Josia Orava, Jussi Enne, Miro Tuovinen, Ariana Karadzha

Buttmote

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and communication technology

Internet of Things (IoT) Project

21 December 2023

Abstract

Author:	Josia Orava, Jussi Enne, Miro Tuovinen, Ariana Karadzha
Title:	Buttmote
Number of Pages:	20 pages + 1 appendices
Date:	21 December 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and communication technology
Professional Major:	Smart IoT
Supervisors:	Joseph Hotchkiss, Teacher

The text section of the abstract is written to fit the space used on the page. The text section uses the “Body Text No Spacing” style.

Keywords: embedded, IoT, smarthome

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

Appendices

List of Abbreviations

UDP:	User Datagram Protocol. Protocol used across the Internet for especially time-sensitive transmissions.
IDE:	Integrated Development Environment
GPIO:	General Purpose Input / Output
IR:	Infrared
IP:	Internet Protocol
LED:	Light Emitting Diode
WiFi:	Wireless Fidelity
TV:	Television
USB:	Universal System Bus
HTTP:	HyperText Transfer Protocol
HTML:	HyperText Markup Language
JS:	JavaScript
CSS:	Cascading Style Sheet
AP:	Access Point
AWG:	American Wire Gauge
UI:	User Interface
MCU:	Micro Controller Unit
FR-4:	Fire retardant woven glass-reinforced epoxy resin
SPIFFS:	Serial Peripheral Interface Flash File Storage
JSON:	JavaScript Object Notation

1 Introduction

The aim of our project was to create a remote controller that will detect when a person sits or gets up on a sofa, turn on and off TV with an IR signal and control smart lights with user defined settings through WiFi using UDP packets containing a JSON message.

2 Methods and Materials

Here we will briefly describe what methods and materials were used in this project.

2.1 Physical design and assembly

Physical design was made in Fusion 360 and mechanical parts were 3D printed from PLA with Creality K1 -printer. Schematics were made in KiCad and the circuit board was hand soldered on a prototype board using through-hole components.

2.2 ESP32 Development Board

The ESP32 Development Board serves as the project's central controller. It enables Wi-Fi connectivity and manages various device functions through its GPIO pins.

2.3 IR Receiver and Emitter

Used for receiving and transmitting IR signals. The IR receiver captures commands from a remote control, while the emitter sends IR commands to control external devices like the TV.

2.4 Pressure Sensor

The pressure sensor is placed under the couch cushion to detect when someone sits down.

2.5 RGB LED

The LED provides feedback about the device's state through a range of different colors.

2.6 Smart Light

The smart light was integrated for advanced lighting control, enhancing user experience with adjustable brightness and automation.

2.7 Buttons

The device has a total of three buttons. The buttons labeled (+) and (-) are used to adjust the brightness of the smart light, while the (R) button is utilized to capture and learn IR signals from remote controls. Additionally, an on/off switch is incorporated for overall device control.

2.8 FreeRTOS

FreeRTOS plays a crucial role in the project, ensuring that different tasks run smoothly and in coordination.

2.9 WiFi

Device uses Wi-Fi to connect to your home network and communicate with all the smart lights that are in the same network.

2.10 Web Technologies in Use

The Buttmote employs basic HTTP requests to facilitate communication with the client's web browser. The client initiates a GET request to the Buttmote's IP address to retrieve Web UI resources, including HTML, JS, and CSS files, from the embedded SPIFFS file system. For status retrieval via GET requests and settings updates via POST requests, the Buttmote strictly adheres to the JSON data format when communicating with the client via the API.

2.11 Libraries

2.11.1 IRRemote

This library was used to send and receive the IR signals and the Preferences library was used to store the signals while the device is powered off.

2.11.2 WiFi & WiFiUDP

These libraries were used to connect to the WiFi and communicate with the lights in the same network by using UDP packets.

2.11.3 ESPAsyncWebServer and AsyncTCP

These libraries allow us to handle multiple concurrent requests without stalling the device. This ensures that the device can keep executing other critical tasks and thus maintain responsiveness to sitting, and can keep updating smart home devices without interruption or interference from the web server. They also provide a great request structure ensuring that clean and readable code is maintained throughout the entire web server portion of the code.

