CSE 147 Final Project Report

Project Title & Team Member Information

ShutEye : Intelligent Energy Usage Monitoring and Mitigation for Your Home

Pranav Mehta, p3mehta@ucsd.edu, A17323782; Aarya Topiwala, atopiwala@ucsd.edu, A17295542

Motivation

Energy costs rise every year, especially as people add more devices to their homes, increasing their overall electricity consumption. However, some of that electricity is wasted, which is bad for the planet and the wallet. Our system lowers energy costs for its users by intelligently determining which appliances consuming energy are being actively used, and providing the option to automatically shut off unused appliances. Many energy monitoring devices already exist, but they are not able to intelligently determine if the appliance connected to them needs power at that moment and shut off power automatically.

Related Work

Our project enhances existing Tapo Smart Plugs by adding intelligent user presence detection to evaluate if connected appliances are in use and turn them off when not in use. While Tapo plugs can measure energy usage and remotely control devices, they cannot turn off intelligently on their own outside of rudimentary timer systems. Existing infrared motion-activated lighting systems inspire our approach, but we extend this functionality to more appliances and also implement smarter detection. An additional drawback of existing infrared systems is that they depend on *motion* not merely proximity. It is a common occurrence for these devices to suddenly turn off if you don't move enough or are away from their line of sight. Our sensors can specifically be used to detect users directionally (for devices you should be directly in front of, like TVs, monitors, or computers) or omnidirectionally (for devices that affect the whole room, like lights, fans, and heaters). To integrate with Tapo devices via an ESP32, we leveraged the open-source tapo-esp32 repository, which reverse-engineers TP-Link's protocol. We also used the python-kasa repo as a reference for the API routes we needed for adding energy monitoring API calls to the tapo-esp32 repo, as it originally only had functions to turn the plugs on and off. Inspired by UWB's use in car key fobs and the opentags project, we decided to explore UWB for our indoor user localization use case. While the opentags docs guided our hardware choices, we relied on Qorvo's demo code for stability and ease of use. We use Qorvo's DWM3000 Ultra-Wideband (UWB) chips for ranging in our presence detection stack, leveraging their Nearby Interaction SDK. We customized Qorvo's base NI Background iOS app and added support for retrieving analytics from our web server. On the embedded side, we implemented UART communication with the ESP32 in a FreeRTOS-based application on the nRF52840-DK. Our ESP32 integrates an OLED screen to display system status, using Espressif's Arduino ESP32 board package and Adafruit's SSD1306 library for simplified firmware development. Our custom firmware connects the ESP32 to WiFi, reads distance data from the ultrasonic sensor and UWB chip to sense user presence, determines device safety status using the flame sensor, displays status updates on the OLED, and most importantly, uses presence detection to control the smart plugs turning devices on or off. Our ESP32 also subscribes to an MQTT topic using the PubSubClient library in order to determine user presence based on a BLE beacon placed in the same room. We leveraged the open-source ESPresense software to set up this BLE beacon. Essentially, one BLE beacon is needed for each room of appliances, and the user can listen on the MQTT topic that corresponds to the room name they set in ESPresense and the name of the device being tracked to determine user presence (we use the user's phone for this). The ESPresense node publishes to an Eclipse Mosquitto MQTT broker running on our Raspberry Pi, which our ESP32-based device connects to. Our Raspberry Pi also runs a MariaDB (MySQL) server with a FastAPI backend, which we send HTTP post requests to from our device to save data and we make HTTP get requests to this server for displaying data in our web and mobile app. To our knowledge, there are no existing energy-saving systems that use ultrasonic or ultra-wideband sensors and allow users to see how their proximity to their devices affects energy usage.

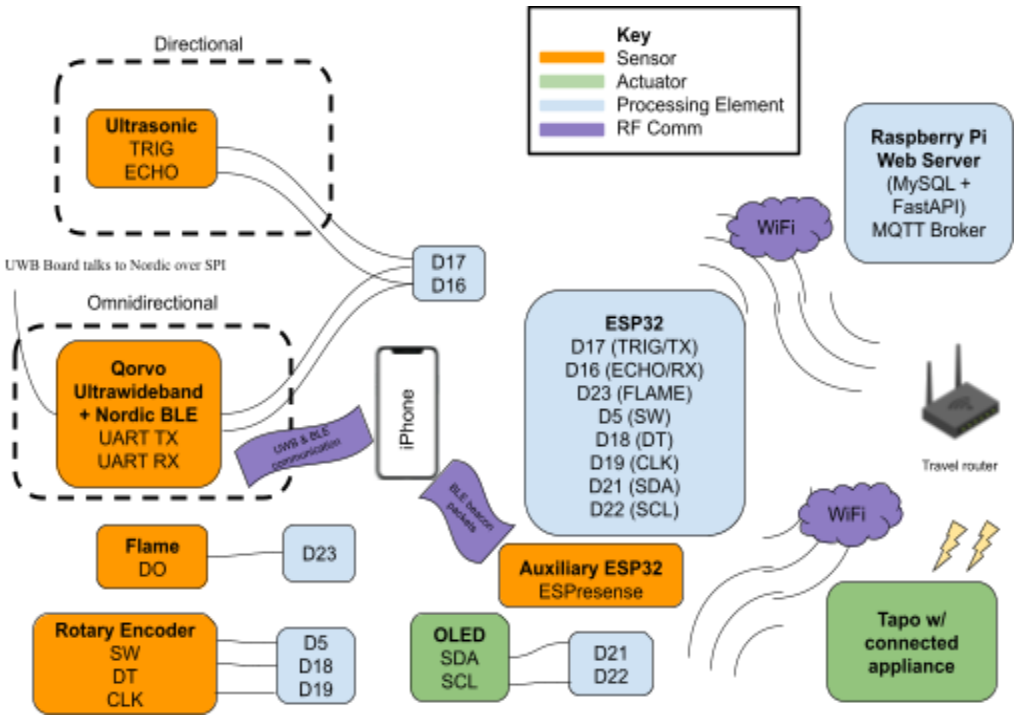
Hardware Components

1. TP-Link Tapo Smart Plugs P115
    a. These function as the primary means of actuation for our system, as they can be used to control and monitor power to lamps, fans, and other connected appliances. The Tapo devices can be controlled over WiFi which is perfect since the ESP32 Dev Module has many WiFi capabilities. Additionally, the Tapo devices have a variety of open-source reverse-engineering projects that we built upon for our project.
2. HC-SR04 Ultrasonic sensors (make use of the RCWL-9300C ultrasonic chip manufactured by RCWL, sensor is open source and produced by a wide variety of Chinese manufacturers)
    a. These function as one way to detect users in a specific direction. This is useful for appliances that are relevant when a user is directly facing them (like monitors or TVs)
3. Espressif ESP32 Wroom32 DevKit V1
    a. These function as the primary WiFi-capable microcontroller within our system, giving it the means to connect to other IoT devices such as the Tapo smart plugs. Additionally, the ESP32 sends smart plug power analytics information to a Raspberry Pi web server to store and display the energy consumption and proximity data for users.
    b. These chips are useful since we realized during development that the nRF52840 does not have WiFi, enabling us to bypass that issue.
    c. An auxiliary ESP32 is used to run ESPresense to provide additional distance data for omnidirectional modules.
4. Nordic NRF52840-DK
    a. Uses the nrf52840, a small microcontroller with Bluetooth support. Used for communicating with the Qorvo board over SPI and pairing and communicating with the smartphone app over BLE.
    b. The NRF52840DK is easy to use with the DWM3000EVB Arduino shield, and Qorvo has provided documentation for this board, which made the process smoother.
5. Qorvo DWM3000EVB Ultra Wideband Evaluation Boards
    a. The DWM3000 Ultra-Wideband chips are used for precise user distance tracking omnidirectionally. The ultrasonic sensor only works in one direction, which may be appropriate for devices like a monitor, but not appropriate for a lamp for example (since the user may be away from its direct line of sight).
6. SSD1306 OLED (SSD1306 chip produced by Solomon Systech)
    a. The OLED module can display information about the sensor to the user. By default the OLED screen displays the range detected on the proximity sensor (ultrasonic or ultra-wideband), the amount of power currently being used by a specific Tapo plug, and the amount of time left in the timeout period before the Tapo plug automatically turns off (if the user is away). When the user adjusts the settings, such as the detection distance or timeout period, the OLED will display this information. Additionally, the OLED will display if a fire (flame) is detected.
7. KY-026 Flame Sensor (sensor is open source and produced by a wide variety of Chinese manufacturers)
    a. These are used for a safety mechanism in our system. In case of an electrical fire with connected appliances, the plug will detect the fire and automatically shut off.
8. KY-040 Rotary encoder (sensor is open source and produced by a wide variety of Chinese manufacturers)
    a. The rotary encoder has a built-in button switch which when pressed allows the user to change the detection distance or the timeout period. After approximately 5 seconds, the settings screen reverts to the default screen which shows the distance, power, and timeout time.
9. Raspberry Pi (Raspberry Pi Foundation)

    a. The Raspberry Pi runs the Mosquitto MQTT broker used by ESPresense for publishing user presence and distance information determined using BLE. Our ESP32-based devices subscribe to this topic to receive Bluetooth distance information.
    b. The Raspberry Pi runs a MariaDB database for storing and retrieving analytics data.
    c. The Raspberry Pi runs a FastAPI web app used to provide a way to store and retrieve data from the MariaDB database over WiFi. It has a simple webpage users can access to view analytics data.
    d. The Raspberry Pi is good for running a data server and MQTT broker since it has a lot of storage on its SD card unlike the ESP32, which is limited to 4MB of Flash, even if using SPIFFS. Its many cores allow for smooth and responsive data stores and fetches, while still having processing power left over to run a simple web app on top. By using the Pi, the user can keep their data local and secure and does not need to worry about the additional complexity/cost of setting up a cloud account. The Pi does consume more power than using the cloud might as a tradeoff, but our goal was to prioritize simplicity and give the user the most control over their system. Another neat feature of our system is that though we use WiFi, since everything is done over the local network, an internet connection is not required to use the system.
10. GL.iNet GL-A1300 Travel Router
    a. The travel router was used to allocate static IPs to our Tapo plugs and the Raspberry Pi data server in order to simplify communication. It was also very useful in ensuring we could keep our entire network infrastructure local.

