ELEC_ENG 326: Electronic System Design II

Northwestern University

LYLA: A Finger-Spelling Interpreter

Sneh Deshpande, Stephen Cheng

June 7, 2024

Contents

List of Figures

List of Tables

1 Abstract

Throughout the quarter, our team, comprised of Stephen Cheng, and myself, Sneh Deshpande, have been working on developing Lyla, originally intended to be used as a sign-language interpreter to facilitate communication between deaf individuals and those able to hear. However, as the quarter progressed, due to various challenges discussed in Section 8, our final product was closer to a American Sign Language (ASL) finger-spelling interpreter. The primary hardware components used in the product at the beginning of the quarter were a camera, microphone, and speaker, although as the quarter progressed an LCD display was also incorporated. A detailed overview of these components is provided in Section 6.1. The final product utilizes a machine learning model, trained on Edge Impulse, to perform inference on letters and numbers signed by users, captured through a camera on-board. Upon performing the inference, a sequence of letters and numbers is sent to a Tornado server, over the Websockets wireless communication protocol, and converted to an mp3 file containing audio of a gramatically correct sentence derived from the sequence of letters and numbers. This mp3 file is played back on a speaker, on-board as well, to enable a deaf individual to 'speak' to a person able to hear. Through conducting interviews, detailed in Section 10, we realized that enabling one-sided communication is not enough, and there must be a component of the product that allows a person able to speak to communicate with the deaf individual as well. Hence, a microphone on-board, can record audio from an individual, sent over Websockets to the same server, to be converted to text and displayed on the LCD monitor on-board. This encompasses a streamlined two-way communication between the two individuals. Further details on the functioning of the product are provided in Sections 6.2 and 7.2. This report describes the final product, and part of the process that led to its development, as well as a thorough evaluation of the performance of our device.

2 Introduction

Lyla is a finger-spelling interpreted that can be used to translate letters and numbers from ASL to enable communication between deaf individuals and individuals able to hear. Approximately 90% of children born deaf are born to parents of hearing. Hence, being able to translate ASL to English can serve as a tool for improving communication between members of the deaf community and those who are able to hear. Furthermore, this project can be expanded as a tool to learn ASL (letters and number for now), for those interested in the language. Beyond this, another motivation for this project was to develop skills in understanding edge computing on embedded systems, and explore task scheduling using real-time operating systems (RTOS).

There are various approaches used by researchers to interpret ASL, or finger-spelling. Starner et. al use "two real-time hidden Markov model-based systems for recognizing sentence-level continuous American sign language (ASL) using a single camera to track the user's unadorned hands." They achieve 92% accuracy when interpreting from a camera mounted on a desk, and 98% accuracy from a wearable device placed on a cap/hat. Furthermore, Munib et. al use Hough transform and neural network to translate static gestures (finger-spelling) from ASL to English "using a data set of 300 samples of hand sign images; 15 images for each sign" and achieving an accuracy of 92.3%. Another approach by Stanford researchers Garcia and Viesca proposes using Convolutional Neural Networks and Transfer Learning to classify hand gestures in ASL to English. They train on pre-trained images from GoogLeNet, and "ASL FingerSpelling Dataset from the University of Surrey's Center for Vision, Speech and Signal Processing" split between color and depth images. The aforementioned product uses a laptop camera and webpage to enable interaction with the product. Our approach was different, as we intended our device to be handheld and mobile with the user; thus, we used a camera placed on-board our PCB to capture static hand gestures. Considering this, instead of training on pre-trained data, where image quality taken from a laptop vastly differ to quality and resolution captured on our camera (OV2640), our model was trained on a MobilNetV2 architecture, on Edge Impusle. Given the limited training period of only 20-minutes on the free tier of the studio, our accuracy was limited as well. Moreover, Starner et. al describe capturing gestures of an angle facing the back of the user's hand; however, given the limited time for developing the project, and training itself, our approach considered gestures facing the front of the hand for simplicity. Since we wanted to implement edge computing we were constrained by the size of the model we were able to access, as deep learning models are strenous on limited memory. Avina et. al use a "deep learning model along with the rolling average prediction" to not just translate static gestures, but ASL words into sentences, which involve motion. To enable two-way communication, our device also has components like a camera, speaker, and LCD display; not present in approaches above, as they use webpages.

3 Design Constraints and Requirements

Within our design, there were several key requirements and constraints that we had to adhere to, in order to develop a product we deemed successful:

1. **Power**: Given that we prioritized mobility of the device for users to carry it along with them, ensuring a low-power application was necessary. Currently, our device is powered by a 3.7 V, 1200 mAh, Lithium-Ion battery (LIPO), and both microcontrollers operate on 3.3 V, as well as our other components. In order to constrain power resources, a key aspect of our design involved switching the power mode of our microcontrollers from *active mode* to *deep sleep mode*, which is a configuration possible on the ESP32 family of devices. A button is used to regulate switching between these modes for now; hence, users have to manually press this button to place the device into sleep after the duration of use. There is also a battery recharging circuit that the user can use to charge the LIPO using a microUSB cable.

2. **Cost**: Ensuring a viable product for a wide variety of consumers suggests that it must be affordable as well. In order to ensure affordability of our device, we used the ESP32 device family, which is a relatively low-cost microcontroller for the applications it provides, such as Wi-Fi, BLE, and 4-8 MB RAM. Our other components are also cost-effective; however, for a commercially viable product, access to a more expensive camera could be fruitful, though not necessary.

3. **Size of the Device**: Constraining the size and weight of our device was essential, considering we desired for users to carry it around with them in their pockets. To enable this, the device is not cluttered with unnecessary components, and the largest component on the device is the speaker, which can replaced with a smaller alternative if deemed necessary.

4. **Choice of Machine Learning Model**: Since the product computes inference on edge, we prioritized size efficiency of the model as well. We used a variation of the MobileNet model architecture called FOMO, which uses a fraction of the processing power and memory as the entire MobilNet architecture for object detection. It can be thought of as the first section of MobileNetV2 followed by a convolutional classifier. This does introduce some inaccuracies; however, as one of our goals was to familiarize ourselves with edge computing, it was a necessary sacrifice to conserve memory resources.

The aforementioned design constraints and requirements were reiterated throughout the design process and adhered to, to the best of our abilities.

4 Engineering Standards

We followed a few engineering standards for our design:

1. Wi-Fi: We used the IEEE 802.11 communication protocol to connect our ESP32 devices to the WebSockets Server.

2. PCB: While we do not explicitly manufacture the PCB, our PCB needed to follow the IPC-6011 Class 1 standard for PCB manufacturing.

3. C++: In writing our program in C++, we followed the ISO/IEC standards for C++.

4. AI: Although we did not intentionally follow the Asilomar AI Principles when creating our machine learnng model, we made sure our data collection methods were ethical since we only collected data on ourselves.

5 Broader Considerations

If our device was refined, we believe the broader impact of our design would be positive. Our device is intended to bridge communication gaps between deaf people and hearing people, so if the device became widely used, it would be able to help people communicate with each other better.

Currently, we do not believe there are any ethical issues surrounding our design. The one possible

data privacy issue is that our camera can take photos, but since the image data are immediately cleared after being classified, the privacy of people who do not want their photo taken are kept safe. One other possible legal issue is that if our dataset gets hacked, the hacker would have access to our photos. However, since our dataset is stored in the device, but rather in Edge Impulse's cloud, the hacker cannot access our dataset easily if they have access to our device. However, there still is a chance they can hack into Edge Impulse's cloud to steal our data.

If we were making our sign language interpreter a real product, we would make sure to address these possible concerns. With regard to the data privacy issue with images being taken unwillingly, we can stress in the product description that none of the images taken are saved and that they are nearly immediately deleted after being taken. With regards to hacking our dataset, we could move our data to another cloud service such as AWS, which may be more difficult to hack into.

6 Design Description

6.1 System Overview

Fig. 1 illustrates an overview of the flow of the design, and the various states and transitions the design holds. As detailed, there are two primary parts of our design: the microphone and speaker system, and the camera and LCD system, which are controlled by the ESP32 and ESP32-S3 respectively. To conserve power, as mentioned in Section 3, the device begins in sleep mode, and is regulated by a button to exit into active mode. The user has two choices: to begin inferencing using the camera to communicate with the individual able to hear/speak, or using the microphone to understand them. If inferencing, the user presses a button on-board, and the camera classifies the letters and numbers signed by the user, and to end inferencing, the user presses the button again. The buffer collected is then sent over Websockets to a Tornado server, running on port 8888. Both microcontrollers are connected to different handlers of the same tornado server as described in Section 6.2.5. The server constructs a gramatically correct sentence using OpenAI API and converts this sentence into an audio file, leveraging the ElevenLabs API. When the file is ready to be played on the speaker, the ESP32 requests this file using HTTP and plays it as an mp3. If the user chooses to enable the microphone using the button available, the microphone records any audio collected until the user presses this button again. As it is collected audio, it simultaneously sends it to the server mentioned above using Websockets for conversion to text, displayed on the LCD display. There are various components working in cohesion in our system; hence, task scheduling and time/resource management is essential, leading us to use a real-time operating system, detailed in Section 6.2.

Before elaborating on software components of the project, a brief overview of the main hardware components is helpful:

1. **ESP32 and ESP32-S3**: The purpose of using the ESP32 and ESP32-S3 is to enable on-chip processing as well as communicate with a server setup in AWS using WiFi. The ESP32 has both capabilities, while the ESP32-S3 has all these functionalities, with more RAM available (8 MB). The project utilizes the PSRAM capabilities of the ESP32-S3 to conduct edge computing, as classification of letters and numbers is performed on chip–enabled by Octal SPI on the ESP32-S3. Another reason for using both microcontrollers is that there are 4 main components that use I2C: microphone, DAC board, LCD monitor and camera; while the ESP32 only has two I2C peripherals on board. Instead of purchasing external peripherals or daisy-chaining external components, for the scope of this project it was simpler to just use two microcontrollers.

