugged when not in use, such as videogame consoles and televisions, can be switched on or off at different frequencies. We still agree that this principle is likely still the best principle; however, we could save money and space by slighting altering it. If we only had one outlet that had a current sensor and could be switched on or off we would save a non-negligible amount of money in the final product. The user would still be able to plug devices, such as alarm clocks, into a different outlet, one that would never need to be turned off, but the ability to monitor the current and status of other, presumably more power hungry devices.

Additionally, if we were to bring this product to market we would need to significantly drop the size of the package. We currently have the entire product in a wooden box; however, if we could make it look much more like a conventional power strip with a smaller footprint it would be more likely that a consumer would buy the product.

If we were to sell the product commercially we would need to significantly improve the performance of the application in a number of ways. First off, we would need to provide encryption so that it would be much more powerful to hack the device. If one were to hack the device then it would be rendered useless. We would also need to implement a complete calendar function that would take into account the days of the week, and could be programmed to be turned off and on in multiple instances weeks prior to the actual need. While this would require a significant amount of time to implement, the overall practicality of the device would greatly increase. Furthermore, we will have to further ensure that the device itself does not consume too much power such that the power it saves is offset by the power used by the auxiliary aspects of the product.

Finally, we would need to significantly improve the wiring of the device. This is for two reasons, the first being safety. Leaving something like this plugged in for an extended period of time without ensuring that all the wiring is excellent would be a major safety hazard and would have a good chance of starting a fire. When implementing the device we had a few instances of shorts, which caused a few of our components to fail. We would need to ensure better wiring to make sure that the consumer would not receive a product that would quickly fail him or her.

# 9. Conclusions

Overall we enjoyed working on this project very much. It was extremely gratifying to get a relatively complicated project to work after such extensive planning and hours of work. We feel that our device represents a legitimate prototype to something that could help save the consumer’s wallet as well as the environment. While hurdles that are mentioned in section 8 must be cleared in order to have a viable product we do not see that as an insurmountable endeavor. Ultimately, the project would have been much more impressive and successful if we had not had the unfortunate sorting of our current sensors the night before we were to demonstrate our final project. Although we did show that they did indeed work and could be integrated into our project we were never able to fully incorporate them. While this problem might have been avoidable if we had fully assembled our project at an earlier time; however, we would have likely come into more problems. Even after we had shorted our current sensors we managed to short other components when we attempted to close the housing of the device.

While we did not fully achieve what we set out to do, we are proud of our final accomplishment. It took much time and coordination to control the power strip from a cell phone. Even six months ago it would have seemed like an incredibly complex challenge to take on with very little idea of what it would require. Each one of us fully immersed ourselves into a subsystem and we were able work exceptionally smoothly in bringing together our final product.

# 

# 10. Appendices

## 10.1 Power Board

If one of the above tests does not work and the application appears to be working normally, the Shown below in Figure 8.1.1 is the schematic of the Power Board. The Power Board features all of the line voltage components, as a way of insulating most of the sensitive components. The main functions of the Power Board are to rectify line voltages (120 Vac) to +5 Vdc, as well as utilizing sensors to capture the current through each set of outlets. These current readings are then passed to an external analog-to-digital converter, which communicates with the microcontroller via I2C.

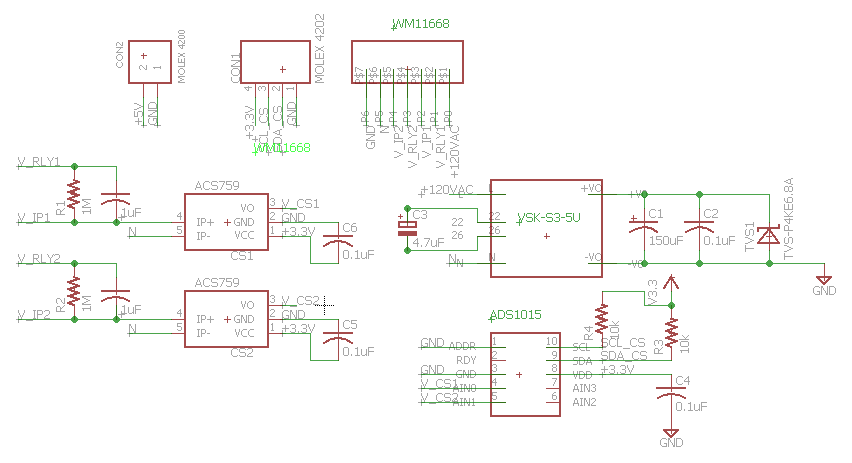


Figure 8.1.1: A Complete Schematic of the Power Board

All components are connected via typical application procedures as described in their data sheets; links to which can be found below. Line voltages are fed into the board using a WM11668 Molex connector. They include 120 Vac power, neutral, ground, lines for the voltages on the positive terminal of both relays (V\_RLY1/2), and negative terminal of each relay (V\_IP1/2). The 120 Vac and neutral lines are fed to the voltage regulator (VSK-S3-5U), which outputs +5 Vdc with respect to ground. Each relay is decoupled using a 1 μF capacitor and 1 MΩ bleeder resistor. The voltages from the negative terminal are then fed into the current sensors (ACS759). The outputs from the current sensors are sent to the external ADC (ADS1015), which will send the I2C signals to the microcontroller.