Hardware Design

The ESP32, rotary encoder, OLED, and flame sensor are all placed on a breadboard, along with the ultrasonic sensor, if this is a Directional module. An Omnidirectional module has a similar setup, but the ultrasonic sensor is replaced with the ultra-wideband sensor which is connected to the Qorvo/nRF module through UART. The Omnidirectional module also has the option of receiving data from a separate ESPresense node (an auxiliary ESP32 running Espresense) through an MQTT broker to enhance its functionality by accounting for the short-range limitation of ultra-wideband. All in all, the system has 5 sensors, 2 actuators, and 2 processing elements.



Software Components and Frameworks

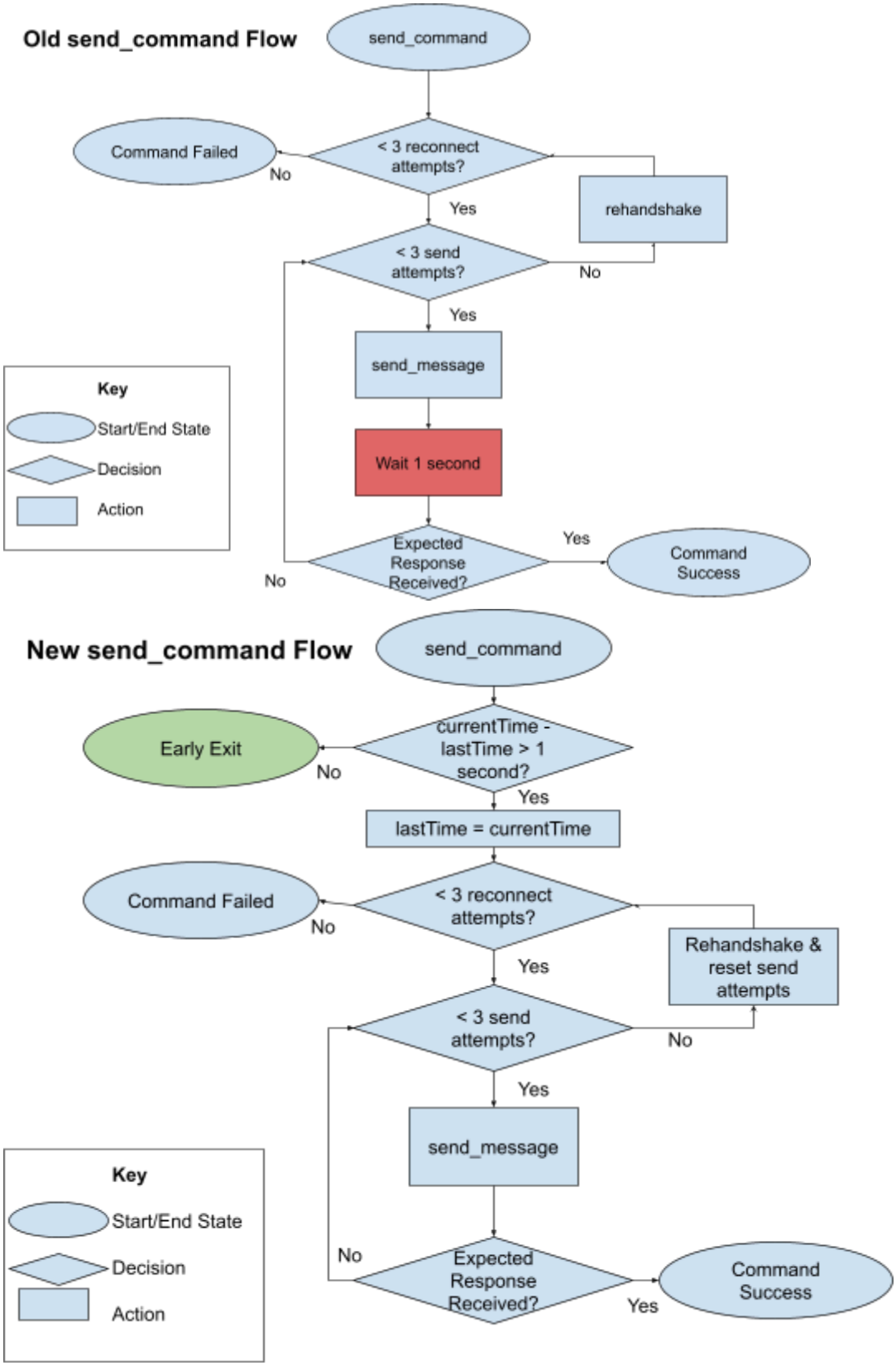
1. Tapo-esp32 (v1) (Unofficial Tapo API Client for ESP32) - This library enabled us to use the ESP32 to control the Tapo Smart plugs. The library reverse-engineered and implemented the KLAP encryption scheme used in the Tapo plugs, saving us a lot of development time. However, by default, the library only supports turning the Tapo plugs on and off, and does not have the functionality to poll the Tapo plug for its energy consumption data. This is the first addition we made, using the information in the python-kasa library as a reference. Another change we made to the library is to refactor it so it does not use delay statements, which hinder prompt updates to the OLED display and make user interaction (from the rotary encoder) impossible. This change will be discussed later in the "Hardware Software Integration" section.
2. Python-kasa (v0.10.2) - We used this library as a reference when we were extending the capability of the tapo-esp32 library mentioned above. This library more extensively reverse-engineers Tapo devices. However, this library is meant to run on more heavy-weight devices that have a command-line interface, like a Raspberry Pi or laptop, not embedded devices like an ESP32. Despite this, we were able to determine the REST API call for polling energy consumption information from the P115 plugs is "get_energy_usage" and format it correctly in order to add this function to the tapo-esp32 library mentioned above.
3. ArduinoJson (v7.3.1) - The Tapo Smart plug REST API calls return information in JSON format and the MQTT messages published by ESPresense also use JSON, so using this purpose-built embedded library to parse this information was a clear choice.
4. HTTPClient (v2.2.0) - This Arduino library allowed us to easily send analytics data from the device to the Raspberry Pi web server using HTTP post requests. We chose to use HTTP for simplicity in designing our data server's REST API.
5. Adafruit SSD1306 (v2.5.13) - This is the library that allows easy use of the SSD1306 OLED displays that we use in our project.

6. ESPresense (v3.3.5) - We ran this software on a separate ESP32 node meant to be used in conjunction with our system. When we were thinking about how our ultra-wideband-based omnidirectional nodes were going to be designed, we thought that using BLE beacons would be a good choice to supplement the ultra-wideband since BLE covers a greater distance in the house and offers a bit of redundancy when ultra-wideband is not working so well. ESPresense has existed for quite some time already and is used in home automation, so we thought it would be good to add compatibility with it to our system, which just meant setting up an MQTT server and making our device an MQTT client to listen to the data published by ESPresense.
7. PubSubClient (v2.8.0) - This library allows creating an MQTT client and subscribing to MQTT topics on ESP32. We use this on our ESP32 devices to listen for MQTT user distance messages published by ESPresense.
8. Eclipse Mosquitto (v2.0.21) - This is a free and open-source MQTT broker that we installed and setup on the Raspberry Pi so that the ESPresense node could publish user distance data detected using BLE beacons. Setup instructions here.
9. MariaDB (v11.7.2) - This is a free and open-source relational database software that we used for our data server. Setup instructions here.
10. FastAPI (v0.115.11) - This is a fast Python framework used for building REST APIs. We used it to design the REST API for our data server. We also use it to host a simple web app used for displaying analytics based on the data we collected on the server.
11. Qorvo Nearby Interactions SDK (v3.2.1) - We leveraged the sample code provided in this SDK to get started with our iOS app and embedded ultra-wideband setup. For the Ultra-Wideband iOS app, we customized the Qorvo NI Background iOS app base code for our product, ShutEye using Xcode, and have added functionality to retrieve analytics data. On the nRF52840-DK board side, we implemented UART communication with the ESP32 chips to communicate ultra-wideband distance data using the hardware abstraction layer (HAL) functions provided in the FreeRTOS project and flashed the updated code using Segger Embedded Studio.