2.11.4 SPIFFS

The SPI Flash File System (SPIFFS) is a file system intended for SPI NOR flash devices on embedded targets. It gives us a reliable way to store files for the web server without having to compile them into the main executable, which would both increase its size and slow down the compilation times.

2.11.5 AsyncJson and ArduinoJson

These libraries enable us to perform parsing, serializing, and type checking on JSON data. This allows us to easily communicate with the client via the API while adhering to the JSON data format, greatly reducing the complexity of the client side scripts.

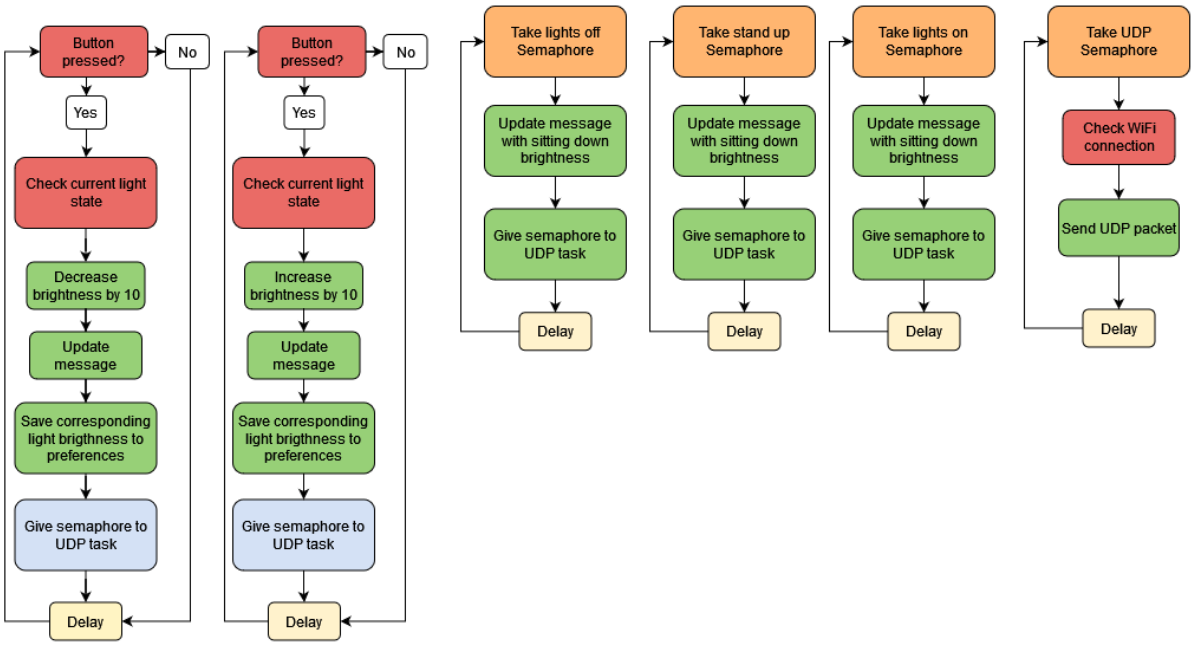
3 Implementation

3.1 Wi-Fi & UDP

The device uses Wi-Fi connectivity and UDP for communication with smart lights on the network. Communication between the device and the smart lights is structured in JSON format. The communication happens so that the device initiates communication with smart lights and the lights only work as a listening device and don’t send any messages to the device.

The JSON format allows for easy modifying of messages, providing a flexible way for integrating various commands and parameters. These messages can include commands to control light brightness, color, and other parameters used by the smart lights. Any new smart lights that are added can be easily controlled because every light has a unique ID starting from 1. So, to control many lights at the same time you just send the same JSON message and increase the ID number between messages.