Relevant Parts:

VSK-S3-5U voltage regulator:

<http://www.cui.com/product/resource/vsk-s3.pdf>

ADS1015 AD converter:

<http://www.ti.com/product/ADS1015>

ACS759 current sensor:

<https://www.arrow.com/en/products/search?datasheets=true&amp;q=ACS759LCB-050B-PFF-T>

WM11668 high voltage connector:

<http://www.digikey.com/product-detail/en/molex-llc/1722861107/WM11668-ND/5360133>

QXK2E105KTPTZH 1μF high voltage capacitor:

<http://www.mouser.com/ProductDetail/Nichicon/QXK2E105KTPTZH/?qs=Dgr3QhzC8eSeM7X7H0jCyw%3D%3D>

P4KE6.8ACT Diode:

<http://www.digikey.com/product-detail/en/P4KE6.8A/P4KE6.8ACT-ND/2770318>

80-ESH475M400AH2AA 4.7μF high voltage capacitor:

<http://www.mouser.com/ProductDetail/Kemet/ESH475M400AH2AA/?qs=sGAEpiMZZMtZ1n0r9vR22bRn0aRzvAvuuOzbX4UeNKk%3d>

667-EEU-FM0J151 150μF capacitor:

<http://www.mouser.com/ProductDetail/Panasonic/EEU-FM0J151/?qs=sGAEpiMZZMvwFf0viD3Y3ffldXPLozayAjs1YqMz%2fUQ%3d>

## 10.2 Microboard

The complete schematic for the Micro Board shown below in Figure 8.2.1. This board is so named because it contains the microcontroller. The main functions of this board are to convert +5 Vdc to +3.3 Vdc to power the microcontroller, output signals to control the relays via a pushbutton or user input, achieve easy programmability of the microcontroller, and read I2C signals from the current sensors on the Power Board.

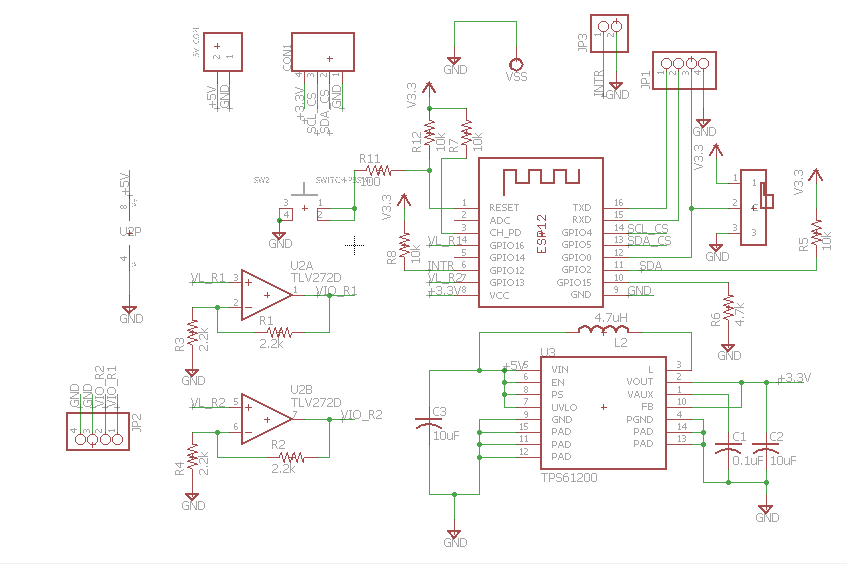


Figure 8.2.1: The Complete Schematic of the Micro Board

All components are connected via typical application procedures as described in their data sheets; links to which can be found below. The board is powered via a +5 Vdc connection to the Power Board. This +5 Vdc signal is converted to +3.3 Vdc via the TPS61201 component. This +3.3 Vdc power is used to power the ESP12 microcontroller, which has Wi-Fi capability. The ESP12 is responsible for turning power on/off to the outlets by controlling the relays. A user controls these relays by toggling them on/off in the mobile application or through the use of a pushbutton mounted on the case of the power strip. This is done simply by giving a digital high or low output on the ESP12’s GIOP pins 16 or 13. Because the relays operate at +5 Vdc, an operational amplifier (TLV272) is used to step up the microcontroller’s outputs to the appropriate voltage level via a simple non-inverting amplifier configuration. Pressing the pushbutton will trigger a falling edge interrupt on GPIO12, which will then toggle power on the relays. To be in programming mode on the microcontroller, GPIO0 must be held low and the microcontroller restarted. To trigger this mode, a sliding switch can hold GPIO0 low and a pushbutton triggers reset. Lastly, the microcontroller must read I2C signals from the current sensors.

Relevant Parts:

ESP12 microcontroller:

<https://cdn-shop.adafruit.com/product-files/2471/0A-ESP8266__Datasheet__EN_v4.3.pdf>

TPS61201 voltage converter:

<http://www.ti.com/lit/ds/symlink/tps61200.pdf>

TLV272 op-amp:

<http://www.ti.com/lit/ds/symlink/tlv272.pdf>

## 10.3 Overall Hardware Design

The overall hardware design must integrate the circuit boards with the external components, mainly the relays, outlets, and pushbutton. It must also connect to a power outlet. Figure 8.3.1 shows the block schematic of the overall system, including the external components. Additionally, Figures 8.3.2 and 8.3.3 show the actual physical build.

