University of Notre Dame
EE 41440 - Senior Design
Professor Mike Schafer

Final Project Report
Group 11 - Infrared Remote Control

By:
Nick Osborne Adrienne Niewiadomski
Elizabeth Gonzalez Collin Finnan

May 6th, 2023

Table of Contents:

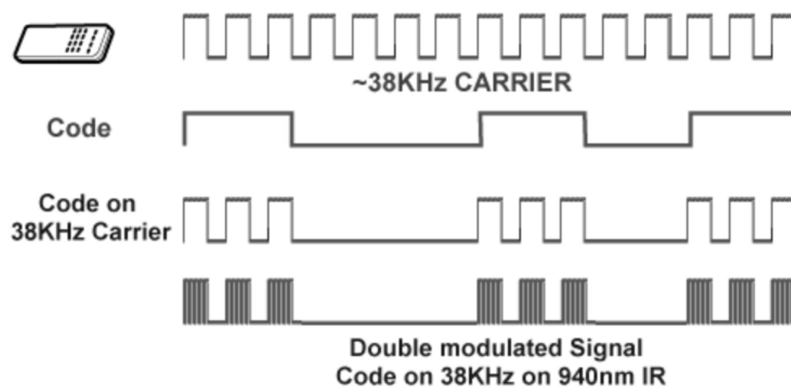
1. Title Page……………………………………………..…(1)
2. Table of Contents………………………………..……(2)
3. Introduction………………………………………(3-7)
4. Requirements……………………………………...(7-12)
5. Detailed Project Description………………………..(12-35)
6. System Integration Testing…………………………(35-37)
7. User Manual……………………………………...(37-42)
8. To Market Changes………………………………(42-48)
9. Conclusions………………………………………(48-49)
10. Appendices……………………………………….(50)

3.) Introduction:

One need not think hard to find issues with their current remote control set up. Oftentimes there are too many remotes for different things, the remotes do not have intuitive hot-keys, the connection is too slow or doesn't have sufficient range, etc. By investigating the fundamental properties guiding a remote control, and building on these properties to create an advanced system, we will provide users with the ability to customize their own remote via an application and / or website that is functional for them specifically.

To begin, we need a brief overview of how a TV remote works. The remote is essentially a transmitter, sending infrared signals to a receiver on the TV. The infrared light is generated via an LED (or more correctly, an IRED since we cannot see the light) with a wavelength that corresponds to that required by the receiver (typically ~940 nm). The LED blinks on and off rapidly, creating binary data that the TV receives and interprets. Because infrared noise exists everywhere (sunlight, body heat, etc.), the IR signal must be modulated to a specific frequency set by the receiver, so that noise is filtered and the signal is amplified. This modulation will be in the range of 32-40 kHz typically (source 1). **Fig. 1** depicts an example signal.

Figure 1. Basics of Remote Control Signal



But how do we generate these signals? We can blink the LED on and off to make nice square waves like this, but how do we know which "code" to make? The answer is unsatisfying, but essentially it depends on the TV being used. For example, many IR receivers use a standardized system like NEC, while others like Samsung TVs have decided to make their own protocol. Luckily, these protocols are quite similar, and both transmitting and receiving are supported within the Arduino architecture (source 2).

We've now introduced the general concepts behind how a TV remote works. We have not, however, gotten into any of the nuances we plan to add to make this primitive model superior. The problems plaguing today's TV remotes have only been accelerated by the advent of smart TVs, where new user freedom has overwhelmed the aged model for controlling the system. The problem description is best seen as a list of problems that correspond to the needed attributes of high level design

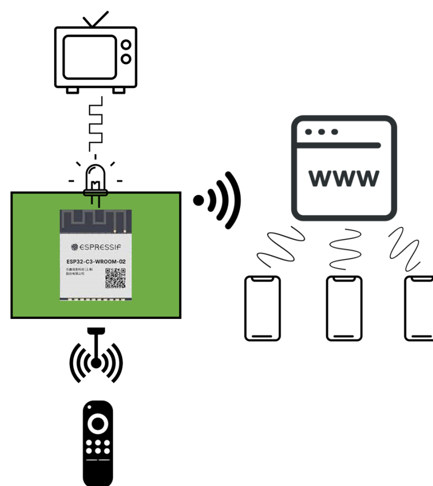
1. Missing remote –
   - There is nothing more frustrating than finishing all your work, sitting down on a comfortable couch, placing your food and drinks on the coffee table in front of you, only to discover the remote is nowhere to be found. Addressing this problem is paramount.
2. Effective range –
   - Although less prevalent in modern remotes, there is always the problem of having your buttons actually transmit. When the battery is low or the angle is just wrong, it can be frustrating to get the TV to do what you want it to.
   - Maybe you want to pause something while outside the room because the task is taking longer than expected. This is currently not possible.

3. Battery lifetime –
    - Transitioning from the previous point, one could also find themselves upset with the battery lifetime of remotes. AAs are used for packaging purposes, but they get used up fairly quickly.
    - There is also no built-in indicator for battery life, so it always comes as a complete surprise.
4. Hot-keys –
    - Today's remotes often come with hot-keys. Want Netflix to pull up right away? Click the Netflix button. But what if you want HBO? Or YoutubeTV? Sure there are ways of programming the remote to pin these to other, unmarked hot-keys, but wouldn't it be nice if I could customize the layout of the remote myself?
    - What if I want a simplistic remote for grandma, which has 4 buttons only, whereas I would like more options with several? Without hiring a universal remote guy to come in, this is impossible.
5. Access to control –
    - There is typically a single TV remote for a whole room. Want to put something on? "Pass the remote!". But tossing the remote across the room could damage the devices inside, and then you're completely out of luck. Fundamentally, you could have as many remotes as you wanted, so long as they weren't being used at the same time. Why limit yourself to having one person in control?
6. External Devices –

- Universal remotes were the fad of the early 2000s, but with new devices cropping up every month, the remote you just paid hundreds of dollars to have set up would become antiquated in less than a year.
- But the idea behind them is still solid. Why have a remote for your speaker and for your TV? Why not put it all in one place? Is there a way to do this that is Easily adjustable for new devices?

Our high level design was influenced by these natural complaints. The proposed solution was a remote rooted in software. A physical PCB with a transmitter would be positioned directly in front of the TV. Users would control the transmitter via a WiFi enabled microcontroller built into the PCB. The microcontroller would host a website emblematic of a standard remote control with the ability to be easily customized in the digital space by users. The PCB would also have an IR receiver, allowing users to program new buttons into their digital remote. The device would be compatible with any IR remote protocol, allowing for compound custom buttons that control various electronics at once. **Fig. 2** provides a simple graphic representing the high level design.

Figure 2. High Level Design of Website Based IR Remote



This proposed solution addresses all the problems highlighted above. The remote would never be missing nor suffer from range issues because the physical transmitter never moves from in front of the television (1-2). The batteries can now be monitored digitally using a readout circuitry from the PCB, so their lifetime is no longer quite as problematic (3). Hot-keys and control over other IR devices is easily addressed using programmable compound buttons; the universality of a website alleviates these common troubles (4,6). Finally, because the base model is a public website, anyone can enter the URL specific to each device and control the TV (5).

The final product achieved a majority of the desired behavior of the high level design. The specifics of what subsystems succeeded and failed will be discussed in later sections, but an overview within the context of the high level design will be provided here. A mounted device that transmits and controls the TV via a universal website was produced, thus fulfilling 1,2,3, and 5. Customizable buttons were produced, but with some limitations. Only 3 buttons were provided for each customizable profile due to memory management for the selected microcontroller. Additionally, there are currently only two supported protocols, NEC and Samsung, again due to memory limitations. The base product is easily adaptable to integrate more protocols, but it would require an external memory unit and more coders than were available. Thus, external devices can be used in the custom buttons, so long as their protocol is NEC or Samsung. The conclusion is that 4 and 6 have been achieved but need to be expanded upon in future projects.

4.) Detailed System Requirements:

The system requirements section has been broken into three sections: functional requirements, user-interface requirements, and safety requirements. The functionality requirements refer to what the system has to do and has nothing to do with the nitty gritty details

of how the system actually works. These are reserved for the user interface requirements, which detail the specific needs of the way humans interact with the device. These requirements are more high-level, and assume that the basic functionality is there. Finally, there are safety requirements ensuring the device's production and usage isn't harmful.

1. Control a TV using the ESP32
    - The most fundamental and basic requirement of the project. A GPIO pin on the ESP32 will output to an IRED subsystem (amplifier + emitting diode, likely).
    - We need to use arduino libraries (reference: https://www.arduino.cc/reference/en/libraries/irremote/), which is compatible with the ESP32, to send codes to a TV receiver.
    - We can prototype this using a breadboard so that work can begin before the PCB is printed and ready.
    - Without being able to do this, the project does not go forward.
2. Control different TYPES of TVs
    - The product is useless if it only works with one type of television. One demonstrated feature is that this device must be quasi-universal.
    - The hardware is universal intrinsically: we just have an IRED subsystem, a microcontroller, and a power supply system.
    - The arduino library has robust support for different protocols: Denon / Sharp, JVC, LG / LG2, NEC / Onkyo / Apple, Panasonic / Kaseikyo, RC5, RC6, Samsung, Sony, (Pronto), BoseWave, Lego, Whynter, MagiQuest.New: Added NEC2 protocol.

    - The only limiting factor: what we do as programmers to implement different versions of code using these different available code bases.
3. Control Via WiFi Based Website
    - The ingenuity of the product comes from the ability to control it via a phone. A functionality requirement must then be that those GPIO pins sending signals are controlled by phones.
    - Our proposed method of doing this is a basic website being published by the ESP32 getting responses from users on their phones.
    - This means the range of the device is simply the range of the WiFi. No one is going to use a TV remote outside their home, so this makes sense for the implementation of the product.
4. Control of External Device
    - Though this is the lowest priority, the proposed solution has immense freedom allowing for the development of universality. Because our ESP32 selection has bluetooth and WiFi, we could easily include support for external devices other than the TV.
    - This functional requirement essentially is a proof of concept: show this solution allows for easier universal remote capabilities than traditional implementations.
    - The types of controllable devices will be limited to IR devices because this is our only actuator subsystem.
5. Memory
    - The code base for IR signals in this device may be quite large at scale. The ESP32 package that we select must have an SPI flash unit with sizable memory.

- Additionally, there will be saved layouts for returning users. If the device loses power, we don't want these users to have to remake their remote / recode their custom buttons! Therefore, the memory device will also be needed to save customized user data.

User Interface Requirements

6. Power
    - Rather than have an additional power cable, the device will be battery powered.
    - Unlike a normal remote, we are not limited by size, so we can throw some additional battery packs onto the PCB. But we do want the design to stay relatively small so no need to go overboard.
    - The ESP32 requires 3.3 V to operate. Likely what we will do is have the batteries and lead them to a regulator that outputs 3.3 V. The input power we've been using this semester is ~5 V from a USB. Our design will take 3 AA batteries in series to power the board.
    - An indicator on the website will report to the user the battery power. Because of this, one of the pins on the ESP32 will need to be an analog input that reports the current value of the battery. When it starts to "droop", the website will alert the user. This requirement is paramount to fulfill one of the major problems facing current TV remotes.
    - There should also be a physical power switch. Typically, we propose the device will just always be on, hence the several batteries in parallel. The ESP32 can be put in an idle power-saving mode which will reduce consumption, but it would be

a hassle for users to constantly have to turn a switch on and off. But, we want that option available (say the family goes for a long trip).