2. **INMP441 (Microphone)**:The INMP441 is a high-performance, low power, digital-output, omnidirectional MEMS microphone with a bottom port. In this project specifically, we are utilizing its I2S capabilities to interact with the microcontroller to collect raw digital audio data. The I2S protocol (Inter-IC Sound) protocol is a standard for digital audio communication used to transfer mono/stereo data between different ICs. It has three primary components:

   (a) Bit-Clock (SCK): this clock signals each bit of the data stream (data is normally transmitted on the rising edge of the clock,

   (b) Word Select (WS): Signals the start of data for each channel where each cycle represents one sample period of left/right channels (in the case of INMP441, which is a mono-channel device,

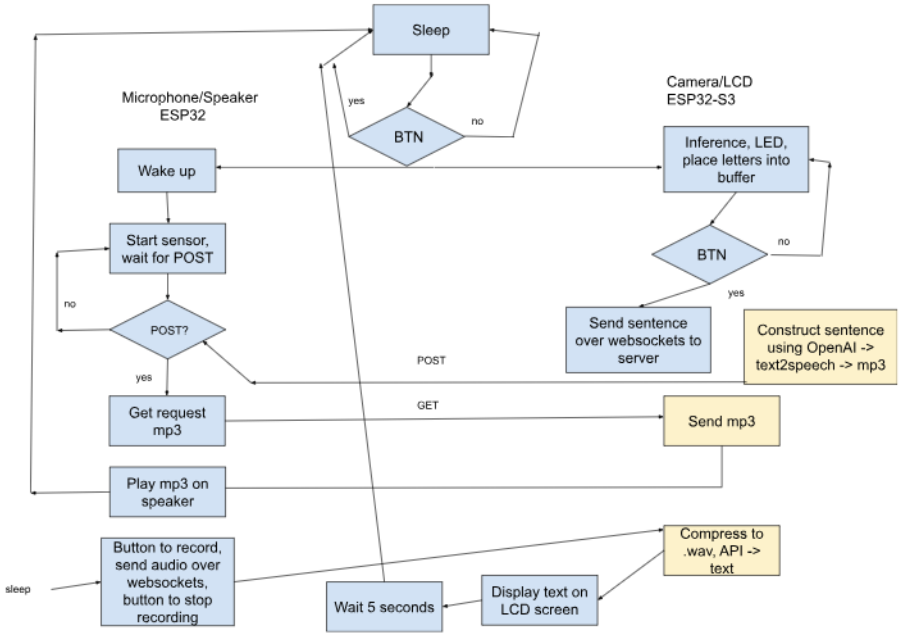


Figure 1: Overview of the flow of the design.

it only samples either the left or the right channel)

(c) Data Line (SD): Line for carrying audio data bits, generally transferred starting with MSB. Our implementation selects data from the left channel for the INMP441, which is achieved by grounding the L/R pin on the device.

3. **Camera (OV2640)**:The OV2640 is a camera module that can be used with the ESP32 in order to capture images using I2C. It operates on both 3.3 V and 1.2 V to capture images, which resulted in the incorporation of a 1.2 V voltage regulator as well. The resolution captured on the OV2640 is 320x240, for both training data and testing the device.

4. **Digital-to-Analog Converter (MAX98357A)**: The DAC board used in the project was the MAX98357A, which has a plethora of example code and libraries on PlatformIO for the ESP32. The DAC facilitated converting the mp3 audio to be played on the speaker in our project. The DAC also uses I2C to communicate with the microcontroller.

5. **LCD Display (B01N0KIVUX)**: The LCD monitor displays characters on the display, using I2C as the main protocol to communicate between the microcontroller and this module.

Other primary components such as the speaker are not discussed thoroughly, as they are considered simpler components and can be replaced by other alternatives as well. Moreover, simpler, more readily available components, such as pushbuttons, voltage regulators, resistors, and capacitors are not mentioned in this overview either as they can be purchased at user discretion, based on the PCB presented in Section 6.3.

The machine learning model used is Edge Impulse's FOMO (Faster Objects, More Objects) model, an object detection model adapted from the first half of MobileNetV2. MobileNetV2 is a deep learning model that uses convolutional neural networks and inception modules that consist of residual networks to perform image classification. The convolutional neural networks repeatedly downproject the width and height of the image while also increasing the number of channels of the data from 3 (RGB) up to 1280. Unfortunately, Edge Impulse did not release the algorithm for FOMO itself.

6.2 Algorithms and Code

The software component of this project uses PlatformIO as a tool to implement software, with code written in Arduino/C++. The project also leverages FreeRTOS, which is part of PlatformIO, as the RTOS

for task execution and program flow. All the hardware components also have libraries in PlatformIO that are easy to implement.

6.2.1 Deep Sleep and Active Mode

As mentioned in Section 3, a requirement of the device was to ensure a low-power application, since the device must be mobile for users to carry it in their pockets. In order to facilitate this, software was used to place the device into deep sleep mode on a button press, and send it to active mode upon a subsequent button press. We left it up to the user to decide when to perform these actions; however, ideally, there would be code in a more refined solution, to automatically sleep after an interval of no activity from the user. Two helper functions were primarily used to check if the button was pressed, and whether the device was in sleep mode, depicted in Fig. 2. The button press function returns a true if the button is pressed, or false if the button is not pressed. The goToSleep function enables the deep sleep mode, after setting an indicator LED to low in order to signal to the user the device being in sleep mode. An RTC data struct was used to keep track of whether the device is in sleep mode or not, since RTC data types are retained even in sleep mode, while other parts of memory are not available.

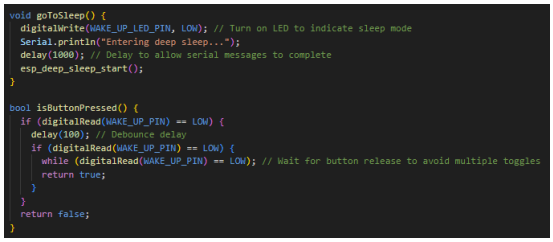


Figure 2: Helper functions for button press and determining if device is in sleep mode.

The code in Fig. 3 portrays the enabling of sleep mode functionality on the ESP32, and checking whether the cause of wakeup was a GPIO pin. If it was, then it toggles the sleep state, which is boolean in an RTC data struct, retained during sleep mode, to indicate whether the device should be in deep sleep or in active mode. It also controls an indicator LED to set it to high if in active mode, a signal to the user.

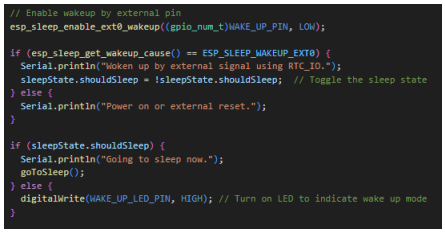


Figure 3: Code in setup to configure sleep mode and wakeup to active mode.

In the loop() function, the ESP32 constantly polls the button to check if it is pressed, and toggles the sleep state, the RTC data struct field in order to indicate sleep or active model. If the sleep state indicates sleep, then it calls the helper function in Fig. 2 to put the device into deep sleep, or toggles an LED high. The reason the loop function was used instead of a separate task like other code, is because the loop function is already a task in this format.

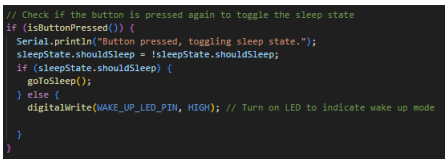


Figure 4: Helper functions for button press and determining if device is in sleep mode.

6.2.2 Wi-Fi

Since we use wireless communication in our design to communicate with a server setup in Tornado, ensuring a stable Wi-Fi connection is crucial. The library our project uses for managing this Wi-Fi connection is the WifiManager library, more specifically, a branch that allows PEAP, in order to sign in to eduroam, which has a Username and Password associated with the address. The Wi-Fi is initially setup in the setup() function of the main file; however, the code instantiates a task to enable reconnection of Wi-Fi as well. The method of connection/reconnecting to Wi-Fi involves connecting to a network setup by the ESP32 ("AutoConnectAP"), and configuring the Wi-Fi on the frontend automatically setup using the library. If the Wi-Fi is somehow disconnected, the ESP32 will automatically attempt reconnect due to a task scheduled using RTOS. This reconnect task is shown in Fig. 5.

```
void wifi_reconnect_task(void * parameters) {
    while (1) {

        if (WiFi.status() == WL_CONNECTED) {
            vTaskDelay(10000 / portTICK_PERIOD_MS);
            continue;
        }

        Serial.println("WiFi disconnected...reconnecting");

        WiFiManager wifiManager;
        wifiManager.autoConnect("AutoConnectAP");

        uint32_t start_attempt = millis();
        while (WiFi.status() != WL_CONNECTED && millis() - start_attempt < 10000);

        if (WiFi.status() != WL_CONNECTED) {
            Serial.println("WiFi reconnection failed!");
            vTaskDelay(20000 / portTICK_PERIOD_MS);
        }
        else {
            Serial.println("WiFi reconnected: " + WiFi.localIP());
        }
    }
}
```

Figure 5: Task to facilitate reconnecting to Wi-Fi automatically upon disconnection.

6.2.3 Microphone and Speaker

Fig. 6 demonstrates the configuration of I2S for the microphone. The audio input is sampled at a rate of 16000 samples per second, sampling 32-bits per sample. Notably, the code shows selecting the right channel for mono-channel input; however, this is a bug with the I2S driver in PlatformIO–the actual channel selected is the left channel, since grounding the L/R pin selects the left channel. The data is collected and transferred using DMA to a buffer initialized in the main file, using an I2S read. There are two DMA buffers allocated, with a buffer length of 1024 each, in order to conserve memory since the audio samples are streamed continously over websockets. Allocating more buffers or increasing the length of the sample buffer is unnecessary; however, when tested, led to memory leakage errors with the ESP32 due to lack of RAM.

```
void setupI2Smic() {
    esp_err_t i2s_error;

    i2s_config_t i2s_config = {
    .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX),
    .sample_rate = 16000,
    .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT,
    .channel_format = I2S_CHANNEL_FMT_ONLY_RIGHT,  // mono channel input (left channel) - bug from library
    .communication_format = (i2s_comm_format_t)I2S_COMM_FORMAT_STAND_I2S,
    .intr_alloc_flags = 0,
    .dma_buf_count = 2,
    .dma_buf_len = buf_len,
    .use_apll = false,
    .tx_desc_auto_clear = false,
    .fixed_mclk = 0};

    i2s_pin_config_t pin_config = {
    .bck_io_num = I2S_MIC_SERIAL_CLOCK,
    .ws_io_num = I2S_MIC_WORD_SELECT,
    .data_out_num = I2S_PIN_NO_CHANGE,
    .data_in_num = I2S_MIC_SERIAL_DATA
    };

    i2s_error = i2s_driver_install(I2S_NUM_0, &i2s_config, 0, NULL);

    if (i2s_error) {
        log_e("Failed to start i2s driver. ESP error: %s (%x)", esp_err_to_name(i2s_error), i2s_error);
        return;
    }

    i2s_error = i2s_set_pin(I2S_NUM_0, &pin_config);

    if (i2s_error) {
        log_e("Failed to set i2s pins. ESP error: %s (%x)", esp_err_to_name(i2s_error), i2s_error);
        return;
    }
}
```

Figure 6: Configuration of microphone using I2S.