Hardware Software Integration

Converting Blocking into Non-blocking delays

One of the first challenges of integrating the tapo-esp32 library with our hardware system (OLED, ultrasonic sensor, rotary encoder, and flame sensor) was the fact that the original codebase used delay statements to repeatedly send commands to the Tapo plugs to ensure the commands would go through. While this would work well for a system that only sends on/off commands with minimal user input, it would not work in our project. Since we attempt to poll the Tapo plugs for their energy consumption, we would encounter a delay almost every cycle. For example, this would make interacting with the rotary encoder impossible for users if every degree of rotation was then met with a one-second delay while the subsequent send_command function was processing. The flowchart below shows how send_command worked in the original tapo_device.h header file. The red box shows the blocking delay statement. To make this function non-blocking, we instead use the millis() function and store when the last time the send_command was run. The green circle in the new flowchart indicates this improvement, where the new function will simply calculate the difference in time and exit early if the same command is trying to be re-sent too often. In the actual code, the send_command functionality is split into two functions, “update” and “sendCommand”. “sendCommand” simply updates the value in a C struct with the new command values, while “update” actually checks the elapsed times, and sends the command to the Tapo. Combined, these two functions essentially upgrade the old “send_command” function. This split is made to add the notion of priority, which we will discuss next.

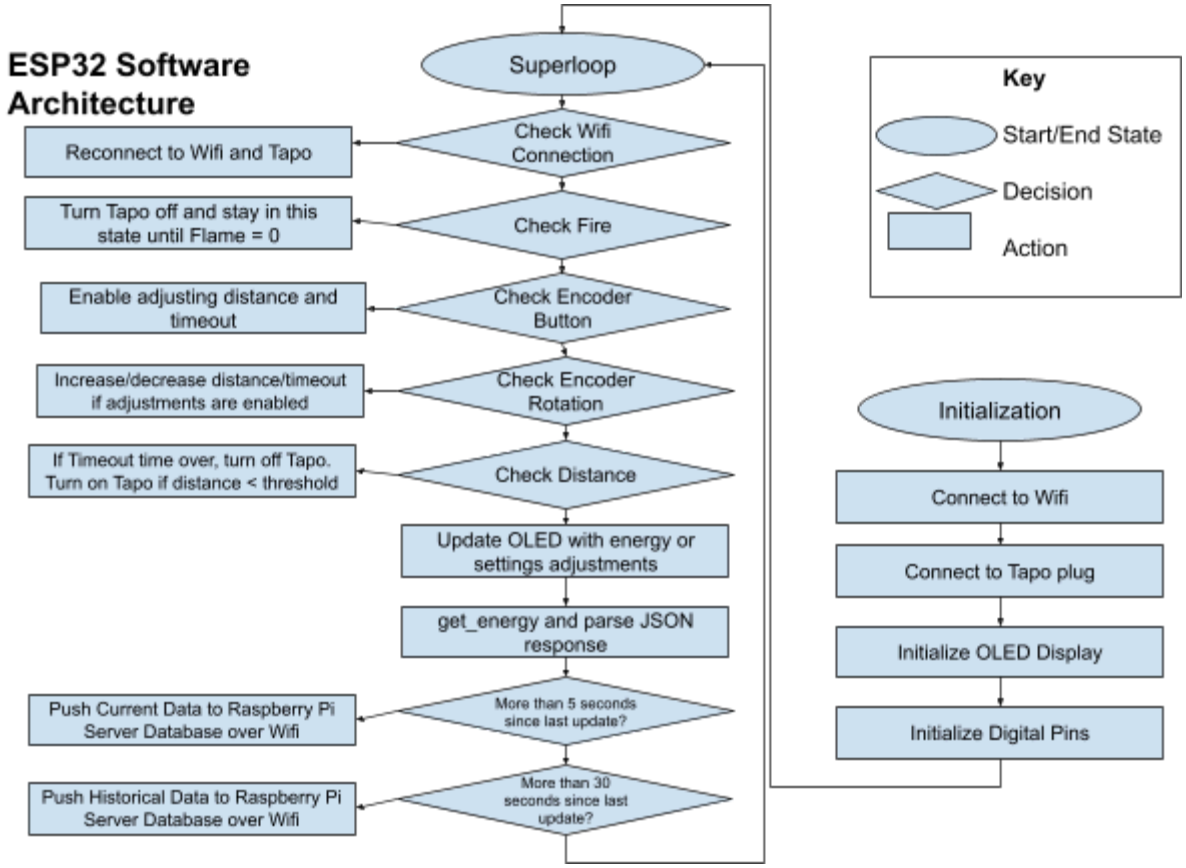

Old send_command Flow
send_command
< 3 reconnect attempts?
Command Failed
No
Yes
rehandshake
< 3 send attempts?
No
Yes
send_message
Wait 1 second
Expected Response Received?
Yes
Command Success
No
Key
Start/End State
Decision
Action
New send_command Flow
send_command
currentTime - lastTime > 1 second?
Early Exit
No
Yes
lastTime = currentTime
< 3 reconnect attempts?
Command Failed
No
Rehandshake & reset send attempts
Yes
< 3 send attempts?
No
Yes
send_message
Expected Response Received?
No
Yes
Command Success
Key
Start/End State
Decision
Action

Priority

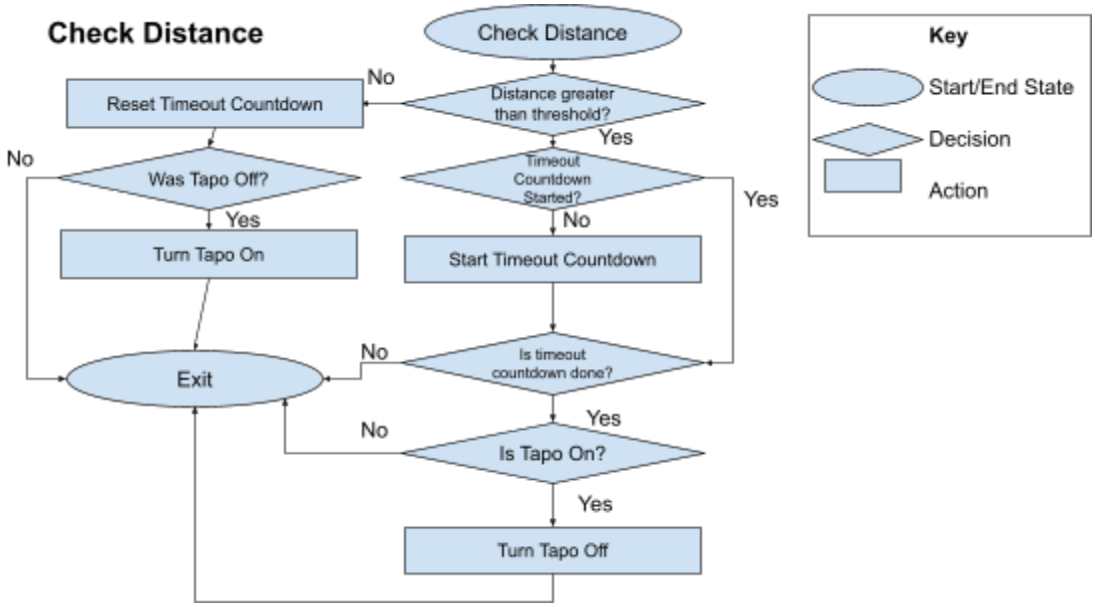
Another change made to the original esp32-tapo library is the addition of priority. Essentially, some commands to the Tapo are more critical than others. If the Tapo receives two commands at the same time or in quick succession, we would like the Tapo to behave predictably. In our project, we decided to add a higher priority to the on/off commands for the Tapo rather than the get_energy commands. We only send the on/off commands to the Tapo once so we do not overwhelm it. For example, if we detect the user is far away, we do not continually send the "off" command to the Tapo, since then it would never execute a "get_energy" command to get the usage data. Since we only send the on/off commands once, we would like to make sure they don't get inadvertently overruled by an upcoming get_energy command. Therefore, we set the priority of get_energy to 0, off to 1, and on to 2. (The difference between the priority of on and off is not significant since these commands are rarely run in close succession). When the Tapo turns on or off from timing out if a person is far away, the on/off command is passed to the sendCommand function, which will override the get_energy command if there is one (since its priority is 0). Likewise, if the next command is get_energy, it will be ignored unless the on/off command has been completed (since priority 0 is less than 1 or 2).