7. Physical Design
    - So far, everything we've discussed could be done on a breadboard. But this is not very durable or elegant. We will make a PCB for our final design.
    - The PCB will have an IRED that must overhang the edge so it is visible. Additionally, the receiver used for custom buttons must have a similar configuration on the other side.
    - A 3D printed, rectangular enclosure will surround the PCB and batteries. The majority of the space will be the power supply, meaning the enclosure will be the same size as the number of batteries that we choose to use.
    - The 3D printed enclosure must have a hole that allows the IRED and the receiver to poke through.
    - The size of the enclosure and PCB must be such that a clip (as seen below) can hold it and attach to the TV.

    - Estimated size will be slightly larger than a hand-held remote, but not by much due to size limitations of the clip / mounting technique.

8. Website

- Quite simply, the website has to be flawless, otherwise this alternative is not worthy of being selected. Therefore, the website needs to be attractive and easy to use. A grandmother should be able to see the website and use it as well or better than she can use her own TV remote.
- The website needs to be able to build profiles. If there is a new user, there needs to be a flow which makes it obvious how to develop a customized button. The button and the profile will be saved under their name.
- The website needs to be able to remember profiles. As discussed in the functional requirements, if someone lives in the home and wants to have their personal remote booted up, the website needs to remember that, otherwise this device becomes a hassle.
- The website needs to allow multiple users at once. A priority queue of sorts will be made which values one user's input above others so interference isn't allowed. This would only arise when both users try to input at the same time.
- The website needs an intuitive bootup. When we first install the ESP32, it won't be on the WiFi. We need an initial way in which the first user enters the network and yields the password. Then, the standard bootup will show two options: "load existing remote, create new remote".

Safety Requirements

- This is a low-power device, so there are no dangerous currents or voltages in the development or operation of the device.
- There are no dangerous pieces of equipment nor machines needed to create this product.
- Lead-based solder may be the only dangerous part of this project.

5.) Detailed Project Description

5.1 - System Theory of Operation:

Though the high-level design version of the system operation was described in the introduction, a more clear description will be presented here. The overarching idea behind the design is to control your TV with your personal devices rather than a TV remote. This solution elegantly solves the majority of the problems listed in the introduction. The implementation can be divided into two very broad groups: hardware and software.

A PCB and enclosing for the board constitute the hardware. On the PCB, an IRED actuator circuit is attached to the output of an ESP32 GPIO. Software defined IR signals are sent to the TV's receiver via this actuator circuit. This constitutes the "remote" functionality of the hardware. Additionally, the PCB has a subcircuit that measures and reports battery power to the ESP32 to be displayed in the software. This addresses battery concerns laid out in the requirements. Finally, the PCB design includes an IR receiver wired to a GPIO pin. This allows customized buttons to be made; a user can point their actual TV remote to the receiver, enter a sequence of commands, and save that sequence as a button on their website remote. The enclosure needs to account for battery packs, openings for transmitted and received IR signals, and attachment to the TV.

The software generally speaking can be divided into three subsections: memory management, IR actuation and receiving functions, and front-end development. Because we need non-volatile memory for user data and storing IR codes, information has to be stored in a SPI based flash that comes with our selected ESP32. See details later in this section. Organizing and

accessing this memory is a challenging task and therefore an entire separate codebase was generated to abstract this process. Next, we needed portions of code designated to sending specific signals. The idea was to abstract this process from the lowest level such that signals could be sent as “POWER OFF” across all protocols. Finally, we used simplistic but effective HTML to create an intuitive website for users to interact with. Very few stylistic leaps were made in the design of the website so that it remained lightweight.

Users interact with the system by button clicking on the website. The website is hosted by the ESP32 microcontroller. When a user clicks on a button, it registers an HTTP request that can be interpreted by the ESP32. Depending on the button, a different type of IR signal will be sent out to the TV. Profile and protocol switching is constituted by webpage navigation. Custom buttons are made by entering a “button name” string into a text field on the site. Doing so prompts a recording functionality where users can point their actual remote to the receiver and enter a sequence of inputs. A binary battery level indicator is found in the bottom corner of the page that tells users if the batteries need to be changed.

NOTE: Throughout this section, we briefly describe how systems were tested individually. However, section 6 is designated to integration testing, which will have more in depth descriptions of subsystem testing too. Subsystem and integration testing were all done on the same PCB, so there was no point in including detailed specifics twice.

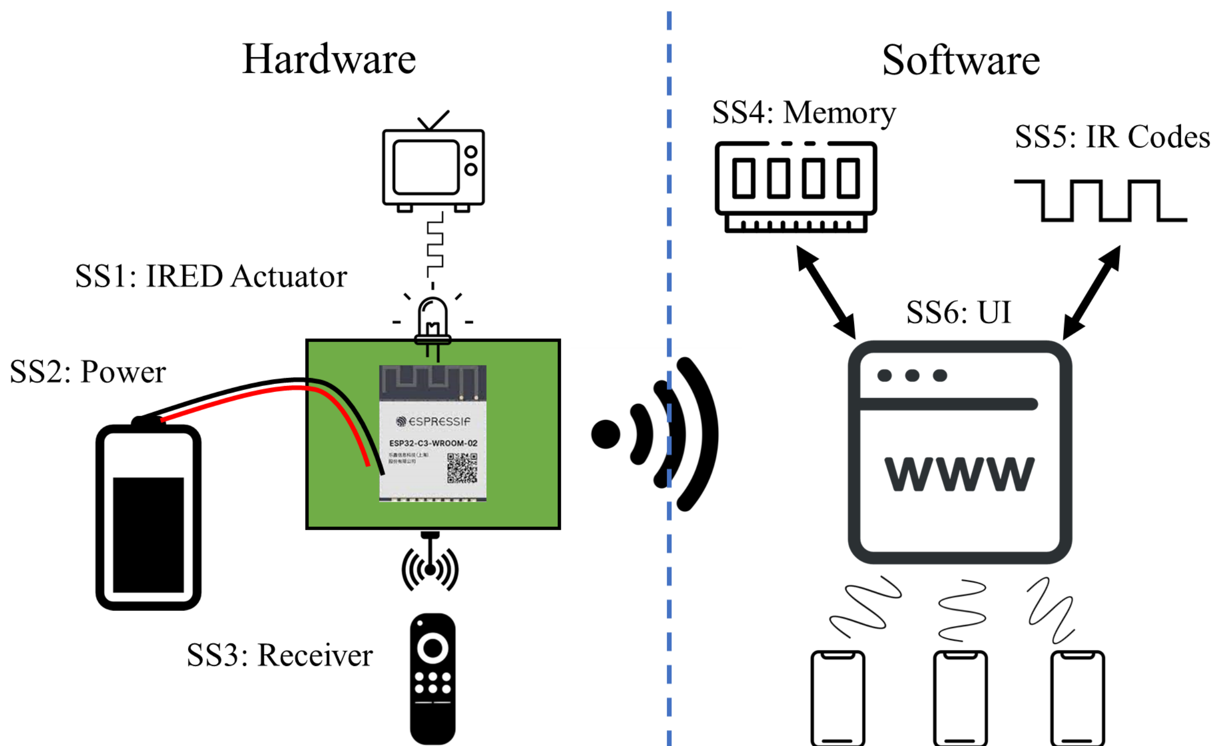
5.2 - System Block Diagrams

The project can be divided into hardware and software. Each of these groups can be subdivided into 3 subsections. Though discussed above, the subsystems (SS) will be made explicit here:

1. SS1 - IRED Actuator. The signal generated by the ESP32 is too weak to drive an IR alone. This is the amplifier + IRED system that connects directly to the ESP32. See details below.
2. SS2 - Power. The board needs power throughout. External batteries provide this power. The power is put through a voltage regulator before being used on the PCB. Additionally, a circuit must be provided to allow for power monitoring without draining the battery. This will be attached to an ESP32 GPIO pin. Finally, the proposed board has the option of being powered via a microUSB for those whose TV's have the ability to output a USB voltage. See details below.
3. SS3 - Receiver. The board needs a receiver, which will require an attachment to the ESP32 and a handful of passives. See details below.
4. SS4 - Memory. Memory management to store IR codes and personalized information is required. Due to the limited memory in this system, special care was taken to efficiently abstract the lower level functionality built into the ESP32. The UI pulls directly from nonvolatile memory to craft its output. Thus, the memory is directly tied to the UI.
5. SS5 - IR Code. Though there is an Arduino package designated to sending IR signals, we want to abstract this process to work across various protocols at the click of a button. This code was then integrated into the UI and attached to specific buttons.
6. SS6 - UI. This is the HTML that users interface with. The buttons are correlated to actual IR code sends and are generated via saved profile data in memory. The layout has default buttons and custom buttons. It is a WiFi based server hosted on the ESP32. Note that this subsystem has two parts: the website itself and the server it runs from.

The annotated diagram is provided below in **fig. 3**. Note that this diagram shows the subsystems as individual components; detailed subsystem information is provided below. The aim of this portion is just to show how they are divided and what function they play in the overall system described in 5.1.

Figure 3. Annotated Subsystem Block Diagram



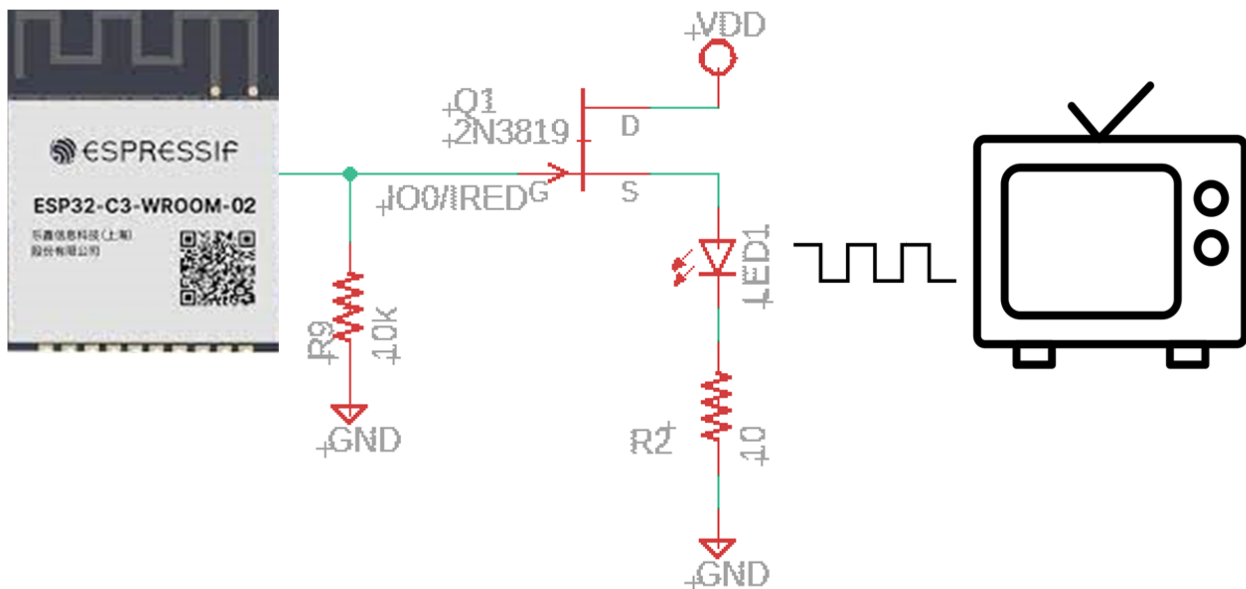
5.3 Detailed Design / Operation of Subsystem 1 - IRED Actuator

The IRED actuator subsystem overall is very simple, though careful detail was needed in its implementation. Generally speaking, the ESP32 cannot drive much current from its GPIO pins. Because of this, we needed to select and wire up not only an IRED but an amplifier circuit as well.

