Datasheet

SightSync

Project Motivation & Background [1.0]

Communication is a fundamental part of daily life, yet millions of people who are deaf or hard of hearing face barriers in understanding spoken conversations, especially in dynamic or group settings. While captioning apps exist, most rely on smartphones, are visually distracting, or require users to constantly hold and check a screen. This creates a disconnect between the conversation and the person trying to participate in it.

This project aims to solve that problem by creating a wearable device that listens to nearby speech and displays real-time captions directly in front of the user's eyes. By combining audio processing, wireless communication, and a compact projection system into a discreet glasses-based design, the device allows users to follow conversations more naturally and independently.

The impact of this technology goes beyond convenience. It provides a more seamless, private, and hands-free way for hearing-impaired individuals to engage in conversations at work, in public, or at home. It helps close the accessibility gap that current tools often leave open and gives users more control over how they interact with the world around them.

Project Goals and Planned Approach [2.0]

High level Goals [2.1]

Key objectives include:

- **Accurate Speech Capture**: Implement directional microphones and filtering techniques to isolate speech from background noise.
- **Real-Time Processing**: Use a low-latency speech-to-text pipeline to convert audio into readable captions with minimal delay.
- **Wearable Display Integration**: Develop or adapt a projection system that can render text clearly and discreetly onto a lens or display surface.
- **Wireless Communication**: Integrate Bluetooth for modularity between the audio capture module and the display hardware.
- **Power Efficiency**: Design a compact PCB and power system to support wireless use and several hours of continuous use without overheating or frequent recharging.
- **User-Friendly Design**: Ensure the device fits comfortably, has intuitive controls (e.g., tactile buttons), and can be worn like typical glasses.

The project will follow an iterative development process. Early stages focused on prototyping and testing each subsystem individually, including the audio pipeline, Bluetooth communication, and display visibility. Once core functionality was established, efforts shifted toward system

integration; designing a custom PCB, optimizing latency, and fitting all hardware components into a functional wearable form factor. Field testing in real-world environments helped validate usability and identify areas for refinement.

By the end of the project, the aim is to deliver a working prototype that demonstrates the full end-to-end system, from audio capture to visual caption display, providing a meaningful proof of concept for future development and potential commercialization.

Planned Approach [2.2]

To meet these goals, we divided the system into five core subsystems:

1. **Audio Capture**: Use an ICS-43434 MEMS microphone with I²S digital output for clean signal capture.

2. **User Interface App:** Route captured audio to a microcontroller and/or smartphone app. The app would use an AI-based API to convert audio to text.

3. **Display**: Use an OLED display mounted in the glasses to reflect the text through a transparent lens or mirror system, acting as a heads-up display.

4. **Power Management**: Design a custom PCB using the TP4056 IC to manage LiPo battery charging, voltage regulation, and noise filtering.

5. **Physical Frame**: Use 3D printing to prototype a glasses frame with modular attachment for electronics and room for all components.

Each subsystem was assigned to team members, with iterative testing planned at each stage before integration. The system architecture was designed to be modular and adaptable—starting with smartphone-based speech processing but allowing future expansion to fully on-device processing as hardware improves.

Actual Implementation [3.0]

Before diving into the specifics of the system implementation, let's first go over a high-level explanation of how our system works. The figure referenced below provides a visual overview:

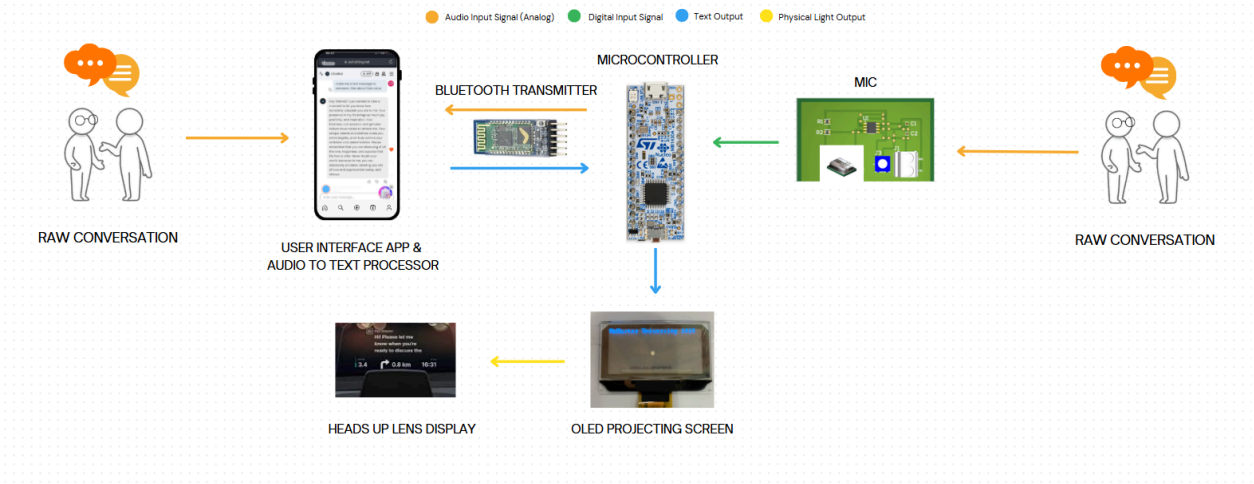


Figure 1: High Level System Design and Data Flow Diagram

First, there are two methods of audio capture. The first involves recording audio through a smartphone running the SightSync application. The SightSync app is an android app that we developed using Android Studio IDE, Java, and Kotlin. The second method uses a MEMS microphone integrated into our hardware system. In the second method, the audio is captured in analog form, converted to digital, and then sent to the STM32 microcontroller. The STM32 collects the audio samples in a buffer and transmits the buffered data to the smartphone application via the HC-05 Bluetooth module.

Regardless of the method used, the end result is the same: the smartphone application receives the audio. At this stage, we utilize Google's Speech Recognition API to convert the audio into text. This text is then sent from the smartphone application to the STM32, one byte at a time serially. The microcontroller processes the text and displays it on an OLED screen. The screen projects the text onto a series of mirrors within the system, which ultimately reflects it into the user's eyes.

This provides a high-level overview of how our system functions and what we aim to achieve. Now, we will explore the implementation details.

System Hardware & Firmware [3.1]

The hardware used in our system consists of four main components:

1. Adafruit ICS 43434 MEMS Microphone
2. 0.96-inch Waveshare OLED Display with an SSD1315 OLED Driver
3. Robojax HC-05 Bluetooth Module
4. STM32 F042K6 Nucleo Microcontroller

Adafruit ICS 43434 MEMS Microphone Overview

The Adafruit ICS-43434 is a digital MEMS microphone that outputs audio via the I²S protocol. It supports a frequency range from 60 Hz to 24 kHz, with a sample rate between 23 kHz and 51.6 kHz. It includes built-in signal conditioning and analog-to-digital conversion, producing 24-bit audio samples.

This microphone supports multiple operating modes, including High Performance, Low Power, and Sleep. For our system, we use the High Performance mode to ensure optimal audio quality for speech recognition. With a high signal-to-noise ratio (SNR) of 65 dBA, the ICS-43434 is well-suited for our application, which relies on clean audio input for accurate transcription.

A block diagram illustrating the internal components and signal flow of the ICS-43434 is shown below.

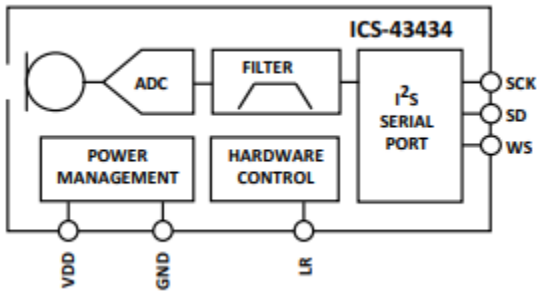


Figure 2: I2S ICS-43434 Microphone Internal Circuit Diagram

As shown, the audio input is passed through an analog-to-digital converter (ADC) and a filter before being output via the I²S interface.

Below are the physical dimensions of the ICS-43434 microphone, measured in millimeters:

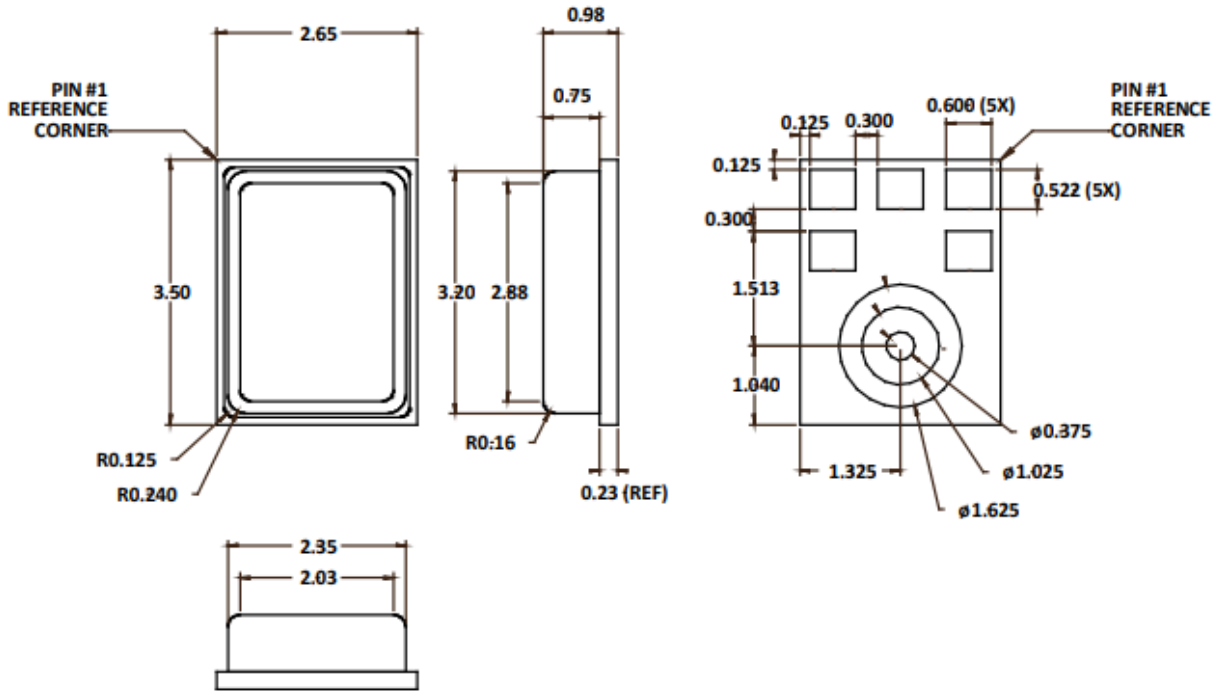


Figure 3: ICS-43434 Microphone Physical Measurements in mm

The ICS-43434 microphone is connected to the STM32 microcontroller using the following ports:

```
// ----------------- MICROPHONE ----------------------------

// SEL = GND
// LRCL = A3
// DOUT = A6
// BCLK = D13
// MICROPHONE VOLTAGE = 3.3v
```

Figure 4: ICS-43434 pins mapped to STM32 ports

Below is an image of the actual Adafruit ICS-43434 device used in our system:

Figure 5: Image of the front of the ICS-43434 microphone

Figure 6: Image of the back of the ICS-43434 microphone

Once the ICS-43434 is connected to the appropriate ports, audio capture can begin. This is achieved through Direct Memory Access (DMA), which allows audio data to bypass the STM32's core processing and avoid unnecessary instruction execution—preserving resources for other critical parts of the code.

Although audio capture is technically possible at this stage, the STM32 won't be able to interpret the audio until both I²S communication and DMA are properly established. These are essential for the STM32 to process incoming audio data correctly.

To enable I²S communication, specific microcontroller registers must be configured with appropriate values. These configurations were set up using STM32CubeMX IDE and are reflected in the code snippet below:

```
static void MX_I2S1_Init(void)
{

  /* USER CODE BEGIN I2S1_Init 0 */

  /* USER CODE END I2S1_Init 0 */

  /* USER CODE BEGIN I2S1_Init 1 */

  /* USER CODE END I2S1_Init 1 */
  hi2s1.Instance = SPI1;
  hi2s1.Init.Mode = I2S_MODE_MASTER_RX;
  hi2s1.Init.Standard = I2S_STANDARD_PHILIPS;
  hi2s1.Init.DataFormat = I2S_DATAFORMAT_24B;
  hi2s1.Init.MCLKOutput = I2S_MCLKOUTPUT_DISABLE;
  hi2s1.Init.AudioFreq = I2S_AUDIOFREQ_16K;
  hi2s1.Init.CPOL = I2S_CPOL_LOW;
  if (HAL_I2S_Init(&hi2s1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2S1_Init 2 */

  /* USER CODE END I2S1_Init 2 */

}
```

Figure 7: I2S Communication Initialization Code in the STM32

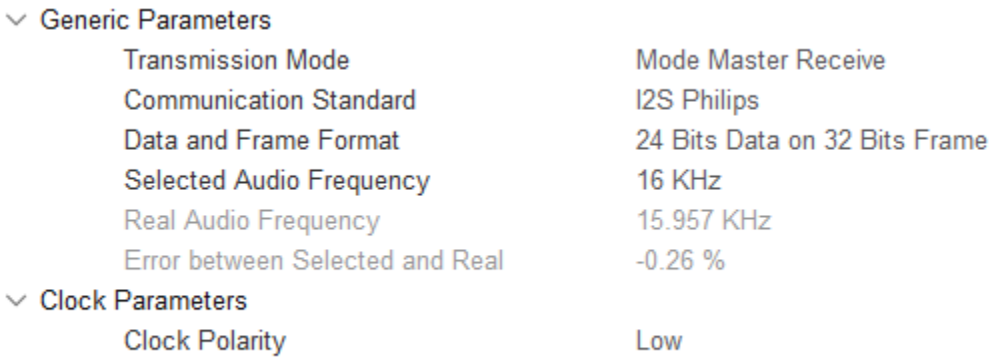


Figure 8: I2S Communication Configuration Settings on STM32CubeMX IDE

As shown, the STM32 drives the ICS-43434 microphone, operating in Master Receive mode, with the microphone acting as the slave. According to the ICS-43434 datasheet, we use the standard Philips I²S protocol with a data format of 24 bits per audio sample and a 32-bit frame—allocating 8 bits of padding.

We sample audio at 16 kHz, which is optimal for speech recognition tasks in AI models. The STM32 provides the Bit Clock (BCLK) and the Left-Right Clock (LRCL) to drive the microphone. In our setup, we utilize only the right channel, since the system is mounted on the right side of the user's face—left and right clocks correspond to the microphone's left and right channels, respectively.

With communication between the STM32 and the ICS-43434 now established, we proceed to configure the DMA controller as shown in the code below:

```
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();
  hdma_spi1_rx.Instance = DMA1_Channel2; // Example channel
  hdma_spi1_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_spi1_rx.Init.PeriphInc = DMA_PINC_DISABLE;   // Peripheral address fixed
  hdma_spi1_rx.Init.MemInc = DMA_MINC_ENABLE;       // Memory address increments
  hdma_spi1_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD; // 32-bit
  hdma_spi1_rx.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;    // 32-bit
  hdma_spi1_rx.Init.Mode = DMA_CIRCULAR;            // Circular mode
  hdma_spi1_rx.Init.Priority = DMA_PRIORITY_VERY_HIGH;   // Adjust as needed
  HAL_DMA_Init(&hdma_spi1_rx);
  __HAL_LINKDMA(&hi2s1, hdmarx, hdma_spi1_rx);

  /* DMA interrupt init */
  /* DMA1_Channel2_3_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel2_3_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel2_3_IRQn);

}
```

Figure 9: I2S DMA Initialization Code in the STM32

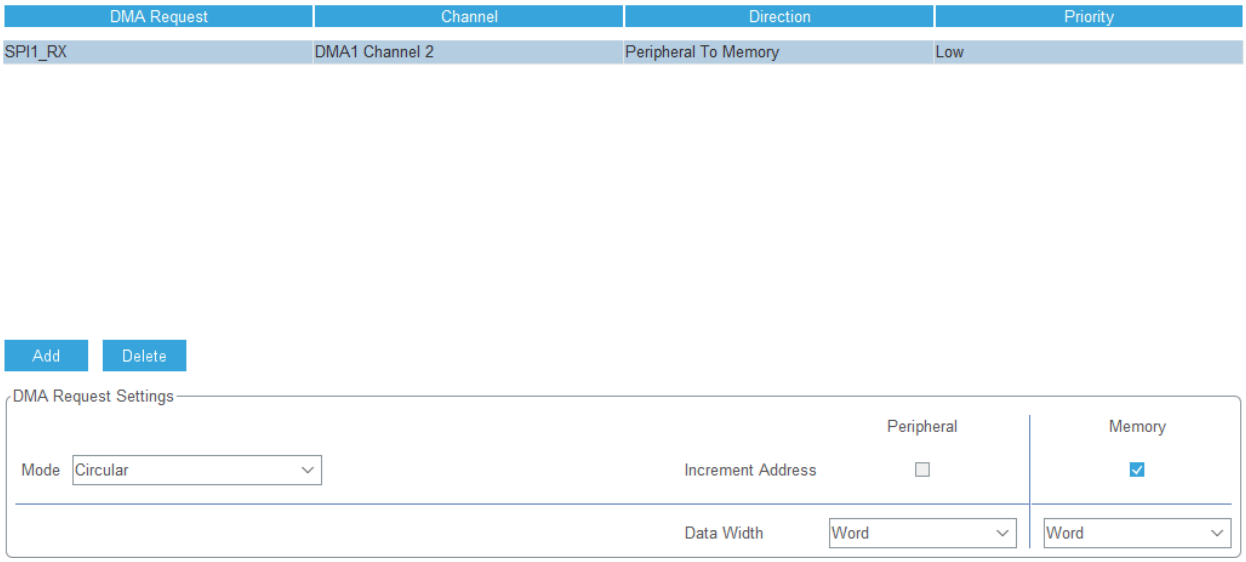


Figure 10: I2S DMA Configuration settings in STM32CubeMX IDE

You may notice that DMA requests are handled through SPI_RX, which refers to SPI receive. This is because I²S communication is implemented over the SPI bus, and in order for I²S to function correctly, SPI must be disabled so the bus can be fully dedicated to I²S.

