Voice Assistant

Hardware Lab

Instructor:
Dr. Ejlali

Mohammad Erfan Salima
400105014
Niki Sepasian
400105003
Asemaneh Nafe
400105285

2024/9/1

Table of Contents

Introduction

This project aims to design and implement an intelligent voice command recognition and execution system using an ESP32 microcontroller, an INMP441 digital microphone, and 4Ω or 8Ω speakers. The system is designed to accurately detect and respond to user voice commands, automating tasks based on the instructions given.

The system consists of three key components:

Wake Word Detection
This component continuously monitors ambient sounds, listening for a specific word or phrase known as the "wake word". Once the wake word is detected, the system activates and prepares to receive and process the user's voice commands. This ensures that the system is always ready to respond without consuming unnecessary power when not in use.

Voice Recording and Command Recognition
After detecting the wake word, the system immediately begins recording the user's voice. The recorded audio is then sent to a server where it undergoes processing to identify the spoken commands. The server uses advanced natural language processing (NLP) algorithms and artificial intelligence (AI) to analyze the audio and extract the user's intended instructions.

Command Execution
Once the server identifies the commands, the ESP32 microcontroller receives the instructions and executes them. These commands could include turning on or off connected devices, playing music, or performing other tasks based on the user's needs. The system is designed to provide quick and accurate responses to ensure a seamless user experience.

The primary components used in this project include the ESP32 microcontroller, chosen for its robust processing power and connectivity features; the INMP441 digital microphone, which captures high-quality audio with precision; and the 4Ω or 8Ω speakers, which are used to provide audio feedback to the user. Additionally, an I2S audio amplifier is employed to ensure that the output audio is clear and powerful enough for various environments.

One of the standout features of this system is its low-power architecture, allowing it to continuously listen for the wake word without significantly draining energy. This is particularly advantageous for Internet of Things (IoT) applications where energy efficiency is crucial.

Development Process Overview

System Design and Planning

The project began with the conceptualization phase, where the primary goal was to create a voice-activated system using the ESP32 microcontroller. The initial design focused on achieving low power consumption while maintaining high accuracy in voice command recognition.

- **Component Selection:** The ESP32 microcontroller was chosen due to its robust processing power, integrated Wi-Fi and Bluetooth capabilities, and suitability for IoT applications. The INMP441 digital microphone was selected for its ability to capture high-quality audio with low power consumption. For audio output, 4Ω and 8Ω speakers were considered to ensure compatibility with different use cases, and an I2S audio amplifier was included to boost the audio signal strength.

- **System Architecture:** A high-level system architecture was designed, mapping out the interactions between hardware components (microphone, speakers, ESP32) and software modules (wake word detection, audio recording, command recognition). This architecture served as the blueprint for subsequent development stages.

Hardware Setup

With the design in place, the next step was to set up the hardware. This involved connecting and configuring the various components.

- **Circuit Design:** A schematic was created to define the connections between the ESP32, the INMP441 microphone. The ESP32's GPIO pins were mapped out to interface with the microphone and speakers, ensuring correct signal routing for audio input and output. We used the following pin setup:

Table 1: Microphone Pin Setup

Microphone	ESP32
L/R	GND
GND	GND
VCC	V3V
SCK	GPIO 32
SD	GPIO 33
WS	GPIO 25

- **Prototyping:** The circuit was first assembled on a breadboard for prototyping. This allowed for easy adjustments and troubleshooting. The INMP441 was connected to the ESP32 via the I2S interface, which provides a direct digital audio connection, reducing noise and improving sound quality.