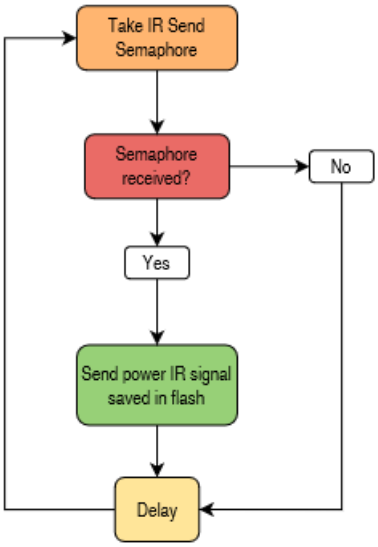
3.2 Lights & UDP tasks



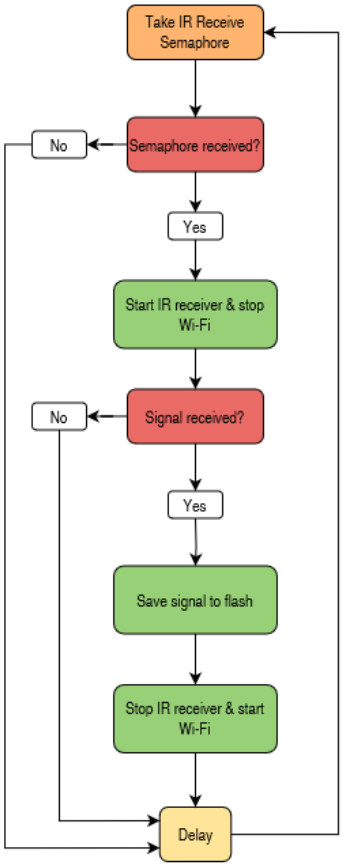
The synchronization between tasks is achieved with the use of binary semaphores. Before a task initiates its execution, it waits for the corresponding semaphore to be given. Each light mode task modifies the JSON message to its specific functionality by changing the needed parameters in the message. The device lets the user save settings to different light states that are: Sitting down, initial stand up, standing up. These are saved in preferences which allows the device to remember user saved settings even when losing power. Messages are made using these saved values. Once the message is ready, the current light task gives a binary semaphore to the UDP task and now the UDP task knows that it can transmit the message to the smart lights.

3.3 IR send task

The IR send task waits for a binary semaphore from the sitting logic task. When the semaphore is received, the task sends the stored IR signal to turn the TV on or off and starts waiting for another semaphore.

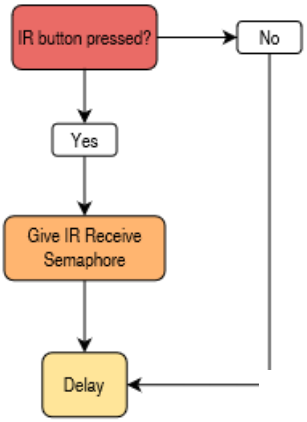


3.4 IR receive task



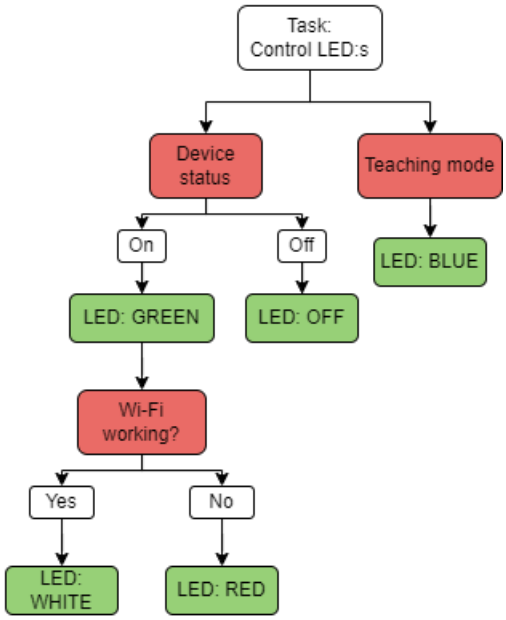
The IR receive task waits for a binary semaphore from IR receive button task. When the semaphore is received, the IR receiver is activated, Wi-Fi is disabled and the task starts waiting for a signal to the IR receiver. Wi-Fi needs to be disabled during receiving to avoid interference from Wi-Fi communication. When a signal is received, it is stored to a structure for the IR send task and to preferences to be saved when the device is powered off. After the signal is saved, the IR receiver is stopped, Wi-Fi is activated and the task returns to waiting for another semaphore.