Upon configuration, another function to read audio samples and place them in the I2S buffer is necessary, shown in Fig. 7. This function uses the i2s_read function written in the I2S driver in PlatformIO to

perform an I2S read, and places them in the buffer that is an input parameter to the function. This function is called in a task instantiated in the main file. The return type of the function is size_t given that the size of the bytes read is necessary for sending the audio samples over Websockets.

```
size_t mic_i2s_to_buffer(int32_t *buffer, int16_t buffer_size) {
  size_t bytes_read;
  esp_err_t read_status = i2s_read(I2S_NUM_0, buffer, buffer_size * sizeof(int32_t), &bytes_read, portMAX_DELAY);
  if (read_status != ESP_OK) {
    Serial.println("Error reading from I2S...");
    return 0; // Handle read error
  }
  return bytes_read;
}
```

Figure 7: Function to read audio samples and place them into a buffer.

Audio collection was performed based on a button press as discussed in Section 6.1. In order to collect audio, when a button was pressed, a variable: 'recording' was flipped to true (initially set to false), and upon a subsequent button press reset to false in the audio collection task. This was configured in the button interrupt handler defined in the main file, as displayed in Fig. 8.

```
// Button handler
void IRAM_ATTR handleButtonPress() {
    // Toggle recording state only if button release is detected
    if (digitalRead(PUSH_BUTTON_PIN) == LOW) {
        buttonPressed = !buttonPressed;
        if (!buttonPressed) {  // Trigger on button release to avoid multiple toggles
            recording = !recording;
        }
    }
}
```

Figure 8: Interrupt handler configured for button pressed for recording audio from microphone.

To collect audio specifically, a task was instantiated and pinned to Core 1, with 10000 bytes of stack memory allocated to it. The choice of the size of stack memory was arbitrary and chosen after trial-and-error testing of which stack size is effective. The audio collection task is detailed in Fig. 9. The way the audio collection task works if:

1. Poll recording flag until it is set to true.
2. If recording, set an LED pin high to indicate to user that microphone is recording.
3. Continuously record until the button is pressed again. Read data from i2s using the function detailed in Fig. 7.
4. Stream data collected in binary format to server using Websockets (data is unpacked on server-side and processed, further elaborated upon in Section 6.2.5.
5. Disconnect the Websocket once the button is pressed and the audio is no longer being collected and break out of the collection loop.
6. Flip the recording variable back to false to disable reading of the microphone data using I2S.
7. If not recording, LED pin indicating recording status is set to low, to signal to user that microphone is not currently recording.

The audio collection task is ran as a task in RTOS with a priority of 1, as it is a time-critical task, since audio must be recorded effectively on button press and must respond to user input in a speedy manner.

To facilitate Websocket communication for the streaming of audio, along with instantiating the Websocket, a Websocket event handler is also defined in the mic.cpp file, and enclosed in Fig. 10. This Websocket handler is a standard defined whenever a Websocket is instantiated, and since we did not intend to do anything special upon any Websocket events, the handler is not particularly used to execute any tasks either.

For the speaker to play audio once the server has prepared the mp3 file, the project leveraged the AudioFileSourceICYStream, AudioFileSourceBuffer, AudioGeneratorMP3, and the AudioOutputI2S libraries. The first one of these facilitates receiving the audio file over HTTP, while the second places it

Figure 9: Task to enable collection of audio and streaming over Websockets.

Figure 10: Websocket handler for microphone.

in a buffer that is then used to create an mp3 format by the third, and the last facilitates the playback of this mp3 format over I2S to communicate with the DAC board, connected to the speaker. Fig. 11 depicts the configuration of the necessary variables using the classes defined by the libraries, and the instantiation of helper functions provided in example code for receiving audio over HTTP. The reason HTTP was used to process audio from the server to the ESP32 was mainly due to the availability of example projects.

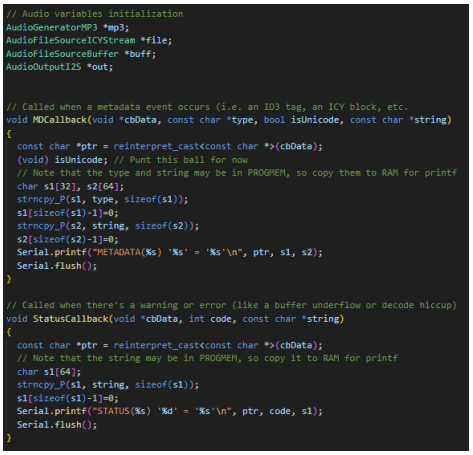


Figure 11: Code to configure audio variables and helper functions used to playback audio on speaker as mp3 file.

Since the mp3 file to be played on the speaker is prepared on the server, for the ESP32 to realize when this mp3 file is ready, in order to use an HTTP GET request to get the mp3 file, a global variable on the server is flipped (further discussed in Section 6.2.5). Hence, a function to check the status of the mp3 file is necessary, and provided in Fig. 18. This function is executed as a task scheduled by the RTOS, also run on Core 1 with priority 1, the same as the microphone task, with 10000 stack space allocated. The function instantiates the ESP32 as an HTTP server, and connects to a handler on the tornado

server that essentially sends a status variable to signal if the mp3 is ready. The HTTP server makes a GET request to check the status of the mp3, and if the mp3 is ready to be played, it adds an item to a shared queue, with another task, shown in Fig. 13. If the mp3 file is not ready yet, the task prints on the Serial Monitor to signal this and closes the HTTP server before restarting. This is essentially a polling mechanism. Although, not the most efficient approach, and starting an HTTP server and making requests consumes power, it is an approach that worked for the scope of this project.

Figure 12: Task to check the status of whether the mp3 is ready on server-side to be requested from ESP32.

As mentioned previously, an item is inserted into a Queue to signal that an mp3 file is ready to be played on the speaker. A Queue is used here, over mutexes or semaphores, as it is shared by both tasks, and when an item is available to be read on the queue, it can easily be used as a signal for the mp3 to be played. Once an item is ready on the queue, the MP3PlayTask request the mp3 from the server, converts it to an mp3 format again from bytes in a buffer, and plays it over I2S on the speaker connected to the DAC board. Once the mp3 is played, it stops playing it back on the speaker. The use of the libraries and example project was especially helpful in this, as we were struggling to understand how to play audio back from an mp3 file on a server.

Figure 13: Task to play the mp3 using the libraries for playing audio on a speaker.

6.2.4 Camera and LCD Screen

The ESP32-S3 program takes images using the OV2640 camera, classifies the images using the deep learning model, sends data in a buffer to a WebSockets server, and displays the text on the HiLetgo screen. The LCD screen uses the Adafruid_GFX and Adafruit_SSD1306 libraries to initialize and display text. The WebSockets client uses the WebSocketsClient library. The camera uses the "esp_camera.h" library. Since these tasks may sometimes be performed simultaneously, we implemented the code using RTOS

In the setup for the ESP32-S3, the device first enters deep sleep since we want the device to conserve

power when not being used. It listens for a pull-up button to be pulled to ground in order to wake up and perform its operations.

Figure 14: WebSockets Task

Once the device is put out of deep sleep, we initialize the camera, LCD screen, Wi-Fi connection, and WebSockets client. The OV2640 camera module required connections to its 8 data pins, XCLK, PCLK, VSYNC, and HREF, 3.0V, 1.2V, and reset. These are connections are specified in a camera config object that are passed to a function from "esp_camera.h" to initialize the camera.

Furthermore, we initialize some other LEDs and buttons. To show that the microcontroller is on, we drive an LED using a GPIO. We set another GPIO to be pulled up and connected to a button, which is toggled when we want to turn sign language classification on or off. The GPIO is connected to an interrupt to accomplish this. We have another GPIO drive an LED on when sign language classification is on, and turn off when sign language classification is off.

We then create 2 tasks using xTaskCreatePinnedToCore(): one for the client connection and one for the sign language classification.

The client connection is simply a while loop through a client.loop() statement, where client is a WebSocketsClient instance, and loop() maintains and checks the connection to the WebSockets server, shown in Figure 14.

The sign language classification task is more sophisticated. If the classification flag is true, we perform the following process, shown in Figure 15.

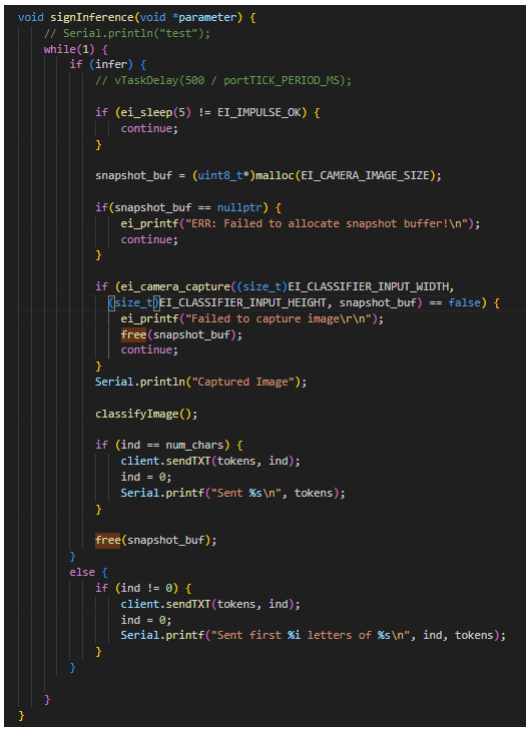


Figure 15: Classification Task

1. Dynamically allocate a buffer in the heap for our image data.
2. Capture an image and check if the buffer is not NULL. If it is not NULL, this means the image was successfully captured.
3. Resize the image data by interpolating the data using a function from the edge impulse library. This is necessary to make the data the correct input size.
4. Pass the image data through a classifier function that takes the input and returns details on the resulting classification made by the result, including the number of objects recognized, their coordinates, and the amount of time it took to run classification.
5. If a classification has been made, then add the character to a buffer. Blink an LED for 1 second to signal to the user that a character has been recognized
6. Check if the buffer has been completely filled, and if so, send the buffer to the WebSockets server so that the data can be processed in the backend. Then clear the buffer.
7. free the image buffer.