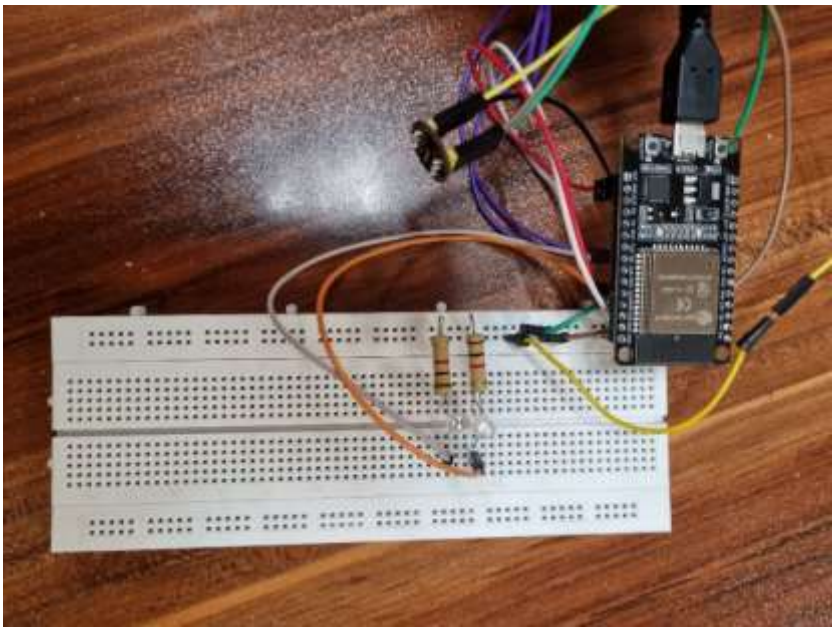


Figure 1 : Final Circuit

- **Testing Connections:** After assembling the hardware, each component was tested individually to confirm correct wiring and functionality. For instance, the microphone's output was checked to ensure it was capturing audio using Arduino's Serial Plotter.

Software Development

Software development was carried out in parallel with hardware setup, focusing on creating the core functionalities of the system.

Wake Word Detection:

The first software module developed was for detecting the wake word. This module runs continuously on the ESP32, monitoring the audio input from the microphone. A lightweight machine learning model was used to recognize the wake word with minimal computational overhead, ensuring the system could run efficiently on the ESP32.

For this step, we need to create something that will tell use when a "wake word" is heard by the system. This will need to run on our embedded devices - an ideal option for this is to use TensorFlow and TensorFlow Lite.

Our first port of call is to find some data to train a model against. We can use the Speech Commands Dataset (Link provided in the Miscellaneous folder.). This dataset contains over 100,000 audio files consisting of a set of 20 core commands words such as "Up", "Down", "Yes", "No" and a set of extra words. Each of the samples is 1 second long.

We have decided to use the word "Marvin" as our wake word.

With our training data in place, we need to think about how we are going to use this data, as it is pretty unlikely that feeding this raw audio to our model would create satisfactory results.

In our Data Generation file, we use various functions in order to turn this audio data into spectrograms. For more in detail information, refer to the file "DataGeneration.ipynb".

After that, we used the generated data to train our neural network responsible for detecting the word "Marvin". For more in detail examination of how we created the model and what setup we used, refer to the file "Train.ipynb".

As the final step, all we needed to do was to convert this model to TFLite (Convert Trained Model to) so we could use it later on in our project.

Audio Recording

Once the wake word was detected, the system was designed to start recording the user's voice. The audio data was buffered and prepared for transmission to a server for further processing. The ESP32's I2S interface was used to capture the audio data digitally, maintaining high fidelity.

Server Communication:

The recorded audio needed to be sent to a server for processing. A Wi-Fi connection was established using the ESP32's integrated Wi-Fi module. The audio data was then transmitted via HTTP POST requests to our Django server. Then this raw audio is processed (saved in .wav format) and sent (along with the necessary headers and authorization information) to its final destination, Wit.AI.

Our application in Wit.AI will then process this audio and send back the appropriate response. This response will get processed in our server and the necessary instructions will then get sent to esp32 for execution.

We utilized Wit.ai to recognize voice commands related to controlling lights in different parts of a home. We created specific applications within Wit.ai to identify and parse commands such as "turn on the kitchen lights" or "turn off the bedroom lights." These applications were trained to recognize various phrases associated with turning lights on or off in different rooms, enabling our system to translate spoken instructions into actionable commands for the ESP32 to execute.