As shown in the DMA initialization code above, we enable DMA on the channel associated with I²S communication and configure it to transfer data from the peripheral to memory. The memory address is incremented with each incoming audio sample, effectively storing the data in an array-like structure. We utilize circular DMA mode to continuously capture audio and automatically trigger a callback function whenever a buffer is filled. Since real-time audio capture is critical to our system, we set the DMA priority to *Very High*. Additionally, we configure the data width to *Word* to match the 32-bit frame size used in our I²S communication.

With I²S communication and DMA now fully configured between the ICS-43434 microphone and the STM32, we can begin implementing the firmware logic to process the captured audio buffers. We start by defining a few variables for the buffers and control flow:

```
#define BUFFER_SIZE 700
uint32_t i2sBuffer[BUFFER_SIZE]; // 2800 bytes 700 elements
volatile uint8_t active_buffer = 0;
```

Figure 11: Declaration of Audio Buffer Size along with the actual Audio Buffer

This is the buffer size, along with the actual buffer, which will store 32-bit elements (each audio sample is 24 bits with 8 bits of padding added by the STM32). Once the buffer is initialized, we

can call the I²S DMA receive function. This function is invoked once in *main.c*, and upon execution, DMA operates in the background to handle audio capture automatically:

```
// Start I2S DMA once here (not inside the loop)
HAL_I2S_Receive_DMA(&hi2s1, (uint16_t*)i2sBuffer, BUFFER_SIZE);
//1400 elements of 16 bit, but receiving 700 32 bit samples = 700 elements
```

Figure 12: I2S Receive DMA function provided by STM32

The reason we cast from uint16_t to uint32_t for the i2sbuffer is due to several factors. First, according to the HAL_I2S_Receive_DMA() function, audio is captured in the following manner:

```
/**
  * @brief  Receive an amount of data in non-blocking mode with DMA
  * @param  hi2s pointer to a I2S_HandleTypeDef structure that contains
  *         the configuration information for I2S module
  * @param  pData a 16-bit pointer to the Receive data buffer.
  * @param  Size number of data sample to be sent:
  * @note   When a 16-bit data frame or a 16-bit data frame extended is selected during the I2S
  *         configuration phase, the Size parameter means the number of 16-bit data length
  *         in the transaction and when a 24-bit data frame or a 32-bit data frame is selected
  *         the Size parameter means the number of 24-bit or 32-bit data length.
  * @note   The I2S is kept enabled at the end of transaction to avoid the clock de-synchronization
  *         between Master and Slave(example: audio streaming).
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_I2S_Receive_DMA(I2S_HandleTypeDef *hi2s, uint16_t *pData, uint16_t Size)
```

Figure 13: I2S Receive DMA function description

Since HAL_I2S_Receive_DMA() expects a uint16_t input for the data buffer, we cast it to a uint32_t audio buffer. The size passed to the function is BUFFER_SIZE, not BUFFER_SIZE * 2, because the documentation for HAL_I2S_Receive_DMA() specifies that the Size parameter refers to the number of 24-bit samples.

Now that HAL_I2S_Receive_DMA() is called with the correct inputs, DMA can function properly. However, we also need to implement a DMA callback function that will be triggered when the DMA buffer is full, allowing us to process the captured data. The DMA callback function used in our system is shown below:

```
void HAL_I2S_RxCpltCallback(I2S_HandleTypeDef *hi2s) {
    // Send buffer via UART (blocking mode)
    HAL_UART_Transmit(&huart1, (uint8_t*)i2sBuffer, BUFFER_SIZE * sizeof(uint32_t), HAL_MAX_DELAY);
}
```

Figure 14: I2S Receive DMA callback function which is called when the audio buffer is full

This is a simple DMA callback function that essentially transmits the data from the buffer via UART to our HC05 Bluetooth module. The Bluetooth module then forwards the data to the smartphone that is running the SightSync app so that the audio data can be translated to text (which will be explained in further detail later).

0.96-inch Waveshare OLED Display Overview

The OLED module used to display text in our system is the 0.96-inch Waveshare OLED display. It operates at 3.3V and supports both SPI and I2C communication interfaces. The display is controlled by the SSD1315 IC and has a resolution of 128 × 64 pixels. The overall dimensions of the module are 26 × 26 mm, with a display area measuring 21.74 × 11.18 mm. Below are front and back images of the OLED display:

Figure 15: Front of the 0.96 Inch OLED Screen

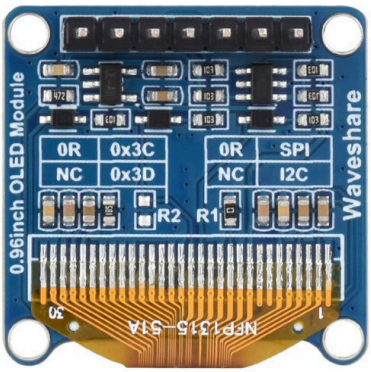


Figure 16: Back of the 0.96 Inch OLED Screen

The available modes of operation are I2C and SPI. Depending on the mode selected in the firmware, the physical wiring must be adjusted accordingly. The OLED module requires seven connections to the STM32, as shown below:

FUNC	SPI Mode	I2C Mode
VCC	3.3V / 5V power input	
GND	Power Ground	
DIN	SPI data input	I2C data input
CLK	SPI clock input	I2C clock input
CS	Chip select, low active	NC
DC	Data/command, low for command, high for data	NC (address selection)
RES	Reset, low active	

Figure 17: 0.96-Inch OLED Screen Pin Details

These pins are connected to the STM32 as follows:

```
// ---------------------- OLED ------------------------------
// OLED CS = A4
// OLED RST = D6
// OLED DC = D3
// IIC SCL = D12
// IIC SDA = D4
// OLED VOLTAGE = 3.3v
```

Figure 18: 0.96-Inch OLED Screen Pins to STM32 ports mapping

DIN carries data from the STM32 to the OLED. CLK provides the clock signal that drives the OLED. CS (Chip Select) enables or disables the OLED depending on whether it is driven low or high. DC (Data/Command) determines the type of data being sent; driving it low allows the STM32 to send commands to the OLED. RESET is used to initialize the OLED during startup.

In our system, we use I2C for communication with the OLED, as SPI is disabled to free the SPI bus for I2S communication, as explained earlier in the ICS-43434 Microphone section of this report. As a result of using I2C, certain hardware configurations must be adjusted. The configurations for both SPI and I2C modes are shown below:

Communication Method	R1	R2
4-wire SPI	0R	**NC** (Please ensure NC)
I2C (0x3C)	NC	0R
I2C (0x3D)		NC

Figure 19: 0.96-Inch OLED Screen Addressing According to Resistors R1 and R2

In our system, we removed the R1 resistor, and R2 was left unconnected by default (No Connect, NC). As a result, both R1 and R2 are disconnected, which sets the OLED's I2C address to 0x3D in the firmware.

This is also a good point to highlight that the OLED is controlled through a driver library written in C. The display comes with multiple driver options for different microcontroller families (e.g., Arduino, ESP32, Raspberry Pi), but in our project, we use the driver library specifically implemented for the STM32. Below is a screenshot of the provided library files:

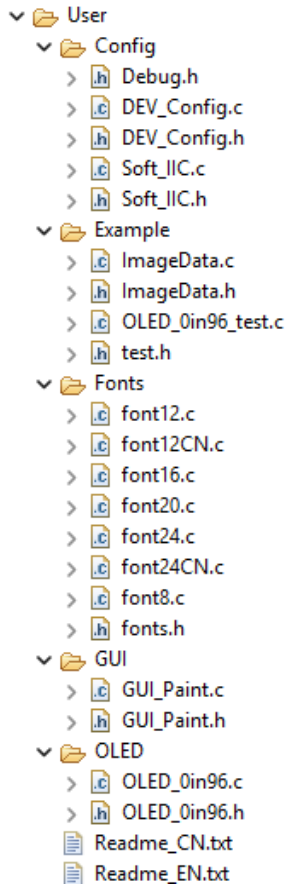


Figure 20: Driver Files Provided by the Manufacturers of the 0.96 Inch OLED Screen

To ensure the STM32 communicates with the OLED using the correct I2C address (0x3D), we modified a few lines of code within the driver files. Specifically, in DEV_Config.h, we updated lines 1, 2, and 3 as shown in the code snippet below:

```
/*****************************************************************************
*************************Hardware interface layer*****************************
* | file            :   DEV_Config.c
* | version         :   V1.0
* | date            :   2020-06-17
* | function        :   Provide the hardware underlying interface
*****************************************************************************/
#ifndef _DEV_CONFIG_H_
#define _DEV_CONFIG_H_

#include "stm32f0xx_hal.h"
#include "stm32f0xx_hal_gpio.h"
#include "main.h"
#include <stdint.h>
#include <stdlib.h>

/**
 * data
**/
#define UBYTE   uint8_t
#define UWORD   uint16_t
#define UDOUBLE uint32_t

#define USE_SPI_4W      0 // Line 1
#define USE_IIC         0
#define USE_IIC_SOFT    1 // Line 2

#define I2C_ADR 0X3D // Line 3
```

Figure 21: DEV_Config.c Code for Specifying OLED Address Used

We enabled IIC_SOFT, which is a custom I2C driver developed by the manufacturers of the SSD1315 integrated chip used to control the OLED. To do this, we set line 1 to 0, line 2 to 1, and changed line 3 from 0x3C to 0x3D, as both resistors R1 and R2 are unsoldered (NC).

Additionally, we made changes in Soft_IIC.h to properly map the GPIO ports of the STM32 to those defined in the OLED driver. The relevant changes are shown below, with the modified lines marked by numbered comments:

```
#ifndef _SOFT_IIC_H_
#define _SOFT_IIC_H_

#include "DEV_Config.h"

#define IIC_SOFT_SCL_PIN        IIC_SCL_SOFT_Pin // 1
#define IIC_SOFT_SDA_PIN        IIC_SDA_SOFT_Pin // 2

#define IIC_SOFT_SCL_GPIO       GPIOB // 3
#define IIC_SOFT_SDA_GPIO       GPIOB // 4

#define __IIC_SCL_SET()     HAL_GPIO_WritePin(IIC_SOFT_SCL_GPIO, IIC_SOFT_SCL_PIN, GPIO_PIN_SET) // 5
#define __IIC_SCL_CLR()     HAL_GPIO_WritePin(IIC_SOFT_SCL_GPIO, IIC_SOFT_SCL_PIN, GPIO_PIN_RESET) // 6

#define __IIC_SDA_SET()     HAL_GPIO_WritePin(IIC_SOFT_SDA_GPIO, IIC_SOFT_SDA_PIN, GPIO_PIN_SET) // 7
#define __IIC_SDA_CLR()     HAL_GPIO_WritePin(IIC_SOFT_SDA_GPIO, IIC_SOFT_SDA_PIN, GPIO_PIN_RESET) // 8
```

Figure 22: Code from Soft_IIC.h file

These lines were mapped to the following GPIO ports in the STM32's main.h file:

```
#define OLED_CS_Pin GPIO_PIN_5
#define OLED_CS_GPIO_Port GPIOA
#define OLED_DC_Pin GPIO_PIN_0
#define OLED_DC_GPIO_Port GPIOB
#define OLED_RST_Pin GPIO_PIN_1
#define OLED_RST_GPIO_Port GPIOB
#define SWDIO_Pin GPIO_PIN_13
#define SWDIO_GPIO_Port GPIOA
#define SWCLK_Pin GPIO_PIN_14
#define SWCLK_GPIO_Port GPIOA
#define IIC_SCL_SOFT_Pin GPIO_PIN_4
#define IIC_SCL_SOFT_GPIO_Port GPIOB
#define IIC_SDA_SOFT_Pin GPIO_PIN_7
#define IIC_SDA_SOFT_GPIO_Port GPIOB
```

Figure 23: STM32 to OLED Pin Definitions

We also modified a line of code in OLED_0in96.c, specifically within a function responsible for writing data from the STM32 to the registers of the SSD1315 OLED controller. Below is the function we updated, along with the exact line that was changed:

```
static void OLED_0in96_WriteReg(uint8_t Reg)
{
#if USE_SPI_4W

#elif USE_IIC_SOFT
    iic_start();
    iic_write_byte(0x3D << 1); // Changed code
    iic_wait_for_ack();
    iic_write_byte(0x00);
    iic_wait_for_ack();
    iic_write_byte(Reg);
    iic_wait_for_ack();
    iic_stop();
#endif
}
```

Figure 24: WriteReg Function to Write Data to the 0.96 Inch OLED Screen

Initially, the code had a random address, but we changed it to 0x3D and shifted it left to match the SSD1315 datasheet.

Once we corrected the code in the OLED drivers provided by the SSD1315 manufacturers, we established communication on the STM32 side by creating GPIO ports for I2C. Since the OLED doesn't support hardware I2C, we use software I2C mapped to GPIO ports as implemented by

the OLED drivers. This was done by creating a GPIO initialization function and setting values to the appropriate registers, as shown below:

```
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_3|OLED_CS_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, OLED_DC_Pin|OLED_RST_Pin|IIC_SCL_SOFT_Pin|IIC_SDA_SOFT_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pins : PA3 OLED_CS_Pin */
  GPIO_InitStruct.Pin = GPIO_PIN_3|OLED_CS_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pins : OLED_DC_Pin OLED_RST_Pin IIC_SCL_SOFT_Pin IIC_SDA_SOFT_Pin */
  GPIO_InitStruct.Pin = OLED_DC_Pin|OLED_RST_Pin|IIC_SCL_SOFT_Pin|IIC_SDA_SOFT_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */
  /* USER CODE END MX_GPIO_Init_2 */
}
```

Figure 25: STM32 GPIO Initialization function

We configured all the GPIO pins as output push-pull, with no pull-up or pull-down resistors, and set all frequencies to low. By default, we also set all the pins to low. With the GPIO pins properly initialized on the STM32 side and the OLED drivers correctly modified, we can now begin displaying pixels on the OLED. To do this, we write code in the main.c function to control the OLED. First, we define a few variables (remember, since we want to display text on the OLED, we need a buffer to hold the text received via Bluetooth UART):

```
#define BUFFER_SIZE 100
uint8_t buffer0[BUFFER_SIZE + 1]; // +1 for null terminator
volatile uint16_t fillIndex = 0;
UBYTE *BlackImage;
```

Figure 26: Declaration of buffer size, buffer for characters, index for the buffer, and the OLED image object

We define a buffer of type uint8_t with a size of 101. This allows us to store a string of 100 characters (101 if you include the null terminator), which is more than sufficient for displaying a word. Additionally, we have a variable called fillIndex, which serves as an increment for the buffer. We also define a pointer to BlackImage, a variable that points to an array representing the OLED screen. To display text on the OLED, we manipulate the BlackImage variable.

The next step is to initialize and turn on the OLED. To do this, we use the following code:

```
OLED_0in96_Init();
Driver_Delay_ms(500);

UWORD Imagesize = ((OLED_0in96_WIDTH%8==0)? (OLED_0in96_WIDTH/8): (OLED_0in96_WIDTH/8+1)) * OLED_0in96_HEIGHT;
BlackImage = (UBYTE *)malloc(Imagesize);
if(!BlackImage) {
    printf("Failed to apply for black memory...\r\n");
    return -1;
}
Paint_NewImage(BlackImage, OLED_0in96_WIDTH, OLED_0in96_HEIGHT, 90, BLACK);
Paint_SelectImage(BlackImage);
Driver_Delay_ms(500);
Paint_Clear(BLACK);
OLED_0in96_display(BlackImage);
```

Figure 27: Code to Initialize and Startup the OLED Screen

All the functions in the code above are API functions that we call (these functions were provided in the drivers, so we didn't implement them ourselves). First, we call the OLED_0in96_Init() function to initialize the OLED registers, followed by a driver delay to allow the OLED time to initialize. We then set ImageSize to the product of the screen's width and height (this represents the size of our 1D array, which covers all the pixels on the screen). Next, we dynamically allocate memory for the BlackImage we defined earlier, using the ImageSize. We also perform a check to ensure the memory was allocated properly. After that, we call a few more functions to display a black screen.

Once the OLED is initialized, we define a few additional variables. To display each word on the OLED screen correctly, we need to increment the x and y positions. For this purpose, we define x_pos and y_pos variables to track the current position on the OLED screen, as shown below:

```
memset(buffer0, 0, sizeof(buffer0));
fillIndex = 0;
uint16_t x_pos = 10;
uint16_t y_pos = 0;
static uint32_t last_input_time = 0;
static uint32_t screen_clear_timeout = 15000;
```

Figure 28: Initializing coordinates for x and y positions of characters as well as screen refresh time and setting buffer elements to 0 along with index to 0

We set all elements in buffer0 (which holds all the characters for a word) to 0, fillIndex to 0, x_pos to 10, and y_pos to 0. These are our default values based on extensive testing. We also define a few additional variables to control when the OLED clears the screen after it's filled with words. We don't want the screen to instantly clear when it's filled, as that wouldn't give the user enough time to read everything. Therefore, we set a last_input_time variable to always capture the current time, and screen_clear_timeout, which determines how long the words will be displayed before the screen clears.