If at any point the classification flag is flipped to false due to the button being toggled, the process above is stopped, any letters already added to the buffer will be sent to the client, and the buffer will be cleared. Then the task waits for the classification flag to become true before continuing.

The LCD screen does not need a task since it only displays text when the WebSocket handler receives a message. When the WebSocket handler receives a message, the message is simply displayed on the LCD screen. This is demonstrated in Figure 16

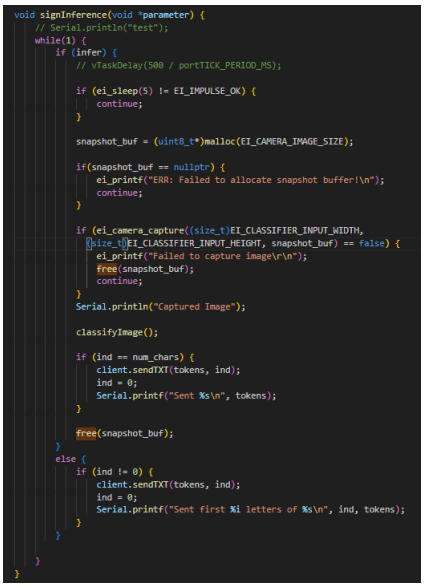


Figure 16: The LCD screen uses the WebSockets Handler

The last thing the ESP32-S3 does is check if the user wants the device to return to deep sleep. Within the void loop() of main.cpp, we have a function that checks if the sleep button has been pressed. Our function checks if the GPIO value is pulled low for longer than 100ms to avoid debouncing. Furthermore, this means our sleep button not only wakes the device up from sleep, but also sets the device to sleep.

6.2.5 Server-Side Code

In the backend, we used Python to run a WebSockets server using the tornado library. There are several handlers that are used, and we will go into detail on each one.

The Text2SpeechHandler WebSocketHandler handles the buffer of characters recognized by classification model on ESP32-S3. Shown in Figure 17, when the Text2Speech handler receives the message string, it passes the variable to a object of the class "Caller", which calls several APIs in succession. The first API call is to OpenAI's GPT 3.5 turbo model, a large language model. Our end goal was to prompt the model interpret the stream of letters and numbers as a grammatically correct sentence, while taking into account possible typos. However, our machine learning model was not reliable in its classifications, which meant our input stream of letters would have too many typos, making it very difficult for a sentence to be reconstructed. Thus, for the time being, our prompt is to return each character in the input separated by spaces. Our second API call is to ElevenLabs, which houses text to speech multimodal models, to convert the text message provided by OpenAI to speech. ElevenLabs returns an object that contains the image data, and the image data is written to an mp3 file. Then a flag that indicates if an mp3 file has been prepared is set to true.

Once the flag for the mp3 file is set to true, a handler to check if the mp3 file is ready, writes "mp3 ready" to its clients. In Section 6.2.3, it was explained how a task to poll if the mp3 is ready on the server makes an HTTP GET request–the code in Fig. 18 details how when a GET request is made to this handler, it writes "mp3 ready" to its client (the ESP32).

The ESP32 makes a GET request to request the mp3 file from the server. Fig. 19 portrays how upon a GET request, the file stored is written to the clients in byte format, and the header has the type 'audio/mpeg' as well.

```python
class Caller:


    def query(self, message):
        completion = self.client.chat.completions.create(
            model="gpt-3.5-turbo",
            messages=[
                # {"role": "system", "content": "Prompt: Interpret the stream of letters and numbers as a grammatically correct sentence. Take into account possit
                {"role": "system", "content": "Prompt: Return spaces in between each character in the input stream. For example, 1a3l should return 1 a 3 l"},
                {"role": "user", "content": message}
            ]
            )
        chat_response = completion.choices[0].message.content
        print(chat_response)

        headers = {
            "Accept": "application/json",
            "xi-api-key": self.xi_key
        }

        data = {
            "text": chat_response,
            "model_id": "eleven_multilingual_v2",
            "voice_settings": {
                "stability": 0.5,
                "similarity_boost": 0.8,
                "style": 0.0,
                "use_speaker_boost": True
            }
        }

        # Make the POST request to the TTS API with headers and data, enabling streaming response
        response = requests.post(self.xi_url, headers=headers, json=data, stream=True)

        # Check if the request was successful
        if response.ok:
            # Open the output file in write-binary mode
            with open(self.output_path, "wb") as f:
                # Read the response in chunks and write to the file
                count = 0
                for chunk in response.iter_content(chunk_size=self.chunk_size):
                    f.write(chunk)
                    print(len(chunk))
                    count += 1
            # Inform the user of success
            print("Audio stream saved successfully.")
            print(count)
        else:
            # Print the error message if the request was not successful
            print(response.text)
```

Figure 17: The Text2SpeechHandler uses a Caller object to processes the classified letters and convert the text into an mp3 file.

In order to handle the audio data streamed from the ESP32 as discussed in Section 6.2.3, a Websocket handler was written on the server-side as well, as shown in Fig. 20. On open, the handler tracks the time that the Websocket was opened, in order to create unique names for audio files after saving them. During streaming, when the ESP32 sends audio samples in binary format, the handler 'unpacks' the bytes sent, and since the audio data is sent in 32-bit format, ensures the right unpacking scheme as well. This audio data is stored in a buffer instantiated in the class. If the audio sample is received, when the Websocket is disconnected after sending all audio data, a helper function, in Fig. 21, saves the audio data to a .wav format and makes an API call to AssemblyAI API to convert the audio to text format. This text is then stored in a class variable called 'transcript' which is written to the LCD display as well.

The helper functions enclosed in Fig. 21 are used in the handler for audio data sent from the ESP32. The save_to_wav function uses the wave library in python the save the raw audio data to a .wav format. The number of channels, sample width, and framerate correspond to configurations made in Fig. 6, with 1 channel, 32-bit data, and 16000 samples per second. The find_latest_wav function is used to ensure that the file with the most recent timestamp is selected for converting to text, using the glob and os libraries in Python. Finally, the convert_audio_to_text function makes an API request to the AssemblyAI API, which transcribes the .wav file into a text format to return it. These functions are called as helpers across the Websocket handler for processing audio.

6.3 Printed Circuit Board (PCB)

Fig. 22 and 23 portray the PCB that we developed for the device. The PCB was the final one of our iterations. This PCB was populated with several pushbuttons as discussed in previous sections to enable microphone recording, deep sleep, and camera inferencing. Miscellaneous buttons were also added in case more functionality was required on our end–this ended up being useful, since we realized that our camera inference button was connected to GPIO 35 which is used to facilitate Octal SPI for the PSRAM in the ESP32-S3; hence, we switched to using the miscellaneous button for our final prototype. Moreover, the PCB also has programming headers for both ESP32s, along with reset and IO0 pins that are used to put

```
void checkMP3status(void *parameter) {

    static int num  = 0;

    HTTPClient http;

    while (true) {
        http.begin(check_mp3_url);
        int httpCode = http.GET();

        if (httpCode == 200) {
            String payload = http.getString();
            if (payload == "mp3 ready") {
                Serial.println("MP3 is ready to be played.");
                // Try to add item to queue for 10 ticks, fail if queue is full
                if (xQueueSend(mp3ready_queue, (void *)&num, 10) != pdTRUE) {
                    Serial.println("Queue full");
                }
                num++;
            } else {
                Serial.println("No MP3 ready yet.");
            }
        } else {
            Serial.println("Failed to connect.");
        }

        http.end();
        vTaskDelay(pdMS_TO_TICKS(2000)); // Check every 5 seconds
    }
}
```

Figure 18: HTTP Handler to notify the ESP32 that the mp3 is ready to be played.

```
# Handler to send MP3 file to ESP32 with Speaker
class GetMP3Server(tornado.web.RequestHandler):
    def get(self):
        print("Sending MP3 file...")
        filepath = "elevenlabs_test.mp3"  # Specify the path to your MP3 file
        self.set_header('Content-Type', 'audio/mpeg')
        with open(filepath, "rb") as f:
            self.write(f.read())
        self.finish()
```

Figure 19: HTTP Handler to send the mp3 file in bytes to the ESP32.

the ESP32s into 'Download' mode, for uploading firmware. The PCB also has the battery recharging circuit included. The final product used the microphone and DAC board modules, rather than breaking down their components, due to time constraints; hence, there are headers available for this. An unforseen issue that rose when testing the PCB was that the headers for the microphone were not the same distance apart as the headers on the PCB, hence wires had to be soldered onto the microphone to be connected to these headers effectively. The PCB performs well when tested and all connections seem to be in place upon testing.

6.4 Enclosure

The enclosure was created in Onshape, Figure 24, and printed using Bambu Studio. It consists of a box with holes for the microphone, camera, buttons, leds, headers, speaker, and charging port. The lid has holes for several screws to join the lid to the box. In order to verify that the the box was designed with the correct dimensions and positions for the holes, we also modeled the PCB in Onshape, and used the slider mate in the Assembly to verify that the box, PCB, and lid can be fitted together.