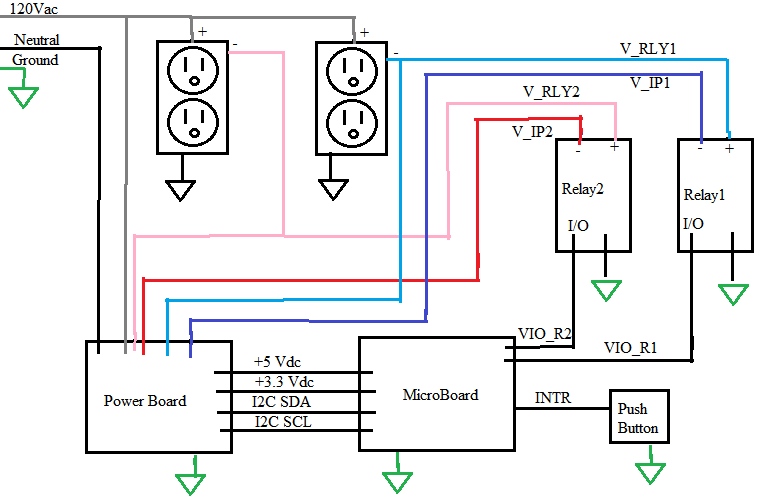


Figure 8.3.1: The Complete Schematic of the Physical Build



Figure 8.3.2: The Exterior of the Physical Build

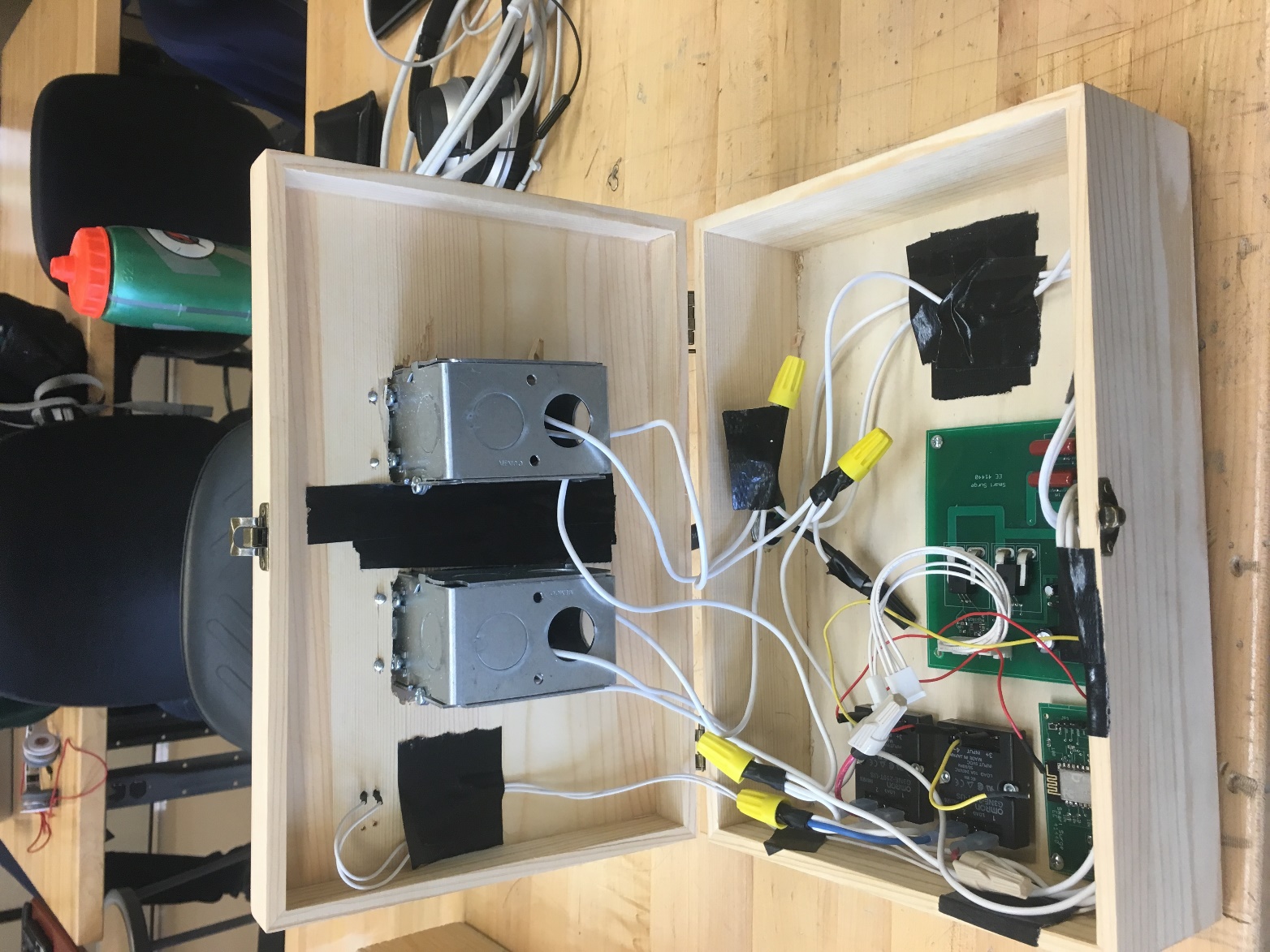


Figure 8.3.3: The Interior of the Physical Build

120 Vac is fed into the device and delivered to both sets of outlets and the Power Board. The outlets’ negative terminals are sent to the relays’ (G3NE-210T) positive terminals, which can toggle power based on the output from the Micro Board. The positive terminal of the relays is also connected to the Power Board, where a bleeder resistor and capacitor help stabilize the circuit. The negative terminals of the relays are then connected to the Power Board, where the current sensors can make readings. The +5 Vdc, I2C SDA, and SCL voltages generated on the Power Board are sent to the Micro Board for processing, while the +3.3 Vdc voltage is sent to the Power Board to power the current sensors.