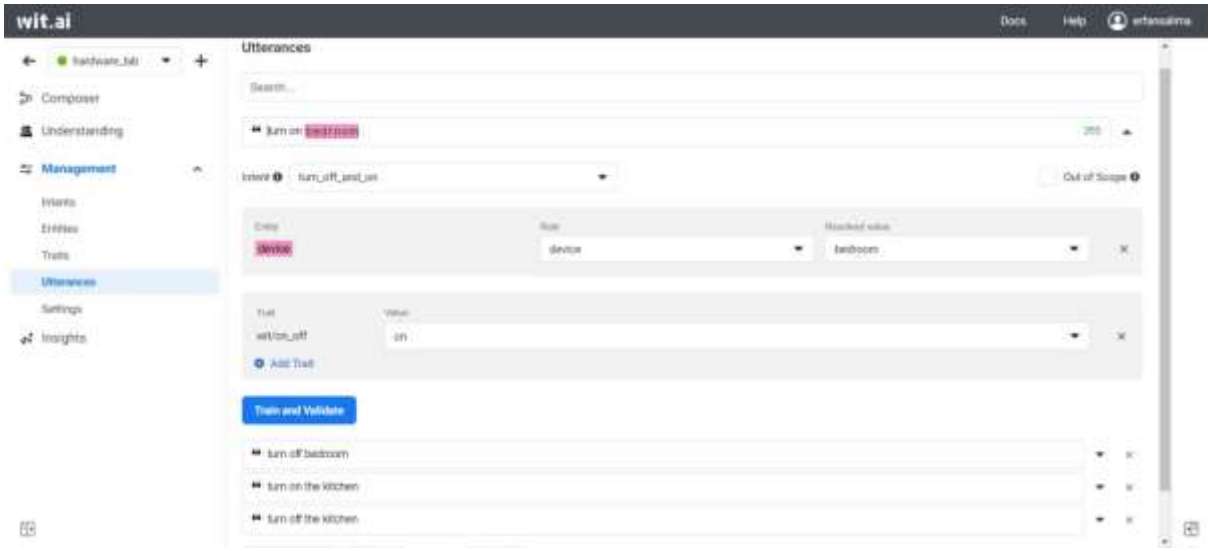


Figure 2 : Wit.AI Application Example

Command Recognition and Execution

Upon receiving the recognized command from the server, the ESP32 parsed the command and executed the appropriate action. This could involve turning on/off an LED, playing a sound through the speakers, or triggering other connected devices. The code was structured to handle multiple commands and provided feedback to the user via the speakers.

Testing and Debugging

Once the initial software was written, extensive testing was performed to ensure the system worked as intended.

- **Functional Testing:** Each functionality—wake word detection, audio recording, server communication, and command execution—was tested independently and then integrated to verify overall system performance. For example, the wake word detection was tested in different noise environments to assess its robustness.

- **Debugging:** During testing, several issues were identified and resolved. For instance, initial tests revealed latency in command recognition due to Wi-Fi transmission delays. To address this, optimizations were made in the data transmission protocol, and buffering

techniques were implemented to reduce the time between command detection and execution.

- **Iterative Improvements:** Based on testing feedback, several iterations of the software and hardware design were made. This included refining the wake word detection algorithm for better accuracy and adjusting hardware connections for more stable performance.

By the end of this phase, the system was able to reliably detect the wake word, record and transmit audio, and execute voice commands. The final prototype was fully functional, meeting the original project objectives.

Challenges Faced During the Project

Throughout the development of this project, we encountered several challenges that required creative problem-solving and iterative testing to overcome. Below are some of the key challenges we faced and how we addressed them:

Microphone Configuration and Audio Sampling Issues

One of the most significant challenges we encountered was getting the INMP441 microphone to work correctly with the ESP32. Initially, we noticed that the output of our neural network remained unchanged regardless of the audio input. This led us to question the validity of the data being recorded.

- **Troubleshooting:** To diagnose the issue, we used various plotters to visualize the audio data, but the results were inconclusive. We suspected that the problem might be related to hardware settings, so we adjusted multiple configurations, such as the I2S communication protocol settings and power supply arrangements.