3.5 IR receive button task



IR receive button task polls the IR receive button 10 times per second. When the button is pressed, the task gives a binary semaphore to IR receive task to signal it to start receiving.

3.6 LED task

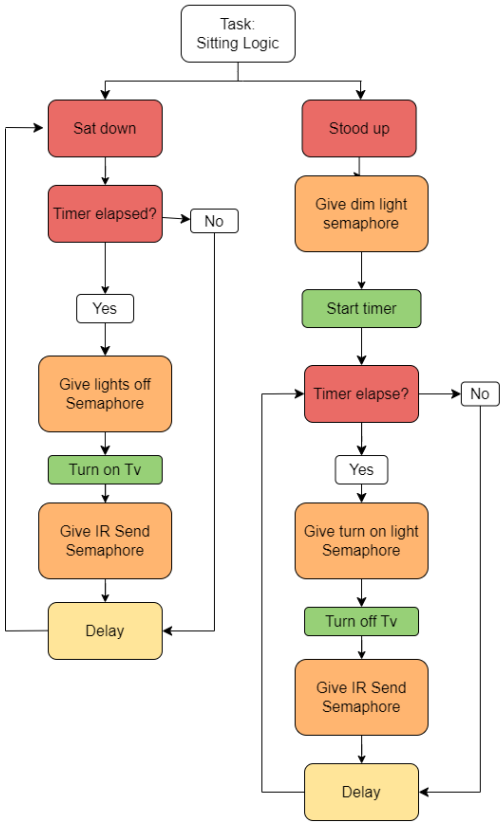


The ControlLedsTask continuously checks the device's state. It starts by turning on the green LED, indicating

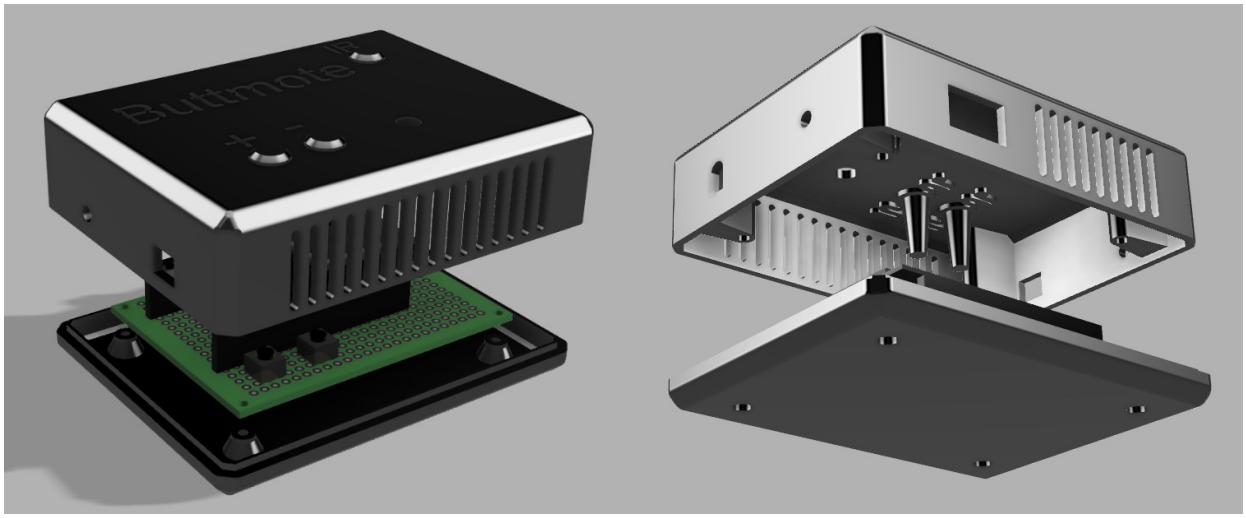
the device is powered on. If teaching mode is signaled, the blue LED lights up. Otherwise, it checks for the WiFi connection, a successful connection triggers the white LED, while a failed connection triggers the red LED.