7 Final Product

7.1 Initial Goal

Before discussing our initial goal for the sign language interpreter, our original idea for the class was to create a device to aid in every day activities by using a Large Action Model. We were initially inspired by a startup called Rabbit AI, which uses a deep learning model to understand user requests and perform actions that a smartphone would typically do. This deep learning model was coined a Large Action Model, since it would comprehend language and execute an action on the application specified. For example, the user would speak into the device's microphone to ask it to schedule an Uber for a certain time, the device's large action model would comprehend the request, and then the model would sign into Uber using the user's credentials and schedule the Uber for the user. In other words, the model can learn how to use an application's interface, and execute the action for the user. However interesting this concept might be, we quickly realized that it was rather infeasible, since a Large Action Model would have been very difficult to create given our time span. Thus, we decided to create a sign language interpreter.

```python
# Handler to convert Audio Data to Text
class AudioWebSocketHandler(tornado.websocket.WebSocketHandler):
    latest_transcript = "No transcript available yet."
    audio_sample_received = False

    def open(self):
        self.start_time = int(time.time())
        print(f"WebSocket opened at {self.start_time}")
        self.audio_samples = []

    def on_message(self, message):
        # Place audio samples into buffer for processing
        samples = struct.unpack('<' + 'i' * (len(message) // 4), message)
        self.audio_samples.extend(samples)
        self.audio_sample_received = True

    def on_close(self):
        print("WebSocket closed")
        if self.audio_sample_received:
            filename = f"output_{self.start_time}.wav"
            # Convert audio samples to WAV file
            self.save_to_wav(self.audio_samples, filename)
            print(f"Saved WAV file as {filename}")
            # Convert WAV file to text
            transcript = self.convert_audio_to_text()  # Call to process the latest file
            print(f"Transcript: {transcript}")
            # Update the latest transcript
            AudioWebSocketHandler.latest_transcript = transcript
            # Send the latest transcript to all clients of Text2SpeechHandler
            Text2SpeechHandler.send_to_all("Message: " + transcript)
            self.audio_sample_received = False
```

Figure 20: Websocket handler to process audio data sent from the ESP32.

Our initial goal for the sign language interpreter was for it to recognize actual English Sign Language words. However, we realized that this would be very difficult to implement, not only due to the sheer number of words in ESL, but also because most words in ESL involve hand movement. Furthermore, classifying motion is a complicated endeavor for large deep learning models, and an even more monumental task for small machine learning models on the edge. Thus, we decided to have the machine learning model classify 24 letters excluding 'j' and 'z', which require motion, and the 10 digits. We wanted our device to be able to recognize the characters, send the characters to a server that would display the images on a web page. The idea behind this was that in a conversation between a deaf person and hearing person, the hearing person could look up what the person is saying by going to the web page.

After interviewing some people, we realized that it would be impractical to display the letters on a website, since if the hearing person wanted the deaf person read text, the deaf person could simply type out what he or she wants to say instead of signing it. Additionally, this would require the hearing person to pull out their phone to navigate to the web page, and we wanted to avoid having either person in the interaction use their phone. Thus, we refined our project goal so that the input stream of characters be passed through the OpenAI and ElevenLabs APIs so that an audio file could be played through a speaker. Thus, our new goal was to have OpenAI interpret the signed characters as a grammatically correct sentence and have ElevenLabs convert the text to speech. Since we anticipated our machine learning model to make some mistakes, we also planned to prompt OpenAI's language models by asking it to take typos into account. We also realized that if the hearing person wanted to communicate with the deaf person, they would have no means of doing so through our device. Thus, we added a microphone and an LCD screen, so that the hearing person can speak into the microphone, have AssemblyAI convert the speech to text, and then display the text on the LCD screen.

For the sake of ergonomics, we originally intended the device to be worn on a lanyard. Thus, the deaf person can carry it around with easily, and when they wanted to use it, it would be within hand's reach at all times.

7.2 Final Product

For our final product, instead of having our API call to OpenAI request that the input stream of characters be interpreted as a grammatically correct sentence, the API call would simply return the input letters and numbers but also include spaces in between each character. Besides this one change, we were able to achieve everything else from our initial goal. We were able to speak into the microphone and have our speech detected and converted to text to be displayed on the LCD. We were able to play audio on our speaker, and we were able to set up WebSockets server in the backend. With regards to

```python
# Save audio samples to WAV file
def save_to_wav(self, audio_samples, filename):
    with wave.open(filename, 'w') as wav_file:
        wav_file.setnchannels(1)
        wav_file.setsampwidth(4)  # Adjust as per the actual bit depth
        wav_file.setframerate(16000)  # Ensure this matches the actual sample rate
        for sample in audio_samples:
            wav_file.writeframes(struct.pack('<i', sample))

# Get latest timestamp to name WAV file
def find_latest_wav(self):
    list_of_files = glob.glob('./*.wav')
    if not list_of_files:
        return None
    return max(list_of_files, key=os.path.getmtime)

# Convert WAV file to text
def convert_audio_to_text(self):
    aai.settings.api_key = "76330e0ff50e43da93f4f15b5dc57fbf"
    # Initialize transcription service
    transcriber = aai.Transcriber()
    # Get most recent .wav file
    audio_url = self.find_latest_wav()
    # If no audio files are found, return
    if audio_url is None:
        print("No audio files found.")
        return
    # Transcribe the audio
    transcript = transcriber.transcribe(audio_url)
    if transcript.error:
        return transcript.error
    # Print the transcript
    return transcript.text
```

Figure 21: Helper functions to support processing of audio data on the server-side.

ergonomics, we decided not to use a lanyard for the device, since we realized that if the device is attached to the lanyard. This meant that when the device is raised to eye level to take images, the lanyard would show up in the pictures taken by the camera, which could make inference less consistent if the lanyard obstructs too much of the image taken. Images of our final product in and out of the enclosure are shown in Table 1.

7.3 Performance and Limitations

Our device was able to execute every task we wanted it to do, though it would do so to varying degrees of accuracy. The task of converting what the user is saying to text to be displayed on the LCD was generally successful. While the API calls to reconstruct the characters classified by the machine learning model worked successfully, the task of recognizing the sign language characters was rather inaccurate.

Our biggest limitation was the performance of our deep learning model. Our deep learning model had an overall test accuracy of less than 60%, shown in Figure 25 and it classified many hand signs incorrectly during inference. For example, "d" might be classified as "1", or "l" might be classified as "2", exhibiting poor precision. Thus, since the stream of letters the model was able to recognize was essentially gibberish, we needed to change our request to OpenAI since it was unrealistic to expect the the model to interpret a sentence from a stream of letters if half of the classified letters were wrong. In addition, due to the poor model performance, it would often not recognize any hand sign when a hand sign was being made, exhibiting poor recall. The full confusion matrix can be seen in Figure 26. Some letters perform much worse than others. For example, characters like s, t, u, w, 8, and 9 have very poor performance. This is probably because some pairs of letters are visually similar, such as s and t or 8 and 0.

Another limitation our final product had was lack of power. While the cameras on the ESP32-CAM devkit worked fine, the OV2640 camera on the final PCB had issues with capturing images. Every image that was captured would have blue and beige horizontal lines scattered across the image, which meant that the image was not being captured entirely correctly. When troubleshooting with the Serial Monitor, we occasionally got brownout errors, which indicated that it might be an issue with a lack of power.

8 Challenges Encountered

Throughout the design process, our team encountered various challenges both in hardware and software development. Primarily, the foremost challenge was the configuration of our all components and test-

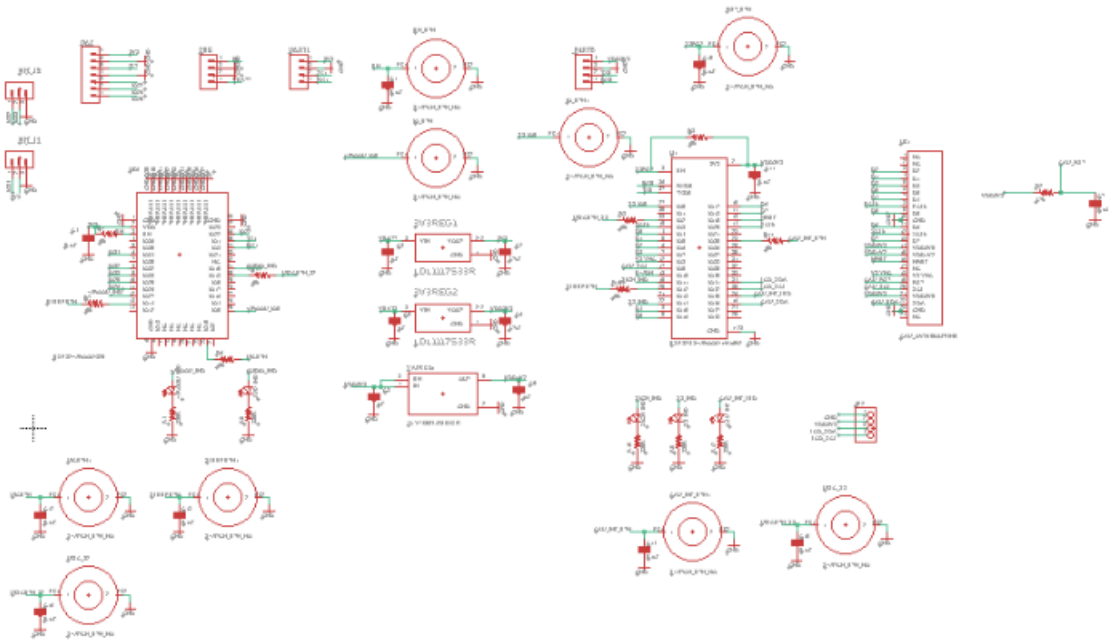


Figure 22: Configuration of microphone using I2S.

ing them on the breakouts. This was difficult given that we were not familiar with the I2S protocol or server-side scripting in great detail; hence, there was a steep learning curve. Furthermore, deciding the configuration of our hardware on our microcontrollers was fundamental due to audio processing and camera capturing, as well as our use of 4 I2C devices. Since the ESP32 has only 2 I2C peripherals, we considered using the ESP32 Pico board, which is essentially the ESP32 core, without the Wi-Fi antenna, to run our camera and LCD display applications on. However, configuring the Pico board proved to be difficult, since the development kit provided would simply reboot each time our code was flashed. Due to time constraints, we were unable to effectively debug this issue, and opted to use the ESP32-S3 board instead, as its configurations were very similar to the ESP32, an environment we were familiar with. There were also issues configuring the PSRAM on the ESP32-WROOM-1 board on our final PCB (corresponds to ESP32-S3), given that the development kit only had 4 MB of RAM, while the chip itself had 8 MB. We had to search online for solutions, and realized that the chip uses Octal SPI which is a configuration to be set in our platformio.ini file.