Initialization and Superloop Architecture

In our initialization step, we first connect to the Wifi network the Tapos are connected to, next, connect to the Tapos, and finally initialize the digital pins for the sensors and initialize the OLED display. The two Tapo plugs were allocated static IP addresses of 192.168.8.9 and 192.168.8.18 using our travel router for ease of connection. The Tapo plugs also first needed to be set up using the Tapo app for the Tapo protocol to work correctly, as we use our app username and password to authenticate to them. The rotary encoder, OLED, flame sensor, and ultrasonic or ultra-wideband + BLE act in a superloop. These elements try to react and update immediately to new user inputs. For example, if the distance detected by the ultrasonic sensor changes, the OLED will update, and the timeout will start counting down if the distance is long enough. Likewise, if the user presses the switch on the encoder, the settings menu will immediately display on the OLED, and the settings update upon the rotation of the rotary encoder. If the timeout period is complete, the ESP32 will immediately send a command to shut off the Tapo and will turn back on if the distance detected is below the threshold again (these override the priority of the get_energy function and therefore bypass the 1-second pause between commands unless two of the same commands are sent in a row). The main functions that are periodic are the polling of energy (once every second) and the upload of the data to the Raspberry Pi Server (5 seconds). The lower the period is, the more often commands are sent to the Tapo, and the more data is sent to the Pi. We can prevent overwhelming both systems by having the period be a couple of seconds. We have decided that a few seconds of old data (from switching a device on/off) are an acceptable tradeoff to having to poll/update very often.



Most of the functions in the above diagram are self-explanatory, such as updating the OLED, or checking the encoder/flame data. We will elaborate on the “Check Distance” function in the below diagram since it incorporates both a timeout effect and the distance being currently measured.

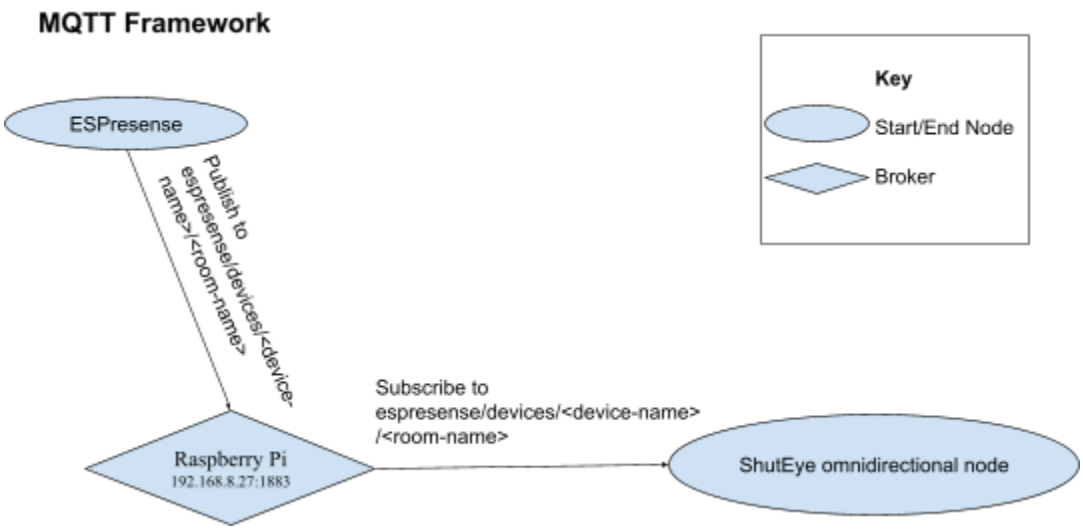


Ultra-wideband Integration

For the ultra-wideband portion of our system, we leveraged the physical hardware of the Qorvo UWB3000 + nRF52840DK on the IoT device side and the Bluetooth and Ultra-wideband chips inside an iPhone 15 on the mobile/user side. To get these two hardware platforms to talk to each other, we leveraged an iOS app that is granted Bluetooth and Nearby Interactions permissions and embedded software compiled using Qorvo's SDK. Going into this process we had no prior experience in iOS app development and had never used Qorvo's SDK before which made it a bit intimidating. We addressed this issue by really taking our time browsing through the two software projects to better understand the purpose of each component, allowing us to determine what was available to us as developers leveraging the API and demo code provided. Through this process, we were able to discover the HAL API provided by Qorvo on the embedded side, which had functions for setting up UART (deca_uart_init) and transmitting data (deca_uart_transmit) in a file called HAL_uart.c. We were also able to discover the exact file where the ultra-wideband distance is parsed and logged on the embedded side, which is called fira_niq.c. Putting this information together, we set up UART in the main.c file using the init function call, then called the UART transmit function from a function in fira_niq.c called report_cb where the ultrawide band distance was logged. Our README is linked here for more detailed information if desired. From there, it was simple to set up an ESP32 as a UART receiver, connect the physical UART TX and RX pins, and test the setup out, which was a success! On the iOS side, we left the logic mostly untouched, save for changes made to customize the branding of the app to our company, ShutEye. We did however add a button later on for pulling up another view for fetching analytics data, which will be discussed in the analytics and web server section.

ESPresense + MQTT Broker Integration

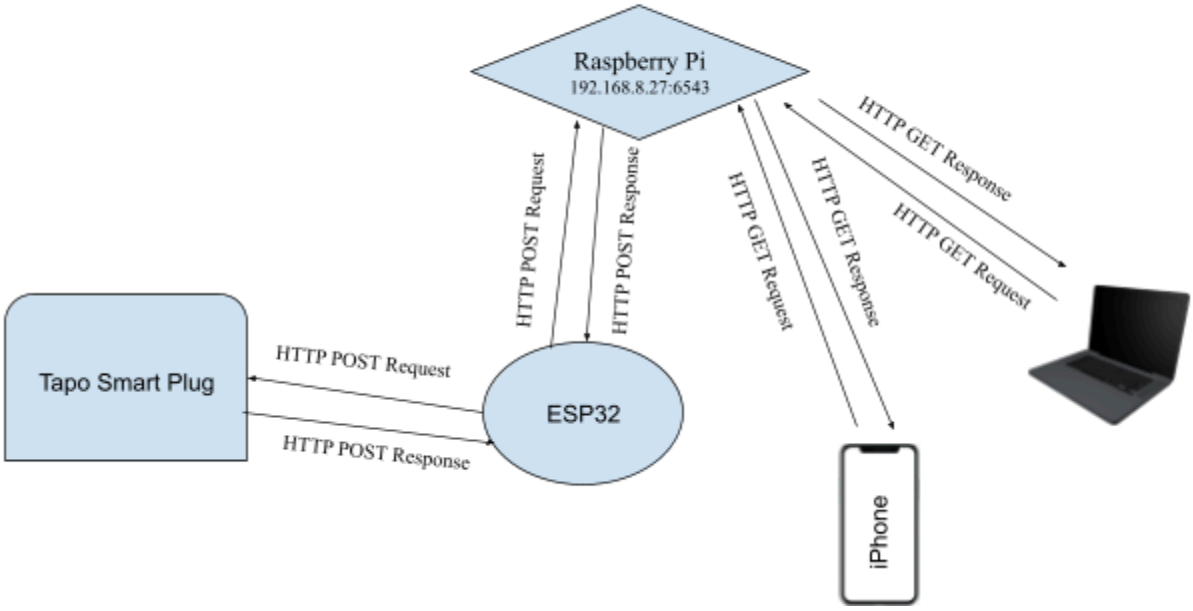
In order to use ESPresense, we needed to first set up a MQTT broker. We searched online, and it seemed that an easy-to-use open-source option was Eclipse Mosquitto, so we searched how to set it up on Raspberry Pi and successfully set it up on the Raspberry Pi hardware (on the default port 1883) without much difficulty following the steps outlined here. We also downloaded a software called MQTT Explorer on our laptop that we used to monitor the ESPresense topic on the MQTT broker to verify that ESPresense was working correctly. Our Raspberry Pi was allocated a static IP address of 192.168.8.27 on our travel router, so we used that as our MQTT host in MQTT Explorer, as well as on the ESP32s. For ESPresense, setup was pretty simple. We chose to go with the simpler approach of using one node per room simply because an ESPresense node per appliance would be overkill if there were multiple appliances in a room. We flashed the firmware available on the ESPresense GitHub using the PlatformIO extension in VSCode, connected to the ESPresense WiFi station access point, and set up the node with our travel router WiFi network details and the MQTT broker details using the captive portal screen that popped up. From there, we followed the setup instructions on the ESPresense website for registering our test iPhone as a tracked device.