The selection of the IRED was the first step. We decided to go with a TSHG6210. This powerful IRED offers larger range than its contemporaries, having an output intensity of 140

mW per steradian. It is operated at 1.5 V and 100 mA, meaning the circuitry had to be designed around those parameters. Because the signal being sent from the ESP32 is binary (**fig. 1**), we could use a simple gate-controlled MOSFET as the amplifying circuit. We needed an NMOS so that the signal did not become inverting, so we decided upon the 2N3819. This MOSFET had plenty of bandwidth to transmit the carrier frequency of 38-42 kHz. The drop across this MOSFET is ~0.7 V because it is silicon, meaning the total available voltage after the MOSFET and IRED is the regulated 3.3 V output minus 2.2 V. To get 100 mA from 1.1 V, we selected a drain resistor of 10 ohms. The completed circuit and subsystem diagram is provided below in **fig. 4**.

Figure 4. SS1 - IRED Actuator Implementation



The subsystem was generated and tested using a breadboard. The effective range of the IRED was demonstrated to be well over 20 feet, completely sufficient for the purposes of this project. The directivity of the IRED is relatively large, operating well over a 20 degree angle only. Because of this, the enclosure openings need only allow the tip of the IRED to show; very

little power exits from other angles. Another consequence of the directivity is that the mounted device must point directly at the TV receiver much like an actual remote. The design of the PCB places the IRED opposite of the receiver so that it points towards the receiver. A pull down resistor was provided on the gate of the MOSFET to avoid charge accumulation while the subsystem is idle.

5.4 Detailed Design / Operation of Subsystem 2 - Power

The device is powered via an external battery pack containing 3 series connected AA batteries. **Fig. 5** depicts the particular model we selected, the Seeed 18650 Battery Holder Case.

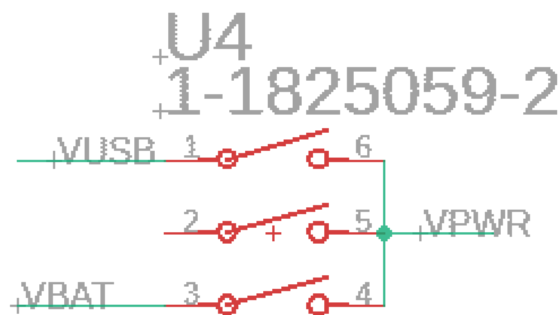
Figure 5. Battery Pack Used in Design

The leads of the pack will be soldered directly to the PCB, and they will all rest within the same rectangular enclosure. The enclosure has an opening on the top that allows for access to the batteries and the PCB. AA batteries are 1.5 V each, so the total input voltage is 4.5 V for this design.

A secondary power option is provided to users in the form of an on board USB mini port. For those that wish to never change batteries, they can provide a wired link from their TV directly to the encasing and plug into the board that way. The USB power line is 5 V, thus allowing for regulators of the same type as the battery implementation. The input voltage is in the same ballpark no matter what. This secondary option is selected via a dip switch. Default

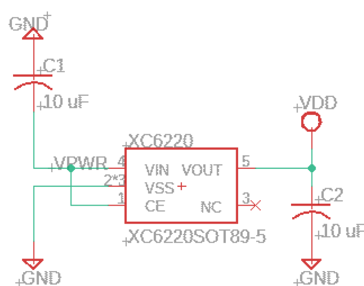
power will be batteries, but when certain users wish to implement the wired connection, they can consult the user manual and find which switch to adjust such that the device runs on USB mini power. The implementation of the selection process can be seen schematically below, where VUSB is the microUSB input voltage and VBAT is the battery input. The output is simply VPWR.

Figure 6. Power Selection Dip Switch



The value of 4.5 V or 5 V was selected due to our regulator. We chose to use the Torex XC6220 linear voltage regulator. The input voltage range is limited to 6.5 V, so we chose to go with AA batteries instead of 9 V batteries. The regulator has an output of 3.3 V, sufficient to power the ESP32, the IRED, and the receiver. The regulator requires a network of passive to power it. See **fig. 7** for the schematic implementation.

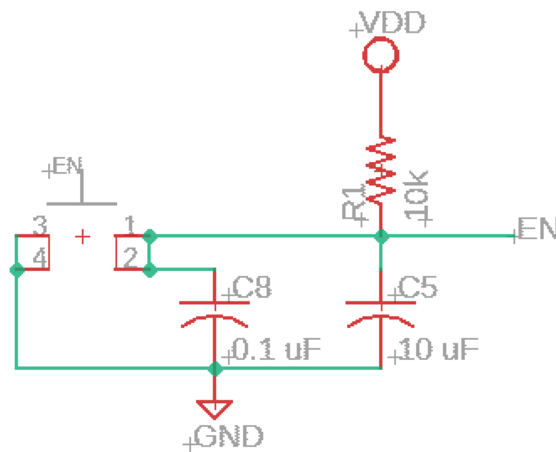
Figure 7. Regulator Schematic



The value and location of passives was decided based on the recommendation of the datasheet for 3.3 V output voltages. This voltage is set to this value because it primarily powers the ESP32.

The ESP32 requires 3.3 V of input voltage and several passive to operate properly upon being powered. Pin 1 is connected directly to VPWR and requires two parallel decoupling capacitors of 10 uF and 0.1 uF. Pin 2 is the “enable” pin, and is attached to a push button circuit that allows users to hard-reset their device mechanically. This circuit requires decoupling capacitors and importantly, a pullup resistor so that enable is set high when the button is unpressed. The schematic for enabling the power of the ESP32 is provided below in **fig. 8**.

Figure 8. Enable Circuit for ESP32 Power



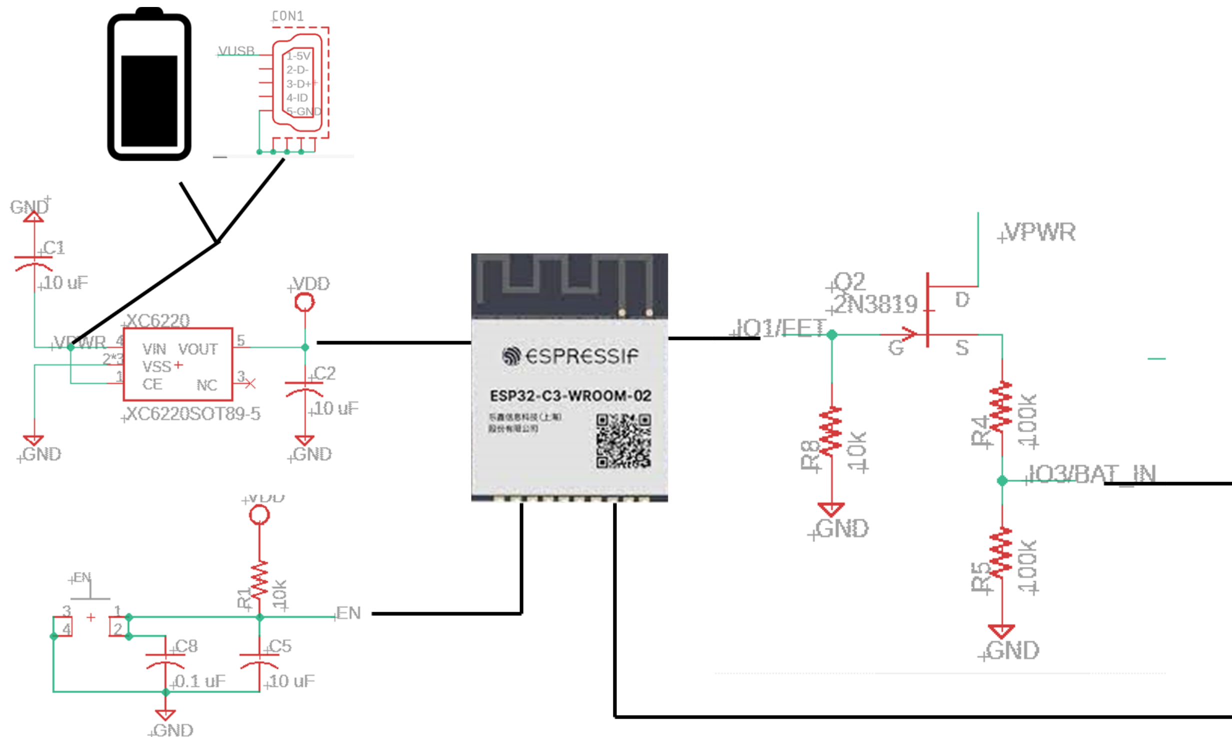
GPIO 9 has a similar push-button circuit to change the type of boot needed. This is included for dev purposes only.

The final thing required within this subsystem is the readout circuitry to monitor the battery level. The input voltage of an ESP32 GPIO pin is ~3 V maximum. Given our battery power is 4.5 V, we need to use a voltage divider to step down this value. This needs to be implemented in a way that does not consume too much power, so a 2N3819 MOSFET was added to the circuit to open and close the voltage divided. We used two 10k resistors to form the voltage divider, with a signal between the signals leading to a GPIO pin on the ESP32. The

selected pin was IO3 because it has a built in ADC for readout purposes. Additionally, another GPIO had to be used to power the MOSFET to allow current flow through the divided. Thus, the “read battery” flow goes as follows: digitally set IO1 to power the MOSFET, allow current to flow through the divider, read the output voltage at the junction with IO3, scale by 2X to account for the divider, output binary status. The threshold for this binary status is 80% charge. This value accounts for any discrepancies in resistor values and losses due to the MOSFET.

This implementation with code was tested and verified using a breadboard. “OFF” current leakage was calculated into a watt-hours value and compared to the capacity of the 3 AA. Leakage power will only constitute 0.004% of total consumption. **Fig. 9** shows the implementation of the readout circuitry within the framework of the entire power subsystem.