Now that all the necessary variables are declared and the OLED is initialized, we can begin displaying words on the screen. To do this, we set up an infinite while loop that continuously displays one word at a time. To ensure that each iteration of the loop displays only one word, we create another while loop within the original one, as shown below:

```
while (1)
{
  uint8_t byte;
  HAL_StatusTypeDef uart_status;
  fillIndex = 0;

  // 1. Check screen timeout first
  if (HAL_GetTick() - last_input_time >= screen_clear_timeout) {
      Paint_Clear(BLACK);
      OLED_0in96_display(BlackImage);
      x_pos = 10;
      y_pos = 0;
      last_input_time = HAL_GetTick();
  }

  // 2. Non-blocking UART receive with 100ms timeout
  do {
      uart_status = HAL_UART_Receive(&huart1, &byte, 1, 100); // 100ms timeout

      if(uart_status == HAL_OK) {
          buffer0[fillIndex++] = byte;
          last_input_time = HAL_GetTick(); // Reset timeout on received byte

          // Check termination conditions
          if (byte == '\n' || fillIndex >= BUFFER_SIZE - 1) {
              break;
          }
      }
      else if (uart_status == HAL_TIMEOUT) {
          // Check screen timeout again during UART wait
          if (HAL_GetTick() - last_input_time >= screen_clear_timeout) {
              Paint_Clear(BLACK);
              OLED_0in96_display(BlackImage);
              x_pos = 10;
              y_pos = 0;
              last_input_time = HAL_GetTick();
          }
      }
  } while (uart_status == HAL_OK);

  // . . . .
```

Figure 29: First part of the while loop responsible for receiving characters via UART

Basically, at each iteration of the top-level while loop, a word is displayed. To do this, we create a do-while loop inside the top-level while loop, which keeps iterating and calling the HAL_UART_Receive function. This function takes a few arguments: an instance of the UART socket, a reference to a byte variable that holds the byte we are receiving, the amount of bytes we want to receive, and the timeout. In this case, we pass &huart1, &byte, 1, and 100 to the function. This means we are accepting 1 byte at each iteration of the do-while loop, with a 100 ms timeout so that the receive function is not blocking. At each iteration, we store the received byte into our buffer0 buffer and increment the fillIndex variable that keeps track of our index. This do-while loop is only broken when it receives a byte that is equal to the newline character

(i.e., '\n') or if the index of the array is greater than BUFFER_SIZE - 1 (which realistically should never happen since no words are more than 100 characters). On the SightSync app, we attach a '\n' character at the end of each string, which helps the STM32 understand when a full word has been received. Once the '\n' has been received, the do-while loop breaks meaning the entire word has been received, and we move on to the next part of the main while loop, which is displaying the word on the OLED. Below is the code of how we start to do this:

```
// . . . .

// 3. Process received data if any
if (fillIndex > 0) {
    // Null termination handling
    if (buffer0[fillIndex - 1] == '\n') {
        buffer0[fillIndex - 1] = '\0';
    }
    buffer0[fillIndex] = '\0';

    // Trim trailing spaces
    char *buffer_str = (char*)buffer0;
    char *end = buffer_str + strlen(buffer_str) - 1;
    while (end >= buffer_str && *end == ' ') *end-- = '\0';
```

Figure 30: Second part of the while loop responsible for eliminating spaces and replacing the newline character with a null terminator

First, we have a safety check which checks if the fillIndex is greater than 0 (this ensures that we are displaying a word and not nothing). Then, we replace the newline character at the end of the word with a null terminator to convert the buffer0 array into a string. We then remove any spaces that come after the word in the buffer (after the null terminator), and replace them with null terminators (this is more of a safety check to make sure that everything after the word is a null terminator). Once this is done, the word in the buffer is ready for display. The rest of the code is essentially meant for displaying the word on the OLED screen.

```
if (strlen(buffer_str) > 0) {
    char *current_word = buffer_str;
    while (*current_word != '\0') {
        // Skip leading spaces
        while (*current_word == ' ') {
            current_word++;
        }
        if (*current_word == '\0') {
            break;
        }

        // Find end of current word
        char *word_end = strchr(current_word, ' ');
        if (word_end == NULL) {
            word_end = current_word + strlen(current_word);
        }

        // Temporarily null-terminate the word
        char temp = *word_end;
        *word_end = '\0';

        // Calculate word width
        uint16_t word_width = strlen(current_word) * Font12.Width;
        int space_needed = (x_pos != 10) ? 1 : 0; // Check if not at line start
        uint16_t space_width = space_needed * Font12.Width;
        uint16_t total_width = space_width + word_width;

        // Check horizontal fit
        if (x_pos + total_width > OLED_0in96_HEIGHT) {
            x_pos = 10;
            y_pos += Font12.Height;

            // Check vertical overflow
            if (y_pos + Font12.Height > OLED_0in96_WIDTH) {
                Paint_Clear(BLACK);
                y_pos = 0;
                x_pos = 10;
            }
            space_needed = 0; // Reset space after new line
        }

        // Draw space if needed
        if (space_needed) {
            Paint_DrawString_EN(x_pos, y_pos, " ", &Font12, WHITE, BLACK);
            x_pos += Font12.Width;
        }

        // Draw the word (cast to char* for display function)
        Paint_DrawString_EN(x_pos, y_pos, current_word, &Font12, WHITE, BLACK);
        x_pos += word_width;

        // Restore buffer and move to next word
        *word_end = temp;
        current_word = word_end;

        // Skip consecutive spaces
        while (*current_word == ' ') {
            current_word++;
            if (*current_word == '\0') break;
        }
```

Figure 31: Third part of the while loop which loops through each character in the buffer and draws it on the OLED screen

To do this, we create a while loop that loops through every letter of the word until it finds the null terminator '\0', and once it finds it, it breaks out of the loop. At each iteration of the while loop, it processes one character from the buffer0 and "draws" it on the BlackImage variable we created earlier (essentially setting some elements of the array to ON). To do this, we first find the end of the word, which is always a space before the null terminator, and replace this space with a null terminator. We then calculate the word width by multiplying the word length by “Font12.width”, which is the width of a character. In our project, we use Font12 because it is small enough and visible enough to be used. Below are the width and height of Font12 characters:

```
sFONT Font12 = {
  Font12_Table,
  7, /* Width */
  12, /* Height */
};
```

Figure 32: Font12 Struct

We then check if we need a space at the end of the word by verifying if we're at the beginning of a new line. If we're at the start of a new line (not a new line character but a literal new line on the OLED screen), we don't need a space, but if we're not, we likely need a space to separate the current word from the previous one. We then calculate the total width, which is the word width plus the space we add, and check if the word can fit on the current line without wrapping to a new line. We don't want the word to break at the end of a line and wrap to the next, so we check if the current x position plus the total size of the word exceeds OLED_0in96_HEIGHT (which, for some reason, corresponds to the maximum width of the OLED). If it does, we move to the next line, set x_pos to 10, and increment y_pos by Font12.Height to go to a new line. If we're at the last line and it's full, meaning y_pos + Font12.Height exceeds OLED_0in96_WIDTH, we reset to the start of the screen, setting y_pos to 0 and x_pos to 10. Once the x and y positions are adjusted, we draw a space (only if we're not at the beginning of a line) by calling the Paint_DrawString function with the x and y positions, a space string, a reference to the font, and the colors for the text and background. Afterward, we call the same function to display the actual word.

Now that the word is stored in the BlackImage variable, it's ready to be sent to the OLED driver to display the text on the screen. To do this, we call the following function:

```
OLED_0in96_display(BlackImage);
```

This function sends the data to the OLED, and the word is displayed on the screen. As mentioned before, this occurs at each iteration of the top-level while loop, where a new word is displayed.

There is one additional part that we skipped, which is the screen timeout and clearing. We coded this to happen every 10 seconds if no activity has been detected on the OLED screen (no new words displayed). The purpose of this is to clear the screen when no words have been detected (so that the user is not looking at the same paragraph from hours ago all day). This ensures they aren't constantly staring at words. Instead, they can see normally without words if no new speech is being captured. To clear the screen, we use the following code:

```
// 2. Non-blocking UART receive with 100ms timeout
do {
    uart_status = HAL_UART_Receive(&huart1, &byte, 1, 100); // 100ms timeout

    if(uart_status == HAL_OK) {
        buffer0[fillIndex++] = byte;
        last_input_time = HAL_GetTick(); // Reset timeout on received byte

        // Check termination conditions
        if (byte == '\n' || fillIndex >= BUFFER_SIZE - 1) {
            break;
        }
    }
    else if (uart_status == HAL_TIMEOUT) {
        // Check screen timeout again during UART wait
        if (HAL_GetTick() - last_input_time >= screen_clear_timeout) {
            Paint_Clear(BLACK);
            OLED_0in96_display(BlackImage);
            x_pos = 10;
            y_pos = 0;
            last_input_time = HAL_GetTick();
        }
    }
} while (uart_status == HAL_OK);
```

Figure 33: do-while loop within the outer while loop, responsible for receiving characters via UART and clearing the screen every 10 seconds

This code was originally meant to receive an entire word, but it also serves an additional purpose: monitoring the activity of the captioning lenses. Each time a byte is received on UART, a counter is reset. We check the uart_status to determine whether data is being received or not. When uart_status is HAL_OK, it means data is being successfully received, and we update last_input_time with the current tick from HAL_GetTick(). On the other hand, if uart_status is not HAL_OK for a while (i.e., when it returns HAL_TIMEOUT), the last_input_time doesn't get updated. This means the difference between last_input_time and the current tick keeps growing.

In the code's else if statement, when the difference between HAL_GetTick() and last_input_time is greater than or equal to screen_clear_timeout (which is set to 10 seconds), the screen is cleared by painting it black. After this, we reset the x_pos and y_pos variables to their starting positions. This mechanism ensures the screen will automatically clear after 10 seconds of inactivity (no new words displayed). The exact screen_clear_timeout value was determined through trial and error to ensure a 10-second delay.

Robojax HC-05 Bluetooth Module Overview

It is clear from the previous sections that Bluetooth is essential for communication between the STM32 microcontroller and the SightSync Android app. This link enables audio transcription and text data transmission. To establish this connection, we needed a Bluetooth module compatible with the STM32. We chose the HC-05 module due to its simplicity and widespread use.

The HC-05 is a Bluetooth SPP (Serial Port Protocol) module designed for easy wireless serial communication. It supports Bluetooth V2.0+EDR (Enhanced Data Rate) and operates on a 2.4GHz radio transceiver, delivering a maximum RF transmit power of +4dBm and a typical sensitivity of -80dBm. The module runs on low power (1.8V), accepts a wide input voltage range (1.8V to 3.6V), and communicates via a UART interface with a programmable baud rate. It also features an integrated antenna and edge connector.

The HC-05 includes software features such as automatic reconnection after 30 minutes of disconnection, auto-pairing with a default PIN code of "0000," and auto-connection to the last paired device on power-up. It also has status LEDs to indicate connection state and supports configurable baud rates. For our project, we used a baud rate of 115200, which provided a stable and fast data transmission rate.

An image of the HC-05 module is shown below:

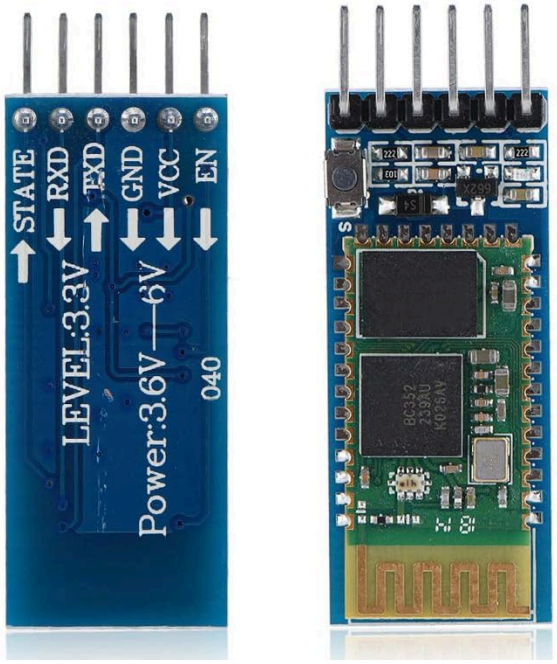


Figure 34: Image of the Back and Front of the HC-05 Bluetooth Module

Before connecting the HC-05 to the STM32 for Bluetooth transmission and reception, we first needed to program the HC-05 using AT commands to change its default baud rate from 9600 to 115200. To accomplish this, we used an Arduino to relay AT commands sent from the Arduino IDE to the HC-05. The code used for this process is shown below:

Change_Baud.ino

```
#include <SoftwareSerial.h>

SoftwareSerial BTSerial(10, 11); //RX | TX
//AT+UART=1024000,0,0
void setup() {
  pinMode(9, OUTPUT); //enable Pin
  digitalWrite(9, HIGH);
  Serial.begin(9600);
  BTSerial.begin(38400); // Default AT mode speed
  Serial.println("Try sending 'AT'");
}

void loop() {
  if (BTSerial.available()) {
    Serial.write(BTSerial.read());
  }
  if (Serial.available()) {
    BTSerial.write(Serial.read());
  }
}
```

Figure 35: Arduino Code to Send AT Commands to the HC-05 Bluetooth Module

We drive pin 9 high, which is connected to the EN pin of the Bluetooth module. This puts the HC-05 into AT mode, allowing it to receive AT commands. Communication is then initiated at a baud rate of 38400—the default rate for AT mode. Using the serial console, we sent the command AT+BAUD=115200 to change the baud rate, and the module responded with a success status.

Once the HC-05's baud rate is properly configured, the next step is to establish communication with the STM32 microcontroller by physically wiring the pins. The HC-05 was connected to the STM32 as follows:

```
// ------------------ BLUETOOTH MODULE -----------------------
// D1 = TX
// D0 = RX
// HC05 VOLTAGE = 5V
// HC05 EN = A2
// HC05 STATE = A5
```

Figure 36: Bluetooth Module Pins to STM32 Ports

Next, we needed to enable UART communication on the STM32 using the appropriate pins. To start, we defined private variables for the EN and STATE pins:

```
#define HC05_EN_PIN    GPIO_PIN_3
#define HC05_EN_PORT   GPIOA //A2
#define HC05_STATE_PIN GPIO_PIN_6
#define HC05_STATE_PORT GPIOA //A5
```

Figure 37: HC-05 EN and STATE pins to STM32 GPIO Ports

While these pins were not used in our project, they were declared in case we needed to troubleshoot or re-enter AT mode to change the baud rate.

The critical pins—TX and RX—required UART to be initialized, which was done in the following code:

```
static void MX_USART1_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */

  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */

  /* USER CODE END USART1_Init 2 */

}
```

Figure 38: STM32 UART Communication Initialization function

We configured the STM32's UART settings with a baud rate of 115200 to match the HC-05. The communication mode was set to asynchronous, with a word length of 8 bits, no parity, and one stop bit. Both transmit and receive directions were enabled. The image below shows the UART configuration as set in STM32CubeMX:

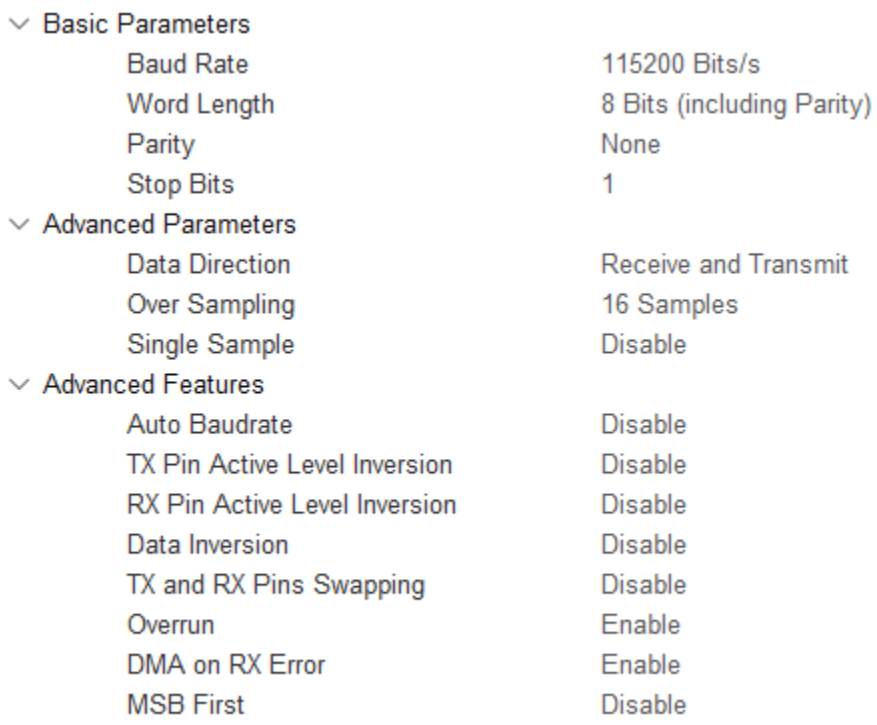


Figure 39: UART Communication Configuration Settings on the STM32CubeMX IDE

The image below shows how the TX and RX pins were configured on the STM32:

Pin Na...	Signal on ...	GPIO outp...	GPIO mode	GPIO Pull-...	Maximum ...	Fast Mode
PA9	USART1_TX	n/a	Alternate ...	No pull-up ...	High	Disable
PA10	USART1_RX	n/a	Alternate ...	No pull-up ...	High	Disable

Figure 40: UART TX and RX Configuration Settings on the STM32CubeMX IDE

We configured the TX and RX pins as alternate function push-pull, with no pull-up or pull-down resistors, and set them to maximum speed. With the Bluetooth module fully set up, the final step is to connect to it. This connection does not require any additional setup on the STM32 side, as the HC-05 handles pairing through its built-in drivers and software.

The connection is established via the SightSync app, which pairs with the HC-05 over Bluetooth. Once connected, data can be sent and received. As described in the ICS43434 microphone and

0.96-inch OLED sections of this report, UART is used for communication between the HC-05 and the STM32. This interface allows us to transmit audio data and receive text, as shown below:

```
HAL_UART_Transmit(&huart1, (uint8_t*)i2sBuffer, BUFFER_SIZE * sizeof(uint32_t), HAL_MAX_DELAY);
```

Figure 41: STM32 UART Transmit Function API

```
uart_status = HAL_UART_Receive(&huart1, &byte, 1, 100);
```

Figure 42: STM32 UART Receive Function API

The HC-05 then either sends data to the SightSync app or receives data from it, relaying it to the STM32 through its built-in drivers.