Relevant Parts:

G3NE-210T relay:

<http://www.digikey.com/product-detail/en/omron-automation-and-safety/G3NE-210TL-US%20DC5/G3NE-210TL-US%20DC5-ND/3460973>

## 

## 10.4 Mobile Application Code

Master Code

Mobile Application:

ViewController.swift

//

// ViewController.swift

// SmartSurge1.0

//

// Created by Adam Rainey on 3/16/16.

// Copyright © 2016 Adam Rainey. All rights reserved.

//

import UIKit

import CocoaMQTT

import EventKit

import Foundation

class ViewController: UIViewController, UITextFieldDelegate {

// MARK: Properties

@IBOutlet weak var nameUsernameField: UITextField!

@IBOutlet weak var namePasswordField: UITextField!

var mqtt: CocoaMQTT?

override func viewDidLoad() {

super.viewDidLoad()

//MARK: Connecting to MQTT

let clientIdPid = "SeniorDesign" + String(NSProcessInfo().processIdentifier)

mqtt = CocoaMQTT(clientId: clientIdPid, host: "senior-mqtt.esc.nd.edu", port: 1883)

//mqtt.secureMQTT = true

if let mqtt = mqtt {

mqtt.username = "SmartSurge"

mqtt.password = "energy"

mqtt.willMessage = CocoaMQTTWill(topic: "/will", message: "dieout")

mqtt.keepAlive = 90

mqtt.delegate = self

mqtt.connect()

}

}

// MARK: UITextFieldDelegate and Keyboard Toggle

func textFieldShouldReturn(textField: UITextField) -> Bool {

textField.resignFirstResponder()

return true

}

override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?){

view.endEditing(true)

super.touchesBegan(touches, withEvent: event)

}

// MARK: Actions

// Trying to connect to settings within the app

@IBAction func goToSettings(sender: UIButton) {

UIApplication.sharedApplication().openURL(NSURL(string: UIApplicationOpenSettingsURLString)!)

}

// Buttons to change the current status of the SmartSurge power display

// Buttons will also send a command to the server to actually perform the function

@IBAction func currentStateSurge(sender: UIButton) {

mqtt!.publish("All", withString: "State")

}

@IBAction func currentStateRelay1(sender: UIButton) {

mqtt!.publish("LeftOutlet", withString: "LeftState")

}

@IBAction func currentStateRelay2(sender: UIButton) {

mqtt!.publish("RightOutlet", withString: "RightState")

}

@IBAction func turnDeviceOn(sender: UIButton) {

mqtt!.publish("All", withString: "On")

}

@IBAction func turnDeviceOff(sender: UIButton) {

mqtt!.publish("All", withString: "Off")

}

@IBAction func turnRelay1On(sender: UIButton) {

mqtt!.publish("LeftOutlet", withString: "LeftOn")

}

@IBAction func turnRelay1Off(sender: UIButton) {

mqtt!.publish("LeftOutlet", withString: "LeftOff")

}

@IBAction func turnRelay2On(sender: UIButton) {

mqtt!.publish("RightOutlet", withString: "RightOn")

}

@IBAction func turnRelay2Off(sender: UIButton) {

mqtt!.publish("RightOutlet", withString: "RightOff")

}

// MARK: Energy Consumption Page Setup

@IBOutlet weak var Relay1CurrentLabel: UILabel!

@IBOutlet weak var Relay2CurrentLabel: UILabel!

@IBOutlet weak var PowerUsedLabel: UILabel!

@IBOutlet weak var EnergySavedLabel: UILabel!

@IBAction func readRelay1Current(sender: UIButton) {

mqtt!.publish("LeftOutlet", withString: "LeftCurrent")

}

@IBAction func readRelay2Current(sender: UIButton) {

mqtt!.publish("RightOutlet", withString: "RightCurrent")

}

@IBAction func readPowerValue(sender: UIButton) {

mqtt!.publish("PowerValue", withString: "100")

}

@IBAction func readEnergyValue(sender: UIButton) {

mqtt!.publish("EnergyValue", withString: "100")

}

// calculate power used function

// Not implemented due to faulty current sensors

// calculate energy

// Not implemented due to faulty current sensors

// Defining label constants for Power On/Off page

@IBOutlet weak var onOffLabel: UILabel!

@IBOutlet weak var onOffRelay1Label: UILabel!

@IBOutlet weak var onOffRelay2Label: UILabel!

// MARK: Connecting to and disconnecting from the Server

@IBAction func connectToServer(sender: UIButton) {

let password = namePasswordField.text

let username = nameUsernameField.text

if username == mqtt?.username && password == mqtt?.password {

mqtt!.connect()

}

else {

mqtt!.disconnect()

}

}

@IBAction func disconnectFromServer(sender: UIButton) {

mqtt!.disconnect()

}

// MARK: Calendar Setup and Schedule information

var submitOnString: String?

@IBOutlet weak var myDatePicker: UIDatePicker!

@IBOutlet weak var selectedDate: UILabel!