During the stage of testing our prototype on the breakout board, we suffered with several Guru Meditation errors, which pointed to memory leaks in our code. Given how heavy audio and image applications are on memory, a lot of debugging was involved in attempting to solve these issues. For example, with the microphone, as we were initially attempting to send data over HTTP POST rather than Websockets; however, allocating a buffer to handle an unknown amount of data was naturally difficult, and lead to memory management issues, which is why we switched over to Websockets. Similar issues with the speaker and camera were also present, which were solved through arduous debugging efforts. Moreover, with our first prototype of the PCB based on our breakout, we realized that more buttons and indicator LEDs were necessary to manage the flow of our design, as shown in Fig. 1; hence, our first PCB iteration was not a complete success–we were only able to use it to test the functionality of our hardware, but not all of our final software working together. Moreover, we believe that having more PCB iterations would have helped us identify these kinds of issues further, which is essentially attributed to our time-management and immense focus on improving software throughout the quarter.

Another main challenge we encountered throughout the quarter was the accuracy of our model for classifying static gestures in ASL. When we tested our model in the CG50 lab, most of the times it was not able to accurately classify signs, which led to a poor demonstration of the final product, as we originally intended for more efficient communication to happen. Essentially, with a more accurate model, we would

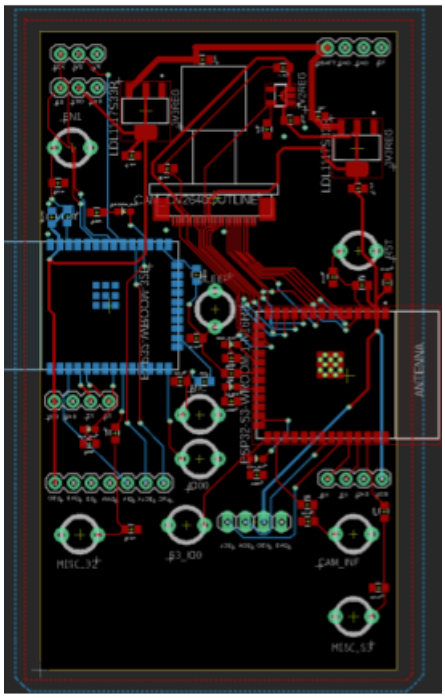


Figure 23: Configuration of microphone using I2S.

have been able to sign letters to form sentences for effective communication; however, since this was not possible, we were only able to obtain what seemed to be random letters classified by the machine learning model onto our LCD display. We experimented with a couple methods to improve the accuracy, by taking images in different environments for training, as we suspected overtraining. However, we believe that even then, we were unable to take a large enough variety of images in different backgrounds. A major issue with this was also the limited training time provided on the free-tier of Edge Impulse (20 minutes), which constrained our model's ability to accurately classify real-time images. Given more time, we could have potentially exported weights outputted after training the model on Edge Impulse, and trained our own model on a GPU available to us, for improved performance. There are also challenges associated with this, given the format of weights and the way the driver provided by Edge Impulse uses these weights. Additionally, even if we were able to train the model using a different pipeline than Edge Impulse, we also had the challenge of collecting training data. Over the course of the quarter, we collected over 8000 images, and it was took a surprisingly long time to label all the images. The process of labeling images for object detection was already expedited by Edge Impulse because of their labeling algorithm, so if we were to label images using other methods, it would have taken an even longer time.

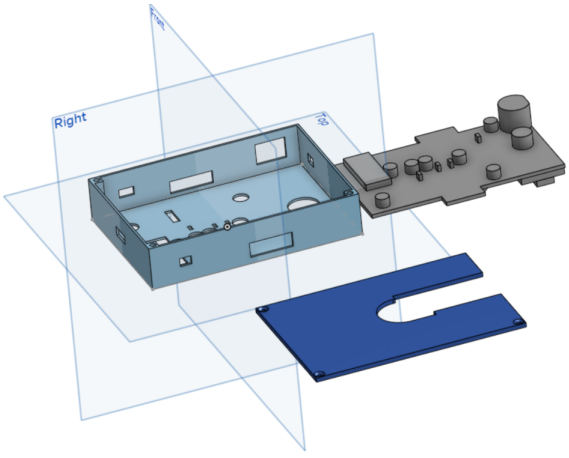


Figure 24: Enclosure created on Onshape

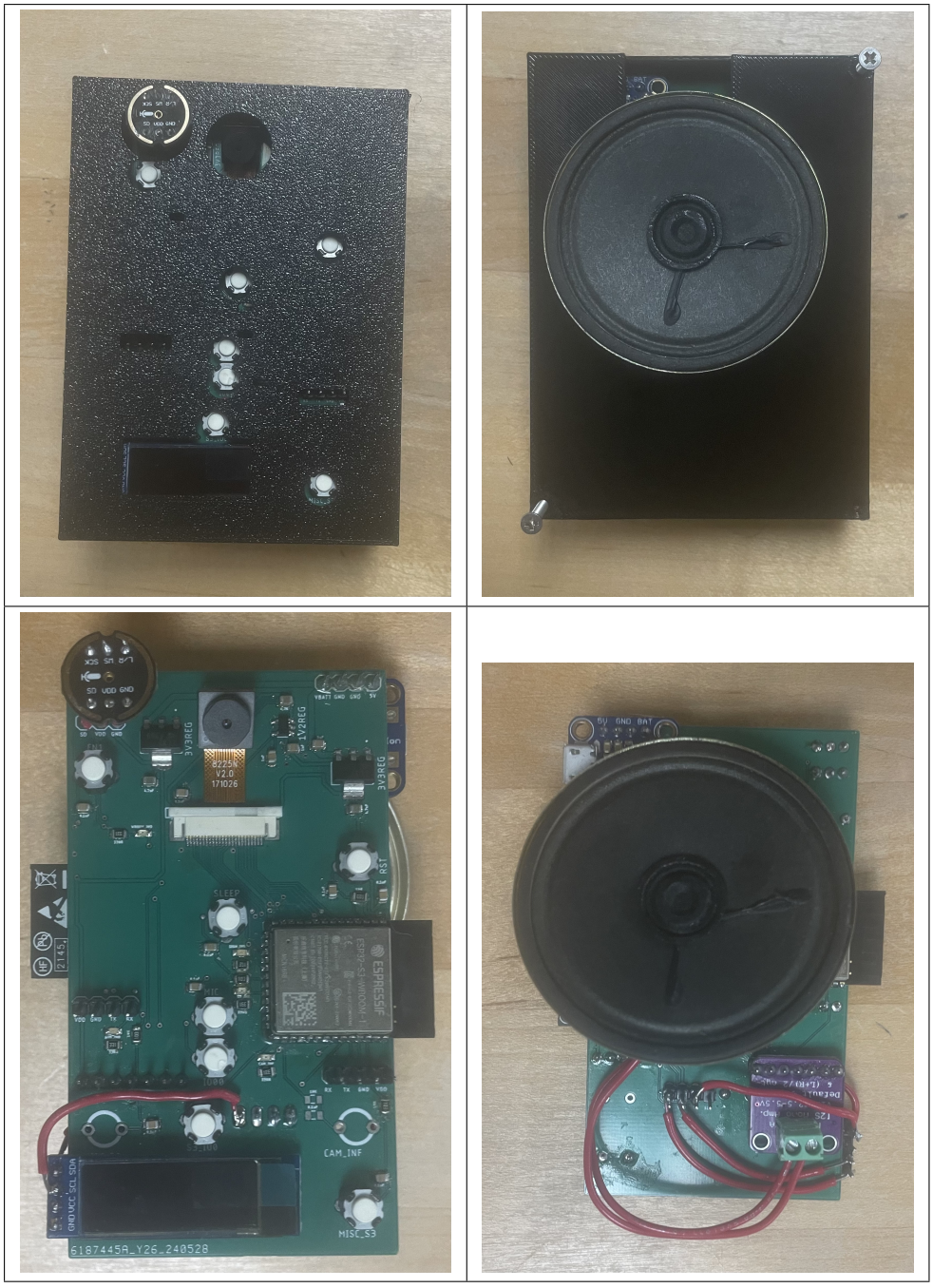


Table 1: Top Left: Front of Device in Enclosure; Top Right: Back of Device in Enclosure; Bottom Left: Front of Device; Bottom Right: Back of Device

Time was a big constraint in making our machine learning model.

Finally, a challenge we encountered with regards to our final software iteration was implementing RTOS. Since neither of us were familiar with RTOS and had little experience implementing it on embedded systems, we watched a reference playlist provided by our instructor. However, being able to understand which scheduling method to use (out of Queues, Mutexes, Semaphores, etc.), proved to be challenging given our limited experience. Being able to plan the overview of the system with our instructor certainly helped and his advice guided us through developing an effective RTOS. Memory management issues with regard to RTOS also arose; however, through trial and error we were able to allocate enough stack space for each of our tasks.

Overall, our project encountered several challenges, some of which (the main ones) are mentioned in

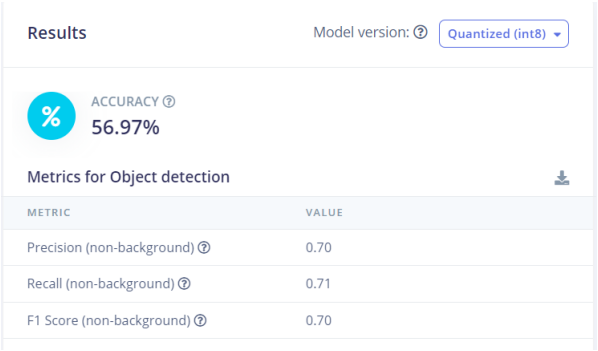


Figure 25: Model Performance on the Test Set

this section. However, through these challenges we were able to learn and explore further, both hardware and software components, in our project and gain a wider understanding of the implementation we were developing.