STM32 F042K6 Nucleo Microcontroller Overview

The previous sections discussed individual components such as the ICS-43434 microphone, a 0.96-inch OLED screen, and the HC-05 Bluetooth module. However, all of these components require a microcontroller to function and be useful. The STM32F042K6 microcontroller is ideal for this purpose. It's also the platform we use for coding and storing our firmware.

To program the STM32F042K6, we use STM32CubeIDE—a free integrated development environment designed for STM32 microcontrollers. This IDE provides the necessary Hardware Abstraction Layer (HAL) drivers and firmware, allowing us to control the STM32's registers.

As shown in earlier sections, whenever we initialize new communication protocols or ports, we typically configure specific registers with predefined values. STM32CubeIDE simplifies this process by offering the following HAL drivers:

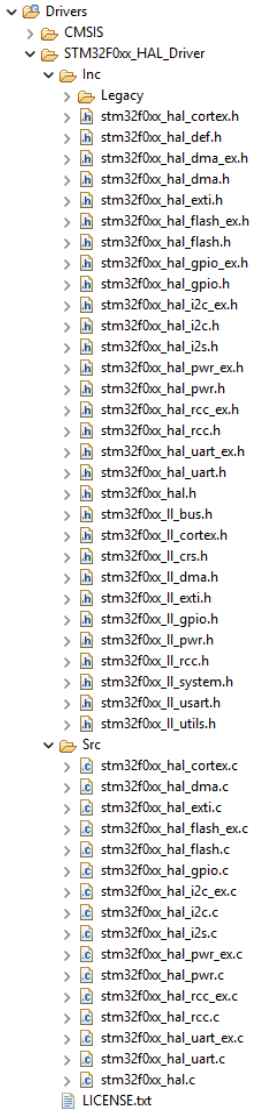


Figure 43: STM32 HAL Drivers

STM32CubeIDE also includes the CubeMX interface, which allows us to configure the STM32—such as setting up pins and peripherals—without writing any code:

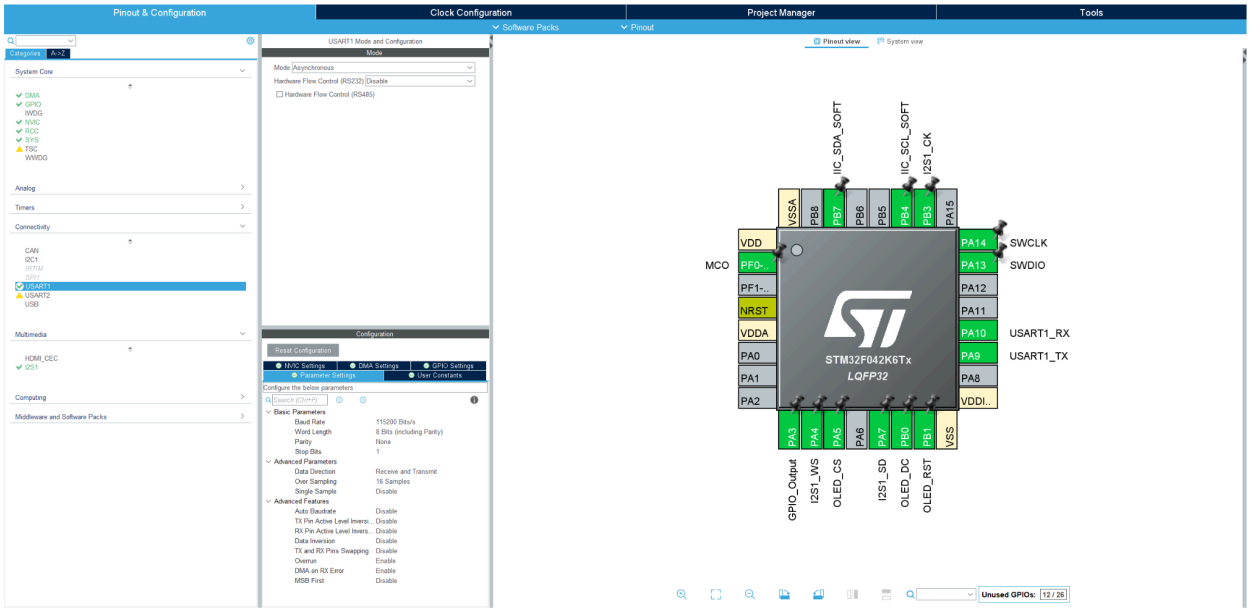


Figure 44: STM32CubeMX IDE Pinout Configuration Interface

It also lets us configure the clock settings for various parts of the system, such as the SPI, UART, and I2S buses. In our case, we set the clock to the maximum supported by the STM32, which is 48 MHz:

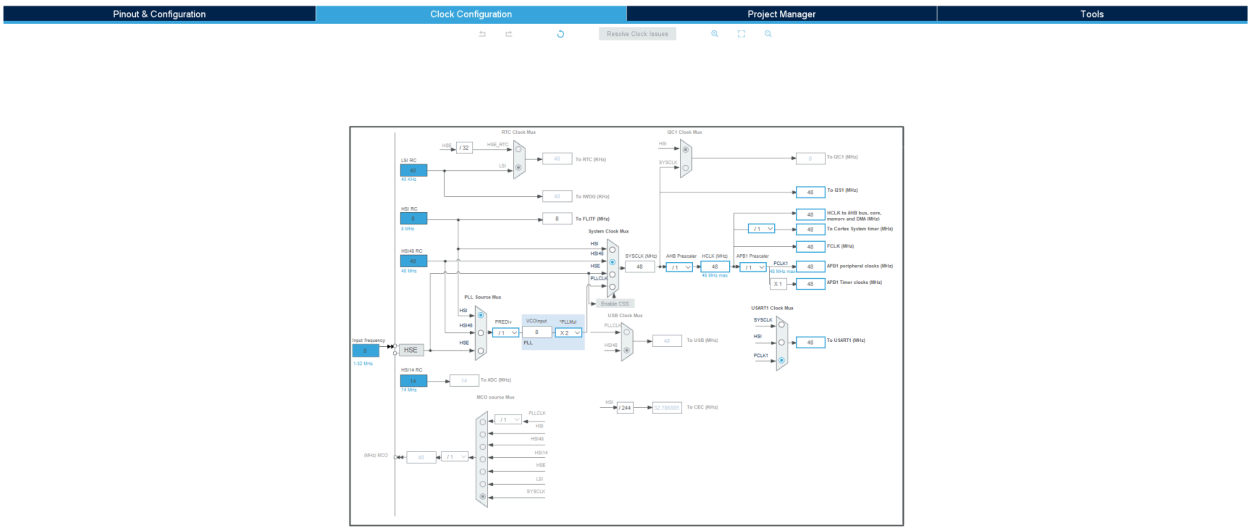


Figure 45: STM32CubeMX IDE Clock Configuration Interface

In terms of general information, the STM32F042K6 is a low-power, high-performance microcontroller from STMicroelectronics. It's built around the ARM Cortex-M0 32-bit RISC core, running at up to 48 MHz. As part of the STM32F0 series, it includes up to 32 KB of Flash memory and 6 KB of SRAM, making it well-suited for compact embedded applications. It also integrates a rich set of peripherals, including USB 2.0 Full-Speed (crystal-less), CAN, USART, SPI, I2C, I2S, ADC, and up to nine timers—all while operating within a voltage range of 2.0 to 3.6 V. Power efficiency is another strong point, with support for low-power modes such as Sleep, Stop, and Standby—ideal for battery-powered designs.

We considered a few other microcontrollers, such as the Arduino Nano, Raspberry Pi Pico, and ESP32. However, each came with drawbacks. The Arduino Nano, for example, only offers 2 KB of SRAM, which isn't sufficient for our program. The Raspberry Pi Pico lacks support for some of the communication protocols we need and doesn't include features like DMA. The ESP32, while powerful, is physically wider and doesn't offer the same level of low-level control over hardware.

The STM32F042K6 hits a sweet spot: compact size, sufficient SRAM, a capable core, and a wide range of communication options, including DMA and interrupt support. It's development environment—STM32CubeIDE—makes working with it even more accessible and user-friendly. Additionally, STM32 microcontrollers are widely used in industry, meaning there's a wealth of community support and documentation available online, which is invaluable for troubleshooting.

Another key advantage of the STM32 is its low-level control over memory and peripherals. By programming in C and directly manipulating hardware registers, we have full control—something not all microcontrollers offer. This flexibility allows us to build the project exactly as we want.

For these reasons, we chose the STM32F042K6 over the alternatives.

The entire STM32 firmware that we wrote can be found in Ahmed's github:

https://github.com/ahmedalduuyher/SightSync-AI-Glasses-Capstone-Project-

Below are images showing the appearance of the STM32F042K6, along with a diagram of its physical dimensions, ports, and:

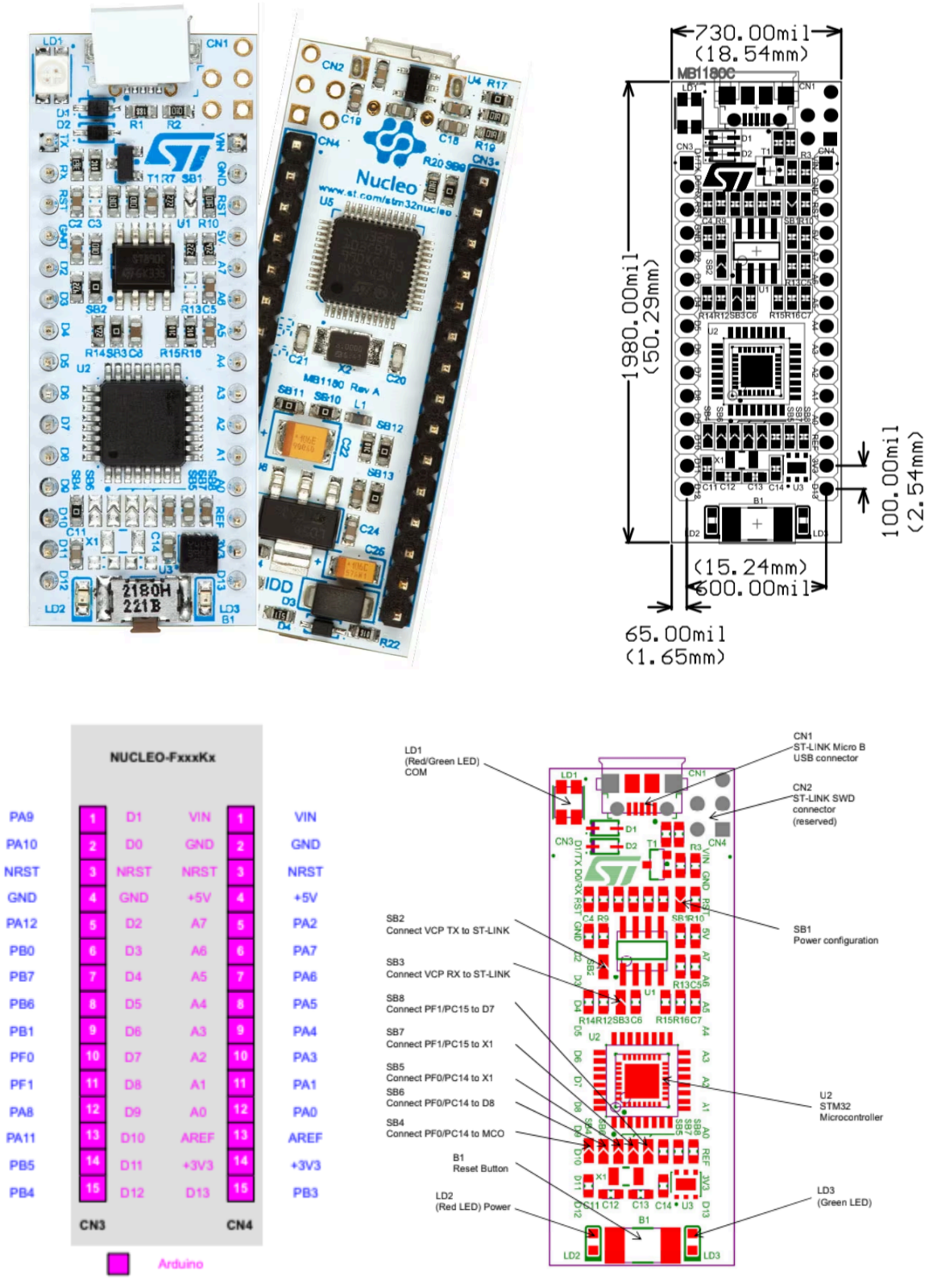


Figure 46: STM32 F042k6 Nucleo Front and Back, Physical Dimensions, Ports

Physical Design [3.2]

The physical design of SightSync balances wearability, functionality, and manufacturability. It has evolved significantly from its initial concept to the final prototype, driven by iterative testing and user feedback. The wearable frame was 3D-printed using lightweight PLA material, with a modular design that attaches to existing eyewear for versatility. Early prototypes revealed challenges with bulkiness, prompting a shift from standalone glasses to a sleeker, modular attachment system. The final frame houses the PCB, STM32 microcontroller, OLED display, HC-05 Bluetooth module, and a 3.7V LiPo battery.

Design 1

For the initial prototype, an open source STL file was 3D printed from Instructables. This was used as a proof of concept. The following images depict the initial prototype design.

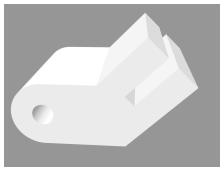


Figure 47: Acrylic Panel Swiveling Mount

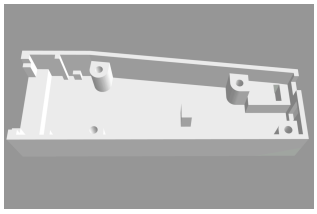


Figure 48: Frame Leg Bottom Half

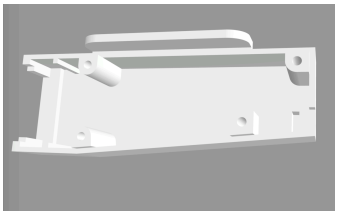


Figure 49: Frame Leg Top Half

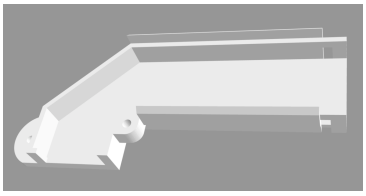


Figure 50: Lens/ Mirror Holding Leg Bottom Half

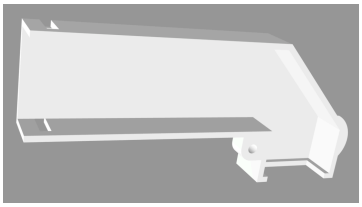


Figure 51: Lens/ Mirror Holding Leg Top Half

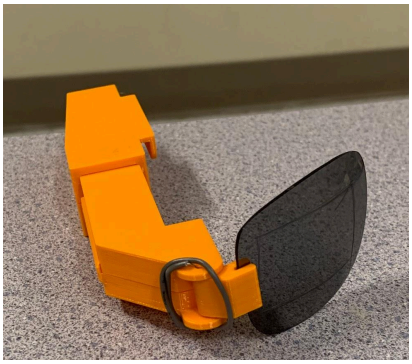


Figure 52: Built Frame Design 1 Prototype with Reflective Lens

Design 2

An alternative frame style and text projection system was also explored. This frame would seat all hardware components on the xz plane as opposed to the xy plane. The arrangement allowed for the frame to take a much narrower form factor, decreasing the amount by which the device extends away from the user. This frame was split into two halves that would snap fit around the right side leg of a pair of glasses. The projection system consisted of a mirror that was mounted at 45 degrees to the OLED panel, which would reflect the panels image onto a magnifying lens cut directly into the lens of the glasses.

This design was ultimately abandoned due to sizing constraints of the PCB and implementation difficulties.

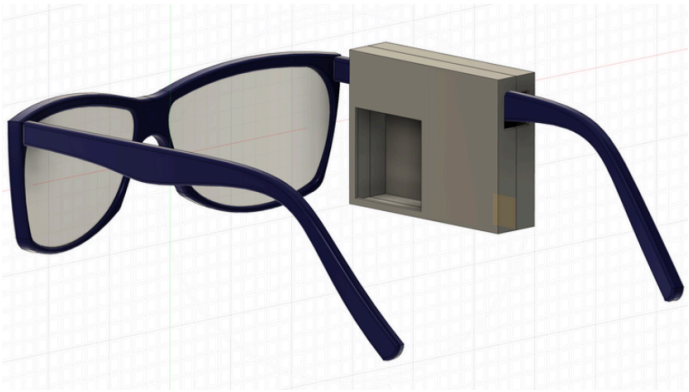


Figure 53: Full Frame Design 2 with Generic Glasses

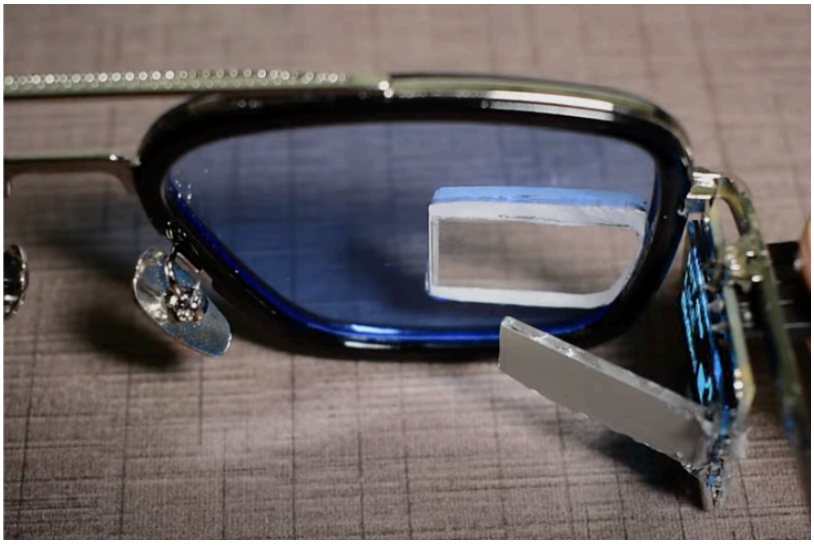


Figure 54: Design 2 Text Projection System Concept

Final Design

For the final product, a new frame was designed from scratch using the Autodesk Fusion360 CAD software. This variation improved on the original prototype design 1 through various key design changes. This includes increased size to comfortably house all necessary components along with wires, an opening at the back face for a charging port and cable, notches to hold the OLED panel in place, and snap fit locks that allow for all components of the frame to be attached and detached easily. Furthermore, the front portion of the frame, where the OLED panel sits, was increased in size to allow for a larger mirror to be fit in.