@IBAction func datePickerAction(sender: AnyObject) {

let dateFormatter = NSDateFormatter()

dateFormatter.dateFormat = "dd-MM-yyyy HH:mm"

let strDate = dateFormatter.stringFromDate(myDatePicker.date)

self.selectedDate.text = strDate

submitOnString = strDate

}

@IBAction func submitDate(sender: UIButton) {

mqtt!.publish("Schedule", withString: "ON")

mqtt!.publish("SmartSurge", withString: submitOnString!)

}

@IBOutlet weak var myDatePickerOff: UIDatePicker!

var submitOffString: String?

@IBOutlet weak var selectedOffDate: UILabel!

@IBAction func offDatePickerAction(sender: AnyObject) {

let dateFormatter2 = NSDateFormatter()

dateFormatter2.dateFormat = "dd-MM-yyyy HH:mm"

let strDate2 = dateFormatter2.stringFromDate(myDatePickerOff.date)

self.selectedOffDate.text = strDate2

submitOffString = strDate2

}

@IBAction func submitOffDate(sender: UIButton) {

mqtt!.publish("Schedule", withString: "OFF")

mqtt!.publish("SmartSurge", withString: submitOffString!)

}

}

// MARK: Using CocoaMQTT as a Delegate

extension ViewController: CocoaMQTTDelegate {

func mqtt(mqtt: CocoaMQTT, didConnect host: String, port: Int) {

print("didConnect \(host):\(port)")

}

func mqtt(mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) {

//print("didConnectAck \(ack.rawValue)")

if ack == .ACCEPT {

//subsribing to a topic

// "Schedule" used for on/off clarification of time

mqtt.subscribe("All", qos: CocoaMQTTQOS.QOS1)

mqtt.subscribe("Energy", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("RightOutlet", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("LeftOutlet", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("PowerValue", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("EnergyValue", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("LC", qos: CocoaMQTTQOS.QOS2)

mqtt.subscribe("RC", qos: CocoaMQTTQOS.QOS2)

mqtt.ping()

}

}

func mqtt(mqtt: CocoaMQTT, didPublishMessage message: CocoaMQTTMessage, id: UInt16) {

print("didPublishMessage with message: \(message.string)")

}

func mqtt(mqtt: CocoaMQTT, didPublishAck id: UInt16) {

print("didPublishAck with id: \(id)")

}

// Connecting the current statuses to respective topics

func mqtt(mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16 ) {

print("didReceivedMessage: \(message.string) with id \(id)")

// Update Status of Relay 1 Button

if (message.topic == "LeftOutlet"){

if message.string == "1" {

onOffRelay1Label.text = "ON"

}

else if message.string == "0" {

onOffRelay1Label.text = "OFF"

}

}

// Update Status of Relay 2 Button

else if (message.topic == "RightOutlet"){

if message.string == "1" {

onOffRelay2Label.text = "ON"

}

else if message.string == "0" {

onOffRelay2Label.text = "OFF"

}

}

// Update status of Relay 1 Current Label

else if (message.topic == "LC"){

Relay1CurrentLabel.text = message.string!

}

// Update status of Relay 2 Current Label

else if (message.topic == "RC"){

Relay2CurrentLabel.text = message.string!

}

// Update current status of entire SmartSurge device

else if (message.topic == "All"){

if message.string == "00" {

onOffRelay1Label.text = "OFF"

onOffRelay2Label.text = "OFF"

onOffLabel.text = "OFF"

}

if message.string == "01" {

onOffRelay1Label.text = "OFF"

onOffRelay2Label.text = "ON"

onOffLabel.text = "ON"

}

if message.string == "10" {

onOffRelay1Label.text = "ON"

onOffRelay2Label.text = "OFF"

onOffLabel.text = "ON"

}

if message.string == "11" {

onOffRelay1Label.text = "ON"

onOffRelay2Label.text = "ON"

onOffLabel.text = "ON"

}

}

// Update current status of Power value label

else if (message.topic == "PowerValue"){

PowerUsedLabel.text = message.string!

}

// Update current status of Energy value label

else if (message.topic == "EnergyValue"){

EnergySavedLabel.text = message.string!

}

}

func mqtt(mqtt: CocoaMQTT, didSubscribeTopic topic: String) {

//print("didSubscribeTopic to \(topic)")

}

func mqtt(mqtt: CocoaMQTT, didUnsubscribeTopic topic: String) {

print("didUnsubscribeTopic to \(topic)")

}

func mqttDidPing(mqtt: CocoaMQTT) {

print("didPing")

}

func mqttDidReceivePong(mqtt: CocoaMQTT) {

\_console("didReceivePong")

}

func mqttDidDisconnect(mqtt: CocoaMQTT, withError err: NSError?) {

\_console("mqttDidDisconnect: \(err.debugDescription)")

}

func \_console(info: String) {

print("Delegate: \(info)")

}

}

## 10.5 Ardunio ESP Code

#include <FS.h>

#include <ESP8266WiFi.h>

#include <PubSubClient.h>

#include <DNSServer.h>

#include <ESP8266WebServer.h>

#include <WiFiManager.h>

#include <WiFiUdp.h>

#include <Adafruit\_ADS1015.h>

#include <Wire.h>

#include <EEPROM.h>

#define SCL 4

#define SDA 5

#define INTR 12

#define leftRELAY 14

#define rightRELAY 13

Adafruit\_ADS1015 ads1015;

#define SERVER\_ADDRESS "senior-mqtt.esc.nd.edu" // server in 213 SR

#define SERVER\_PORT 1883 // standard port

#define ESP8266\_LED 5

float right\_curr;

float left\_curr;

int iii = 0;

unsigned long timer;

unsigned long timehold = 0;

unsigned long NTPtime;

int currentMemoryAddress = 50;

int scheduleMemoryAddress = 150;

unsigned long epochalarm = 0;

String SET = "NOTYET";

WiFiClient wf\_client; // instantiate wifi client

PubSubClient client(wf\_client, SERVER\_ADDRESS); // pass to pubsub

unsigned int localPort = 2390; // local port to listen for UDP packets

/\* Don't hardwire the IP address or we won't get the benefits of the pool.