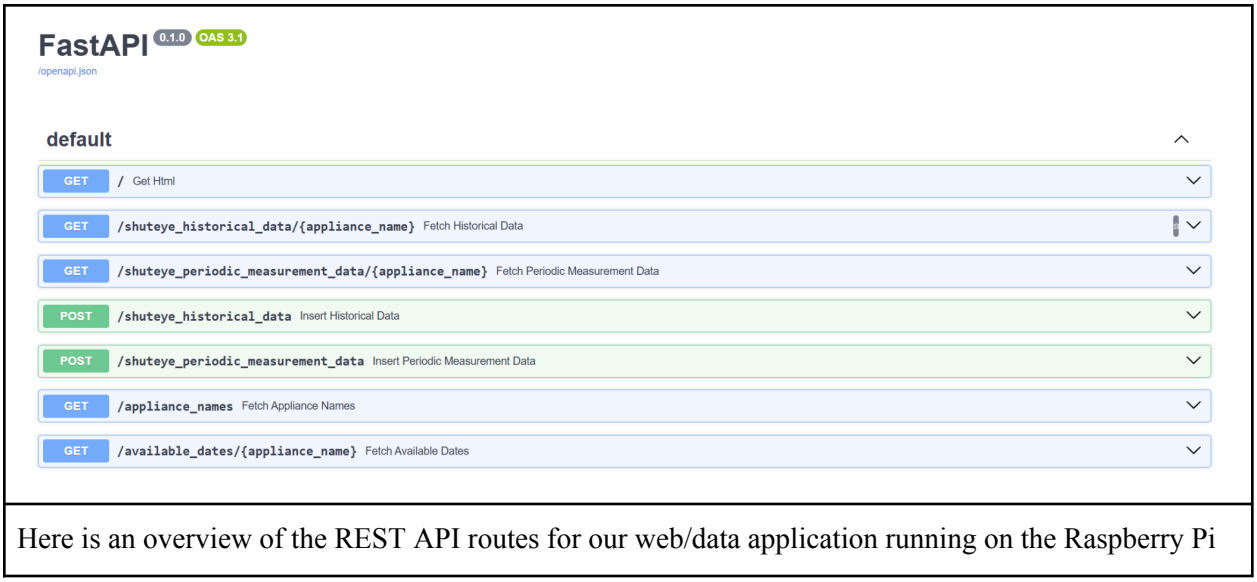


Web/Data Server + Analytics Integration

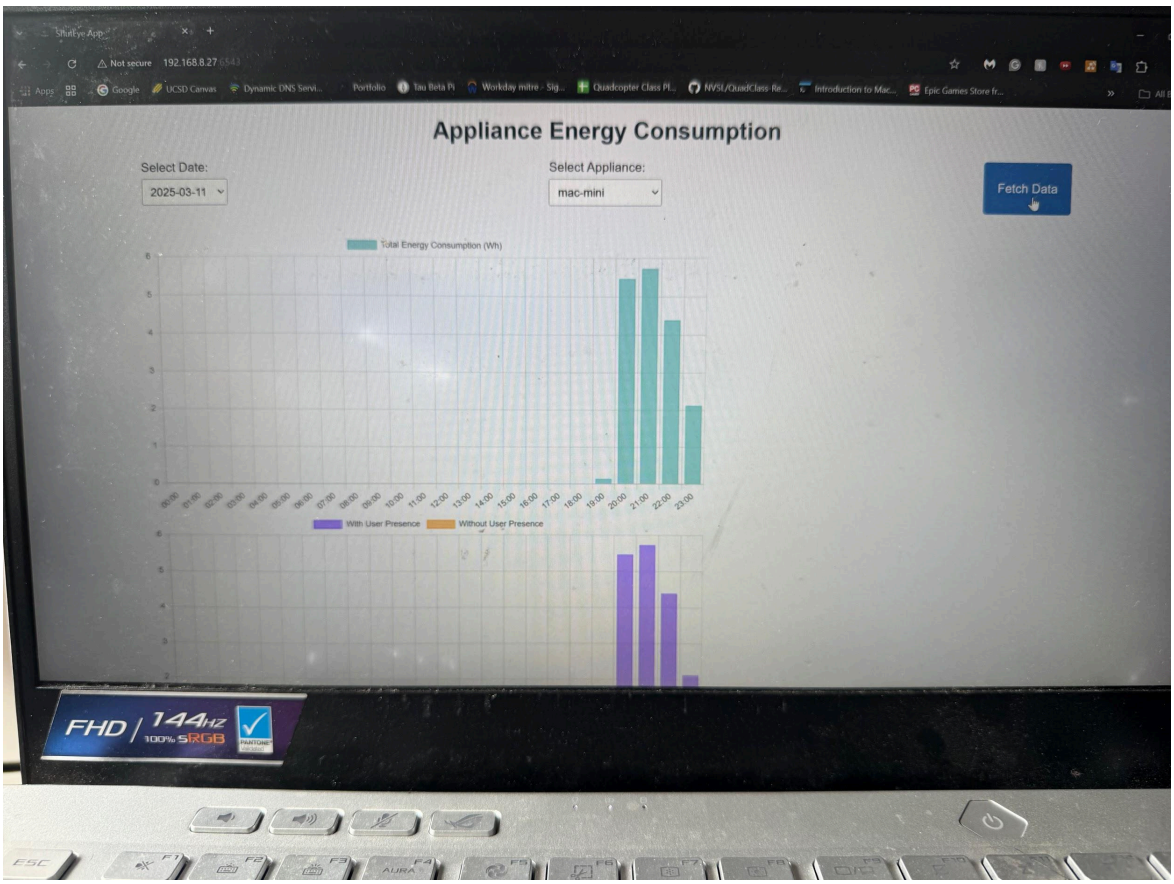
To get started with our data server, we first needed to install the free version of MySQL, MariaDB, on the Raspberry Pi hardware, so we followed the instructions here to do so. We had some prior experience with FastAPI and MySQL, so we were able to quickly write a Python script with all the required REST API routes for our web and data server and then got the server running pretty smoothly. Most of our time in this step went into fine-tuning the data processing code and UI display logic simultaneously in the JavaScript code of the web app and the Swift code of the mobile app, since we needed to ensure that first of all we were correctly aggregating and displaying data values, but also that the analytics were in an efficient visual form we could utilize for our report. Our application-level software on the server processes the sensor data analytics and passes it to a web interface that shows proximity sensor data and energy consumption at once, which allows users to see which devices consume a lot of power when actively used, or which devices consume a lot of power when not in use.

HTTP Framework

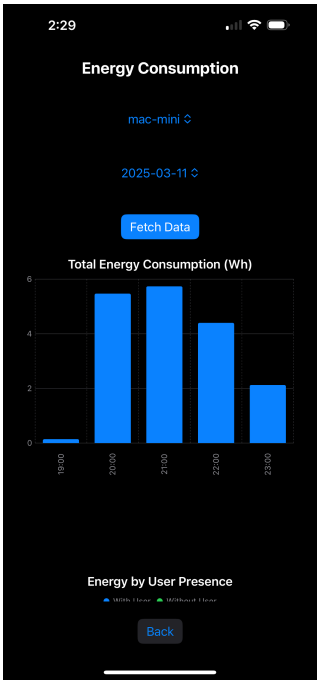




Here is an overview of the REST API routes for our web/data application running on the Raspberry Pi

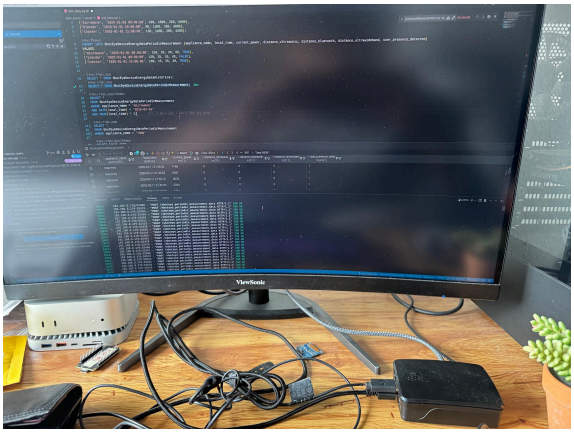


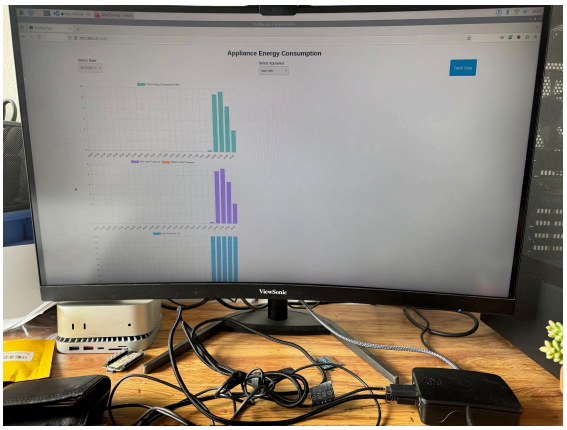
Here is a picture of us accessing the analytics dashboard on the web app from a different computer