3.7 Sitting task

The sittingLogictask checks the state of the pressure sensor. If the sensor detects sitting, and the timer is not yet triggered, the task sends a binary semaphore to turn off the lights, and another semaphore to send the IR signal that turns on the TV. When standing is detected, a timer starts, if the timer elapses, the task sends a binary semaphore to turn on the lights, and another to send the IR signal to turn off the TV.

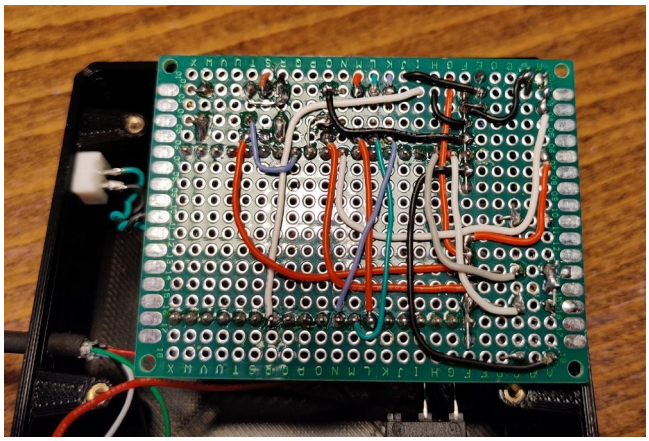


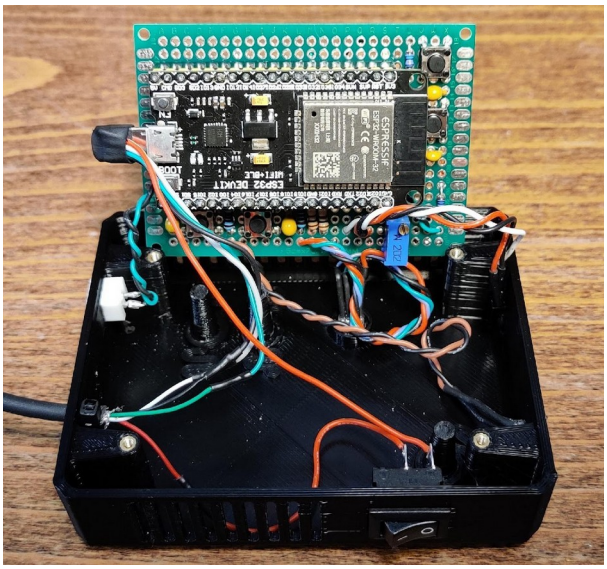
3.8 Enclosure

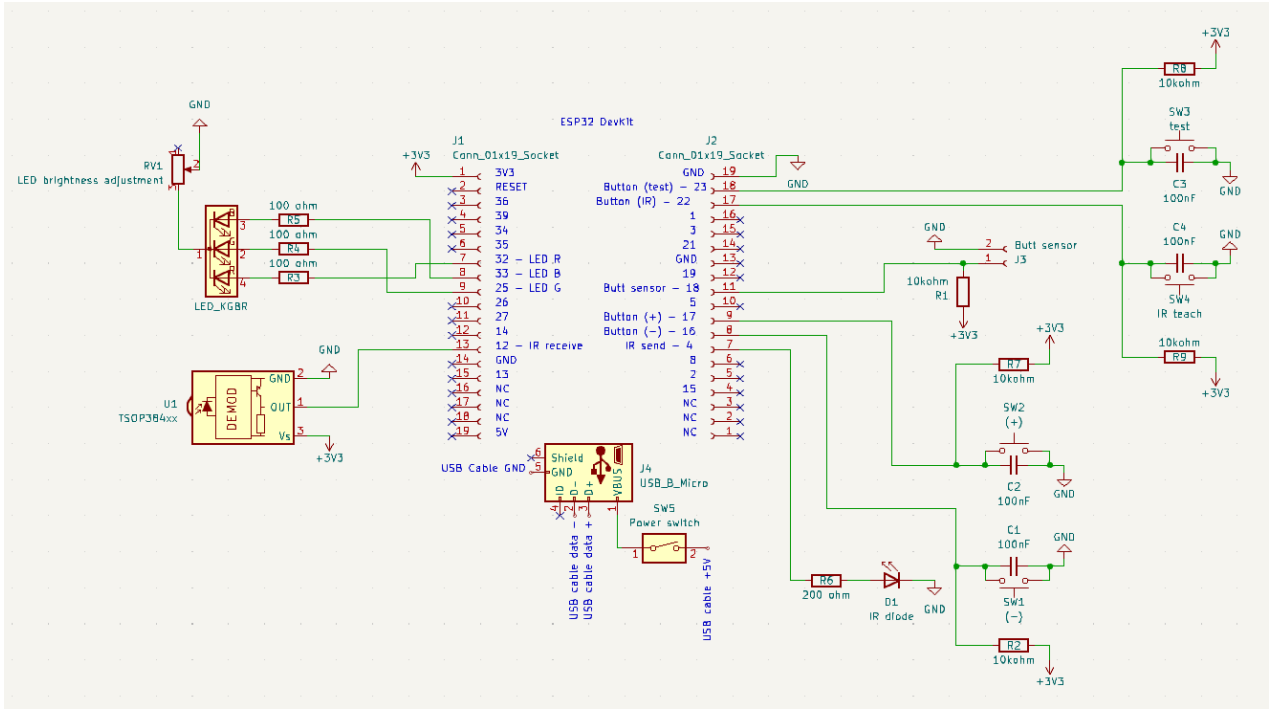


The enclosure is a two part design and it is held together with four hidden M2 allen screws that thread into brass inserts. The screws also hold the circuit board in place as it is sandwiched between the two halves of the enclosure. The buttons use a compliant mechanism to improve the feel and to prevent accidental presses, as the tactile switches on the circuit board are quite sensitive. The enclosure has openings for the IR LED, IR receiver, indicator LED, power switch, USB cable and pressure sensor connector and those were glued in place. Labels for the buttons were embossed to the top surface for better user experience. The enclosure has cooling slots on its sides to ensure sufficient cooling of the chip and they also improve the looks. Silicone pads were glued to the bottom to help keep the device stationary during use.

3.9 Electrical design