\* Lookup the IP address for the host name instead \*/

//IPAddress timeServer(129, 6, 15, 28); // time.nist.gov NTP server

IPAddress timeServerIP; // time.nist.gov NTP server address

const char\* ntpServerName = "time.nist.gov";

const int NTP\_PACKET\_SIZE = 48; // NTP time stamp is in the first 48 bytes of the message

byte packetBuffer[ NTP\_PACKET\_SIZE]; //buffer to hold incoming and outgoing packets

// A UDP instance to let us send and receive packets over UDP

WiFiUDP udp;

void callback(const MQTT::Publish& pub) {

// handle message arrived

Serial.print("Message arrived [");

Serial.print(pub.topic());

Serial.print("] ");

Serial.println(pub.payload\_string());

Serial.println();

String message\_buff = pub.payload\_string();

// char RightOutlet;

if (pub.topic() == "RightOutlet"){

if(message\_buff == "RightOn"){

setRightOutHigh();

String rightS = String(readRightState());

client.publish("RightOutlet", rightS);

}

if(message\_buff == "RightOff"){

setRightOutLow();

String rightS = String(readRightState());

client.publish("RightOutlet", rightS);

}

if(message\_buff == "RightCurrent"){

String currR = String(readRightCurrent(300));

client.publish("RC", currR);

}

if(message\_buff == "RightState"){

String rightS = String(readRightState());

client.publish("RightOutlet", rightS);

}

}

if (pub.topic() == "LeftOutlet"){

if(message\_buff == "LeftOn"){

setLeftOutHigh();

String leftS = String(readLeftState());

client.publish("LeftOutlet", leftS);

}

if(message\_buff == "LeftOff"){

setLeftOutLow();

String leftS = String(readLeftState());

client.publish("LeftOutlet", leftS);

}

if(message\_buff == "LeftCurrent"){

String currL = String(readLeftCurrent(300));

client.publish("LC", currL);

}

if(message\_buff == "LeftState"){

String leftS = String(readLeftState());

client.publish("LeftOutlet", leftS);

}

}

if (pub.topic() == "SmartSurge"){

if(message\_buff == "Off"){

setLeftOutLow();

setRightOutLow();

}

}

if (pub.topic() == "Schedule"){

if(message\_buff == "OFF"){

SET = "OFF";

Serial.println(SET);

}

if(message\_buff == "ON"){

SET = "ON";

Serial.println(SET);

}

}

if (pub.topic() == "All"){

if(message\_buff == "On"){

setLeftOutHigh();

setRightOutHigh();

client.publish("All", "11");

}

if(message\_buff == "Off"){

setLeftOutLow();

setRightOutLow();

client.publish("All", "00");

}

if(message\_buff == "State"){

int RLS = readLeftState();

int RRS = readRightState();

if (RLS == 0){

if (RRS == 0){

client.publish("All", "00");

}

if (RRS == 1){

client.publish("All", "01");

}

}

if (RLS == 1){

if (RRS == 0){

client.publish("All", "10");

}

if (RRS == 1){

client.publish("All", "11");

}

}

}

}

if (pub.topic() == "SmartSurge"){

// String Date = message\_buff; //"dd-MM-yyyy HH:mm";

String Date = message\_buff;;

String dayStr = Date.substring(0,2);

String monthStr = Date.substring(3,5);

String yearStr = Date.substring(6,10);

String hourStr = Date.substring(11,13);

String minuteStr = Date.substring(14,16);

int day = dayStr.toInt();

int month = monthStr.toInt();

int year = yearStr.toInt();

int hour = hourStr.toInt();

int minute = minuteStr.toInt();

byte dayb = char(day);

byte monthb = char(month);

byte yearb = char(year);

byte hourb = char(hour);

byte minuteb = char(minute);

int diff\_d = day - 28;

int diff\_y = year -2016;

int diff\_m = month-4;

epochalarm = (((((24\*diff\_d)+hour-4)\*60)+minute)\*60)+1461816000UL;

Serial.println("Alarm time is:");

Serial.println(epochalarm);

Serial.println("EST time is:");

}

}

void setup() {

// put your setup code here, to run once:

Serial.begin(9600);

while (!Serial) {

; // wait for serial port to connect. Needed for native USB port only

}

Serial.println();

EEPROM.begin(512);

ads1015.begin();

WiFiManager wifiManager;

//exit after config instead of connecting

wifiManager.setBreakAfterConfig(true);

//reset settings - for testing

// wifiManager.resetSettings();

// WiFi.begin("SDNet","CapstoneProject");

// while(WiFi.status() != WL\_CONNECTED){

wifiManager.setAPCallback(configModeCallback);

if (!wifiManager.autoConnect("SmartSurge", "Vampire")) {

Serial.println("failed to connect, we should reset as see if it connects");

delay(3000);

ESP.reset();

delay(10000);