The optical projection system underwent two major iterations. The initial design used a plano-convex lens and acrylic mirror that was abandoned due to collimation issues and excessive bulk. The finalized design employed a 45 degree-angled mirror reflecting the image of the text from the 0.96-inch OLED display (128x64 pixels) onto an acrylic panel into the user's field of view with minimal distortion. The system achieved readability at a distance of ~5cm. The acrylic mirror was selected to reduce the ghosting.

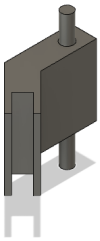


Figure 55: Acrylic Panel Swiveling Mount

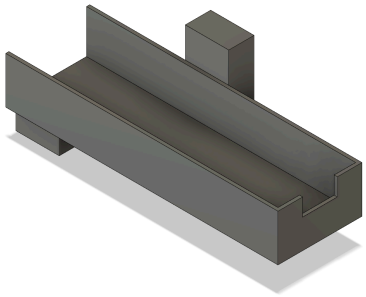


Figure 56: Frame Leg Bottom Half

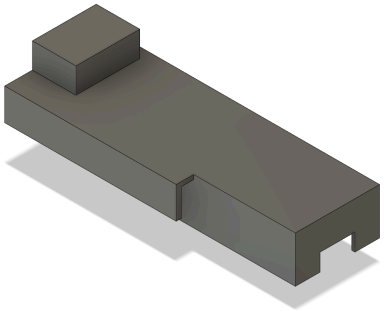


Figure 57: Frame Leg Top Half

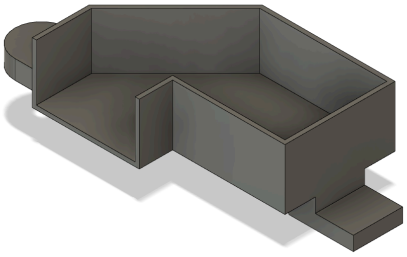


Figure 58: Lens/ Mirror Holding Leg Bottom Half

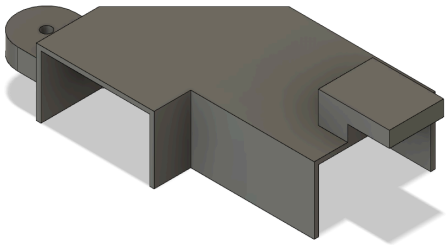


Figure 59: Lens/ Mirror Holding Leg Top Half

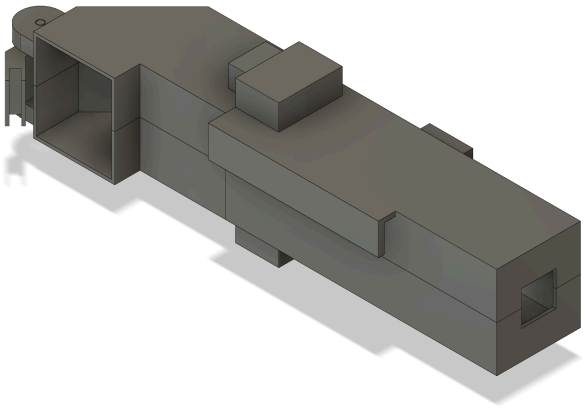


Figure 60: Full Frame Assembly

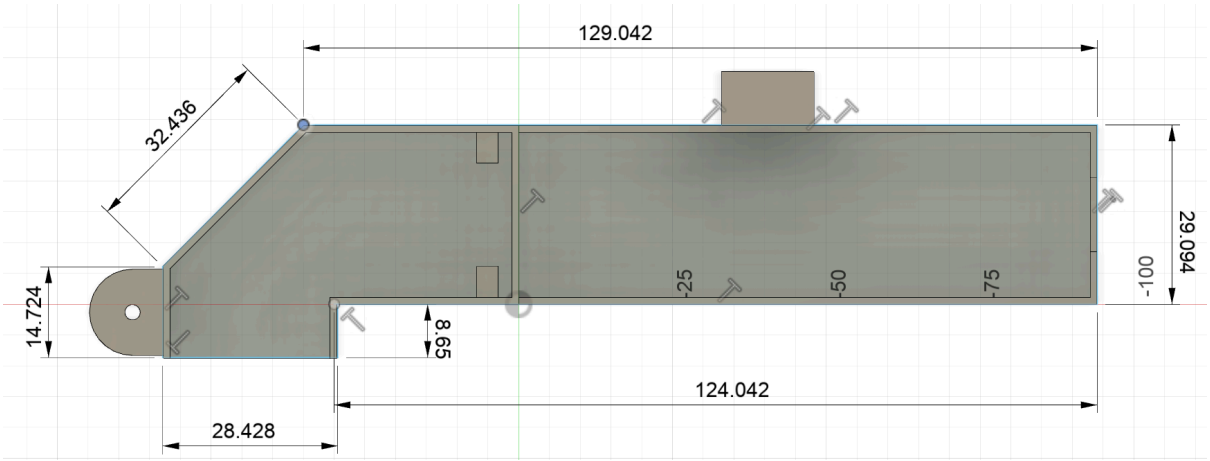


Figure 61: Full Frame Top View with Dimensions (in millimeters)

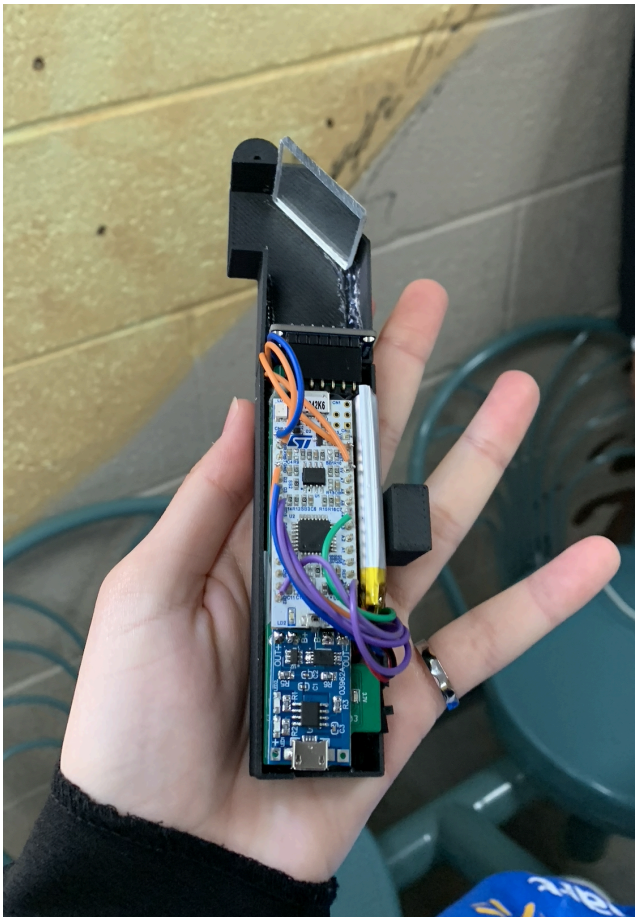


Figure 62: Assembled Frame with Hardware Components

The following table outlines the functional design criteria achieved by the final design.

Design Criteria	
Priority	**Description**
Clear Visibility of Text	Sharpness and ease of readable text projected by the OLED screen without obstructing vision**.**
Alignment Precision	Accurate positioning of the OLED screen and lens to ensure text appears in the correct location within the user's view.
Minimal Power Consumption	Efficient use of energy in the electronics, such as the OLED screen, to maximize battery life.
Ergonomic Integration	Lightweight and balanced design to prevent strain on the wearer's ears during prolonged use.
Modularity	Compatibility of the lens system with various eyewear frames, allowing secure attachment and flexibility.
Durability	Robustness of materials to withstand daily use, minor impacts, and exposure to environmental conditions.
Aesthetic Design	Sleek and modern appearance that avoids bulkiness and maintains visual appeal.

System Circuit Design and Integration [3.3]

In the previous section, the focus was on how the system's core components such as the microcontroller, Bluetooth module, OLED screen, and microphone were programmed to work together. This section explores how those components are powered, connected, and physically brought to life through the system circuitry. Circuit design and integration refers to the electrical system that supplies the necessary voltages, handles communication between modules, and supports key features like charging and user control.

Designing this system required a number of important decisions. Should the device be powered through a wall outlet, or should it run independently with an internal battery? Would it be enough to manually wire each connection and fit everything into the frame, or should the design aim to feel like a finished product with clean layout, rechargeability, and user control?

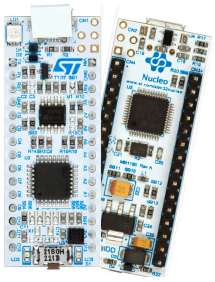
To give users the ability to move freely without being connected to an external power source, the team decided to build a self-contained custom printed circuit board powered by an internal battery. This involved delivering the correct voltage levels to each component, integrating a charging system, managing communication between modules, and allowing the user to control power through a manual and software ON/OFF switches. This section will detail the design choices, trade offs, and technical decisions involved in what resulted as a compact and portable electrical system built into lightweight and comfortable frames.

Current Draw Calculations

Before selecting components or designing the power system, it was essential to calculate the current draw of each module in the device. These calculations help estimate the total power consumption of the system under normal and peak operating conditions. Understanding how much current each component uses ensures that the battery, voltage regulators, connectors, and trace widths on the PCB can all safely support the load without overheating and risking component failure. Datasheets were thoroughly analyzed for each of the components involved in our design to make sure current ratings were not exceeded, which may lead to permanent damage.

These current draw calculations were also key for estimating battery life of the selected battery and selecting appropriate charging circuits, both of which will be covered in the next section after this. On the flip side, without considering these calculations, there would also be a risk of underpowering the system, in which case some components may not turn on altogether.

The design process began with calculating the current requirements of each major component to ensure proper sizing, appropriate component ratings, and reliable power delivery throughout the system. A full breakdown of these values is provided in the table below, with all numbers referenced from official datasheets or manufacturer documentation.

Component	Photo	Current Draw at Normal Operation	Current Draw at Peak Operation
MEMS ICS-43434 Mic Breakout Board		230 uA (0.23mA) in low power mode	490 uA (0.49mA)
0.96” Waveshare OLED Screen		25 mA when ON	
Bluetooth Module		30 mA while operating	40mA during transmission
STM32 Microcontroller		100 mA during normal operation when CPU, I2C, I2S, LEDs are in use.	300 mA maximum
ICs, LEDs, and passives	–	30 mA	
Total Worst Case Current Draw: ~395mA			
Current Draw during Typical Operation: ~195.49mA			

The STM32 Nucleo board, responsible for logic and communication, was estimated to draw approximately 100 milliamps during typical operation. The ICS-43434 MEMS microphone added a minimal load of about half a milliamp, while the HC-05 Bluetooth module peaks at 40 milliamps during transmission. The OLED display was expected to draw between 25 milliamps. Altogether, the total worst-case current draw for the system was calculated to be approximately 395 milliamps.

This figure became the baseline for all power-related design choices. For example, voltage regulators were chosen to be rated at least 30% above 395 mA to allow for headroom and avoid stress on the regulator during peak load. For the battery, a 3.7V LiPo cell was chosen with a minimum capacity of 1000 mAh. At 200 mA of continuous draw, this provided a theoretical runtime of roughly 5 hours. PCB trace widths were sized using trace calculators based on the estimated current and 1 oz copper weight. Wider traces were used on power lines to ensure safe current handling and minimize voltage drop. All connectors and headers were selected to exceed the 395 mA threshold accordingly, with built in buffer room of 30%. Each of these topics will be covered in following sections.

Thermal performance was considered for each component by analyzing power dissipation and the expected current draw through each branch of the circuit. To verify safe operating conditions, the standard power formulas were used frequently throughout the design process:

$$P = IV = I^2 * R = V^2 / R$$

Battery Selection and Runtime Analysis

The team began by researching battery types commonly used in wearable technology. The goal was to find a lightweight power source that would not generate excessive heat, pose safety concerns, or feel uncomfortable during extended use. Lithium Polymer (LiPo) batteries stood out due to their flat form factor, relatively high energy density, and widespread use in compact consumer electronics. Their flexibility in shape and capacity made them an ideal choice for integration into a wearable frame without compromising safety or comfort.

One of the primary design goals was achieving at least 8 hours of continuous use, allowing users to wear the device throughout the day without interruption. To meet this requirement, the team initially selected a 2000mAh LiPo battery that appeared compact enough for the intended frame size. However, during the PCB layout and physical integration phase, it became clear that this battery was the tallest component in the system and ultimately dictated the minimum height of the glasses frame.

This prompted a second round of battery sourcing focused on finding alternatives with a similar capacity but reduced height. Although several smaller options were considered, including 800mAh and 600mAh cells, which also came with significantly reduced runtime. After

comparing size-to-runtime tradeoffs with each, a 1000mAh battery was selected. While this new option reduced the device height by only 4mm, it was half the thickness of the original 2000mAh battery, which plays to our advantage in slimming the frame design. This thinner, lighter battery course had the benefit of easier wearability and better visual design. It would also suffice for demoing purposes at the capstone expo, as a 1000mAh battery would run for approximately 4 hours continuously. Our design also involved including a manual switch to control current draw and extend the battery life as much as possible throughout the day.

Charging IC Analysis

In order to implement charging functionality into the device, a battery management system was required. This included an integrated circuit to monitor the battery's charge level, temperature, and fault conditions, along with overcurrent and overvoltage protection. A USB port was also necessary to allow charging from an external power source such as a wall outlet or computer. The team began designing this custom protection and charging circuit in the early stages of PCB development, based on a charging IC called the BQ21040. This IC is a compact linear charger designed for lithium-ion and lithium-polymer batteries, typically used in space-constrained portable applications. It can operate from either a USB port or an AC adapter, and features input overvoltage protection for compatibility with low-cost, unregulated adapters. It manages charging in three stages: conditioning, constant current, and constant voltage. It includes built-in temperature monitoring and will automatically reduce charge current if thermal thresholds are exceeded. The fast charge current is programmable through an external resistor. Below is the internal functional block diagram of the IC.

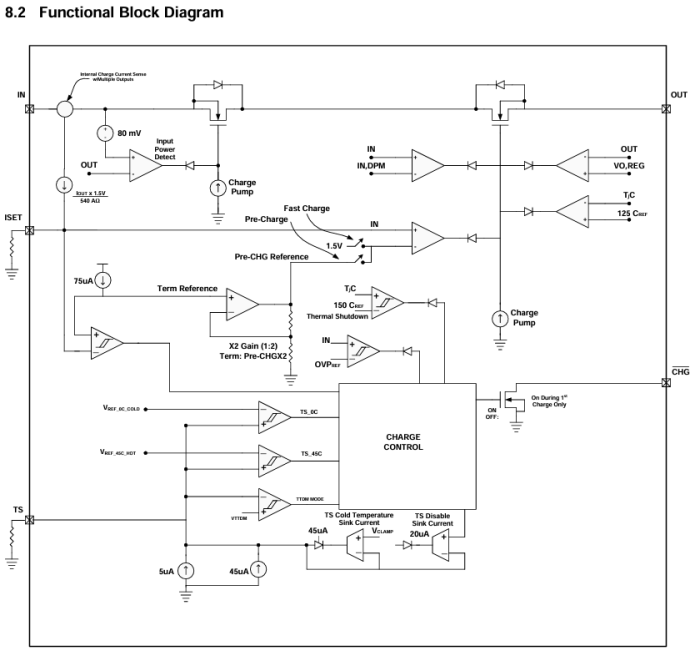


Figure 63: Internal Circuitry of BQ21040 Charging IC

The team looked into two possible charging ICs: the BQ21040 and the TP4056. Both were compatible with single-cell lithium polymer batteries, but each came with its own advantages and limitations. A comparison was made based on key features such as charging current, protection features, complexity of implementation, and more in the table below.

	BQ21040	**TP4056**
Input Voltage Range	4.5V to 6.5V	4.0V to 8.0V
Charge Current	Programmable up to 800mA	Programmable up to 1A
Power Efficiency	High efficiency due to low quiescent current.	Less efficient, higher quiescent current.
Charging Profile	Includes thermal regulation and accurate CV/CC charging profile.	Similar CC/CV profile but less advanced thermal handling.
Protection Features	Built-in protections: overvoltage, overcurrent, and thermal regulation.	Built-in protections: overvoltage and thermal shutdown.
Power Path Management	Supports direct load power while charging.	No power path management, direct battery output only.
LED Indicators	External resistor needed for status indication.	Built-in charge/standby LED pins.
Cost	Slightly higher due to advanced features.	Cheaper and widely available.
External Circuitry Required	Full breakout required, will add additional design time.	Charging module available with USB port, only pins for mating with the module are required.
Decision: TP4056 Based Charging		

As part of the early design phase, a charging circuit was designed using the TP4056 charging IC. This initial schematic explored how the TP4056 could be implemented directly into our board to manage lithium polymer battery charging. The circuit below reflects our first attempt at integrating charging functionality into the system.