The device is powered through a USB cable, which also serves as the interface for programming the ESP32. A solderable micro USB connector was connected to the USB connector on the board and the wires from the USB cable were soldered to it, 5V wire through the power switch. The IR receiver is powered with 3.3V and the data pin is connected to pin 12 on the ESP. The IR LED is connected to pin 4 and it has a 200 ohm resistor to limit the current. The indicator LED is connected to pins 25, 32 and 33 and there are 100 ohm resistors to prevent drawing too much current from the pins. There is

a 2 kohm potentiometer on the ground connection to be able to adjust the brightness of the LED to a suitable level. There are 4 buttons on the circuit board and those connect to pins 16, 17, 22 and 23 on the ESP. One of these was not used in the final design. The buttons have 10 kohm pullups and 100 nF capacitors for debouncing. The design was hand soldered to a FR-4 prototype board using through-hole components and 30 AWG silicone wire.

3.10 Web Server

The web server starts by initializing the SPIFFS file system in order to fetch the required files to be sent to the client, after which a Wi-Fi connection with the set credentials is made. These initialization steps are also required by other parts of the Buttmote codebase. The web server then begins the process of starting the proper server itself by creating a static AsyncWebServer instance on port 80, which is the default HTTP port. It creates a route to handle the root page of the website, which is the site that loads when the IP address is visited directly. It uses this route to serve the index.html file while also leaving room for future implementations of the access point and captive portal. Routes for the style.css and main.js files are also created in a similar fashion, however these routes are just fully serving static files as there is no need to expand their implementation.