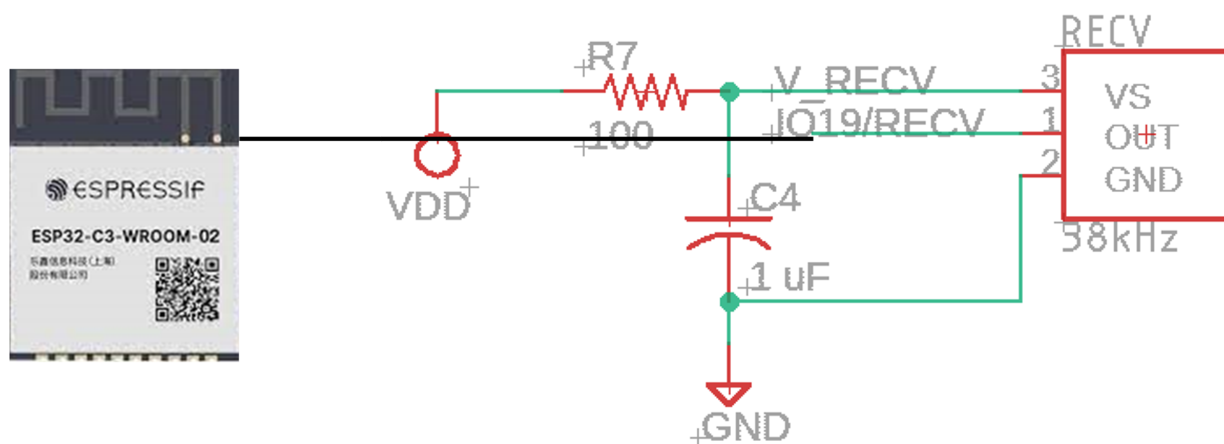
Figure 9. Power Subsystem Overall



5.5 Detailed Design / Operation of Subsystem 3 - Receiver

The receiver we selected needed to work in the range of 38-42 kHz as a carrier. We decided to go with the Vishay TSOP34438. The datasheet for this device came with a recommended circuit layout that we implemented for our design. The device comes with 3 pins: input voltage, ground, and output signal. The input voltage needed a sequence of passive so the device operated in a stable manner. The output was attached to a GPIO pin on the ESP32. See the implementation below in **fig. 10**.

Figure 10. Receiver Subsystem Schematic



The layout and values of passives were taken directly from the manufacturer. The subsystem was built and verified on a breadboard. An IR remote was pointed at the receiver from a distance comparable to a family room. Code was uploaded to an ESP32 to print out the indicated value. Additionally, the IRED circuit seen in previous sections was built. The code automatically outputs the signal to the IRED circuit, thus verifying both circuits. For example, if the receiver was given the “volume down” button from a remote, the ESP32 should read that and output it to the IRED. When the volume went down twice, once from the initial press, once from the IRED, we knew both circuits operated correctly.

Before continuing to the next section, we will briefly present a summary of the parts list needed for this project. Table 1 shows the list of non-passives.

Table 1. Parts Required for IR Remote Hardware

Name	Q	Description	Package	Link
ESP32-C3	1	Microprocessor	MODULE_ESP32-C3-WROOM-02-H4	Link
XC6220	1	Voltage Regulator	XC6220SOT89-5	Link
USB-MINI-B	1	USB-C connector	USB-MINI-FCI10033527	Link
SWITCH-PB4X4	2	Push button	PBSMT4X4M	Not needed
PINHD-1X4	1	4x1 Pin Header	1X04N	Not needed
PINHD-1X10	2	1X10 Pin Header	1X10	Not needed
2N3819	2	N-Type Transistor	TO92	Link
TSHG6210	1	IRED	T-1¾	Link
TSOP38238	1	Receiver	TSOP382	Link
1-1825059-2	1	Switch	SWSO6_325X138P100	Link
LampVPath - 3AA	1	AA Battery Holder	N/A	Link

5.6 Detailed Design / Operation of Subsystem 4 - Memory

Nonvolatile memory is needed to hold IR codes and user data. For our project, we used the "nvs_flash.h" and "nvs.h" packages, designed for the ESP32 nonvolatile memory. Documentation can be found here. The functions in this memory system are very simple and we were required to implement our own abstractions for our specific purpose.

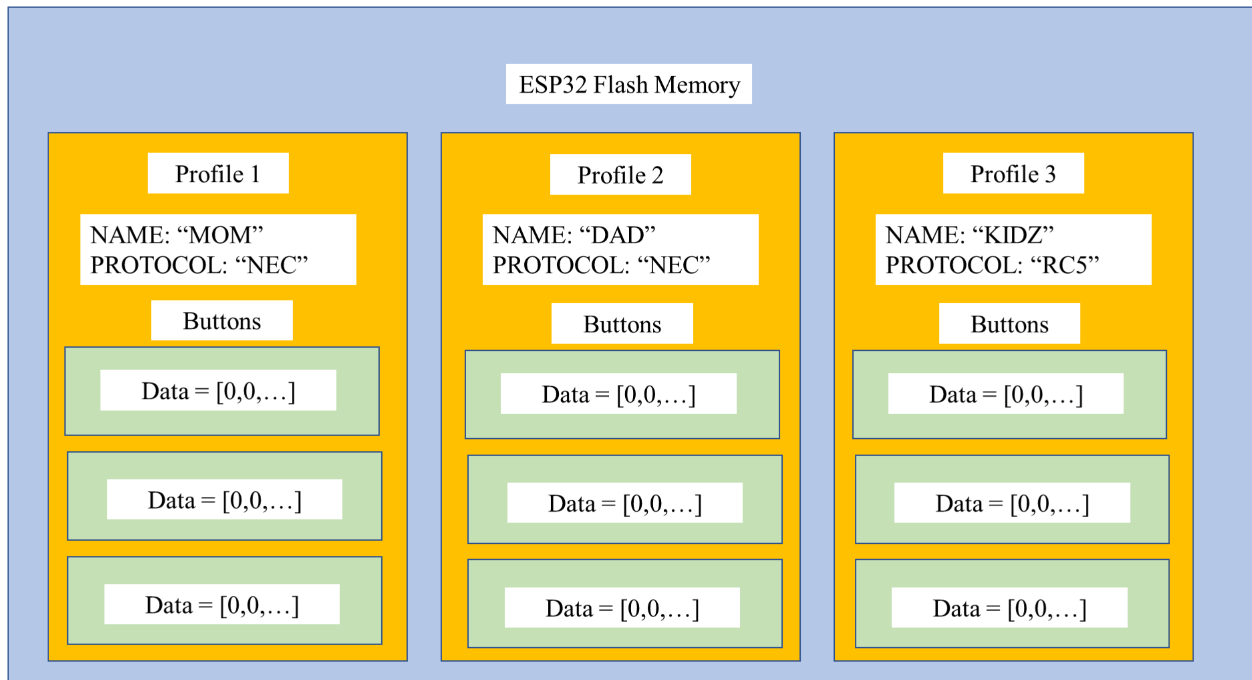
Generally speaking, this memory system works like a giant hash table. Four main functions from the libraries were used:

1. `nvs_get_u64(nvs_handle_t my_handle, const char * key, uint64_t data)`
2. `nvs_get_str(nvs_handle_t my_handle, const char * key, char * data)`
3. `nvs_set_u64(nvs_handle_t my_handle, const char * key, uint64_t data)`
4. `nvs_set_str(nvs_handle_t my_handle, const char * key, char * data)`

These functions allowed us to store two types of data: strings and unsigned integers. This is all we really needed. We used strings to store the names of custom buttons and profiles. Similarly, we used unsigned integers to store specific button data that would be sent as an IR code. From these base functions, we built our entire memory system.

We started by envisioning the final UI and what information we would need to store. We knew we wanted profiles and that each profile would need a number of buttons. Thus, we decided to generate 3 profiles in memory and each profile has 3 customizable buttons. Each profile needed a name and a default protocol. Similarly, each button needed a name and protocol. Buttons also needed to store the actual data that would be sent out as an IR signal. These values are stored in an array of unsigned integers, initialized to zero. There is storage for 5 total signals within a single button press. An overview of stored userdata can be seen in **fig. 11**.

Figure 11. Userdata Hierarchy Within Memory



Abstractions started at the button level. We began by creating “flash_write_button” and “flash_read_button” functions that wrote and stored button data contained within a button structure.

5. `flash_write_button(int profile_number, int button_number, button b)`
6. `button b = flash_read_button(int profile_number, int button_number)`

Button and profile numbers were used to generate specifically formatted strings. These strings became the “keys” used within the NVS functions provided above. For example, if we wanted to load the first custom button from profile 1, we would call `b = flash_read_button(1, 1)`, where “b” is the button structure holding its metadata. Eventually, when we wish to send IR signals, we will load specific buttons and access their data using `b.data[N]`.

The next layer of abstraction occurs at the profile level. Profiles can also be written and read, stored within the profile structure as a container. They hold the data seen within **fig 11**.

7. `flash_write_profile(int profile_number, profile p)`
8. `profile p = flash_read_button(int profile_number)`

The read and write functions for profiles read and write their own buttons using functions 5 and 6. They read and write their "name" and "protocol" information using functions 1-4. With a system in place for accessing and organizing user data, the memory system was essentially complete. See fig. 12 for the c-structures that store the user data explicitly.

Figure 12. Profile and Button Structures Within C

```c
struct button
{
  char name[100];
  char protocol[100];
  uint64_t data[5];
};

struct profile
{
  int number;
  char name[100];
  char protocol[100];
  button buttons[3];
};
```

Read and write functions were tested extensively. They were uploaded to an ESP32, and within a loop, profile, button, and arbitrary data was updated using these functions. The iterative uploads would follow a pattern that depended on the previous value. We could monitor the progress of the pattern to ensure memory was being read / written correctly. The code also would reset the device every so often to verify the updates were nonvolatile. We also printed out the amount of

data consumed during the process to ensure we were not overloading the flash. Additional memory helper functions were developed that reset buttons to NULL values, reset profiles to NULL values, and to generate “example” profiles with junk data for demo purposes. Print functions were also developed for command line debugging.

5.7 Detailed Design / Operation of Subsystem 5 - IR Code

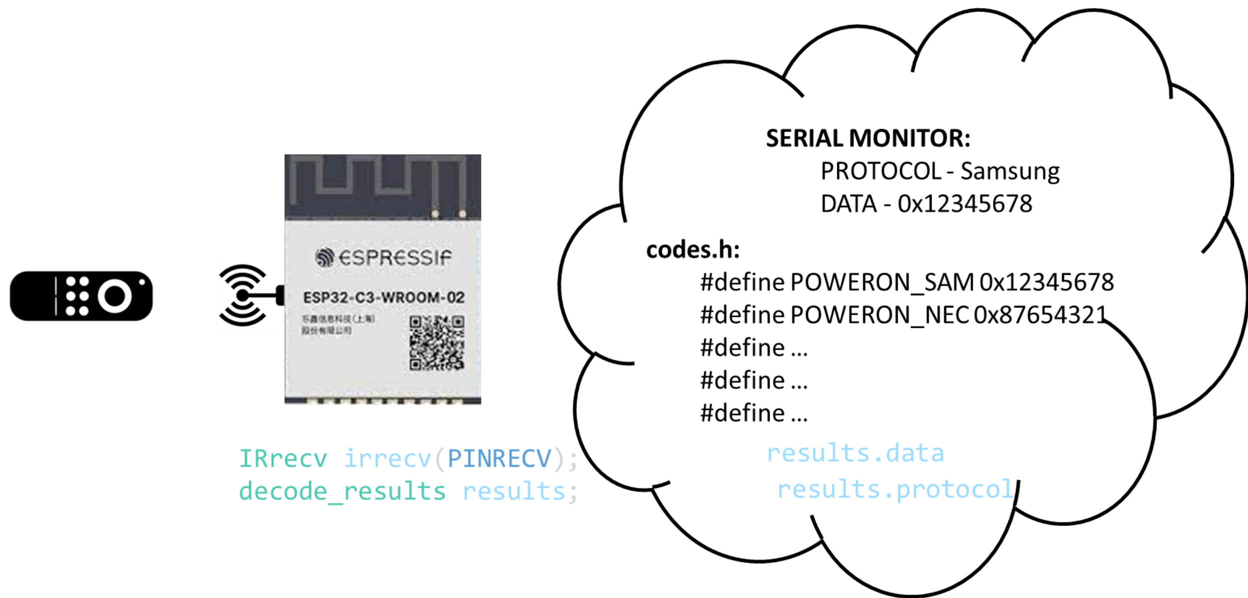
The library used to actuate and receiver IR signals using the ESP32 was the IRRemote.h library provided by Armin Joachimsmeyer. Early in the process, we found that an additional package was needed for compatibility with the ESP32 specifically, thanks to the demo project provided here. The additional package is found here and is built into our codebase. Generally speaking, we used these packages for two very simple purposes: receiving data and sending data.