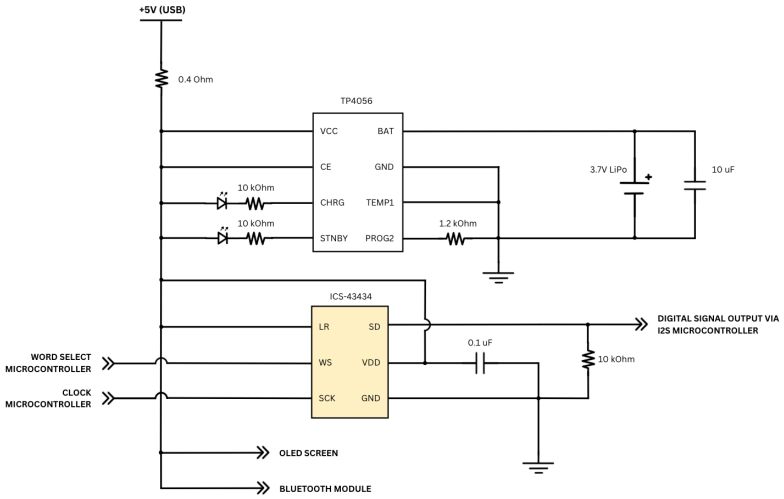


Figure 64: First Charging Circuit Design Using TP4056 Charging IC

However, after further research into the design, the team identified a prebuilt lithium polymer charging module that offered a simpler and fully tested solution. This module used the TP4056 charging IC and included a built-in micro USB port. It was preconfigured with a 1.2 kOhm resistor to limit the charge current to 1000 milliamps, which matched the rated capacity of the selected 1000 milliamp-hour battery. As a result, the battery could be fully charged in approximately one hour. This made the module an ideal fit for the project timeline by minimizing the number of components that needed to be procured and reducing the time spent designing a custom charging circuit, while still meeting performance requirements.

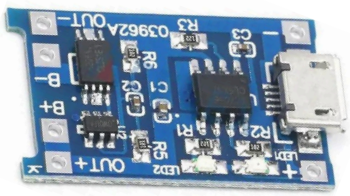
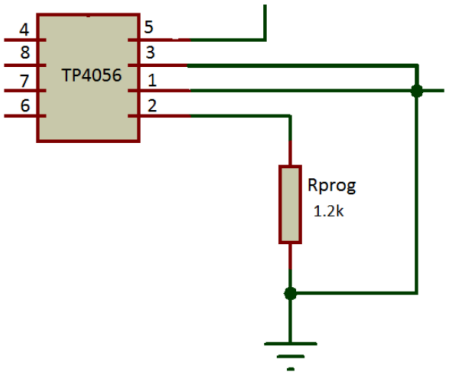


Figure 65: TP4056 Based Charging Module



Rprog Current Setting

R_{PROG} (k)	I_{BAT} (mA)
10	130
5	250
4	300
3	400
2	580
1.66	690
1.5	780
1.33	900
1.2	1000

Figure 66: Description of Programmable Charging Currents based on Resistance

The main tradeoff was that the TP4056 module does not support simultaneous charging and usage, meaning the device cannot be used while charging. Additionally, the module occupied more space on the PCB than a custom-designed solution. Measuring 17 by 29 millimeters, it offered less flexibility in component placement compared to integrating the charging circuitry directly onto the board, where the team could have optimized the layout of resistors, capacitors, indicator LEDs, and the USB port location.

Voltage Regulation

In addition to estimating the current draw of each component, each major component in the system required a specific voltage to function reliably. Supplying too little power would result in unstable or non-functional behavior, while too much could damage the components. The voltage requirements were as follows:

Component	Voltage Requirement
OLED Screen	3.3 Volts
Microphone	3.3 Volts
Bluetooth module	3.6 - 6 Volts (5 Volts)
STM32 Microcontroller	5 Volts

To meet these requirements, the power delivery system had to convert the single-cell 3.7V lithium polymer battery output into both 3.3V and 5V rails. This involved sourcing appropriate voltage regulation ICs that could step down the voltage to 3.3V for the OLED and microphone, and step up to 5V for the STM32 and Bluetooth module. Each regulator had to be compliant with both the input voltage range of the battery and the current demands of the downstream components.

The selected voltage regulator was the Texas Instruments LP38511-ADJ, a low-dropout linear regulator used to provide a clean and stable 3.3V output. This component supports an input voltage range of 2.25 to 5.5 volts and can supply up to 800 milliamps of continuous current, which is more than sufficient for the system's worst-case load of approximately 260 milliamps. The output voltage was configured using external resistors connected to the ADJ pin, allowing for precise tuning based on the needs of the connected components.

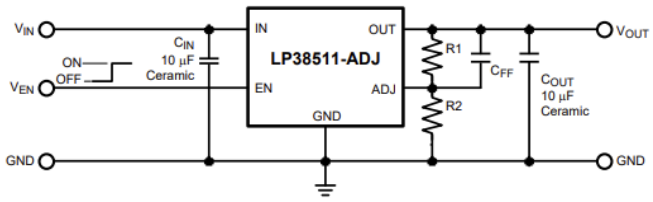


Figure 67: LP38511 LDO Regulator Typical Application Circuit

To boost the single-cell lithium polymer battery to a stable 5V rail, the ISL9111A synchronous boost converter was used. This IC operates with a low start-up voltage of 0.8V and a minimum input of 0.7V, making it ideal for battery-powered designs. The fixed 5V output version was chosen to simplify implementation, and it provided sufficient current for the 5V components in the system, including the STM32 and Bluetooth module. A 4.7 microfarad input and output capacitor were used as recommended in the datasheet for voltage stability and ripple control. The ISL9111A also includes built-in short-circuit, over-temperature, and over-voltage protection.

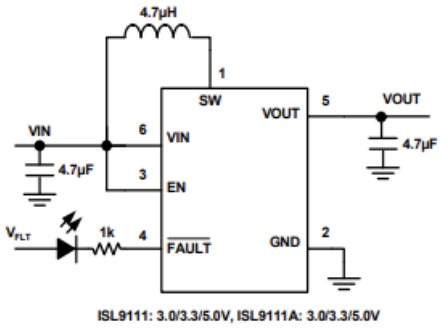


Figure 68: ISL9111A Boost Switching Regulator Typical Application Circuit

Even if components were to operate off of 3.7V, such as the bluetooth module which has an input voltage range of 3.6 to 6V, these boost and low drop voltage regulators provide us stability; a reliable continuous voltage that can safely handle spikes on turn on and turn off. This is highly desirable in our case to protect our core components and keep our device functioning consistently for our customer.

Schematic Routing

Tools

Both the schematic and PCB layout for the device were created using Altium Designer, chosen for its powerful tools, extensive component libraries, and the team's existing familiarity with using it, which helped speed up development and reduce the learning curve during the design process.

To import schematic symbols, the team used an external tool called Altium Library Loader, which integrates directly with Altium Designer. This tool connects to a large online database of schematic symbols and 3D models from various part suppliers and libraries. It allowed components to be added quickly and accurately, without needing to manually search for, download, and assign each symbol or footprint individually. This saved time and ensured that every part used in the schematic had a verified symbol and a matching 3D model for accurate placement and spacing during PCB layout.

Voltage Regulating Circuits

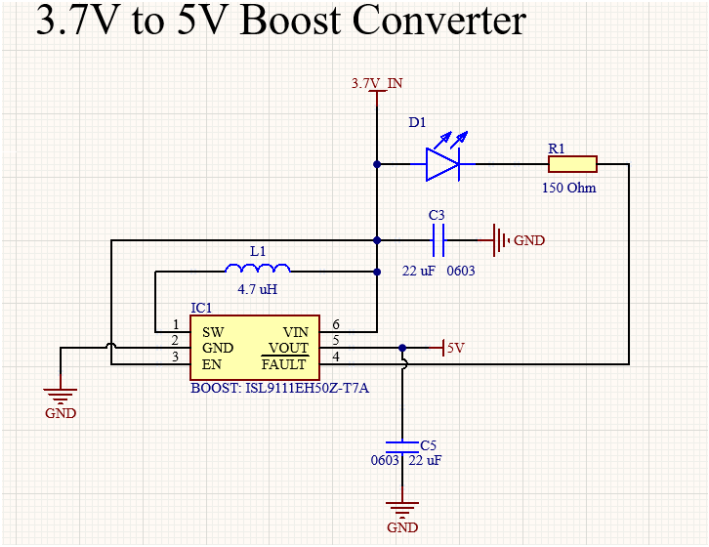


Figure 69: Boost Converter Circuit Schematic

To supply a stable 5V rail from the 3.7V lithium polymer battery, the ISL9111EH50Z-T7A boost converter was used. This IC was selected for its high efficiency and ability to start up from low input voltages, making it ideal for battery-powered applications. The input voltage (3.7V_IN) enters the circuit and passes through a 4.7 mH inductor (L1), which is essential for energy storage and voltage boosting. A 22 microfarad input capacitor (C3) was placed close to the IC to help filter noise and smooth the incoming voltage. The output voltage is regulated to 5V at pin 5 of the IC and stabilized with a 22 uF output capacitor (C5) to ensure consistent delivery to

downstream components. A logic-level enable pin (EN) is connected to ground to keep the converter always active during operation. The converter also includes fault detection on pin 4, though this pin was left unconnected in the current design. An orange LED (D1) with a 150 ohm series resistor (R1) was added to indicate when the 5V rail is active for easier debugging and functionality testing.

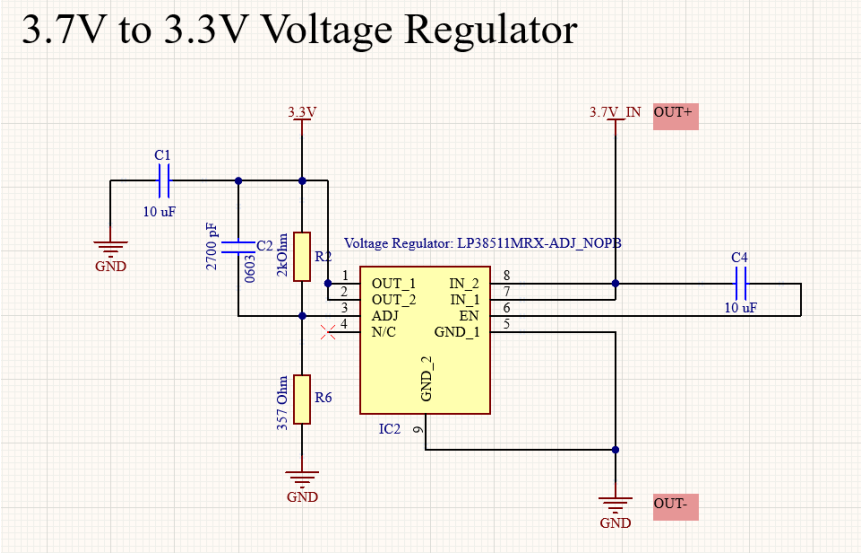


Figure 70: Low Drop Linear Voltage Regulator Circuit Schematic

To supply a stable 3.3 volt rail for noise-sensitive components such as the MEMS microphone and OLED screen, the LP38511MRX-ADJ low dropout (LDO) regulator was used. This adjustable regulator was configured to output 3.3 volts using two external resistors (R2 and R6), which formed a voltage divider connected to the ADJ pin. The selected values (2.60 kOhms and 357 ohms—set the output voltage according to the regulator's internal reference. Input capacitors C4 (10 mF) and output capacitors C1 (10 mF) were added close to the regulator for stability and to reduce voltage ripple. A 2.7 nF capacitor (C2) was also included near the feedback loop for added filtering.

The 3.7V supply entered through the regulator’s input pins, and the regulated 3.3V output was distributed to all connected components that required it. The enable pin (EN) was tied high, keeping the regulator active whenever power was applied. This LDO was chosen for its low noise, high accuracy, and strong thermal performance in a compact layout. With a maximum current capability of 800 mA, the LP38511 comfortably supported the system’s needs without risk of voltage sag or instability.

Core Component Connections

The STM32 Nucleo board serves as the central controller of the system, handling communication, logic, and control for all connected modules. To mount it onto the PCB, 15-pin

female headers were placed in the schematic, corresponding to the STM32's dual 15-pin male headers on each side. The headers were straight-angle through-hole components, meaning the microcontroller would sit parallel to the surface of the board and the STM32 would be easily removable any time for separate testing, rather than being permanently soldered in place. To ensure compatibility, mechanical drawings of the STM32 were referenced to confirm the 0.1-inch (2.54 millimeter) pin pitch, and physical measurements were taken to verify the header model before importing and adding it to the bill of materials.

With the microcontroller being the brains for our logic and operation of each core component, there had to be wiring connections in the schematic for the communication to take place. Careful consideration to the position of the microcontroller was taken so as to follow the pin out of the microcontroller exactly and not flip the orientation of the headers.

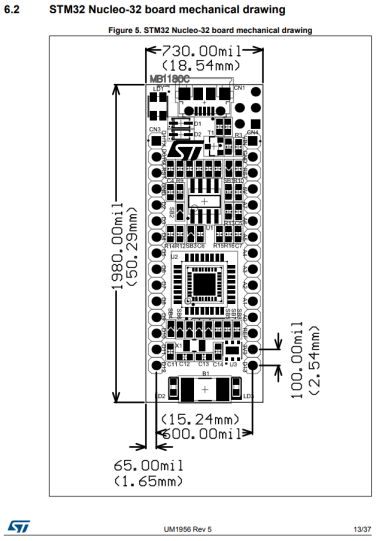


Figure 71: STM32 Mechanical Drawing

GPIO pins were assigned for each major component based on their communication protocols and functional needs. The ICS-43434 MEMS microphone required three I2S lines: LRCL (left-right clock), DOUT (data out), and BCLK (bit clock), all routed to compatible STM32 pins to ensure clean audio signal transmission.

The OLED display required a set of digital control lines, including RESET, DC (data or command), CS (chip select), CLK (clock), and DIN (data in). Each of these was connected to a dedicated GPIO pin, allowing for flexible software control of screen refresh and data transmission. A 7 pin right angle female header schematic footprint was imported to mate with the 7 male pins of the OLED screen and provide power from the

For the Bluetooth module, the STM32's UART transmit (TX) and receive (RX) pins were used to enable serial communication with the HC-05. In addition to these, two additional GPIOs were

assigned to the module's STATE and ENABLE pins. These allowed the microcontroller to monitor Bluetooth connection status and enable or disable the module programmatically.

This complete pin setup from the microcontroller described above mirrored the manual wiring and programming that was used during initial testing and translated into the final PCB implementation.

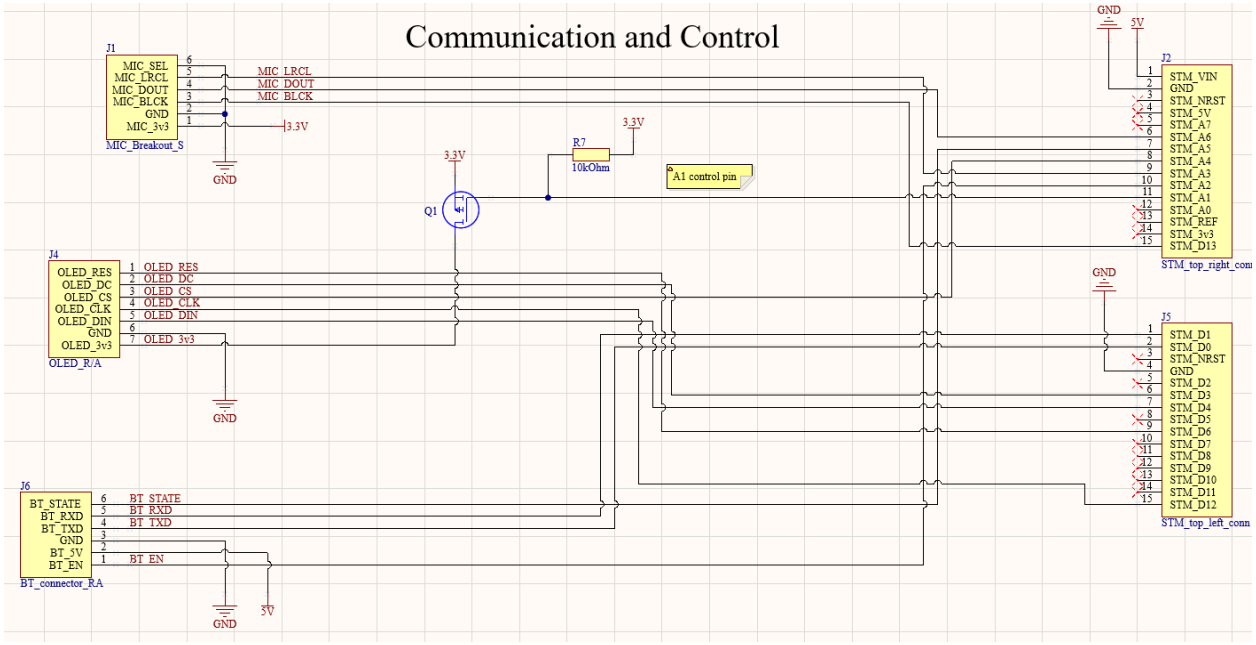


Figure 72: Schematic Wiring for STM Communication with Core Components

```
#define OLED_CS_Pin GPIO_PIN_5
#define OLED_CS_GPIO_Port GPIOA
#define OLED_DC_Pin GPIO_PIN_0
#define OLED_DC_GPIO_Port GPIOB
#define OLED_RST_Pin GPIO_PIN_1
#define OLED_RST_GPIO_Port GPIOB
#define SWDIO_Pin GPIO_PIN_13
#define SWDIO_GPIO_Port GPIOA
#define SWCLK_Pin GPIO_PIN_14
#define SWCLK_GPIO_Port GPIOA
#define IIC_SCL_SOFT_Pin GPIO_PIN_4
#define IIC_SCL_SOFT_GPIO_Port GPIOB
#define IIC_SDA_SOFT_Pin GPIO_PIN_7
#define IIC_SDA_SOFT_GPIO_Port GPIOB

#define HC05_EN_PIN    GPIO_PIN_3
#define HC05_EN_PORT   GPIOA //A2
#define HC05_STATE_PIN GPIO_PIN_6
#define HC05_STATE_PORT GPIOA //A5
```

```
// ----------------- BLUETOOTH MODULE -----------------------
// D1 = TX
// D0 = RX
// HC05 VOLTAGE = 5V
// HC05 EN = A2
// HC05 STATE = A5

// ----------------- MICROPHONE -----------------------------

// SEL = GND
// LRCL = A3
// DOUT = A6
// BCLK = D13
// MICROPHONE VOLTAGE = 3.3v

// ---------------------- OLED -----------------------------
// OLED CS = A4
// OLED RST = D6
// OLED DC = D3
// IIC SCL = D12
// IIC SDA = D4
// OLED VOLTAGE = 3.3v
```

Figure 73: Pre-programmed Pin Assignments for STM32

A software-controlled power switch was implemented using a P-channel MOSFET to control power to the OLED display. The source of the MOSFET was connected to the 3.3 volt power rail, and the drain was connected to the power input of the OLED. The gate was tied to GPIO pin A1 on the STM32, with a 10 kOhm pull up resistor connected between the gate and the 3.3 volt rail. This resistor ensured that the gate would remain high, keeping the MOSFET off by default when the microcontroller was not actively driving the pin. When the GPIO pin was driven low, the voltage difference between the gate and source turned the MOSFET on, supplying power to the OLED. When the pin was driven high again, the MOSFET turned off, cutting power to the screen. This setup allowed the OLED to be toggled on or off through a control in the user interface app, giving users the option to enable or disable captions as needed. This can be seen in figure 72 above, at the centre of the image.