We then define routes for the API as these require special handling of JSON and related variables. These routes are quite dynamic and thus they have their own handler functions. The handler for the /api/status path first creates a DynamicJsonDocument instance and a character buffer to store the data to be sent, each of these use roughly a kibibyte of memory. It then populates the JSON document with data containing the current state of the Buttmote and serializes it to a string to be sent to the client. It sends the JSON string with a status code of 200 indicating that the request was successful. The handler for the /api/settings path takes in the JSON document sent by the client by using a special AsyncCallbackJsonWebHandler. It then checks for the presence of the

required settings and checks that they are of the correct type. If all these checks are successful it updates the settings and responds with status code 200 indicating that the request was successful.

3.11 Client Side Web UI

On the client side after the main page is loaded it periodically polls the /api/status path with GET requests every second and updates the UI to match accordingly based on the received JSON data. It displays whether or not a person is currently sitting on the Buttmote and also has a display for which state the light should currently be experiencing, the light however does not react to the real brightness of the bulb and would have been fixed if we had more time as adjusting the individual brightness of each state was added very late into development. This is also the reason that the brightnesses of the states cannot be updated from the Web UI. The Web UI however does allow the user to update the idle time to change between states and to enable or disable the IR functionality. It achieves this by sending a POST request to the /api/settings path containing the formatted JSON data.If it receives and error it alerts the user about what went wrong and continues operation as normal.

3.12 Challenges

3.12.1 PlatformIO

The initial challenge surfaced during the integration of Visual Studio Code with PlatformIO library. Despite our microcontroller not being directly supported in the board libraries of PlatformIO, we attempted manual configuration of the toolchain to flash code to the device. Unfortunately, these efforts were not successful, leading us to transition to the Arduino IDE, which offered native support for our specific board.

3.12.2 Amazon Alexa

The second significant challenge came up as we explored possibilities with Amazon Alexa. Initially, we aimed to use Alexa for interaction with our lights. However, we encountered a limitation that Alexa lacked the functionality to activate lights in response to our sensor triggers. To overcome this, we tried to explore the integration of a third-party program to act as a sensor activated with webhooks. However, the Arduino Cloud IDE posed limitations with webhooks, forcing us to rethink our approach. Finally, we decided to go for a direct connection to Wi-Fi and communicate with smart lights using UDP and JSON. This solution proved to be the way to go and aligned with our project requirements, overcoming the challenges posed by the initial choice control through Alexa and the limitations of webhooks in the Arduino Cloud IDE.

3.12.3 IR sender

When testing the first prototype, it did not turn the TV on. An oscilloscope revealed that the signal was present on the MCU pin connected to the IR LED, so it was likely that the SFH203 IR LED from Partco was defective. It was changed to a different generic Chinese IR LED that was able to turn the TV on and off successfully from up to 4m away.

3.12.4 Pressure sensor

The pressure sensor was not sensitive enough to detect a person sitting through a cushion, so it needed to be modified. Five 3D printed plastic discs with a diameter of 70mm and height of 15mm were glued on top of the sensor to concentrate the pressure. With this simple modification the sensor was able to detect a sitting person through many different kinds of cushions.

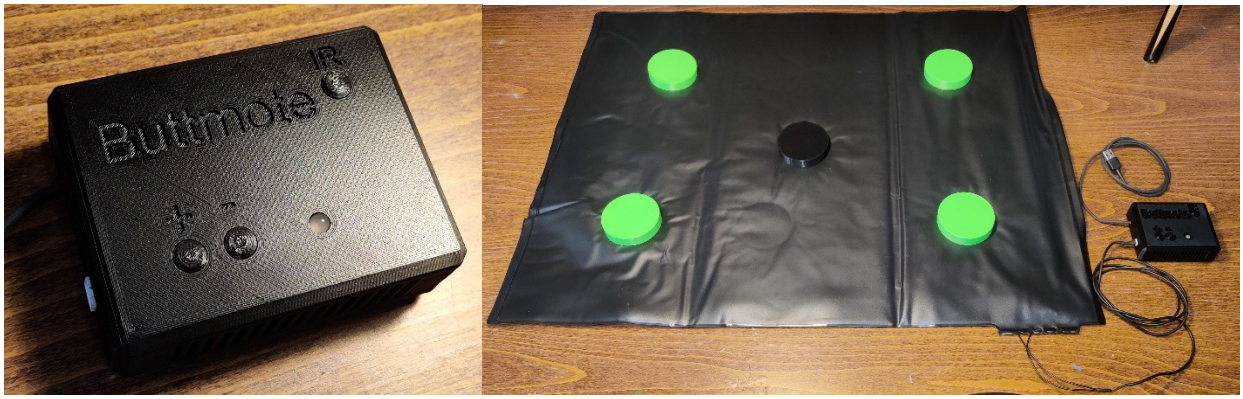
3.12.5 Access Point (AP) And Captive Portal