IR code data was collected manually from the receiver. The receiver was attached to a specific GPIO pin on the ESP32 which used the receive code to interpret and print results. To set up the receiver code, we created a receiver object and specified the pin the ESP32 should be listening on. Upon being sent data, the object automatically updates its metadata. The protocol and unsigned integer data corresponding to the button pressed can be found as metadata, as well as an “available” field that indicates when new data has been made. Using a large switch statement, we would identify the protocol in newly received signals and decode the sent data.

The printed data was then tabulated within a centralized “codes.h” file. We created lists for Samsung and NEC based data for various default buttons. These will later be used when a certain profile having a certain protocol wishes to click a non-customized button like “POWER ON” or “VOLUME UP”. Only two protocols were developed for this demo due to time and

manpower constraints. The code is easily adaptable beyond two protocols. A figure showing the flow of receiver data collection and tabulation is shown below in **fig. 13**.

Figure 13. Methodology to Obtain IR Codes and Store Them



Signals are received and decoded in a similar manner for customized buttons. The only difference is that instead of being stored within a .h file manually, they are stored within a button structure using the read and write functions described before.

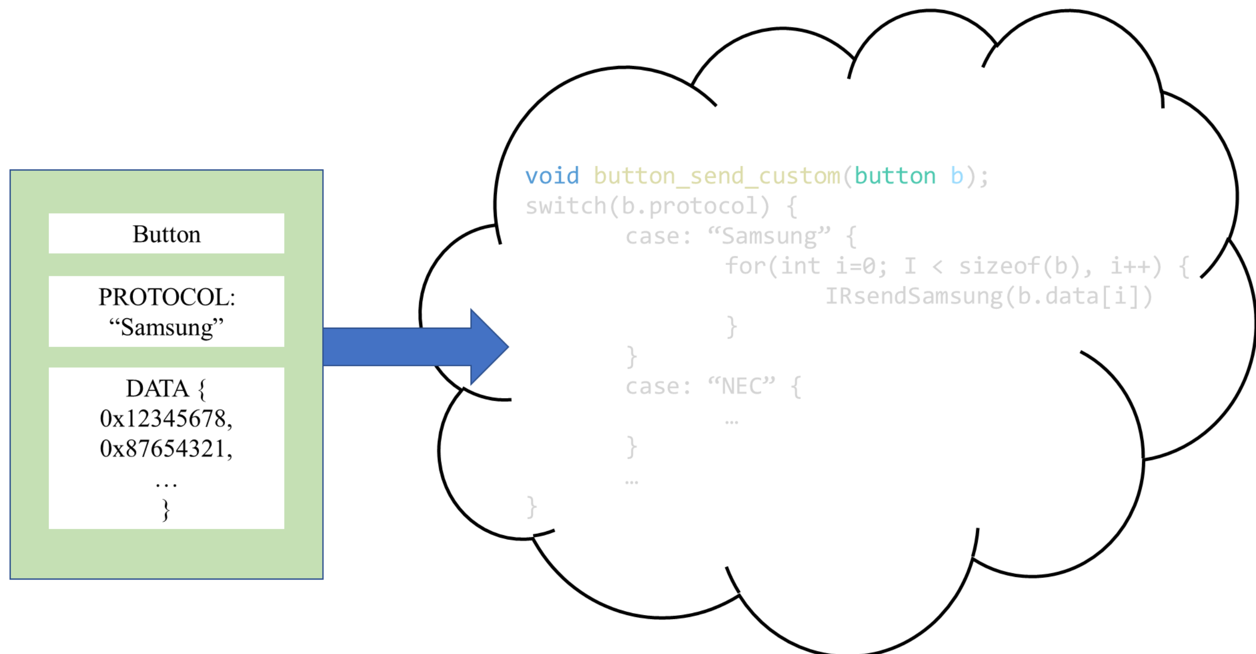
With custom codes stored within buttons / profiles and default codes stored within the codes.h file, we are now ready to send signals. Two "send" functions were developed:

1. `void button_send_custom(button b);`
2. `void button_send_default(char * protocol, int b);`

The function in (1) takes an input of a button and sends the data accordingly. A switch statement determines which protocol to send as using the button.protocol field. Then, a for loop running over the length of the button's data iterates, pausing for 2 seconds between each button to account for TV speed. Pseudocode for this process is provided in **fig. 14**. Default data sends are simpler as they transmit only a single piece of data. A similar switch statement is used, but this

time it reads the protocol of the profile, not the button since these are the default buttons meant for the primary TV. Then, single pieces of data are just sent without the need for reading buttons or iterating. The input button is given as an enumerated integer. This was done for simplicity, but the “integer” is really a string corresponding to a button such as “POWER” or “VOLUME UP”.

Figure 14. Sending Custom Data Using the IR Send Functions

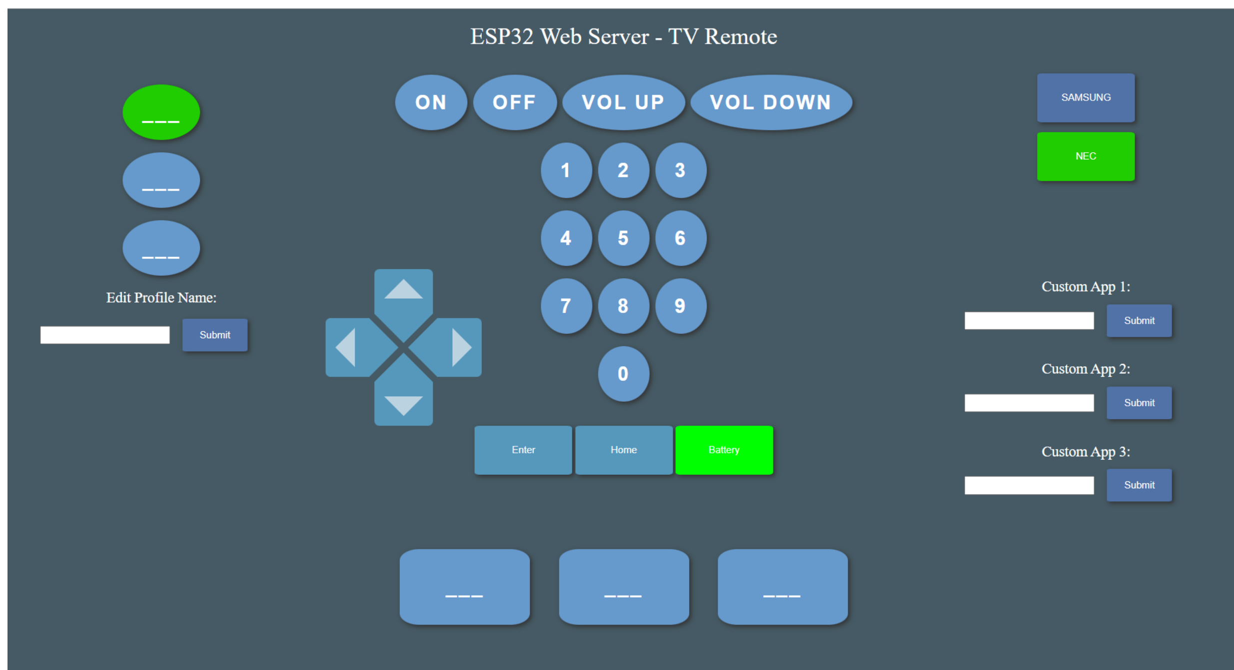


With the memory architecture in place, sending data becomes relatively simple. As seen above, it is really just a matter of selecting the correct protocol within the IRRemote.h package and putting the unsigned integer within the data field. Custom buttons have a small caveat where multiple pieces of data are sent at once, but this is easy to implement with a delay. Again, data is easily received using the objects provided by the IRRemote.h software. We simply take this received data and put it into our button framework.

5.8 Detailed Design / Operation of Subsystem 6 - User Interface (Website)

An image of our lightweight UI can be seen in fig. 15. It was developed using raw HTML so that we could control each aspect of code being uploaded to the ESP32. This was done in an effort to save space as the CPU has limited on chip memory. Additionally, this is an electrical engineering project, so the lower-level functionality rather than aesthetic appearance was the focus of the site.

Figure 15. User Interface for the Digital TV IR Remote



The functionality and implementation of each button will be discussed starting from the top left moving to the bottom right.

In the top left is the profile panel. Here, we have three blank profile buttons and a text-entry field to change profile information. When one of the profiles is selected, it highlights the button to green so you can know which profile you're on. While on a particular profile, you can update its name using the textbox field. The default protocol and custom buttons can also be changed while on a particular profile and changes will carry over upon reset. When the profile is

switched, it will change the corresponding buttons and protocol data. Profile structures are loaded within the HTML code and profile names are applied as a variable within the button. Again, displayed here is just a blank demo.

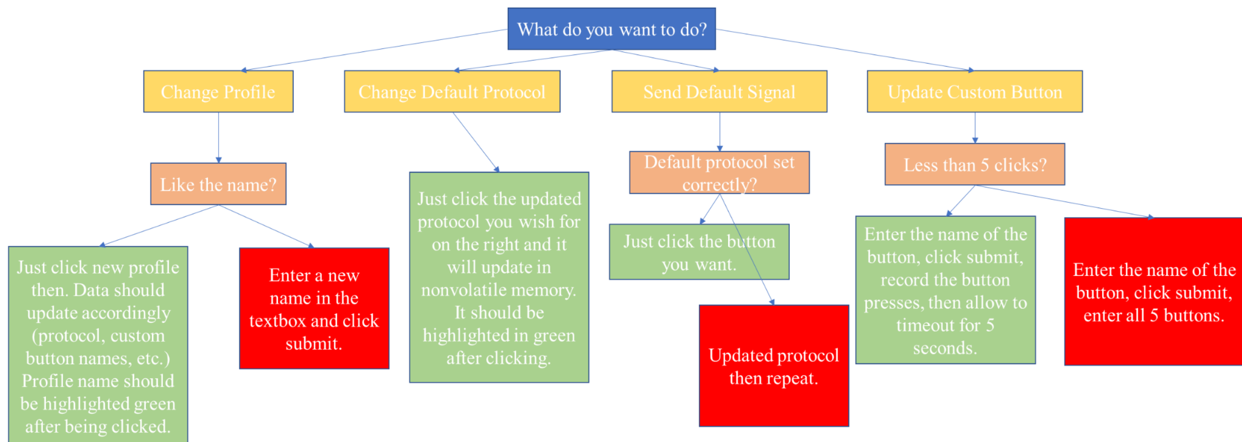
To the right of the profile panel are the default buttons. These are the prototypical buttons that every remote has and needs. The buttons automatically send data corresponding to the profile's given protocol. If the user gets a new TV on a different protocol, these buttons will still work so long as they change their protocol. Data sent from these buttons is done using the functions described in the previous section. Enumerated labels are passed as arguments for a given button. When the HTTP request goes through to the ESP32 server, based on which button was pressed, a different enumerated code is sent to the function, thus sending the correct data to the TV. To the right of the default buttons are the default protocols. Based on which protocol is selected for that profile, it will show up in green. Protocol changes are automatically updated to the profile and therefore nonvolatile memory.