- **Solution:** After considerable testing, we discovered that the root cause was the sampling rate. We initially set the sample rate to 4000 Hz, which was insufficient for capturing the full range of audio frequencies needed for accurate voice recognition. By increasing the sample rate to 44,100 Hz, we were able to capture higher-quality audio, which in turn improved the accuracy of our neural network's output. This adjustment resolved the issue and allowed us to proceed with reliable audio data.

Filtering of wit.ai and Connectivity Issues

Another major hurdle was the fact that wit.ai, our chosen voice recognition service, was filtered and inaccessible in our country. This presented a significant challenge, as we needed to find a way to connect the ESP32 to wit.ai despite these restrictions.

- **Failed Workarounds:** Initially, we attempted to bypass the filtering by configuring a proxy or DNS settings directly on the ESP32. However, these approaches failed to establish a connection with wit.ai, as the ESP32 could not properly route the data through the proxy or DNS settings.

- **Solution:** To overcome this, we decided to create an intermediate Django server that would act as a bridge between the ESP32 and wit.ai. The ESP32 would send audio data to our Django server, which was hosted on a laptop with an active VPN connection. The server would then forward the data to wit.ai for processing. This solution effectively bypassed the filtering issue and allowed us to continue using wit.ai for voice recognition.

Limited Storage for Voice Recording

The ESP32 lacks sufficient onboard storage, and without an SD card, we faced significant limitations in recording audio longer than one second. This constraint posed a challenge for capturing complete voice commands, which typically require longer recording durations.

- **Raw Audio Streaming:** Given the storage limitations, we decided to send the audio data directly as a raw stream rather than storing it locally on the ESP32. This approach allowed us to capture and transmit longer audio sequences in real-time without needing to store them on the device. While this solution required efficient handling of data streams and network communication, it enabled us to continue with the project without being constrained by the ESP32's storage limitations.

Code Walkthrough

Our code for this project is split into two main components: the ESP32 firmware and the Django server. Each part has a distinct role in the overall system—ESP32 handles wake word detection (using), audio capture, and device control, while the Django server processes the audio data (saves the raw audio in .wav format), interacts with the wit.ai API for speech recognition, and sends back commands to the ESP32 after processing the response from Wit.AI.

ESP32 Code

The ESP32 firmware is designed to handle the real-time processing of audio signals, detect specific wake words, record audio, and execute commands received from the Django server. The firmware consists of several key parts, each responsible for a specific task:

Wi-Fi Setup

This block of code is responsible for creating the Wi-Fi connection between ESP32 and our Django server. Setup function handles the initialization of the ESP32, setting up the pins for controlling the lights and connecting to the Wi-Fi network using the provided SSID and password. This connection is crucial for communicating with the Django server.

```
1
2  void setup() {
3    Serial.begin(115200);
4    pinMode(lightPin, OUTPUT);
5    pinMode(lightpin2, OUTPUT);
6
7    digitalWrite(lightPin, LOW);
8    digitalWrite(lightpin2, LOW);
9
10   // Connect to Wi-Fi
11   WiFi.begin(ssid, password);
12   while (WiFi.status() != WL_CONNECTED) {
13     delay(1000);
14     Serial.println("Connecting to WiFi...");
15   }
16   Serial.println("Connected to WiFi");
17
18   i2s_driver_install(I2S_NUM_0, &i2s_config, 0, NULL);
19   i2s_set_pin(I2S_NUM_0, &pin_config);
20   i2s_zero_dma_buffer(I2S_NUM_0);
21 }
22
```

Figure 3 : ESP32's Initial Setup:

I2S Configuration for Audio Capture

This code block sets up the I2S interface on the ESP32, specifying parameters like the sample rate, bit depth, and buffer sizes. The I2S configuration is critical for correctly capturing the audio data that will later be processed for wake word detection or sent to the server.

```
1
2  i2s_pin_config_t pin_config = {
3      .bck_io_num = I2S_SCK,
4      .ws_io_num = I2S_WS,
5      .data_out_num = 20, // We don't need data out
6      .data_in_num = I2S_SD
7    };
8
9  i2s_config_t i2s_config =  {
10     .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX),
11     .sample_rate = SAMPLE_RATE,
12     .bits_per_sample = i2s_bits_per_sample_t(16),
13     .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
14     .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_I2S),
15     .intr_alloc_flags = 0,
16     .dma_buf_count = 10,
17     .dma_buf_len = I2S_BUFFER_SIZE,
18     .use_apll = false
19   };
20
21 i2s_config_t i2sMemsConfigBothChannels ={
22     .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX),
23     .sample_rate = 44100,
24     //.sample_rate = 16000,
25     .bits_per_sample = i2s_bits_per_sample_t(16),
26     .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
27     .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_I2S),
28     .intr_alloc_flags = 0,
29     .dma_buf_count = 10,
30     .dma_buf_len = 1024,
31     .use_apll = false
32   };
```

Figure 4: ESP32 and I2S Microphone Configuration Data

Wake Word Detection Using a Neural Network

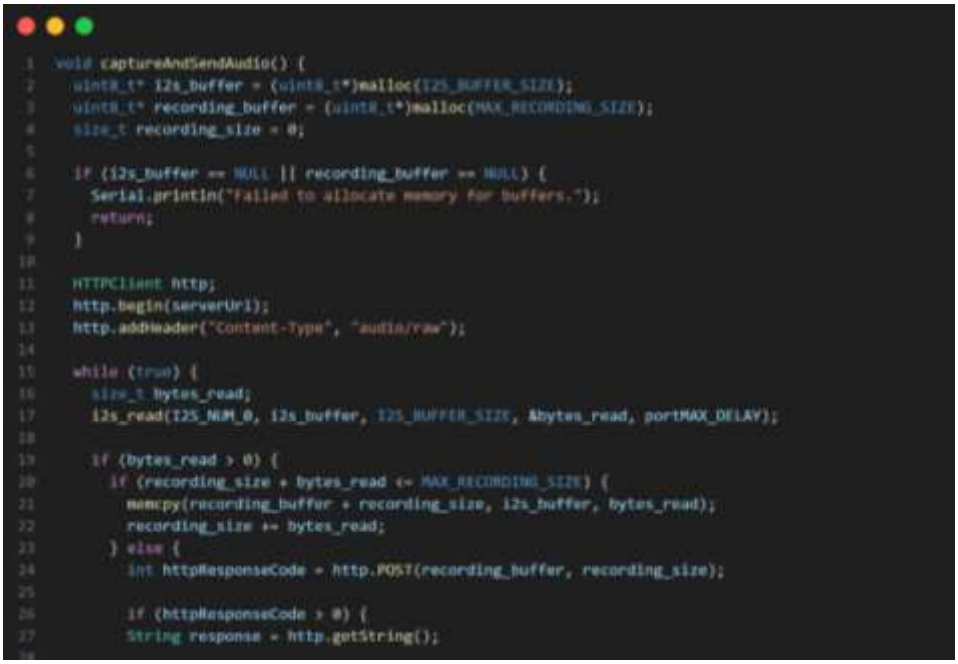
This function runs a neural network model to analyze audio data captured by the I2S interface. It converts the audio into a spectrogram and passes it to the neural network to detect if the wake word has been spoken. If the wake word is detected, the system becomes active and starts recording the subsequent voice command.

```
float getNeuralNetworkOutput(I2SSampler *sample_provider) {
    NeuralNetwork *m_nn = new NeuralNetwork();
    Serial.println("Created Neural Net");

    AudioProcessor *audio = new AudioProcessor(AUDIO_LENGTH, WINDOW_SIZE, STEP_SIZE, POOLING_SIZE);
    Serial.println("Created audio processor");

    RingBufferAccessor *reader = sample_provider->getRingBufferReader();
    reader->rewind(16000);

    float *input_buffer = m_nn->getInputBuffer();
    audio->get_spectrogram(reader, input_buffer);

    delete reader;
    float output = m_nn->predict();
    Serial.println("Neural Network Output:");
    Serial.println(output);

    delete m_nn;
    m_nn = NULL;
    delete audio;
    audio = NULL;

    uint32_t free_ram = esp_get_free_heap_size();
    Serial.printf("Free RAM: %u bytes\n", free_ram);
    Serial.println(output);
    return output;
}
```