Here is a picture of us viewing the same analytics data in our mobile app

Experiments

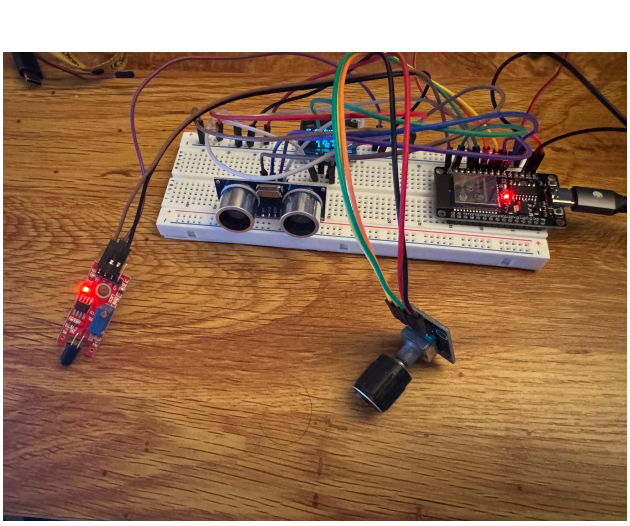


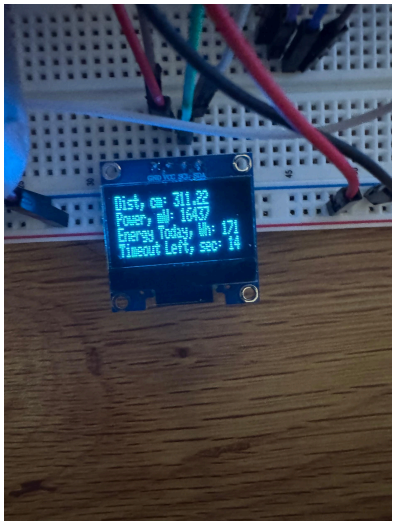


Pictures of the Raspberry Pi data server setup

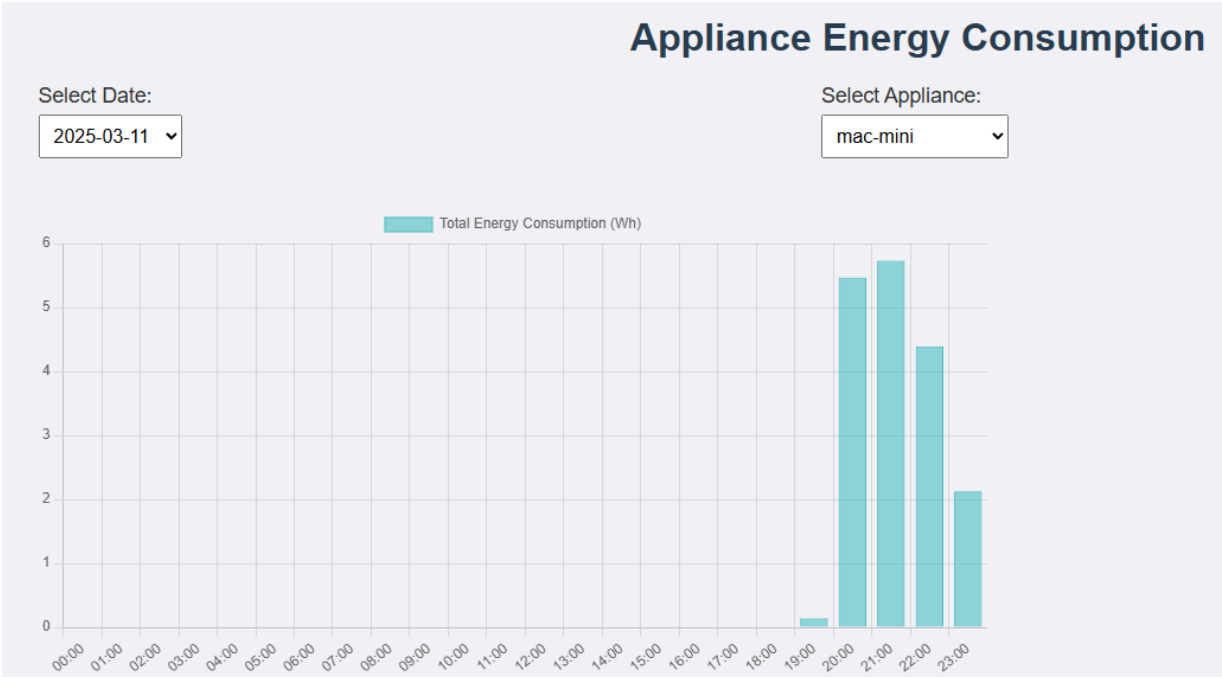
Directional
We were able to perform two experiments, one using the directional setup (with the ultrasonic sensor) and one with the omnidirectional setup (with the ultra-wideband sensor). For the directional setup, in one trial, we monitored the energy usage of a Mac mini over the course of the day with no energy-saving techniques. In another trial we enabled our ultrasonic sensor to turn off the Tapo plug of the Mac it was

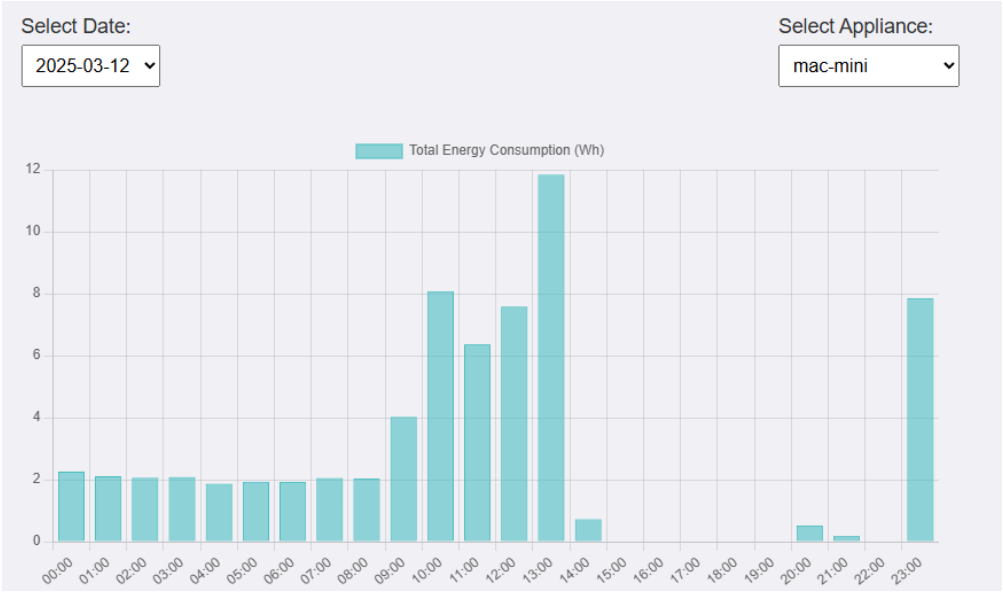
connected to. This allows us to save the energy the Mac consumes when it is asleep during the day while the user is outside (going to work or school for example) or asleep.