Indicator LEDs

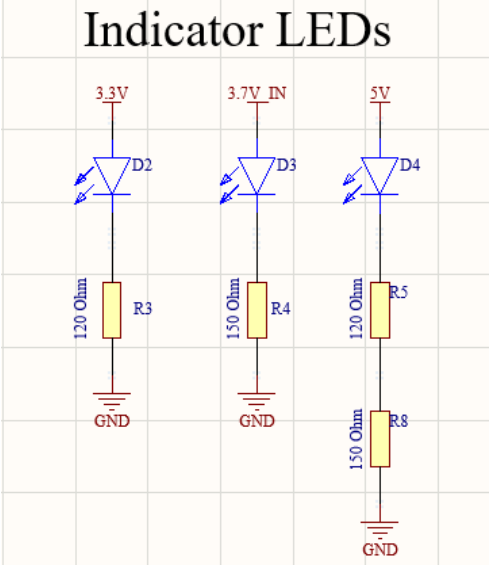


Figure 74: Indicator LED Circuit Schematic

LEDs were added to provide the team visual confirmation that key voltage rails were active during testing. Three green LEDs were used to represent the 3.3 volt, 3.7 volt battery input, and 5 volt power lines. Each LED was placed in series with a current limiting resistor to prevent damage and control brightness. Green LEDs have a forward voltage drop of 2.1V. Thus the calculations were as follows:

$$I_D = (3.3V - 2.1V) / 120\ Ohm = 17.5\ mA$$
$$I_D = (3.7V - 2.1V) / 150\ Ohm = 10.7\ mA$$
$$I_D = (5V - 2.1V) / 270\ Ohm = 10.7\ mA$$

A 120 ohm resistor was used for the 3.3V line to maintain the current at 17.5 mA; within the safe 10-30mA range but still bright and identifiable. For the 3.7V line a 150 ohm resistor was used, keeping the current at 10.7 mA, and a total resistance of 270 ohms was used for the 5V line. These indicators were especially useful during debugging and ensured that power delivery was functioning correctly throughout the system.

Power Input

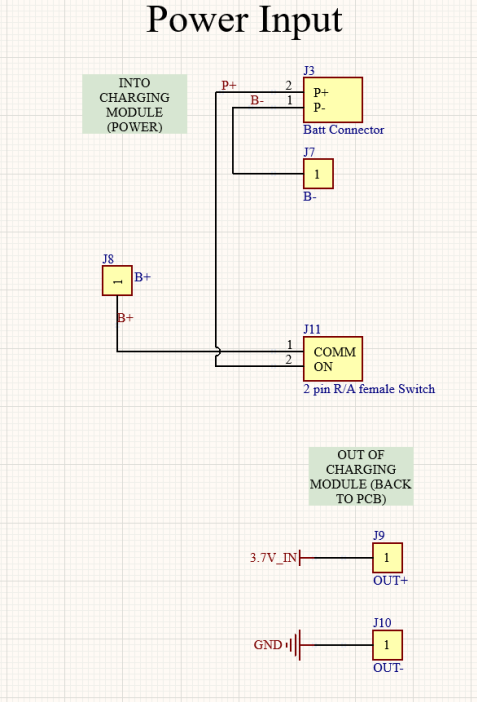


Figure 75: Power Input Circuit Schematic

The device was powered using a lithium polymer battery connected through a 2-pin JST connector with a 2 millimeter pitch, matching the exact model recommended by the battery manufacturer. Proper polarity was critical, as reversing the positive and negative lines could damage the charging module or other components. Power from the battery first passed through a manual ON/OFF switch, allowing the user to control whether the system received power at all. From there, power flowed into the charging and battery protection module, which handled charging safety, overcurrent protection, and regulated output.

Final Schematic

Below is a screenshot in Altium Designer of our final wiring connections. This is the schematic version that was imported into the PCB document to begin planning the layout.

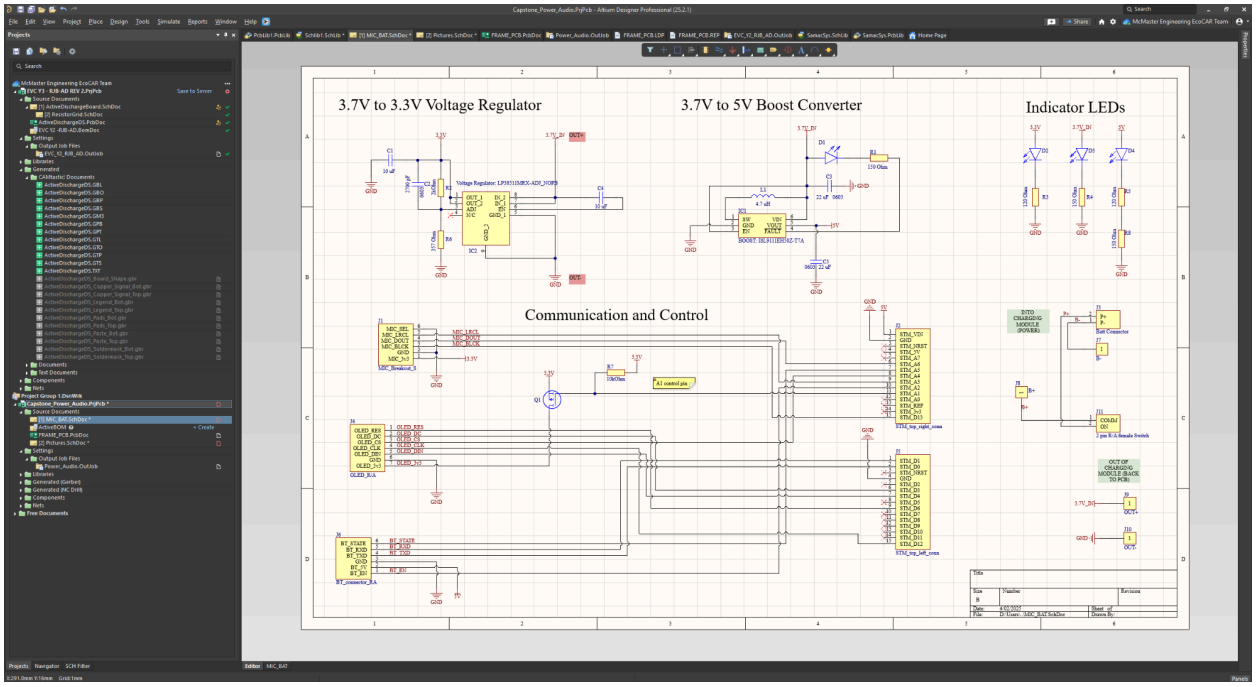


Figure 76: Final Wiring Schematic before PCB Layout Design

PCB Layout Strategy

Large Components

The PCB was designed not just as a functional circuit, but as the physical backbone that all components would mount to in the most compact and controlled way possible. Rather than allowing components to float freely inside the frames with loosely soldered wires, which could lead to short circuits or disconnections, the board served as a clean and structured platform. This approach made the system more reliable, safer, and significantly easier to test and assemble.

An enormous number of measurements were taken throughout the design process. While we had accurate 3D footprints for the headers, most of our core components lacked official 3D models, so we had to rely on mechanical drawings, datasheets, and manual verification. The headers helped establish accurate pin spacing and width dimensions for each component, giving us confidence in lateral alignment. However, height and length had to be carefully measured and accounted for to prevent any overlap or mechanical interference during final assembly.

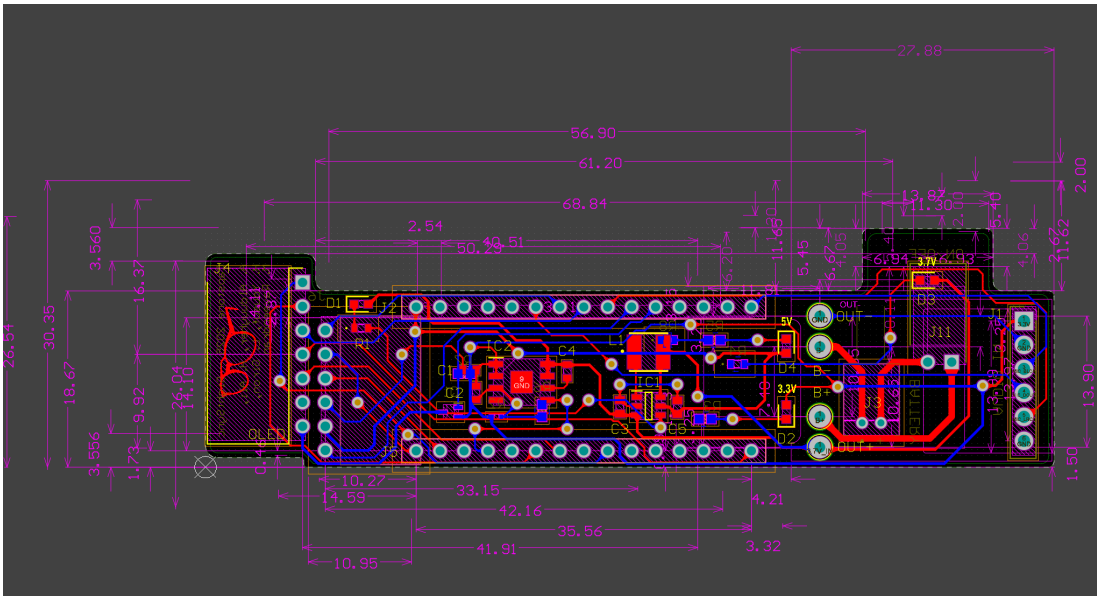


Figure 77: PCB Mechanical Layer

The STM32 Nucleo board was the widest component, so its width set the constraint for the rest of the layout. All other components were required to fit within this boundary. The only exception was the manual ON/OFF switch, which had to stick out from the edge of the PCB to be physically accessible once mounted in the frames. Precise placement of the charging module's USB port was also critical. The USB port needed to align with a cutout in the frame to allow users to plug it in for charging. This meant positioning the module exactly at the edge of the PCB, with careful measurements to ensure the port was neither recessed too far nor protruding awkwardly.

The OLED screen was placed to face the “front” of the glasses (toward the barrel section in the frame) so it could project text toward the user’s eye. The Bluetooth module, while similar in length to the STM32, had right-angle pins instead of downward-facing ones. It was mounted on the underside of the board, parallel to the surface like a cantilever, using a right-angle female header.

The charging module and the microphone were positioned on opposite sides of the board but within the same general region. The charging module was raised slightly above the board to leave room for the ON/OFF switch and battery connector underneath. A dedicated slot was also left on one edge of the board to house the 30mm battery, ensuring it could be neatly secured alongside the rest of the components.

Below is a rough sketch of the PCB from the bottom view, showing the spatial layout and how each packaging decision came together.

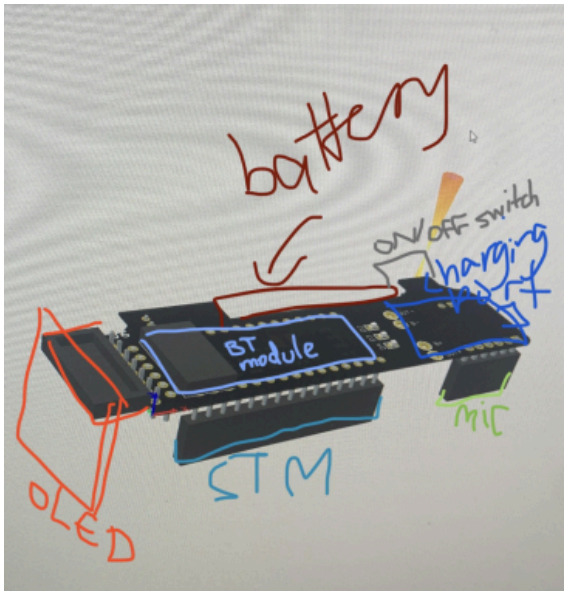


Figure 78: Proposed Physical Layout

Trace Design

Once all critical components were placed, the remaining smaller and passive components were positioned around and underneath the STM32, distributed across both the top and bottom layers of the board. Due to the compact layout and limited surface area, many vias were used to route signals between layers and navigate around dense regions of the PCB.

Bypass capacitors were placed as close as possible to the input and output pins of key components to help smooth voltage fluctuations and filter out electrical noise. These placements were made carefully to ensure low-impedance paths and proper decoupling, especially for sensitive components like the microphone and voltage regulators. Special care was also taken in trace design to avoid sharp 90-degree corners, which can cause signal reflections or manufacturing issues. All trace paths were smoothed with angled bends or arcs for better signal integrity and professional-grade layout standards.

Power traces that carried higher current from voltage regulators to power-hungry components were made wider than standard signal traces. These were sized at 0.35 millimeters, compared to the 0.15 millimeters used for regular data or control signals. This ensured safe current handling, reduced voltage drop, and helped with thermal management along key power delivery paths.

Final Touches

Once all components were placed and routed, final design elements were added to improve both functionality and usability. Labels were placed on the silkscreen to indicate the Z-space or height

clearance needed for key components such as the microphone, charging module, Bluetooth module, OLED, LEDs, battery, and microcontroller. These labels were helpful during assembly and gave clear visual cues for where and how each component should be mounted.

To reinforce the structural integrity of the board and improve its electrical performance, polygon pours were added on both the top and bottom layers. These pours were connected to the ground plane to reduce stray inductance, lower overall noise, and help with thermal dissipation. They also added a layer of mechanical strength to the thin PCB, especially in areas with fewer routed traces.

A final design rule check (DRC) was run in Altium to verify that all spacing, clearances, and silkscreen placements were within acceptable limits. One issue was flagged where two pads on small resistors were too close together, which was corrected by increasing the distance between them. Most passive components used a 0603 package size, which, while small, was manageable for hand soldering during assembly.

Documented Layout Evolution

Figure 79: First Rudimentary PCB Layout, Post Part Import (3D View)

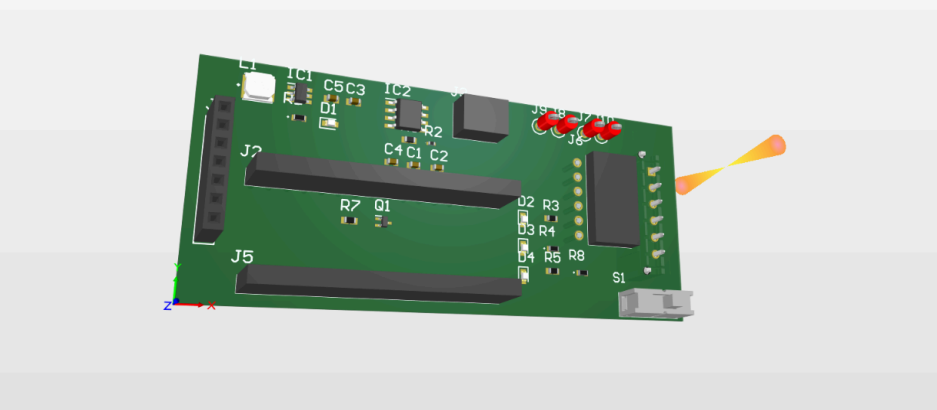


Figure 80: Second Rudimentary PCB Layout (3D View)

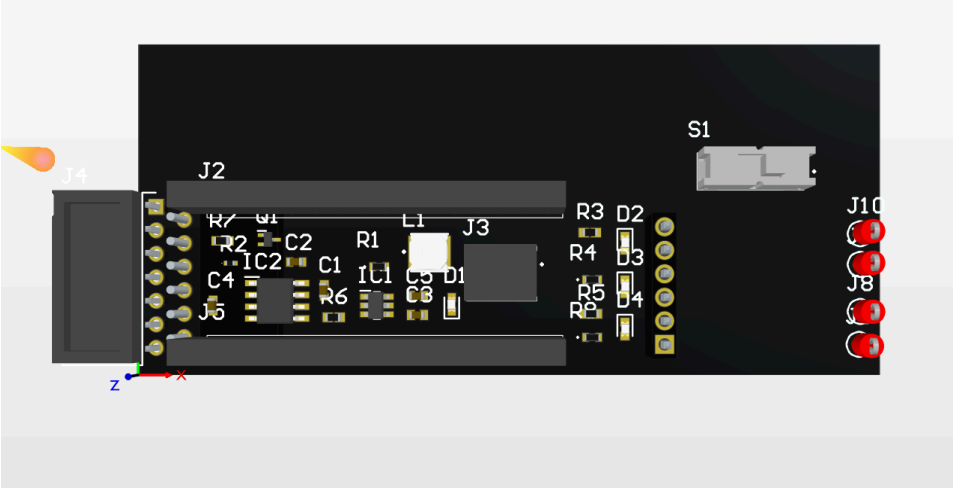


Figure 81: Third Revision PCB Layout (3D View)