We ran into a few issues with the access point and captive portal implementation due to a possible lack of coordination within the team or unforeseen circumstances with the device memory. This resulted in us having to drop the feature altogether and settle with hardcoding the network credentials. The access point implementation was tested and working when run from its own implementation alongside the web UI, however when combined with the rest of the project it all fell apart and we were simply unable to fix the issue due to time constraints. The working version of the access point implementation allowed the user to connect to a passwordless access point on an Android device and be seamlessly redirected to configure the Wi-Fi credentials and IP address of the smart bulb.

3.12.6 ESP8266

In the early stages of the project, we faced some challenges with the initial microprocessor we chose, the ESP8266. We quickly realized that it required an older version of the Arduino IDE and specific versions of certain libraries. However, even after meeting these requirements, we continued to encounter issues. This led to the decision to switch from the ESP8266 to the ESP32.

4 Results



After resolving the initial IR LED issue, the device successfully controls TV functions up to a distance of 4 meters. The replacement with a generic IR LED was crucial for this functionality.

Modifications to the pressure sensor, particularly adding 3D printed discs, significantly improved its ability to detect pressure. This modification allowed the device to detect a person sitting through various cushions.

The implementation of Wi-Fi and UDP for communication with smart lights was efficient. Using JSON for messages enabled flexible control over the smart lighting system.

The web server, built using ESPAsyncWebServer and AsyncTCP, proved stable and could handle multiple requests. The client-side interface is straightforward, allowing users to monitor and control the Buttmote easily.

The project faced several challenges, such as issues with the PlatformIO library and integrating Amazon Alexa. Switching to Arduino IDE and direct Wi-Fi communication was key to overcoming these issues.

5 Conclusion

The project overall turned out to be a success. Despite some early bumps, we managed to get these components to work together, creating a solid and efficient home automation system. The ESP32 was reliable and teaming it up with the Arduino IDE was a game-changer. It led to a setup that was not just smart but also user-friendly, making managing a smart home a breeze for user

6 Appendix 1: API documentation

/api/status

After sending a GET request it responds with a JSON message containing data about the state the device is in currently. The data is as follows:

sitting

A boolean signifying whether a person is sitting on the device.

remoteEnabled

A boolean signifying whether or not the IR transmitter is enabled.

minutesSat

A floating point value signifying the amount of time a person has been sitting on the device.

minutesStood

A floating point value signifying the amount of time a person has not sat down on the device.

bufferMinutes

A floating point value signifying the currently set buffer time. This is the time the device will wait to begin executing the second task once stood up.

Example Typescript Interface

```typescript
interface ButtmoteState {
    sitting: boolean,
    remoteEnabled: boolean,
    minutesSat: number,
    minutesStood: number,
    bufferMinutes: number
}
```

Example Responses

```json
{
    "sitting": true,
    "remoteEnabled": true,
    "minutesSat": 1.5,
    "minutesStood": 0.0,
    "bufferMinutes": 5.0
```

```
}
```

```
{
    "sitting": false,
    "remoteEnabled": false,
    "minutesSat": 0.0,
    "minutesStood": 15.0,
    "bufferMinutes": 10.0
}
```

/api/settings

After sending a POST request it responds with a JSON message that contains data about whether or not the settings were updated successfully. It uses standard HTTP status codes with the responses.

Example Request

```
{
    "remoteEnabled": true,
    "bufferMinutes": 15.0
}
```

Example Responses

```
{
    "error": false,
    "msg": "OK"
}
```

```
{
    "error": true,
    "msg": "Bad Request!"
}
```