Below the default buttons is the "Battery" button, currently highlighted in green. Each time the HTML is updated from a button press, a function called "read_battery()" is called. This function returns a boolean value indicating if the battery level is sufficient, per the circuit diagram provided in a previous section. If the boolean is true, the button appears green. If the boolean is false, the button appears read, indicating the batteries must be changed. Because HTML is just a big string, we manipulate the button choice using string concatenation. In the same row as the battery are default buttons for "HOME" and "ENTER".

Below the battery button are the 3 custom buttons. Right now they are blank, but they can have any customized string built into them. To record a new button, enter the name you wish the button to have then hit "submit". This will trigger the code to enter "record" mode, where the IR

receiver listens on a continuous loop for remote data. It will time out after 5 seconds and / or when 5 total inputs have been given. 5 is the limit for the sequence length of input data. A flow chart for all possible options of what to do with our UI is provided below in **fig. 16**.

Figure 16. Flow Chart for How to User Digital IR Remote



Again, pressing any button on this UI triggers an HTTP request on the ESP32's server side. Thus, the implementation becomes very easy. Based on the textbox / button selected, we can decompose the request and identify which function is needed. Through a series of "if statements", each and every HTTP request type is assigned to either update profile / button data or send out IR signals. Because of the simple nature of the HTML, the functionality is implemented easily.

The user interface website was tested by generating raw HTML and linking each button press or text entry to a print statement within the code. For example, if a user hit the enter button, we would test HTML functionality by printing "ENTER BUTTON PRESSED" to the serial monitor. This allowed us to develop the memory code separately from the UI code; we did not need the IR functionality to be integrated until later.

5.9 Detailed Design / Operation of Subsystem 6 - User Interface (Server)

The website seen in 5.8 was developed on the ESP32's own host server. This is why the design is lightweight; the ESP32 has limited on-chip memory and larger websites slow down the response time. The website was hosted using the WiFi.h and ESPAsyncWebServer.h packages. The first step in hosting the site is to connect to WiFi. This becomes complicated as the WiFi network name and password cannot be hardcoded in the design.

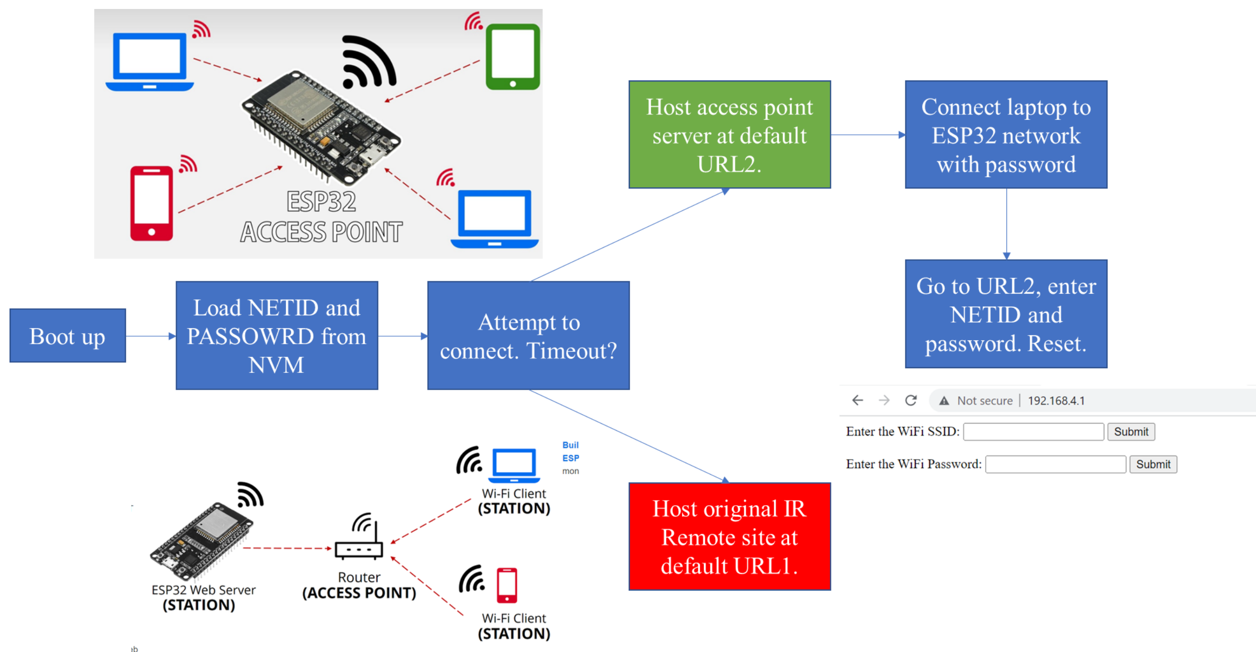
We solved this issue by using nonvolatile memory. Upon restart, the ESP32 will read values for NETID and PASSWORD from its own nonvolatile memory and connect using that. There is a slight hiccup in this plan, though. What happens when the device is moved to a new WiFi network? It will continually try to enter the WiFI credentials over and over incorrectly, never actually connecting to the WiFi. We needed a mechanism to reset the WiFi credentials when moving to a new location.

We achieved this using a timeout functionality. If there is more than 10 seconds of failure connecting to the WiFi, the ESP32 will host its OWN WiFi network. This is called an access point, with implementations using the ESP32 detailed here. Essentially, the ESP32 will show up as its own network on your laptop or phone when trying to connect to WiFi. When locations are changed, users will be instructed to boot up the device as normal, wait 10 seconds for the WiFi to timeout, then connect to the "ESP32" network using the password "password". The network need not be secure, hence the terrible password.

While on its own network, the ESP32 will host a simplistic site meant only for entering WiFi credentials. Users can access this network by entering in a hardcoded URL specific to the device. Once both credentials are entered, the values are written to nonvolatile memory and the website is shut down. The users are then instructed to reset the ESP32 and this time it will boot on the correct network. When it reboots, it will automatically connect to the correct network and

can host the remote site. This site will be on a similar, but different, URL that is also a default value for each ESP32. Thus, users will receive instructions with the two URLs as default parameters (URL1 - for the IR Remote, URL2 - for the credentials website). If credentials are entered incorrectly, the process will simply repeat. A flow chart of how WiFi is connected is provided in **fig. 17**.

Figure 17. WiFi Boot Up and Entering Credentials for New Locations



After WiFi is established, a simple server is declared using the asynchronous codebase. HTTP requests are particularly easy to handle in this syntax. The ESP32 receives user input using the following line:

```
server.on("/get", HTTP_GET, [] (AsyncWebServerRequest *request)
```

A series of “if statements” digests the “request” parameter by searching it for substrings corresponding to the particular button selected.

```
if (request->hasParam(PARAM_INPUT_3)) {
```

When a particular if statement is satisfied, one of two things happens: an IR signal is sent or profile data is updated. See the previous section to differentiate which buttons do what. Afterwards, the ESP32 sends an HTTP request to the user and publishes the updated site back to them. Again, everything on the server side revolves around the profiles. These determine what text is outputted on the buttons, what color certain buttons are, what data values are being sent, etc. Because of this prevalence, the profile is updated to nonvolatile memory after every update action.

This server functionality was tested by entering incorrect values for “netid” and “password” manually into memory, then repeatedly running the bootup procedure. The procedure worked as planned, with a few caveats. The ESP32 struggled to maintain its proxy hosted WiFi. Oftentimes, when entering information into the temporary website, the WiFi would disconnect and you would have to reconnect and begin again. However, we believe this is because of the breadboard that the kit board rests upon. The giant plate of metal blocks the antenna signal, causing slow / bad WiFi. This will hopefully not be a problem in the final design, as the antenna rests above free space.

6.) System Integration Testing (SIT)

The integration test was performed on the breadboard just as the individual subsystem tests were. The receiver, transmitter, and battery readout circuits were constructed on the breadboard using discrete components listed in table 1 and wired together to the GPIO pins on the ESP32. The regulator circuit was already constructed and verified last semester. The exact ESP32 used in the SIT was not the C3 designed within the final PCB; however, extensive research went into ensuring their compatibility. Passives were made to match those found in the

final design. Therefore, other than the parasitic capacitance added from the breadboard and mismatch of ESP32s, the SIT hardware was directly emblematic of the final design.

Software was also integrated for SIT. Buttons that previously only outputted text to the serial monitor were now attached to IRsend commands and memory functions. The implementation of the custom buttons was the only nuanced integration beyond a few lines of code. Upon entering the name of the custom button and selecting enter, the code enters a while loop. The while loop times out after a parameter TIMEOUT, and otherwise collects receiver data and adds it to the memory stack. This implementation had issues; clients who do not interact with the website for several seconds eventually disconnect, thus terminating the while loop and ending the data collection prematurely.

The outline for SIT is given below:

1. Preload incorrect WiFi data into non-volatile memory.
2. Turn on the ESP32.
3. Verify that WiFi fails to connect after 10 seconds.
4. Verify that the access point network named “ESP32” appears in the computer’s WiFi options.
5. Connect to the “ESP32” WiFi using password “password”.
6. Copy link from serial monitor and enter it into the browser. Verify the WiFi entry page boots up.
7. Enter information for NETID and PASSWORD as “SDNet” and “CapstoneProject”.
8. Verify that the webpage does not change until both entries are added.
9. Reset ESP32.
10. Verify that WiFi properly connects after reboot.

11. Copy link from the serial monitor and enter it into the browser. The remote control page should now open.
12. Begin by testing profiles. Enter new profile names for each of the three accounts.
13. Change the protocol from default to Samsung or NEC on each of the three accounts. Verify the changes maintained as you swap accounts.
14. Attempt to turn the TV on while on the wrong protocol.
15. Switch to correct protocol and turn on TV.
16. Show functionality of various default buttons.
17. Unplug battery indicator. Verify the battery icon turns red.
18. Record a new button by naming it “Volume Up+”. Press the up volume button on a physical remote 5 times. Verify recording stops after 5.
19. Verify the functionality of this new button by using it on the TV.
20. Reset the ESP32 and reboot the website. It should do so automatically. The profile data and custom button should still be the same.