}

// }

//if you get here you have connected to the WiFi

Serial.println("connected...yeey :)");

Serial.println("local ip");

Serial.println(WiFi.localIP());

pinMode(ESP8266\_LED, OUTPUT);

// Serial.begin(115200);

delay(10);

Serial.println();

Serial.println();

Serial.println(F("Modified pubsub client basic code using modified pubsub software"));

client.set\_callback(callback);

pinMode(INTR, INPUT); //Set interrupt pin to input

pinMode(leftRELAY, OUTPUT); //Set left relay pin to output

pinMode(rightRELAY, OUTPUT); //Set right relay pin to input

Wire.begin(SDA, SCL); //Initialize I2C on SDA & SCL pins

ads1015.begin(); //Initialize ads1015 operation

ads1015.setGain(GAIN\_ONE); //Set ads gain to 2, 1 mV/bit

attachInterrupt(12, intrChangeOut, FALLING); //Set interrupt to trigger on falling edge

// Open serial communications and wait for port to open:

udp.begin(localPort);

}

void EEPROMWritelong(int address, unsigned long value)

{

//Decomposition from a long to 4 bytes by using bitshift.

//One = Most significant -> Four = Least significant byte

byte four = (value & 0xFF);

byte three = ((value >> 8) & 0xFF);

byte two = ((value >> 16) & 0xFF);

byte one = ((value >> 24) & 0xFF);

//Write the 4 bytes into the eeprom memory.

EEPROM.write(address, four);

EEPROM.write(address + 1, three);

EEPROM.write(address + 2, two);

EEPROM.write(address + 3, one);

}

unsigned long EEPROMReadlong(int address)

{

//Read the 4 bytes from the eeprom memory.

unsigned long four = EEPROM.read(address);

unsigned long three = EEPROM.read(address + 1);

unsigned long two = EEPROM.read(address + 2);

unsigned long one = EEPROM.read(address + 3);

//Return the recomposed long by using bitshift.

return ((four << 0) & 0xFF) + ((three << 8) & 0xFFFF) + ((two << 16) & 0xFFFFFF) + ((one << 24) & 0xFFFFFFFF);

}

//Read the current through the right outlets

unsigned long storeCurrentData(unsigned long timer, int\* currentMemoryAddressPNT) {

timehold = timer;

int currentMemoryAddress = \*currentMemoryAddressPNT;

float rightCur = readRightCurrent(300);

float leftCur = readLeftCurrent(300);

unsigned long NTPtime; //NEED TO OBTAIN VALUE OF NTP time

EEPROMWritelong(currentMemoryAddress, NTPtime);

EEPROM.write(currentMemoryAddress + 4, rightCur);

EEPROM.write(currentMemoryAddress + 5, leftCur);

EEPROM.commit();

if (currentMemoryAddress >= 110) {

currentMemoryAddress = 50;

}

else {

currentMemoryAddress = currentMemoryAddress + 6;

}

\*currentMemoryAddressPNT = currentMemoryAddress;

return timehold;

}

void readCurrentData() {

String rightCur, leftCur, measTime;

for (int i = 50; i >= 110; i = i + 6) {

measTime = EEPROMReadlong(i);

rightCur = EEPROM.read(i + 4);

leftCur = EEPROM.read(i + 5);

//SEND THIS TO THE APP

}

}

void checkCalender(unsigned long NTPtime) {

byte relayState;

for (int i = 150; i <= 250; i = i + 5) {

unsigned long onoffTime = EEPROMReadlong(i);

unsigned long comp = abs(onoffTime - NTPtime);

if (comp <= 120) {

relayState = EEPROM.read(i + 4);

switch (relayState) {

case 0:

setLeftOutLow();

setRightOutLow();

break;

case 1:

setLeftOutLow();

setRightOutHigh();

break;

case 2:

setLeftOutHigh();

setRightOutLow();

break;

case 3:

setLeftOutHigh();

setRightOutHigh();

break;

}

}

}

}

void setCalender(String onoffTimeStr, int\* scheduleMemoryAddressPNT, byte relayState) {

unsigned long onoffTime = onoffTimeStr.toInt();

int scheduleMemoryAddress = \*scheduleMemoryAddressPNT;

if (scheduleMemoryAddress >= 250){

scheduleMemoryAddress = 150;

}

EEPROMWritelong(scheduleMemoryAddress, onoffTime);

EEPROM.write(scheduleMemoryAddress + 4, relayState);

EEPROM.commit();

\*scheduleMemoryAddressPNT = scheduleMemoryAddress + 5;

}

void loop() {

if (WiFi.status() == WL\_CONNECTED) {

if (!client.connected()) {

if (client.connect("mydevice")) {

Serial.println("Connected to MQTT server");

client.subscribe("SmartSurge");

client.subscribe("Schedule");

client.subscribe("All");

client.publish("Smart","Success");

client.subscribe("LeftOutlet");

Serial.println("Just subscribed to LeftOutlet");

client.subscribe("RightOutlet");

Serial.println("Just subscribed to RightOutlet");

}

}

/\* wait for incoming messages \*/

if (client.connected())

client.loop();

}

unsigned long TTime = checktime();

String time = String(TTime);

if (epochalarm > 0){

if(TTime > epochalarm){

if( SET == "ON"){

setLeftOutHigh();

setRightOutHigh();

SET == "NOTYET";

Serial.println("Alarm Turned Ports On");

epochalarm = 0;

}

if( SET == "OFF"){

setLeftOutLow();

setRightOutLow();