Figure 5 : ESP32's Neural Network

Audio Capture and Transmission

This function captures the audio data after the wake word is detected and sends it to the Django server. The audio data is transmitted in chunks to ensure that it fits within the memory constraints of the ESP32. Once the audio is sent, the ESP32 waits for a response from the server, which indicates whether the lights should be turned on or off.

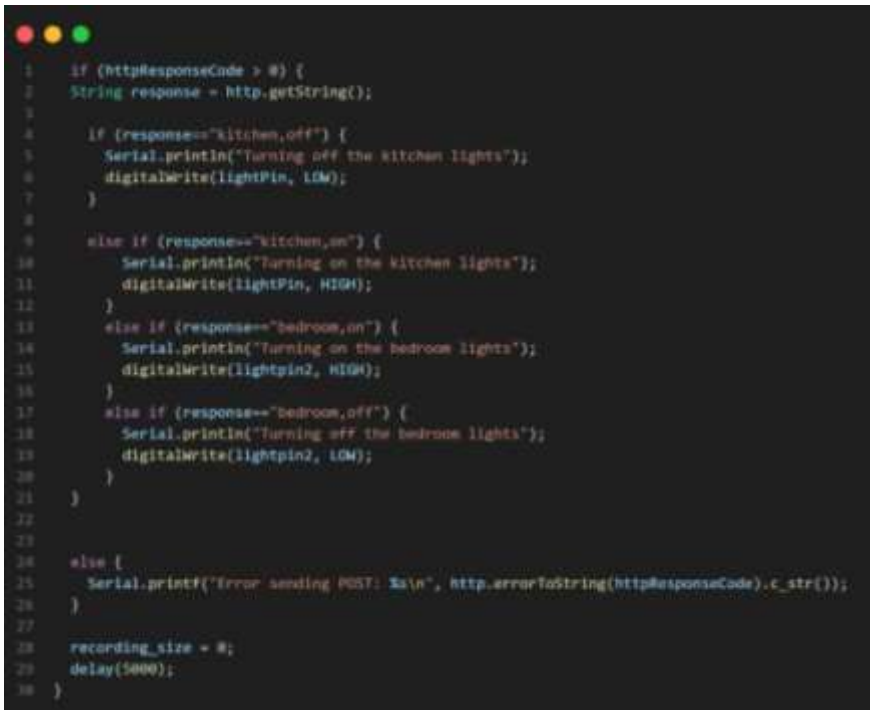


```
void captureAndSendAudio() {
  uint8_t* i2s_buffer = (uint8_t*)malloc(I2S_BUFFER_SIZE);
  uint8_t* recording_buffer = (uint8_t*)malloc(MAX_RECORDING_SIZE);
  size_t recording_size = 0;

  if (i2s_buffer == NULL || recording_buffer == NULL) {
    Serial.println("Failed to allocate memory for buffers.");
    return;
  }

  HTTPClient http;
  http.begin(serverUrl);
  http.addHeader("Content-Type", "audio/raw");

  while (true) {
    size_t bytes_read;
    i2s_read(I2S_NUM_0, i2s_buffer, I2S_BUFFER_SIZE, &bytes_read, portMAX_DELAY);

    if (bytes_read > 0) {
      if (recording_size + bytes_read <= MAX_RECORDING_SIZE) {
        memcpy(recording_buffer + recording_size, i2s_buffer, bytes_read);
        recording_size += bytes_read;
      } else {
        int httpResponseCode = http.POST(recording_buffer, recording_size);

        if (httpResponseCode > 0) {
        String response = http.getString();
```