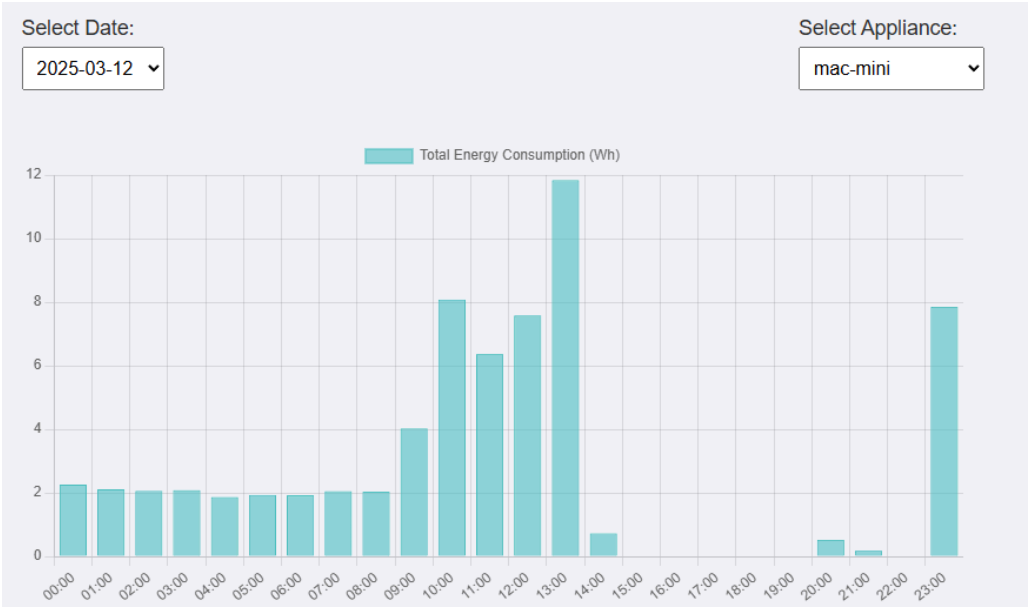


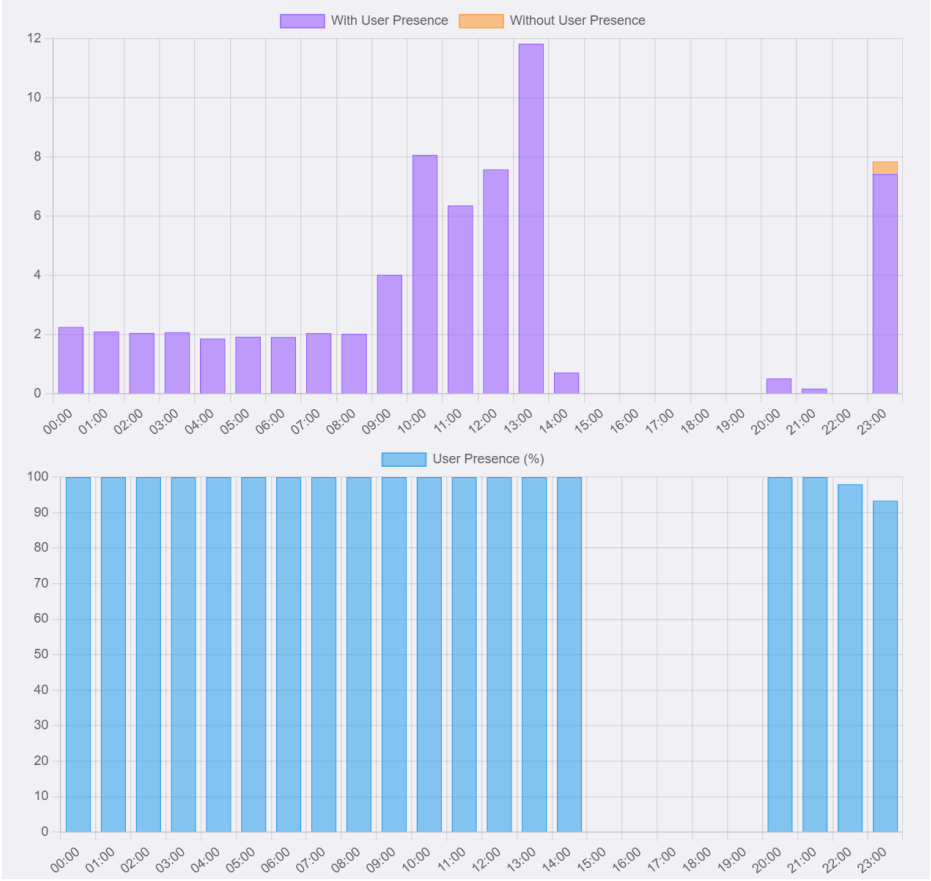
Left: The ultrasonic setup with the rotary encoder, OLED, flame sensor, and ESP32.
Right: A closeup of the OLED showing the default information screen.





This is our baseline for the Mac mini experiment, where the user switches on their Mac mini after coming home from school. Each bar represents the total consumption that hour. There was no energy usage earlier in the day because the Mac mini was off. Starting around 7:50 PM, the user begins using the Mac mini, indicated by the higher spikes in energy consumption. Their Mac mini then went into sleep mode overnight due to inactivity when they went to bed (10:40 PM), which consumed about 2 watts of average power when idling. The user then wakes up and begins using their Mac mini again around 9:35 AM, which is why the total energy consumption spikes.



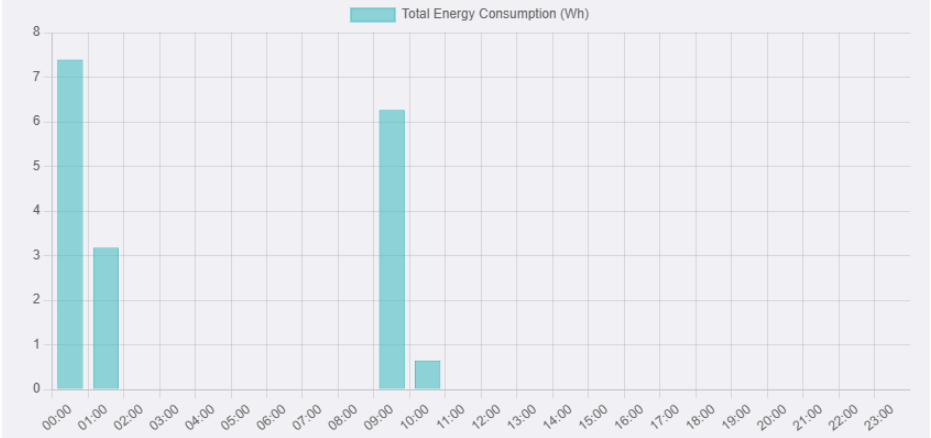


Select Date:

2025-03-13

Select Appliance:

mac-mini



Top two graphs: The total energy usage of the Mac mini with the ultrasonic system active
Middle graph: The total energy, with breakdowns for when users are present (purple) or not (orange)
Bottom graph: The percentage of time each hour when someone is near the device.

The user begins continually using their Mac mini on March 12 at 11 PM, so we use that start time as a reference and continue our measurements into the early morning of March 13. When the user was not near the device (they had gone to bed), the ultrasonic detected it and switched off the plug connected to the Mac mini. This happened around 1:30 AM, so the user was present for about half of that hour, and the in-use energy consumption spike matches that at a little over 3 watt-hours of energy consumed. Between 1:30 AM and 9:00 AM, the plug was switched off by our directional module, so no energy was consumed. The user then begins using the Mac again from 9 AM to 10:10 AM. The directional node was switched off at that moment (at 10:10 AM) and did not take any more measurements, which is why the user presence percentage is 100% for that hour since the user is present for 100% of the measurements collected for that hour.

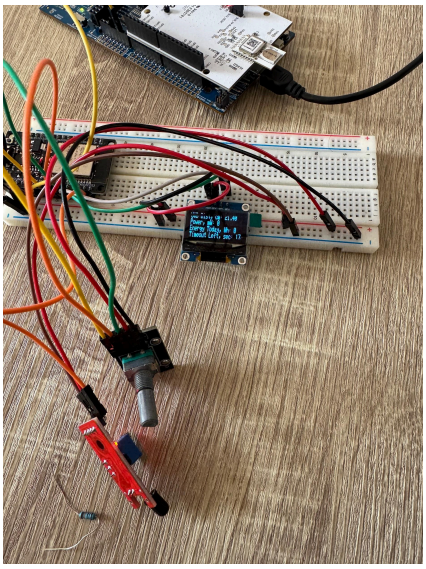
The graphs above show the results of the directional Mac mini experiment. For the sake of the experiment, we will assume an average active power consumption of 7.5 W (actual power consumption may vary based on how the Mac mini is being used) and an idle/sleep power usage of 2 W, which we determined using our baseline data. We will assume that the user actively uses the Mac mini for 4 hours in the morning and 4 hours in the night for a total of 8 hours of use for the first two entries in our table, and the rest of the time, the Mac mini is sitting idle while the user is either at school or sleeping. This use time

is an approximation and may vary based on the classes the user has that day (assuming they are a student), their workload that day, and their sleep habits. With this assumption of 8 hours of active use and 16 hours of idle time, with the Mac mini left on the entire time, the calculation for the first row of the table is derived at such: (8 hours active)*(7.5 W average active power consumption) + (16 hours idle)*(2 W average idle power consumption) = 60 Wh + 32 Wh = 92 Wh total. For the second row, we assume that the user leverages the timer functionality of the Tapo plugs to set a timer to turn the plug off between 2 AM and 9 AM since the user is unlikely to be using it during that period. This saves 2 Wh of energy for every hour the plug is off, which is 7 hours, saving 14 Wh of energy total, or 15.2% of the original total energy consumption. Now for the energy consumption recorded using the ShutEye system. We used ShutEye from 9 AM on March 12 to 9 AM on March 13 and recorded the energy consumption over that timeframe. One thing to notice is that one of these bars is a very large value of 12 Wh of energy, which may be attributed to the fact that the user was watching a video on their Mac mini during that time. However, there are also smaller bars when they are doing other tasks. This is why we maintain that 7.5W of average active power consumption is an approximation, as it can fluctuate quite a bit based on what the user is doing. Regardless, our system shut off the power to the Mac when the user left their desk to go to school at around 2:15 PM on 3/12 and turned the power back on around 8 PM when the user got back to their desk. However, since the Mac mini was powered off using the plug, the power button on the Mac mini needed to be pressed, and the user only did this around 11 PM since they were doing other work until then. The user then used their Mac mini until 1:30 AM on the morning of 3/13, with a break to use the restroom early on. The user then retired to bed, and the power to the Mac mini was shut off until 9 AM of 3/13 when the user woke up and came back to their desk. Altogether, adding up all of the energy consumption bars between those hours gets us a total of about 58 Wh of energy. That means our system saved about 34 Wh from leaving the Mac mini on continuously, or 37% of the original total energy consumption. Our system saved 142.9% more energy than simply using the Tapo timer feature! Since our goal was to have 20% additional energy savings, we have met our goals.