SET == "NOTYET";

Serial.println("Alarm Turned Ports Off");

epochalarm = 0;

}

}

}

}

unsigned long checktime(){

WiFi.hostByName(ntpServerName, timeServerIP);

sendNTPpacket(timeServerIP); // send an NTP packet to a time server

// wait to see if a reply is available

delay(1000);

int cb = udp.parsePacket();

if (!cb) {

// Serial.println("no packet yet");

return 0;

}

else {

// THIS Serial.print("packet received, length=");

// THIS Serial.println(cb);

// We've received a packet, read the data from it

udp.read(packetBuffer, NTP\_PACKET\_SIZE); // read the packet into the buffer

//the timestamp starts at byte 40 of the received packet and is four bytes,

// or two words, long. First, esxtract the two words:

unsigned long highWord = word(packetBuffer[40], packetBuffer[41]);

unsigned long lowWord = word(packetBuffer[42], packetBuffer[43]);

// combine the four bytes (two words) into a long integer

// this is NTP time (seconds since Jan 1 1900):

unsigned long secsSince1900 = highWord << 16 | lowWord;

// THIS Serial.print("Seconds since Jan 1 1900 = " );

// THIS Serial.println(secsSince1900);

// now convert NTP time into everyday time:

// THIS Serial.print("Unix time = ");

// Unix time starts on Jan 1 1970. In seconds, that's 2208988800:

const unsigned long seventyYears = 2208988800UL;

//const unsigned long seventyYears = 2208974400UL;

// subtract seventy years:

unsigned long hours4 = 14400;

unsigned long epoch = secsSince1900 - seventyYears - hours4;

return epoch;

}

}

//Interrupt routine for the push button

void intrChangeOut() { //Interrupt toggle on Pin 12

delayMicroseconds(500000);

if (r1 & r2 == 0) { //If both relays are off, turn power on

digitalWrite(leftRELAY, HIGH);

digitalWrite(rightRELAY, HIGH);

} else { //Else, turn both relays off

digitalWrite(ESP8266\_LED, HIGH);

digitalWrite(rightRELAY, LOW);

}

Serial.println("You've been interrupted");//Slight delay for bounce

}

void configModeCallback (WiFiManager \*myWiFiManager) {

Serial.println("Entered config mode");

Serial.println(WiFi.softAPIP());

//if you used auto generated SSID, print it

Serial.println(myWiFiManager->getConfigPortalSSID());

}

//Turn on the left outlets

void setLeftOutHigh()

{

digitalWrite(leftRELAY, HIGH);

}

//Turn on the right outlets

void setRightOutHigh()

{

digitalWrite(rightRELAY, HIGH);

}

//Turn off the left outlets

void setLeftOutLow()

{

digitalWrite(leftRELAY, LOW);

}

//Turn off the right outlets

void setRightOutLow()

{

digitalWrite(rightRELAY, LOW);

}

//Obtain the current state of the left outlets

int readLeftState()

{

return digitalRead(leftRELAY);

}

//Obtain the current state of the right outlets

int readRightState() {

return digitalRead(rightRELAY);

}

//Read the current through the left outlets

float readLeftCurrent(int samples) //Input is number of samples

{

float adc0; //Declare Variables

float data\_corr;

float data\_raw = 0;;

int i;

// For Loop measuing the current for the specified number of samples

for (i = 0; i <= samples - 1; i = i + 1) {

adc0 = abs(ads1015.readADC\_SingleEnded(0)); //adc1 is number from right outlet

data\_raw = max(adc0, data\_raw); //Ensure maximum value is found

}

data\_corr = (data\_raw - 1650) / 26.4 / 1.41421; //Gain = 2, 1 mV/bit, 26.4 mV/A, sqrt(2) for RMS value

return data\_raw;

}

//Read the current through the right outlets

float readRightCurrent(int samples) //Input is number of samples

{

float adc1; //Declare Variables

float data\_corr;

float data\_raw = 0;;

int i;

// For Loop measuing the current for the specified number of samples

for (i = 0; i <= samples - 1; i = i + 1) {

adc1 = abs(ads1015.readADC\_SingleEnded(1)); //adc1 is number from right outlet

data\_raw = max(adc1, data\_raw); //Ensure maximum value is found

}

// data\_corr = (data\_raw - 1650) / 26.4 / 1.41421; //Gain = 2, 1 mV/bit, 26.4 mV/A, sqrt(2) for RMS value

return data\_raw;

}

unsigned long sendNTPpacket(IPAddress& address)

{

// THIS Serial.println("sending NTP packet...");

// set all bytes in the buffer to 0

memset(packetBuffer, 0, NTP\_PACKET\_SIZE);

// Initialize values needed to form NTP request

// (see URL above for details on the packets)

packetBuffer[0] = 0b11100011; // LI, Version, Mode

packetBuffer[1] = 0; // Stratum, or type of clock

packetBuffer[2] = 6; // Polling Interval

packetBuffer[3] = 0xEC; // Peer Clock Precision

// 8 bytes of zero for Root Delay & Root Dispersion

packetBuffer[12] = 49;

packetBuffer[13] = 0x4E;

packetBuffer[14] = 49;

packetBuffer[15] = 52;

// all NTP fields have been given values, now

// you can send a packet requesting a timestamp:

udp.beginPacket(address, 123); //NTP requests are to port 123

udp.write(packetBuffer, NTP\_PACKET\_SIZE);

udp.endPacket();

}