Figure 6 :ESP32's Audio Transmission

Command Execution

This snippet demonstrates how the ESP32 interprets the server's response and performs the corresponding action, such as turning on or off the lights in different rooms. The response is parsed, and the appropriate GPIO pins are toggled.



```
    if (httpResponseCode > 0) {
    String response = http.getString();

      if (response=="kitchen,off") {
        Serial.println("Turning off the kitchen lights");
        digitalWrite(lightPin, LOW);
      }

      else if (response=="kitchen,on") {
          Serial.println("Turning on the kitchen lights");
          digitalWrite(lightPin, HIGH);
        }
        else if (response=="bedroom,on") {
          Serial.println("Turning on the bedroom lights");
          digitalWrite(lightpin2, HIGH);
        }
        else if (response=="bedroom,off") {
          Serial.println("Turning off the bedroom lights");
          digitalWrite(lightpin2, LOW);
        }
    }


    else {
      Serial.printf("Error sending POST: %s\n", http.errorToString(httpResponseCode).c_str());
    }

    recording_size = 0;
    delay(5000);
}
```

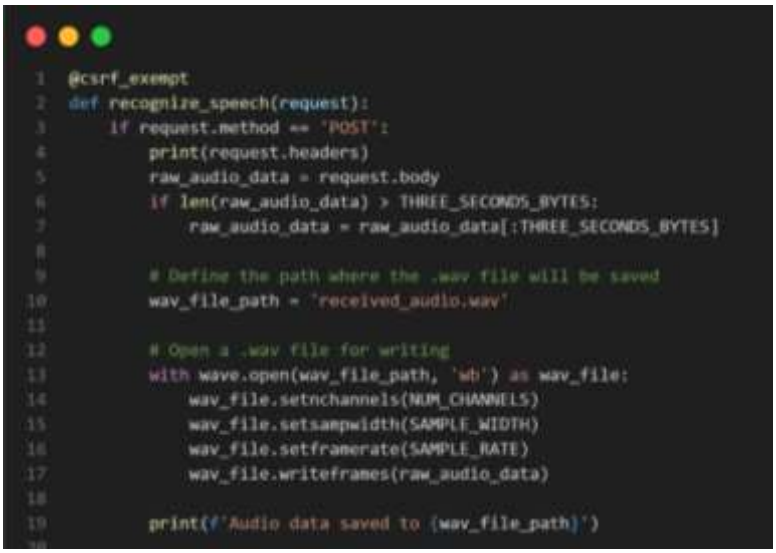
Figure 7: ESP32's Command Execution

Server

The server acts as an intermediary between the ESP32 and the Wit.AI API. It receives raw audio data from the ESP32, sends it to wit.ai for speech recognition, and returns the interpreted command back to the ESP32.

Receiving Audio Data

This section of the code handles the incoming POST request from the ESP32. It extracts the raw audio data from the request body, saves it as a .wav file, and prepares it for further processing. The audio is trimmed to three seconds if it's longer, ensuring the data sent to wit.ai is manageable.



```
@csrf_exempt
def recognize_speech(request):
    if request.method == 'POST':
        print(request.headers)
        raw_audio_data = request.body
        if len(raw_audio_data) > THREE_SECONDS_BYTES:
            raw_audio_data = raw_audio_data[:THREE_SECONDS_BYTES]

        # Define the path where the .wav file will be saved
        wav_file_path = 'received_audio.wav'

        # Open a .wav file for writing
        with wave.open(wav_file_path, 'wb') as wav_file:
            wav_file.setnchannels(NUM_CHANNELS)
            wav_file.setsampwidth(SAMPLE_WIDTH)
            wav_file.setframerate(SAMPLE_RATE)
            wav_file.writeframes(raw_audio_data)

        print(f'Audio data saved to {wav_file_path}')
```

Figure 8 : Server's Audio Receiver

Sending Audio to wit.ai

This block sends the saved audio file to wit.ai via an HTTP POST request. The response from wit.ai, which contains the recognized speech in JSON format, is then decoded for further processing.

```
        with open(wav_file_path, 'rb') as f:
            audio = f.read()

        headers = {
            'authorization': 'Bearer ' + wit_access_token,
            'Content-Type': 'audio/wav',  # Specify raw audio content type
        }

        # Make an HTTP POST request to Wit.ai
        resp = requests.post(API_ENDPOINT, headers=headers, data=audio)
```