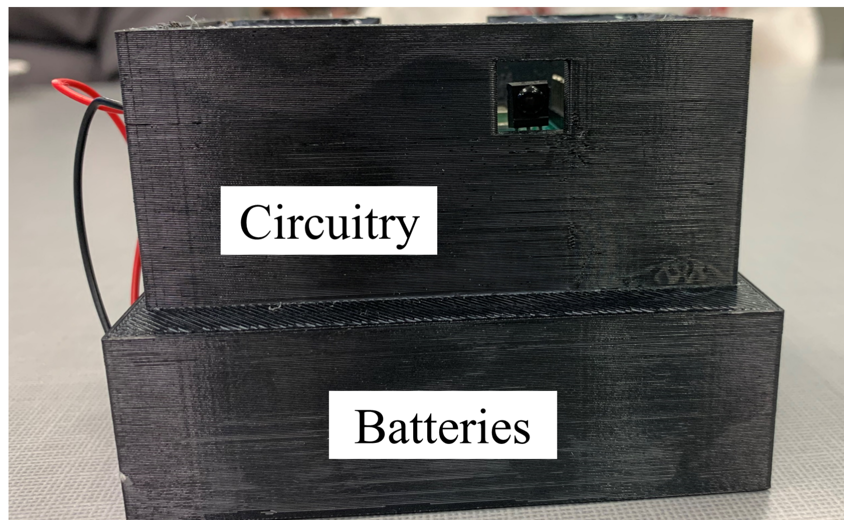
By doing these tests, we verify full functionality of our original goals. The device allows users to control the TV from their personal device, addressing the majority of the design requirements. The UI is intuitive and needs little to no explaining. The device works across various IR devices. Buttons can be customized for compound actions, giving users the freedom we envisioned. Battery level can be more easily monitored before they are fully dead. These will all make for a more relaxing TV experience, per the requirements.

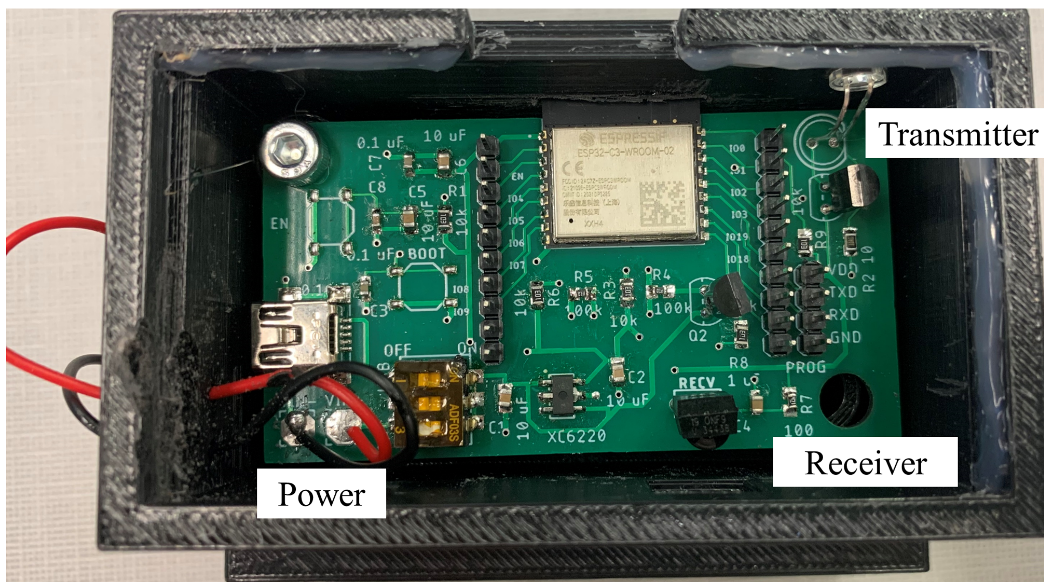
7.) User Manual / Installation Manual

7.1 - How to Install Your Product:

Product installation is meant to be incredibly simple. The delivered product is seen below in figure 18.

Figure 18. Physical Product that Users Receive Upon Delivery. (a) Horizontal View Showing Battery Pack Stacked on Device and (b) Inside of Device Highlighting Where Transmitter and Receiver Are Located





To begin, open the lid on the bottom portion of the device. This is the battery pack which holds the 3 AA batteries that power the device. Once they are installed, the device will automatically boot up as the dip switch will be preset for batteries. If, however, a user wishes to power their device using a microUSB, the dip switch will have to be adjusted. The switch is presented in position 3 to the right with switches 1 and 2 to the left. When using power from the microUSB, switch 3 ought to be to the right while switch 1 is positioned left.

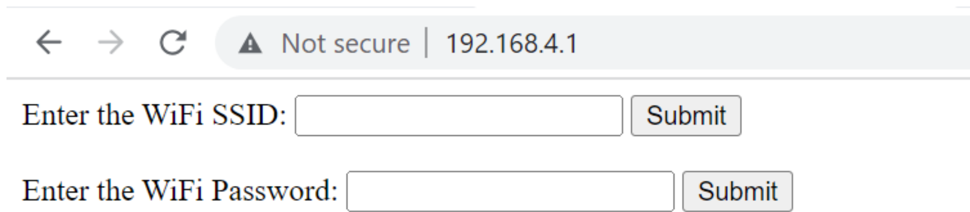
Once powered, the device should be placed with the IRED / transmitter pointing at the TV and the receiver able to be pointed at with a remote. Ideally, the device should be placed nearer to the TV's receiver, but its effective range is roughly 30 feet when accurately positioned. The IRED functions the same way a traditional TV remote does, so if the positioning would work for a TV remote, it will work for our product. The receiver is on the opposite side of the device relative to the transmitter, thus the device should be placed between the user and the TV when programming custom buttons.

7.2 - How to Setup Your Product:

The product requires WiFi to run properly. When first purchased, users must enter their personal WiFi credentials for the device to work. Once positioned with batteries, the device will boot up, attempt to connect to the network unsuccessfully, and after 10 seconds, launch its own WiFi network entitled "ESP32". The password is simply "password". Connect your personal device to this network.

Once online the "ESP32" network, you can access a web-page designated to enter your personal WiFi credentials. The IP of this site changes for each ESP32. For this device, it is **192.168.4.1**. Entering this number into the browser navigates you to the page shown in Fig. 19.

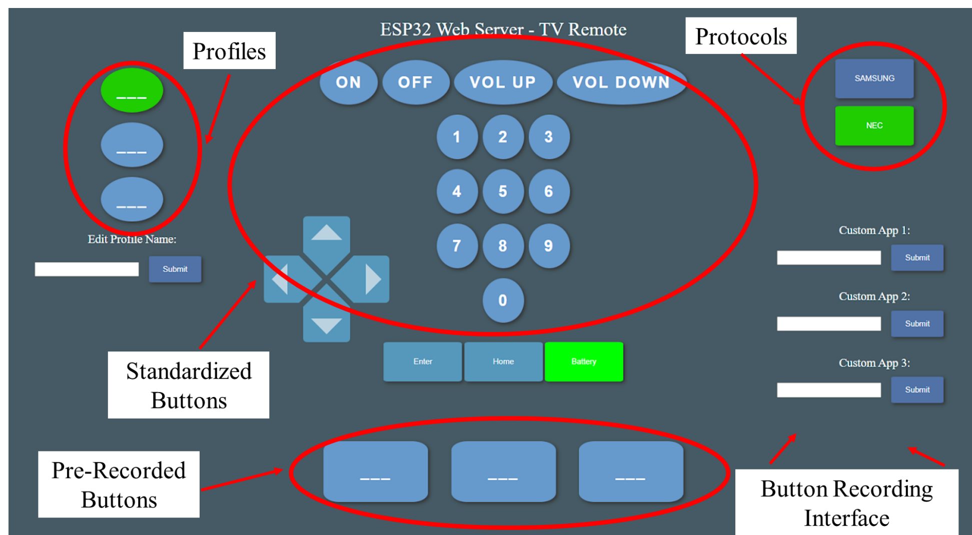
Figure 19. ESP32 Access Point Page for Entering WiFi Credentials



Upon entering both your SSID and your network password, the page will display a terminating message and ask you to restart your device. To do so, just disconnect the batteries and reconnect them. The device will now properly boot up on your WiFi network, where you can navigate to the actual site.

Once on the correct WiFi, the device will never enter the access-point mode again. The WiFi credentials are stored in memory, and will only need to be adjusted when the device is moved to a new network. To access the main webpage after being on the WiFi, navigate to the following link: **192.168.10.147**. This is again unique to the specific hardware and will be updated for each edition of this manual that comes with devices. The webpage that appears can be seen in **Fig. 15** but is repeated here with annotations for the user's benefit in **Fig. 20**.

Figure 20. Annotated UI for Web-Based IR Remote Control



7.3 - How can the User Tell if Your Product is Working?

The annotated portions in **Fig. 20** have specific functionality that indicate to the user that the product is working. First is the “profiles” section. Depending on the selected profile, customized features will be different. A functioning product should update these features when switching across various profiles. Additionally, the names of profiles should be interchangeable. Users can enter a new name for the currently selected profile using the prompt below the profile buttons. This should update the profile highlighted in green to have a new name. Profile data is non-volatile, meaning when the device turns off and turns back on, it will have the same profile names upon boot up.

The next section to discuss is the protocols. The device currently supports Samsung protocol, which works for Samsung IR products, and NEC, which works for a variety of devices like LG and Sharp. The currently active protocol is highlighted in green. Users can switch their profile’s protocol by clicking on the button corresponding to the given protocol. They can test that the update is working using the default buttons. If, for example, a user was working with an NEC TV but had their protocol set to Samsung, the power button would not turn on their television. However, if they switched the protocol to NEC, the power button and all the other default buttons would work perfectly.

If there was a button missing from the default layout that users wish to use, they can add it to their profile using the recording interface. Here, three text boxes allow users to enter a phrase describing their desired button. After typing, users should press enter, which will begin recording their button presses. Users should point their actual TV remotes at the receiver on the device and enter a sequence that encapsulates their custom required button. Users can enter up to

5 compound button presses of various protocols - even non-NEC or non-Samsung. The data stream will be saved in non-volatile memory and be tied to that profile. To test the functionality, simply navigate to the bottom of the user-interface and click the corresponding button. It will now have the user-entered text within the box instead of the default “---”. Ensure that it correctly performs the compound recording.

The final indicator of a functioning system is the battery box. It is integrated with the default buttons, and highlighted in green. When the battery level sags below a certain threshold, users will be notified that they should swap them out. The battery button turns bright red when the level droops, thus alleviating the user from the frustration of being surprised by dead batteries. When the device boots back up, its WiFi credentials are already stored as is the profile and button data, so everything will function properly simply after swapping the needed batteries.

7/4 - How the User Can Troubleshoot Your Product?

Currently, troubleshooting is the weakest part of this product. There is no display built into the device to communicate with users if something fails to work. If something goes wrong, and the website is not booting as it should, users should first change the batteries to fresh ones. If this goes wrong, users should check if the device is attempting to open an access point network by looking through available WiFi networks and searching for “ESP32”. If neither of these fix the problem and the device will not host its remote website, the device is broken and a member from the group must address the issue by reprogramming the board or fixing damaged hardware.

8.) To Market Design Changes

There are several improvements that could be made to this design. Limitations in its development were due the scale of implementing a universal device. It would not have been possible to have created these changes given the scope of this project; however, thinking about these changes provides great insights into future group projects.