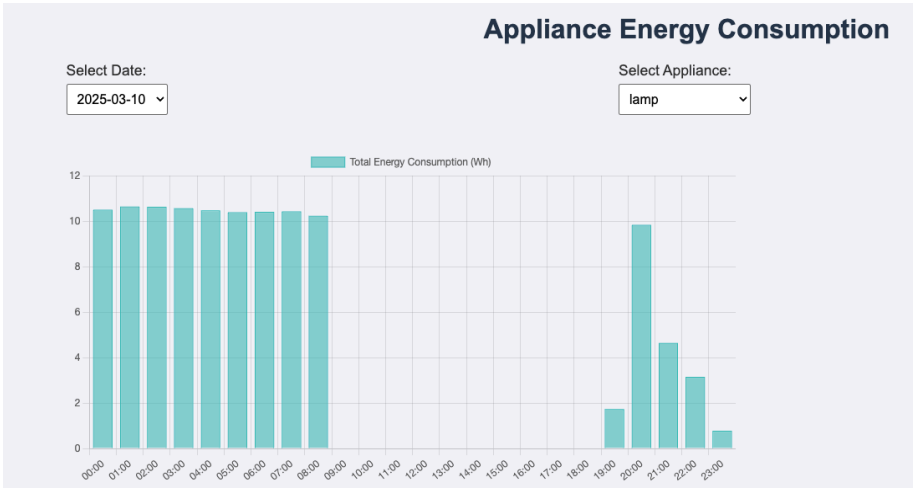
Situation	Approximate Energy Usage	Energy Saved	Savings (%)
Leave Mac mini on continuously throughout all 24 hours of the day (sleep mode when not in active use)	92 Wh	0 Wh	-
Turn the Mac mini off between the hours of 2 AM to 9 AM using a timer	78 Wh	14 Wh	15.2% from leaving on
ShutEye (from 9 AM 3/12 to 9 AM 3/13)	58 Wh	34 Wh	37% from leaving on, **142.9% additional savings!**

Omnidirectional

In another experiment, we used the omnidirectional setup with a lamp set up in a living room. In the first trial, we leave the lamp on overnight. This represents a case where someone forgets to turn off the lights, as is often the case. In the second trial, we use the ultra-wideband sensor to detect when someone is near the light and to turn it on/off accordingly. Since ultra-wideband measures proximity, not motion, it does not have the weakness that common infrared detection devices which sometimes intermittently turn off if someone doesn't move often.




Left: The breadboard setup with the Qorvo ultra-wideband board connected to the nRF board
Right: The setup in context with the lamp being controlled in the background.



This is our baseline for the lamp experiment, where the lamp is left on accidentally overnight, but then shut off as the user leaves their residence.

Top graph: The total energy usage of the lamp with the ultra-wideband system active
Middle graph: The total energy, with breakdowns for when users are present (purple) or not (orange)
Bottom graph: The percentage of time each hour when someone is near the device. For example, at 00:00, someone was near the lamp 90% of the time, which means no one was there 10% of the time. Notice that this time's corresponding energy usage is lower than 01:00 or 02:00 where someone was near it 100% of the time.

The graphs above show the results of the omnidirectional lamp experiment. The lamp was placed in the living room, and on March 10th, it was accidentally left on overnight. When the user woke up, they turned off the lamp before heading outside at 8:30 AM. When they came back around 7:40 PM, they started using the lamp again. Now let's compare this to March 11th, when the ShutEye system is active. On March 11th, our user was working throughout the night, until 3:10 AM (typical for college students sometimes). When they went to bed, they exited the living room, and by doing so, the ultra-wideband sensor no longer detected their presence and shut off. At 8:30 the user woke up and left the residence, once again, the ShutEye system no longer detected them and thus shut off the lamp until about 8:00 PM when they returned. Since user behavior between 2 days can be inconsistent, we will focus on the data from 00:00 to 18:00, which are the most consistent periods of activity. First we notice that the lamp uses about 10Wh of energy each hour. Therefore if the lamp were left on from 00:00 to 18:00, it would've used 180 Wh of energy. However, we can mitigate some energy usage with user behavior (remembering to turn off the lights), or with a timer, since the user consistently leaves their residence at 9:00 am. With this rudimentary measure, the lamp uses 90 Wh of electricity and saves 90 Wh. Now, our ShutEye system turns off the electricity at 3 AM when the user leaves the room, and turns it back on at 8 AM when they come back. Therefore, they now use about 55Wh of energy instead. As the table below summarizes, simply timing turning off the lights saved 90Wh of energy. If we wanted to meet our goal of 20% additional energy savings, we'd have to save 90*1.2 = 108 Wh of energy. However, ShutEye actually saved 125 Wh of energy, a 38.8% increase from 90Wh! Therefore in this experiment, we have met our goals.

Situation	Approximate Energy Usage	Energy Saved	Savings (%)
Leave the lamp on from 00:00 to 18:00	180 Wh	0 Wh	-
Turn the lamp off at 8:30 AM manually or with a timer	90 Wh	90 Wh	50% from leaving on
ShutEye	55 Wh	125 Wh	69.4% from leaving on; **38.8% additional savings!**

Responsivity

We also perform measurements on the responsivity of the on/off commands of the Tapo. We use a stopwatch to approximately measure how long it takes for the Tapo to turn on/off in 5 trials. In each trial, we move away from the ESP32 to trigger the timeout and subsequent shutoff, and then move close to the ESP32 to trigger the tapo "on" command. Additionally, we measure how long it takes for the Tapo to turn off in response to a flame sensor activation to evaluate our safety system. We find that the response is in the magnitude of a few seconds, which is acceptable considering the system communicates to the Tapo wirelessly over Wifi. Note that the timeout period we set by default is 15 seconds. Therefore the average for turning off the Tapo is about 16.7 seconds, which means it actually takes about 1.7 seconds to actually turn off after the timeout period is over. Since turning the Tapo off in response to flame detection is a safety feature, we bypass the timeout period in this situation.

Responsiveness Experiment Data

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
Tapo On	2.21 sec	1.98 sec	1.60 sec	2.06 sec	1.46 sec	1.86 sec
Tapo Off	16.93 sec	16.67 sec	17.13 sec	16.65 sec	16.60 sec	16.7 sec
Tapo Off (flame)	1.66 sec	1.70 sec	1.78 sec	1.72 sec	1.61 sec	1.69 sec

Conclusion

In this project, we have introduced our product named ShutEye. This system allows users to extend their Tapo smart plugs with new capabilities not currently present in the market. Smart plugs can currently be controlled remotely, monitor energy, and even offer rudimentary automation in the form of timers. ShutEye allows users to now use proximity sensors to save even more energy throughout the day when appliances are left running accidentally, or on idle mode. Our proximity data collection also empowers users to see which appliances use lots of power when they are not near it to make more informed purchasing choices for future more energy-efficient appliances. We were able to demonstrate that ShutEye can save over 20% more energy compared to using the Tapo's timer feature through our experiments with the lamp and Mac Mini. We also successfully tested the responsivity of our system and found it to be acceptable.

Now to dive into how our project demonstrates key concepts from this class. In our project, we leverage three of the serial communication protocols discussed in class, UART, I2C, and SPI. We leverage UART to talk to the Nordic board with the ultra-wideband sensor mounted since it is the simplest to set up and we only want a point-to-point link. We also want to get fast transmission of distance data, and UART can be faster than I2C depending on the baud rate used. The SSD1306 OLED screen is designed to be used in a system with many other sensors, which is why it makes use of I2C, which is bus-based and allows for many devices on the same bus as long as they have different addresses. The connection between the ultrawide-band evaluation board and the Nordic board leverages SPI because SPI has very high transfer speeds and the dedicated clock signal of SPI makes it more reliable for wireless communication, where timing is crucial, which is why SPI is used commonly for RF chips. We also leverage two wireless communication protocols discussed in class, Bluetooth and WiFi, while also making use of a third not discussed, ultra-wideband. In modern embedded systems, particularly IoT devices, some sort of wireless communication is usually present, since they make data collection and device monitoring much simpler. WiFi and Bluetooth are pretty ubiquitous as they have become much cheaper to implement, and we thought ultra-wideband would be cool to use since it has seen some major developments recently, particularly in the smartphone space, and it fits the needs of our project pretty well. Our system also has a safety feature making use of the flame sensor which we thought was pretty important because of how important safety is in embedded systems. Another key feature of embedded systems is energy efficiency, and our system has been designed with that in mind. Our nodes make use of ESP32s instead of Raspberry Pis, which are power-efficient yet WiFi-capable microcontrollers, saving us a lot of energy in terms of how much our system consumes. In addition, our system's purpose is to help users save energy in their homes, which it successfully accomplishes. Another topic we discussed in class was sensors and actuators, which determine how embedded systems interact with the world, but also how people can interact with embedded systems. We made sure to address user interactability as a feature some embedded systems have, through the use of our rotary encoder, OLED screen, and web and mobile interfaces.

Some future directions for this project include using the app or website to control the sensor settings (like the distance threshold or timeout period). This would add a new layer of complexity to the project since

the ESP32s will now act as receivers to REST API requests (in addition to transmitting data). Another future avenue is to exploit the uniqueness of the individuals nearby using ESPresense and the Bluetooth fingerprint of their device. This way we can offer multiple checks before turning a device on automatically. For example, for a monitor, we currently only check the ultrasonic sensor to see if someone is in front of it to turn the device on/off. In the future, we can make an additional check with the ultra-wideband chip to detect if its owner is in front of the monitor, to prevent inadvertent activations when someone else walks by. Some other features we could include to make this a more finished product would be to use ESP-IDF and FreeRTOS instead of Arduino to make our ESP32 code more optimal and write code for the ESP32 so that when the ShutEye device powers on, it sets up in WiFi access point mode so that the user can connect to it and enter all the details for the WiFi connection and MQTT subscription into its captive portal instead of hard coding it. We would also integrate all of our microcontrollers and sensors onto a custom PCB and design a 3D-printed shell for the system.