9 Planning and Organization

9.1 Gantt Chart

The Gantt Chart, figure 27, lists the rough schedule of how we completed our tasks.

9.2 Final Bill of Materials

The Bill of Materials, figure 28 was consistently updated throughout the quarter as we purchased more parts. Some items we had used that were not listed include the ESP32 Cam breakout, the ESP32-S3 module, and the ESP32-Wroom-32E devkits since we did not need to purchase these.

9.3 Communication Amongst Team Members

The communication amongst team members was exceptional throughout the quarter, as we were able to organize meeting times effectively both amongst each other, and with the course instructor. We were also able to help each other with problems we were having on our individual tasks, leading to a better understanding of the overall project as well. Moreover, at any stage if we felt we needed to discuss or change the trajectory of the project (which happened very often, especially early on), it was an equitable discussion.

9.4 Task Distribution and Management

The distribution of tasks was effective throughout the quarter as well. Stephen was primarily in-charge of ensuring the camera and LCD functioned as expected, while Sneh took care of the microphone and speaker. We wrote firmware and server-side code for our respective responsibilities, and helped each other clarify any doubts and confusions.

10 Market Research

11 million people in the USA are deaf, which is around 3.6% of our population. Currently, there is no device on the market that does sign language interpretation. Instead, people rely on sign language itself. We wanted this device to be a means to bridge communication between people who do not know sign language with deaf people.

We also interviewed 3 people throughout the project. We first interviewed Prof. Sahakian from the ECE Department, who is fluent in ESL. He explained to us that the most feasible option for our device was for it to be able to recognize letters and numbers. We also asked him if he thought it would be feasible for the language model to recognize hand signs from the back, to which he replied that it was

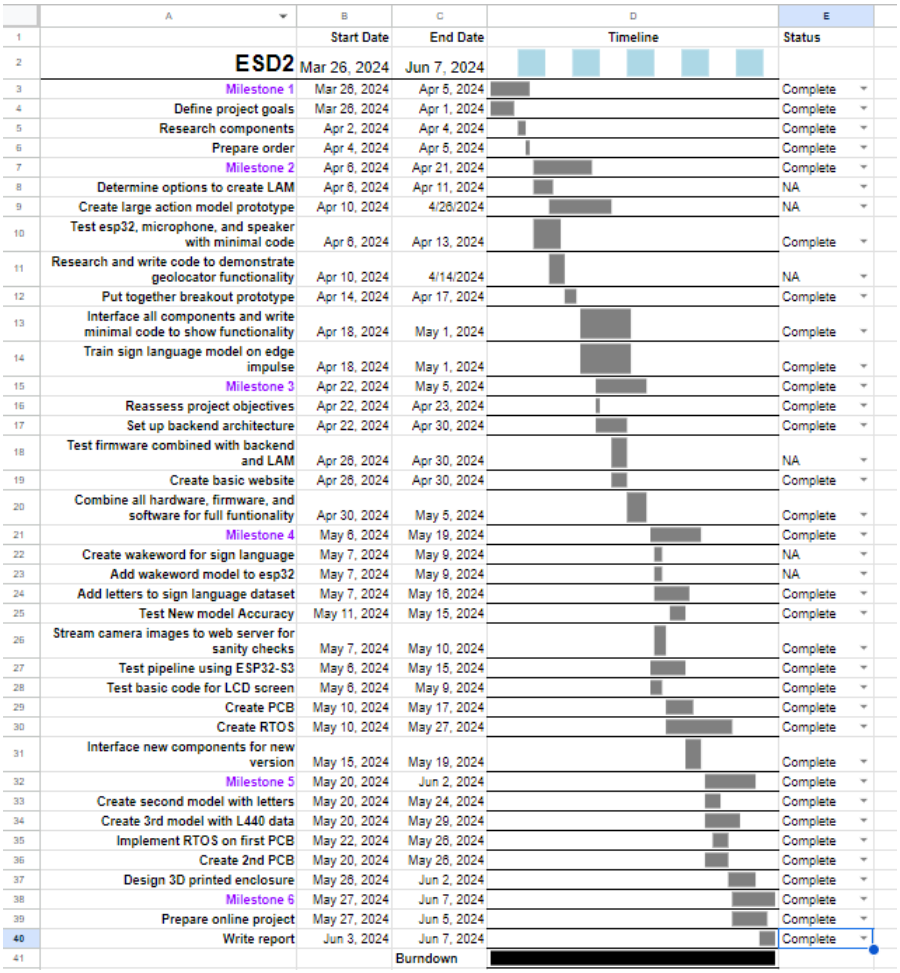
	F1-SCORE	PRECISION	RECALL
BACKGROUND	1.00	1.00	1.00
0	0.27	1.00	0.15
1	0.95	0.90	1.00
2	0.64	0.50	0.88
3	0.80	0.74	0.87
4	0.65	0.54	0.81
5	0.70	0.87	0.59
6	0.70	0.65	0.76
7	0.78	0.69	0.90
8	0.62	0.60	0.64
9	0.50	0.42	0.62
A	0.76	0.80	0.73
B	0.77	0.83	0.71
C	0.70	0.65	0.76
D	0.85	0.81	0.89
E	0.76	0.62	1.00
F	0.76	1.00	0.62
G	0.53	0.43	0.69
H	0.77	0.88	0.68
I	0.75	0.63	0.95
K	0.91	0.88	0.94
L	1.00	1.00	1.00
M	0.83	0.76	0.93
N	0.75	1.00	0.60
O	0.75	0.60	1.00
P	0.83	0.77	0.89
Q	0.81	0.70	0.95
R	0.35	0.75	0.23
S	0.13	0.33	0.08
T	0.13	1.00	0.07
U	0.48	0.50	0.47
V	0.33	1.00	0.20
W	0.69	0.69	0.69
X	0.94	0.89	1.00
Y	0.79	0.79	0.79

Figure 26: Model Confusion Matrix

possible but difficult since hand signs are meant to be recognized from the front. Thus, we settled on having the hand signs be recognized from the front.

We also interviewed Sam Liu, one of my friends who is interested in pursuing a career in audiology. After telling him about our initial goals, he proposed that the device should also have a feature that allows the hearing person to communicate with the deaf person. Thus, from interviewing him, we decided to include the microphone and LCD screen pipeline.

We also interviewed Lance Choi, a student on campus who has deaf grandparents. He told us that the device should be able to be used without accessing a smartphone, since the moment a smartphone is involved, the purpose of our device would be diluted since smartphones would also be able to allow a deaf person and hearing person communicate with each other. From this interview we decided to no longer use a website in our final product, since the hearing person would need to be able to access the website, likely using a smartphone.



	A	B	C	D	E
1		Start Date	End Date	Timeline	Status
2	**ESD2**	Mar 26, 2024	Jun 7, 2024		
3	Milestone 1	Mar 26, 2024	Apr 5, 2024		Complete
4	Define project goals	Mar 26, 2024	Apr 1, 2024		Complete
5	Research components	Apr 2, 2024	Apr 4, 2024		Complete
6	Prepare order	Apr 4, 2024	Apr 5, 2024		Complete
7	Milestone 2	Apr 6, 2024	Apr 21, 2024		Complete
8	Determine options to create LAM	Apr 6, 2024	Apr 11, 2024		NA
9	Create large action model prototype	Apr 10, 2024	4/26/2024		NA
10	Test esp32, microphone, and speaker with minimal code	Apr 6, 2024	Apr 13, 2024		Complete
11	Research and write code to demonstrate geolocator functionality	Apr 10, 2024	4/14/2024		NA
12	Put together breakout prototype	Apr 14, 2024	Apr 17, 2024		Complete
13	Interface all components and write minimal code to show functionality	Apr 18, 2024	May 1, 2024		Complete
14	Train sign language model on edge impulse	Apr 18, 2024	May 1, 2024		Complete
15	Milestone 3	Apr 22, 2024	May 5, 2024		Complete
16	Reassess project objectives	Apr 22, 2024	Apr 23, 2024		Complete
17	Set up backend architecture	Apr 22, 2024	Apr 30, 2024		Complete
18	Test firmware combined with backend and LAM	Apr 26, 2024	Apr 30, 2024		NA
19	Create basic website	Apr 26, 2024	Apr 30, 2024		Complete
20	Combine all hardware, firmware, and software for full funtionality	Apr 30, 2024	May 5, 2024		Complete
21	Milestone 4	May 6, 2024	May 19, 2024		Complete
22	Create wakeword for sign language	May 7, 2024	May 9, 2024		NA
23	Add wakeword model to esp32	May 7, 2024	May 9, 2024		NA
24	Add letters to sign language dataset	May 7, 2024	May 16, 2024		Complete
25	Test New model Accuracy	May 11, 2024	May 15, 2024		Complete
26	Stream camera images to web server for sanity checks	May 7, 2024	May 10, 2024		Complete
27	Test pipeline using ESP32-S3	May 6, 2024	May 15, 2024		Complete
28	Test basic code for LCD screen	May 6, 2024	May 9, 2024		Complete
29	Create PCB	May 10, 2024	May 17, 2024		Complete
30	Create RTOS	May 10, 2024	May 27, 2024		Complete
31	Interface new components for new version	May 15, 2024	May 19, 2024		Complete
32	Milestone 5	May 20, 2024	Jun 2, 2024		Complete
33	Create second model with letters	May 20, 2024	May 24, 2024		Complete
34	Create 3rd model with L440 data	May 20, 2024	May 29, 2024		Complete
35	Implement RTOS on first PCB	May 22, 2024	May 26, 2024		Complete
36	Create 2nd PCB	May 20, 2024	May 26, 2024		Complete
37	Design 3D printed enclosure	May 26, 2024	Jun 2, 2024		Complete
38	Milestone 6	May 27, 2024	Jun 7, 2024		Complete
39	Prepare online project	May 27, 2024	Jun 5, 2024		Complete
40	Write report	Jun 3, 2024	Jun 7, 2024		Complete
41			Burndown		