8.1 - Server Hosting Website:

In an ideal to-market strategy, the ESP32 would not be the webhost for the user interface. By using a remote server, several limitations and problems would be addressed including memory challenges, response speeds, and connection difficulties. Let's begin with memory challenges. In its current state, the memory of the system is extremely limited, using an SPI bus to communicate with an on-chip flash memory. The total data storage is on the order of kilobytes, severely impeding scalability. For this reason, only 3 profiles were allotted with each profile only having 3 custom buttons. Additionally, profiles do not have any of the customization that comes with typical digital profiles. Users cannot, in its current state, select their background, their button colors, their profile icon, etc.

By implementing a remote server, users could have greater customization due to larger memory banks. A list of potential aesthetic and functional changes to the final product is provided below assuming an unlimited amount of memory:

- The ability to add and delete profiles up to "N" profiles.
- Profile icons that can be pictures or emblems.
- Changing profile color schemes.
- Changing profile backgrounds.
- The ability to add and delete custom buttons up to "N" buttons.

- The ability to move and store the location of different buttons on a single profile.
- Extend the number of inputs a single custom button can have beyond 5.
- The addition of a "bank" of prewritten custom buttons like "Netflix" or "Hulu" that could be added at the whim of the user.

These features would make the feasibility of this product much more realistic versus a competitor like Roku. Because of on-chip memory limitation, many of the original features were ruled out. A secondary solution would be an external flash memory attached to an SPI bus on the PCB design. However, because there are other benefits to a remote server, and because extra memory would be more easily available through the remote server, the "to-market" strategy does not involve on-chip memory.

The external server would also allow for better response speeds. Though the ESP32 has its own antenna and is able to connect to the WiFi, its ability to host a website is limited. As previously discussed, when users go idle while recording a custom button, there is often an interrupt that causes the program to stop recording until the user clicks or interacts with the website again. Additionally, there is general lag / slowness when clicking or entering texts as a result of the ESP32's design. Though it has the ability to interact with the WiFi, it is still just a microprocessor. In a to-market environment, products have to be competitive and seamlessly work under all circumstances. For ensured reliability, the server should be hosted on a designated computer somewhere on a server farm.

Finally, the usage of an external server allows for easier user connectivity. In its current state, users have to program the ESP32 to connect to their home WiFi. This involves a series of steps including the usage of an access point with spotty reliability. Additionally, the actual WiFi IP changes across different networks. Because of this issue, the "website" is not a standalone IP

that users enter but something they have to have given to them each time they change their WiFi. A better implementation would be to have a single webpage that is synchronized with an individual device. The device would still need to be manually connected to the internet, but this could be done through the universal web page rather than the access point. By using an external server, the connection and location of the webpage would be much easier for the user. Results of this discussion are summarized below in **fig. 21**.

Figure 21. Summary of Benefits from Using Remotely Hosted Server in To-Market Product Implementation

Benefits of a Remote Server Versus a Locally Hosted Server		
Memory	**Responsivity**	**Connection**
• Using a remote server allows for remote access to memory servers. • Additional features using unlimited memory include: ➢ The ability to add and delete profiles up to "N" profiles. ➢ Profile icons that can be pictures or emblems. ➢ Changing profile color schemes. ➢ Changing profile backgrounds. ➢ The ability to add and delete custom buttons up to "N" buttons. ➢ The ability to move and store the location of different buttons on a single profile. ➢ Extend the number of inputs a single custom button can have beyond 5.	• ESP32 server times out occasionally, resulting in: ➢ Truncated customized buttons being recorded ➢ The need to double send commands (one click to reboot server, one to send command) • Quick multi-clicking is difficult too. Pressing volume-up rapidly in sequence, for example, is not feasible. • Response speed changes minute to minute. This is aggravating for users.	• Allows for single URL across various WiFi networks • Allows for centralized hub for all WiFi connections • Eliminates need for access point hosted website to enter WiFi creditials. • Simplifies process for users.

8.2 - Web Page Design:

Due to the memory constraints and limitations of our coding background, the website was designed using raw HTML. Because of this, the aesthetics of the website were very basic. User entry fields were exclusively either button presses or text. Effort went into the styling of the

buttons, but the advanced functionality provided by other HTML tools was not taken advantage of. Additionally, advanced features seen in contemporary TV applications like profile pictures, color schemes, etc. were missing. In a full scale implementation, the website would need more functionality than just text boxes and button presses. Modeling after an application like Netflix, we would want a robust profile setup with a home-page, advanced customization, etc.

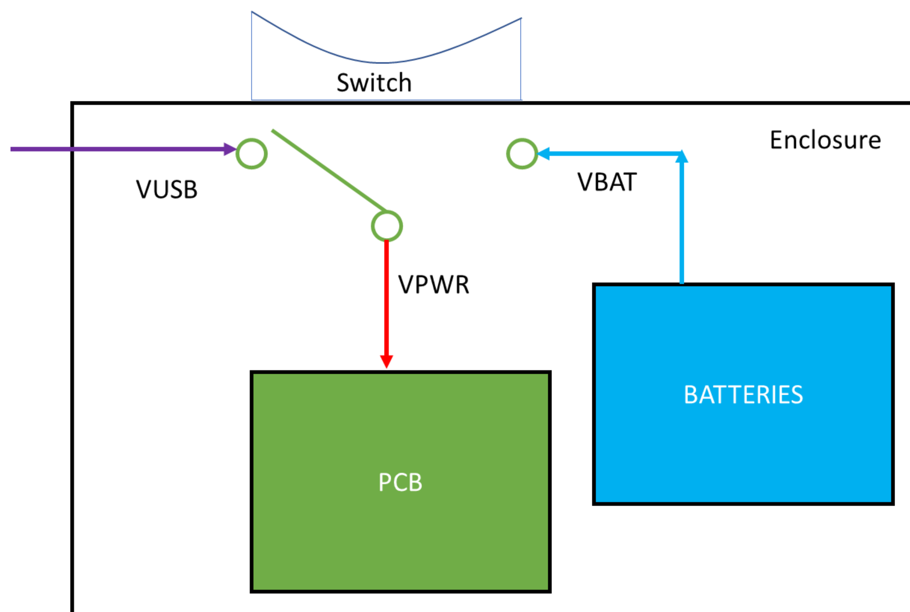
This could be done using an HTML builder used by contemporary web developers. After creating the full scale website using click-and-drag tools, we could export the raw HTML generated from the site and synchronize button clicks with IR functions. This has already been done with our current site on a smaller scale, so integration would not be a problem. Much of the customizations that add to user experience comes down to software alone; because of this, development externally would not cause integration issues beyond what we have already experienced. Designing a full scale site within this new context of web dev would require an entire brainstorm and project development in itself. The number of potential features is limitless.

8.3 - PCB Design:

The PCB design for this project was made with the thought of this product being a development board. Pin headers and push buttons were included for testing / bootup purposes. In a final design, pin headers would not be needed because no debugging would be occurring. Additionally, bootup push buttons would not be included because the device would boot in the same way each time. The device would be programmed on a factor line, where pads on the PCB would be exposed and pressed down via a programmer. This eliminates the need for pin headers and will confine the strapping pins to a single value.

The switch allowing users to have a USB-micro power source versus batteries would still be included. This freedom is imperative to give users the ability to avoid battery issues and / or avoid cables in general. The entire purpose of the device is to yield choices to the classic remote design; therefore, the choice of power is important to us as designers. That being said, the switch selected for this project is small and overly complicated. There are three dip switches with the middle switch doing nothing. User error would allow both switches to be down at the same time, which would put the batteries and micro-USB together in parallel, completely derailing the power scheme. In the same vein, both switches could be off at once and the device would have no power whatsoever. The switch itself has a size issue; adjusting the dip switches would be a pain because the enclosure would have to be opened, and its location on the actual PCB makes for a finicky situation. Instead, a single switch should be positioned on the outside of the device that dictates “BATTERY” on one side and “USB” on the other. A generic implementation can be seen below in **fig. 22**.

Figure 22. Single-Switch Exterior Implementation for Final Market Design



The rest of the PCB components can remain the same. With the assumption that an external server would be used for the website, no additional memory would be required. Additionally, this assumption then enforces the requirement of a WiFi enabled chip like the C3. Beyond connecting to the website for signal information, however, the PCB simply acts as a hub for receiving and transmitting signals. The final design still needs a regulator for power, a subsystem for reading out the power, a subsystem for transmitting data, and a subsystem for receiving data. Because these subsystems function properly already, there is no need to adjust them. Just like real TV remotes, IRED transmitters will be through hole components with legs so their position can be adjusted to form fit the enclosure.

9.) Conclusions

The initial requirements for this project were met with specific limitations. To briefly rehash, our requirement were:

1. Control a TV using the ESP32
2. Control different TYPES of TVs
3. Control Via WiFi Based Website
4. Control of External Device
5. Have Profiles that Persist in Non-volatile Memory
6. Power with Batteries and Have an Intuitive Indicator
7. Implement with a PCB and 3D Printed Container
8. Generate an Intuitive Website UI that …
    a. Builds profiles
    b. Remembers profiles

c. Allows multiple users at once

d. Has an intuitive bootup

Each of these requirements was addressed in some form or another with specific implementations addressed in previous sections. Because of this, the project was a net success.

There are obvious needs for improvement, with each iterative requirement having various degrees of difficulty. Protocols need to be expanded for true universality, which is easily done within the code. Additional profiles and buttons need to be added, which is also an easy software fix. The PCB needs to be adjusted so it is a final product, not a demo board, simply by removing the header pins / buttons and moving the dip switch.

More difficult changes involve the UI and server. For the purposes of this educational project, it was convenient and sufficient to host the site on the ESP32; however, in future works, special care should be taken to use an external server with better responsiveness, memory, and feature implementation. The attractiveness and functionality of the website can certainly be improved upon to give users a more fluid UI experience. This will require larger memory banks and more sophisticated HTML that was beyond the scope of this particular project.

Members of the group got to apply their knowledge from the entire spectrum of electrical engineering. This device encompasses low level semiconductor concepts related to IR transmission / receiving across various bandgaps and wavelengths. It also involved circuit and electronics design, pulling from our sophomore year coursework in that domain. We also learned about signal modulation and processing within the IRRemote.h file for different carrier frequencies, thus pulling on our knowledge of signals and systems. Certain group members worked intimately with the microcontroller in low-level c-code, harking to experiences in embedded systems and various introductory coding classes. Finally, high-level HTML design

was used which was developed both in CSE electives and through online resources and research. The project was a robust, full scope integration that culminated in a holistic, all encompassing body of work. Group members had to use knowledge from nearly half of their undergraduate studies, making it a fantastic capstone project.

For potential students next year looking to continue this work, we have provided our codebase and board in the following section. The direction we recommend people take this project is via software. We figured out many hardships about the IRRemote.h file and became experts in how to use it effectively across various protocols for both signal sending and receiving. What new students should now do is abstract that process and hone in on server and styling functionality. Add the features we did not have time for to make this prototype a usable, household product.

10.) Appendices

1. Complete hardware schematics can be found here. It is our group GitHub. The saved file ".boards.zip" has all the needed board and CAM files. If the link isn't working, copy and paste the following: https://github.com/collinfinnan/Group11IRRemote
2. Complete Software listings can be found at the same GitHub repository as linked above.
3. Relevant parts or component data sheets can be found in **Table 1.**, found earlier in the documentation with links provided already.