Figure 9: Sending Audio to Wit.AI

Processing Wit.AI Response

Here, the server parses the JSON response from wit.ai to extract the relevant command (e.g., "turn on") and the device (e.g., "kitchen"). It then sends this information back to the ESP32 in a simple format, such as "kitchen,on" or "bedroom,off", which the ESP32 can easily process.

```
        response_text = resp.content.decode('utf-8')
        json_objects = response_text.strip().split('\r\n')

        parsed_objects = []
        for json_str in json_objects:
            if json_str:
                try:
                    data = json.loads(json_str)
                    parsed_objects.append(data)
                except json.JSONDecodeError as e:
                    return JsonResponse({'error': f"JSONDecodeError: {e} - Failed to parse: {json_str}"}, status=400)

        # Return the first FINAL_UNDERSTANDING object found
        for obj in parsed_objects:
            if 'type' in obj and obj['type'] == 'FINAL_UNDERSTANDING':
                value = str(obj['traits']['wit$on_off'][0]['value'])
                device = str(obj['entities']['device:device'][0]['value'])
                print(device)
                print(value)
                return HttpResponse(device + ',' + value)

        return JsonResponse({'error': 'No FINAL_UNDERSTANDING found'}, status=400)

    return JsonResponse({'error': 'Invalid request'}, status=400)
```

Figure 10 : Processing Wit.AI Response

Interpretation of results

In this project, the ESP32 device, coupled with the Django server, successfully implements a voice-activated system to control devices such as lights. The results can be interpreted across several key aspects:

Wake Word Detection and Audio Capture

- The ESP32 should accurately detect the specified wake word ("Marvin" in this case) and initiate audio recording once the wake word is recognized.

  The neural network embedded within the ESP32 correctly identifies the wake word, as evidenced by the system proceeding to record audio only after detecting "Marvin." This indicates that the neural network model is effective in distinguishing the wake word from other background noise or speech. The only observed downside to our model was the fact that because of the low quality of our microphone, sometimes it took more than one try to capture the attention of our system.

Audio Transmission and Server Communication

- The ESP32 should capture the audio following the wake word and successfully transmit it to the Django server for processing.

  The ESP32 captures the audio and sends it to the server in raw format. The transmission is confirmed to be successful as the Django server receives the audio, processes it into a .wav file, and forwards it to wit.ai for speech recognition. This seamless communication ensures that the system's voice recognition functionality is operational.

Speech Recognition and Command Parsing

- The Django server, using wit.ai, should accurately recognize the voice command (e.g., "turn on the kitchen lights") and parse it into actionable instructions (device and action) for the ESP32.

  The server correctly interprets the commands received from wit.ai. The parsed commands, such as "kitchen,on" or "bedroom,off", are sent back to the ESP32, indicating that the server correctly processes and understands the speech input. The wit.ai API effectively translates natural language into structured data that the ESP32 can act upon.

Device Control Execution

- Upon receiving the command from the server, the ESP32 should execute the corresponding action, such as turning on or off the lights in the specified location.

  The ESP32 performs the expected actions, toggling the GPIO pins to control the lights based on the received command. This successful control indicates that the entire pipeline—from wake word detection to device control—operates as intended.

Conclusion

The results demonstrate that the system effectively bridges voice commands to physical actions, utilizing the ESP32 for real-time audio processing and the Django server for advanced speech recognition via wit.ai. The observed outcomes align with the expected results, showing that the system is robust in detecting wake words, processing voice commands, and controlling devices accordingly.

Future Improvements

While the system is functional, potential enhancements could include:

- **Improved Neural Network Accuracy:** Further training of the neural network model could enhance wake word detection accuracy in noisy environments.

- **Extended Command Set:** Expanding the vocabulary and device compatibility could make the system more versatile.

- **Real-time Feedback:** Providing real-time feedback to the user, such as an acknowledgment sound upon wake word detection, could improve the user experience.