Figure 27: Gantt Chart

	A	B	C	D	E	F	G	H	I
1	Item Desc.	Mfg. Part #	Unit Price	1000 Unit Price	Quantity	URL	In Final Design	Total Unit Price	Total Bulk Price
2	Microcontroller	ESP32-S3	$15.000	$15.000	1	https://www.digik	Yes	$69.28	$59.63
3	Camera	OV2640	$2.110	$2.110	1	https://www.tem	Yes		
4	Microphone	INMP441	$6.790	$6.790	1	https://www.ama	Yes		
5	Speaker	CDM-20008	$1.800	$1.017	2	https://www.digik	Yes		
6	GSM Modem	SIG100	$5.270	$5.270	1	SIG100-IC Yama	No		
7	GPS Receiver	RYS3520	$7.200	$7.200	1	https://www.digik	No		
8	DAC Board	MAX98357	$3.150	$3.150	1	Amazon.com: Hi	Yes		
9	LCD 16x2	FNK0079	$7.950	$7.950	1	https://www.ama	No		
10	ESP32-pico-v2-02	ESP32-PICO-V3	$2.500	$2.500	1	https://www.digik	No		
11	Battery Charging Circ	1904 (MCP73831)	$6.950	$6.950	1	https://www.digik	Yes		
12	Lipo Battery 3.7V	PRT-13851	$5.500	$5.500	1	https://www.digik	Yes		
13	Microphone	INMP441	$6.790	$6.790	1	https://www.ama	Yes		
14	LCD 3.3V i2c	B01N0KIVUX	$6.490	$6.490	1	https://www.ama	Yes		
15	1.2 V Regulator	TLV70012DDCR	$0.480	$0.10	10	TLV70012DDCR	Yes		
16	3.3 V regulator	LDL1117S33R	$0.81	$0.38	10	LDL1117S33R S	Yes		
17							No		

Figure 28: Bill of Materials

11 Conclusion

Through this course, I was able to explore a plethora of new concepts: the use of real-time operating systems, understanding the I2S protocol, server-side scripting, and AWS instances and resource management, as technical skills. By engaging in this project, and improving the software, I was able to learn about task scheduling in RTOS, and although we did not incorporate mutexes and semaphores in our software, by watching the playlist recommended by our instructor, I was able to gain a lot of knowledge of concepts I had not previously encountered, or had limited interaction with. Moreover, the entire process of debugging, both hardware and software, was reiterated, and I gained more experience in it. Understanding how to strip down the parts of a problem when an error came up, and then building it slowly back up to see what exactly was causing the error, was a known method of debugging from 326; however, practicing it on a larger scale definitely improved my skills.

Furthermore, I was able to experience implementing edge computing on embedded systems through this project. Although our group did not create a novel model or architecture for the purpose of converting ASL to English, I was able to understand quantization of machine learning models to run them on our chip, and also interact with the driver created in Edge Impulse. This experience was definitely fruitful to me, in giving me a taste of edge computing. Moreover, understanding the problems and challenges, as well as limitations, behind edge computing was an eye-opener. With the multitude of memory issues that I had to debug, as well as understanding the limitations of our model, training, and overall performance on-chip, compared to simulation on a validation set was completely new to me.

Besides purely technical skills, working with a partner on a quarter-long project also had an impact on my time-management, communication, and organization skills. Although our communication was highly effective, and we distributed tasks in an equitable fashion, being able to organize meetings, and common work timings was a challenge that we had to overcome. I learned how to work on a shared Github repository, which, although it does not sound as complicated, can be a useful skill when working in a large team especially. Navigating the completion of our project towards the end meant that we had to put in several late nights of work; however, this helped me realize how important efficient time-management is throughout the project.

Given the chance to do the project from scratch, one thing I would definitely do is start earlier! I cannot emphasize enough the importance of starting earlier, as it would have shifted our timeline up by almost an entire week. This would mean that we would have another attempt at ordering a third PCB, and debugging any issues we had with our second order. Also, starting early would mean, instead of using modules for our microphone and DAC board, attached to the speaker with headers, we could make our design more compact and use the integrated-chips themselves with the rest of the application circuit. Moreover, having more time, would have allowed us to explore improving the accuracy of our model further as well, as this was a constraint on our project. Additionally, I start training our model as soon as possible, and take more images in different settings to avoid the issue of overtraining we encountered. I would also explore the opportunity to obtain the weights generated by the model in our library imported, and try to train the model on a GPU outside of Edge Impulse, to overcome the training time limitation imposed by using the free-tier of the studio.

Finally, there are various features I would add had there been more time. Primarily, rather than using button to regulate the inference and recording, I would try to automatically detect whether a user is speaking into the microphone, or the user has stopped signing, to start and stop these tasks. Moreover, I would also detect periods of inactivity to enable the deep sleep mode somehow, rather than having the user press yet another button. This is especially helpful, because we want our device to have a high ease-of-use, and incorporating so many buttons creates a barrier for that. Furthermore, I would try to detect static gestures from the back of the hand, so that the user can wear the device around their neck, and they do not have to hold it in one hand and sign with the other hand. This would make the device much easier to use for them as well. Given a much longer amount of time, I would see if we can use computer vision methods to detect motion of the hands and classify actual ASL words, rather than simple static gestures involved in finger-spelling. This would really make our product robust and commercially viable. The more immediate next steps would be to improve the model accuracy by collecting more data and training for longer however.

12 Class Feedback

Did you learn as much as you hoped in this class?

Not only did I learn all that I hoped in this class, I learned 10x more than I expected from this class. With 326, there is a lot of support through the instructor and other students as we are working on the same project. However, in this class, since each product is unique and has different requirements, navigating an almost-independent project in the time given was a valuable experience for me. Moreover, I was able to experiment with more advanced techniques such as RTOS and edge computing through this class–experiences that will serve me in the future as an aspiring embedded systems developer. Moroever, just through the sheer amount of debugging done on our project, I feel like I understand the hardware and software concepts used in the project like the back of my hand. I am able to look at problems now, and understand a general approach to debug them, even if they are not related to my own project. This class has been the most valuable class throughout my experience at Northwestern so far.

Do you have any suggestions for improvement of the class format or structure to increase learning?

As mentioned in discussions with the instructor, I feel like it would be valuable to have mini-courses on things like RTOS, or Edge Computing, for us to learn more theoretical concepts that we can apply in our own projects. Even having guest lecturers would add an interesting element to the class, by learning from people with actual industry experience in developing and researching new products. I feel like I never really worked during the allocated class time, and always preferred to work at night, so unless I was in a meeting with the instructor, this time was free generally. Also, I think something future students might benefit from is having a hard deadline to order their first PCB. We ended up ordering our first PCB so late, that we were only able to complete two iterations of our PCB. Having a hard deadline earlier in the quarter would incentivize students to be more proactive with developing their PCB designs, and help them in the end to test more thoroughly. Although this is implemented for each student, as this class goes, some groups may have different timelines, but I think overall students would benefit from having some sort of expectation for when the PCB design for their first iteration is due.

This quarter, we all worked with the ESP32 by default. Did you enjoy this experience, or would you have preferred to work with another MCU?

I enjoyed working with the ESP32 thoroughly. The fact that it is such an easy chip to work with, and has an active community of developers online with several example projects, made it much simpler to debug small issues that we encountered, such as the Octal SPI issue mentioned in the report. I do feel like there are other chips that are used more commonly across industry, such as the ones provided by Nordic Semiconductor, but the ESP32 is a good starter microcontroller I think for those trying to learn without a very steep learning curve, due to the online support. As someone who has worked with nrf microcontrollers before, the support for those compared to ESP32s is not as great and widespread, even though a lot of industry applications use them.

Sum up your thoughts to this project, the class, and your overall experience.

This class was so enjoyable for me, and I know it was for a lot of other students as well. As an instructor, the support Ilya shows to his students can't be put into words. He is always there for us, even if it is a Saturday night at 9pm and we ask an urgent question on Campuswire or through email. This class has also helped me find a community in ECE that I did not have before. Even though I have only known some of my classmates for 3 months, being in CG50 with them day and night, we have formed a bond that goes beyond the classroom. And for that, I am really thankful to Ilya, for motivating us to keep working hard and learn as much as we can. Other project courses at Northwestern that I have taken have seemed more like classes, but 327 feels more like a 'startup'. Everyone working on their own projects but discussing their progress with each other, asking each other for help–and Ilya can be thought of as our CEO/CTO, keeping us accountable and helping us when needed. This project has especially helped me learn new aspects of embedded systems that I am interested in, such as edge computing and embedded ML, which I will continue to explore beyond this course. Overall, this course never made me feel like I was working on a class, rather I was constantly learning, struggling, and curious, which is

something I will always appreciate.

Please share anything else that did not have a specific section in the report, especially any considerations that may be important in helping me assess your work throughout the quarter. Since this part of the report is not shared with your teammates, please feel free to be completely honest.

Both my teammate and I worked really hard to create a final prototype, and I understand that it was not able to form actual sentences in the end because of poor model performance and accuracy. This is something that we were trying to improve from the moment we got our first model working, but unfortunately could not get it to that point.

13 References

Avina, V.D.; Amiruzzaman, M.; Amiruzzaman, S.; Ngo, L.B.; Dewan, M.A.A. An AI-Based Framework for Translating American Sign Language to English and Vice Versa. Information 2023, 14, 569. https://doi.org/10.3390/info14100569

T. Starner, J. Weaver and A. Pentland, "Real-time American sign language recognition using desk and wearable computer based video," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 12, pp. 1371-1375, Dec. 1998, doi: 10.1109/34.735811. keywords: Handicapped aids;Wearable computers;Hidden Markov models;Cameras;Speech recognition;Face recognition;Computer vision;Computer Society;Real time systems;Pattern recognition,

Munib, Q. et al. (2006) American sign language (ASL) recognition based on Hough transform and neural networks, ScienceDirect. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0957417405003040 (Accessed: 07 June 2024).

Garcia, B. and Viesca , S.A. (no date) Real-time American Sign Language Recognition with Convolutional Neural Networks , Stanford University. Available at: https://cs231n.stanford.edu/reports/2016/pdfs/214_Report.pdf (Accessed: 07 June 2024).