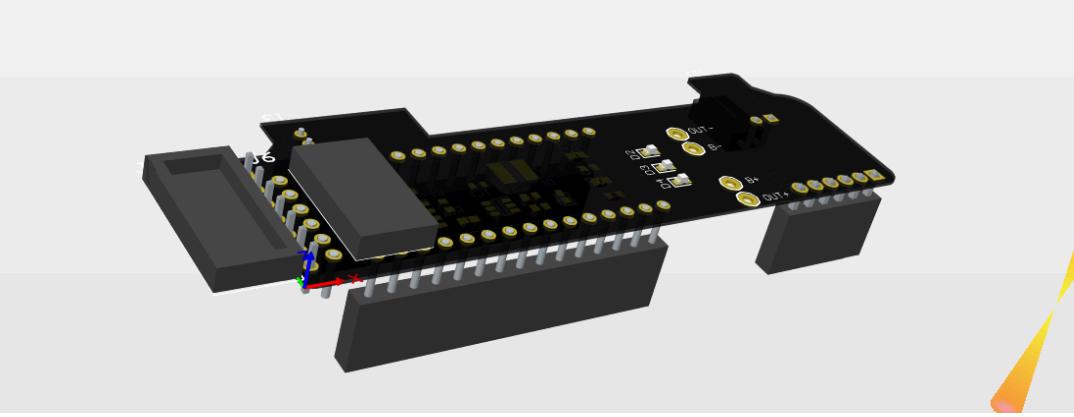


Figure 82: Fourth Revision PCB Layout (3D View)

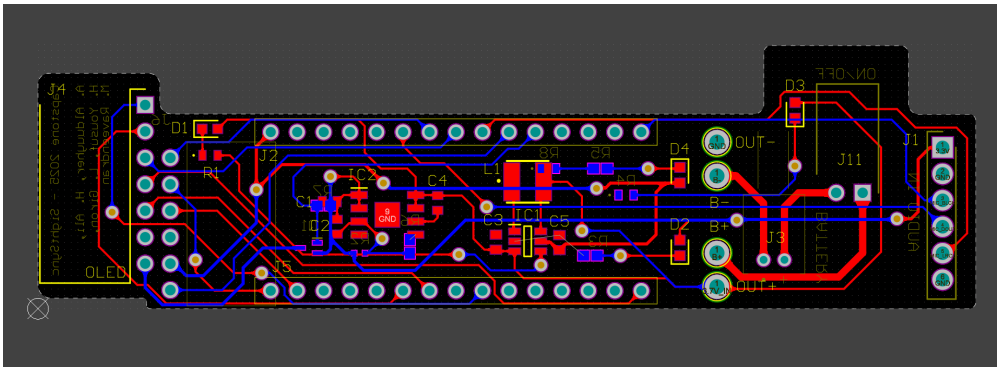


Figure 83: Final Pre-Polygon Pour Wiring (2D View)

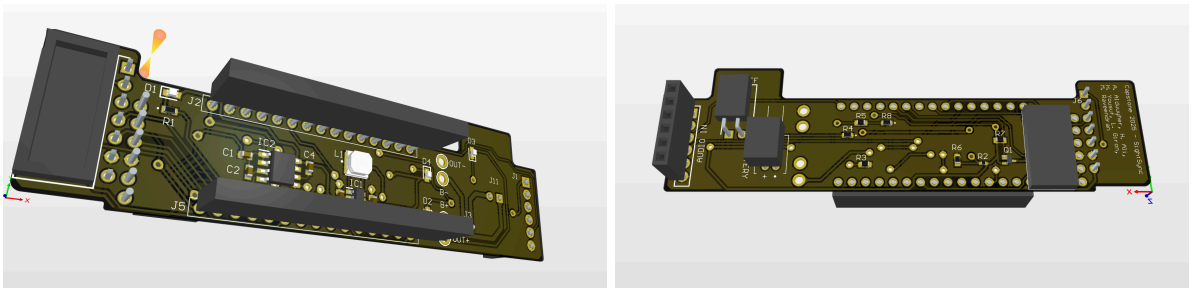


Figure 84: Top/Bottom Side Final Revision PCB Layout (3D View)

Part Procurement and BOM Management

Gerber files were exported in a zipped folder and pre-checked using a PCB manufacturing validation tool provided by OSHPark. This tool displays each layer of the PCB as manufacturers would see it, including the top and bottom silkscreen, board outline, drill holes, copper layers, and solder mask layers. It also generates a visual preview of the finished board, showing exposed copper areas in gold simulating the final manufactured appearance.

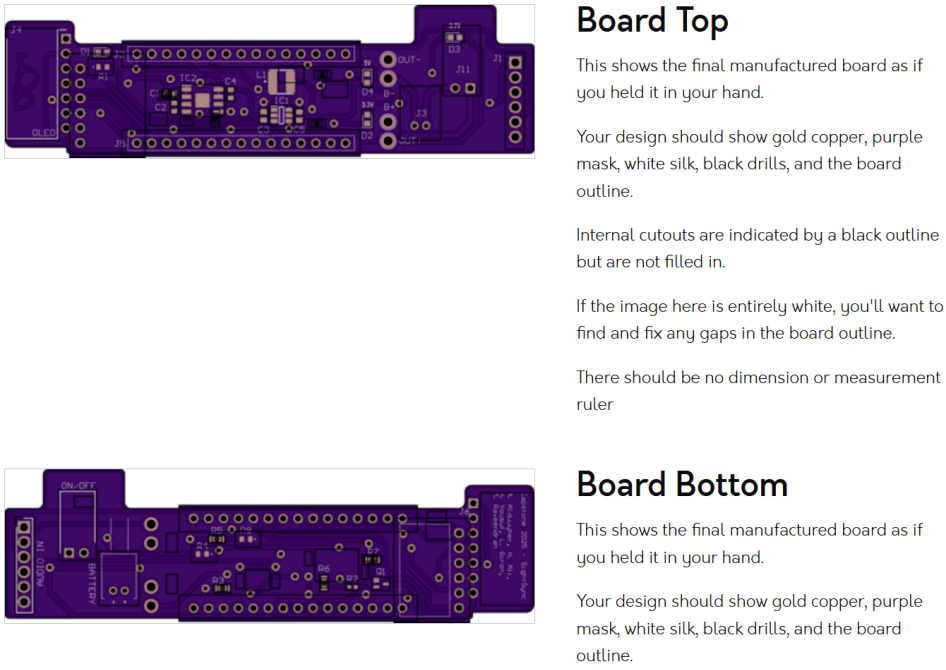


Figure 85: OSHPark PCB Sample View

After inspecting the files, the team went through several iterations to address missing solder pads, misplaced holes, or other layout issues. Each updated version was reuploaded and rechecked using the same tool. Once the board passed visual verification and no errors were

flagged, the final version was submitted through JLCPCB's online web store for fabrication. This manufacturer, based in China, had an average lead time of approximately seven days, including fabrication and shipping. Since the capstone expo was only six days away at the time of ordering, considerable time was spent evaluating different manufacturers. Other options such as PCBWay, FastPCB, QualiEco, Circuit Central, and Siber Circuits and more were considered. As a backup plan, the team placed the order with JLCPCB while continuing to contact local manufacturers by phone and email to request quotes and check for faster lead times. We will discuss in detail how we solved this issue in section 4.0 of this report. The final board order cost $31.66 and included 1 ounce copper weight, no tenting on the vias to allow for hand soldering, and a standard green solder mask.

Part procurement took place immediately after submitting the PCB order. Since the entire design process had been carefully documented, the required components were already tracked and organized. During the schematic design phase, parts were selected based on availability, electrical requirements, and mechanical dimensions, often using exact part numbers to ensure accurate 3D footprints during layout. By the time the PCB was finalized, the complete bill of materials (BOM) in Altium reflected all confirmed components. At this stage purchasing was all that was left, which involved adding the selected parts to carts across suppliers such as DigiKey and Amazon. The full parts list is provided in the following table.

	Part	**Function**	**Unit Price**	**Order Quantity**
1	6 pin 0.1" pitch gold pin female header	MIC Straight Mating connector	$0.71	2
2	15 pin 0.1" pitch tin pin female header	STM Straight Mating connector	$1.35	4
3	7 pin 0.1" pitch gold pin female header	OLED Straight Mating connector (Spare)	$0.80	1
4	7 pin 0.1" pitch gold pin right angle female header	OLED Right Angle Mating connector	$1.01	2
5	6 pin 0.1" pitch gold pin right angle female header	Bluetooth Module Right Angle Mating connector	$0.95	2
6	2 pin 2 mm pitch female header	Battery Connector	$0.11	10
7	Linear voltage regulator	Step 3.7V down to	$2.93	3

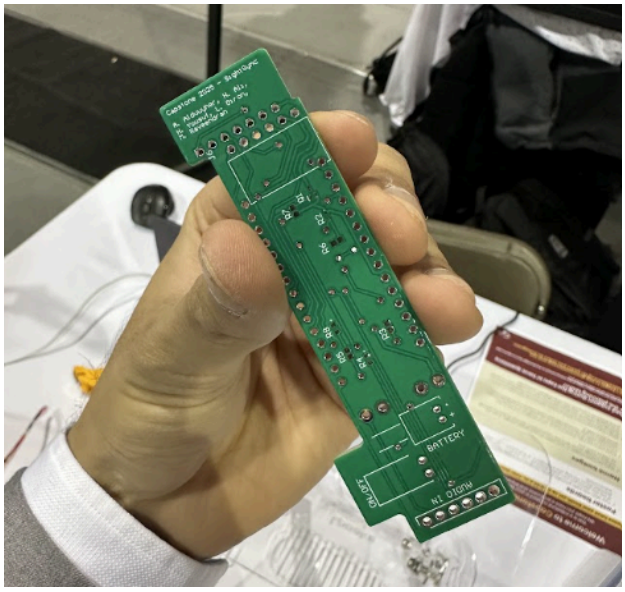
		3.3V		
8	4.7 uH inductor	Current spike control for voltage boost circuit	$0.58	3
9	22 uF ceramic capacitor	Noise filtering for voltage boost circuit	$0.07	10
10	Orange LED	Fault indication	$0.18	10
11	150 Ohm resistor	Current limiting for voltage boost circuit	$0.09	10
12	2kOhm resistor	Current limiting for voltage regulator circuit	$0.05	10
13	357 Ohm	Current limiting for voltage regulator circuit	$0.16	10
14	10 uF capacitor	Noise filtering for voltage regulator circuit	$0.20	6
15	2700 pF capacitor	Noise filtering for voltage regulator circuit	$0.15	5
16	Boost switching voltage regulator	Step 3.7V up to 5V	$3.61	2
17	Manual slide switch	ON/OFF control	$1.21	2
18	2 pin 0.1” pitch gold pin right angle female header	Manual switch right angle mating connector	$0.60	3
19	P-channel mosfet	Software power control	$0.36	4
20	Green LED	Power indicator	$0.15	10
21	10kOhm resistor	Current limiting for software control	$0.03	10
22	120 Ohm resistor	Current limiting for	$0.03	10

		power indication		
23	Single pin male header	Mounting connection for charging IC	$0.65	2
24	Charging IC (pack of 3)	Battery protection and charging circuit	$8.00	1
25	3.7 V LiPo Battery	Power for wireless use	$21.00	1
26	PCB from JLCPCB (pack of 5)	Board for packaging and power delivery	$32.48	5
Total: $122.59				

Testing and Results

Testing was limited due to time constraints, as the final PCB arrived just one day before the capstone expo. With only a few hours available, the board had to be hand-soldered and tested on the same day. Despite the rush, the system powered on successfully and functioned as intended, though a few issues emerged during testing. The manual on and off switch required being toggled twice to activate the system, likely due to initial capacitance buildup in the power path.

The device was tested by several demo users at the expo event and performed well without any issues throughout the entire day, even in real use cases with a few individuals having hearing impairments. The battery lasted approximately 4.5 hours on a full charge and reached 30 percent charge within 20 minutes, confirming the effectiveness of the selected charging circuit.



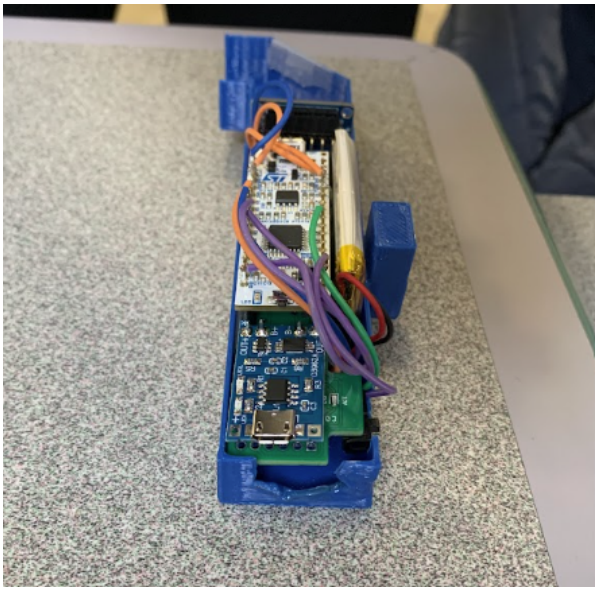


Figure 86: Manufactured PCB, Pre and Post Assembly

Software Development [3.4]

Taking inspiration from XRAI glasses, a similar competitor, it was deemed that due to time and resource constraints, an external API will conduct the speech to text conversion. This is best done utilizing a speech recognition library, in which we are able to utilize an open source API, such as Google speeches speech to text conversion using wifi to conduct the speech to text conversion. Android Studio was used as a development platform, in which it enables easy debugging and testing of our app. The coding language used is java and xml and we utilize androidx, android, reactivex and other libraries to use our app. The app utilizes programming language data structures, such as arrays, floats, strings, characters and integers, along with loops and if, else if and else statements, to handle interrupts and condition based logic.

In general, the app will have a simple user interface, in which one can start up the app, connect their phone to the microcontroller through bluetooth and begin the speech to text processing. It will receive audio data from our microphone, to then ultimately transfer our data to the glasses' microcontroller and OLED screen through bluetooth communication. It will also have a power on and power off button to control the OLED screen, a transcription button to open a new screen to show transcribed text and a filter button to allow filtering of the transcribed conversations.

The app was made utilizing a github repository by "The Frugal Engineer", where aspects of the bluetooth communication setup and of writing data through bluetooth seen in the connectThread and connectedThread public classes, were greatly modified. The original repository was used to send data based on a temperature sensor and was modified to allow continuous sending of data through bluetooth.

The comprehensive app code can be found in our github repository. It contains the user interface written in xml and the backend software written in java, along with two thread files that will run in the background, to allow multiple processes running in parallel.

App Development

MainActivity.java

This is our main java class, in which it will control all functionalities on our primary view of our app. The following methods were defined and used within our main class, onCreate(), onRequestPermissionsResult, searchDevices(), checkBluetoothPermissions, speak(), startListening(), processRecognizedText(), onResume() and onDestroy().

The most important method is the onCreate() method, this initializes all the buttons and textViews that will be used in our app and assigns variables to them, such as on_off = findViewById(R.id.on_off);. This is a line of code utilized to initialize the on and off button to our xml files that are used to sync our user interface with our backend code. We add getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON); to ensure our device does not sleep or turn off while in use. We also handle android specific permission handling, to check for audio and microphone permission.

```
92      if (ActivityCompat.checkSelfPermission( context: this, Manifest.permission.RECORD_AUDIO) != PackageManager.PERMISSION_GRANTED) {
93          ActivityCompat.requestPermissions( activity: this, new String[]{Manifest.permission.RECORD_AUDIO}, requestCode: 1);
94      }
```

Figure 87: Permission Handling.

The buttons are made utilizing the variables defined earlier and use an onClick(View view) method to allow lines of code to activate once the button is pressed. The key buttons in our main screen are buttonGotoSecond, on_off, clearValues, stopButton, connectToDevice, searchDevices, and speak button.

The button to go to the second screen, simply allows the user to move from the main screen to the second screen and sends over the textView variable, textToShow, to then display our transcribed text on our second screen.

```
buttonGoToSecond.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Text to send to SecondActivity


        // Start SecondActivity and pass the text
        Intent intent = new Intent( packageContext: MainActivity.this, SecondActivity.class);
        intent.putExtra( name: "text_key", textToShow);
        startActivity(intent);
    }
});
```

Figure 88: Button to go to the next screen.

The next button is our on_off button, which is an imagebutton, that will change images based on whether our OLED screen is on or off. The logic utilized are global variable flags, that will check to see if the integer logic is true or false and will correspondingly display text if the screen is on or off.

```
// Set a listener event on a button to clear the texts
on_off.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if(flag1 == 0){
            processRecognizedText(".");
            flag1 = 1;
            on_off.setImageResource(R.drawable.on_1);
            btReadings.setText("Turning Screen On...");
        }
        else{
            processRecognizedText(".");
            flag1 = 0;
            on_off.setImageResource(R.drawable.off_1);
            btReadings.setText("Turning Screen Off...");
        }
    }
});
```

Figure 89: On and Off Button.

The next button is our clear button, which will set all textView to an empty string (“”) and will set our threads connectedThread and connectThread to null. This will clear all background processing that has been happening when our app is processing text and sending it to our device. It will also stop our speech recognition and transcription and disconnect from bluetooth.

```
// Ensure arduinoBTModule is also reset so you must rescan
arduinoBTModule = null;
if(isListening) {

    speechRecognizer.stopListening();  // Stop listening
    speechRecognizer.cancel();        // Cancel any ongoing recognition

}
connectToDevice.setEnabled(false);
isListening = false;
isRecognitionActive = false;
```

Figure 90: Clear all resources used, to reset the app.

The next button is our stopButton, which will stop our speech to text recognition. It first has a logic check, in which we see if speech recognition is active and if it is active, it will disconnect our device.

```
stopButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        btReadings.setText("Speech Recognizer is Off...");

        if(isListening) {

            speechRecognizer.stopListening();  // Stop listening
            speechRecognizer.cancel();        // Cancel any ongoing recognition

        }

        isListening = false;
        isRecognitionActive = false;
    }
    });
```

Figure 91: Stop button to stop the speech recognizer.

There are two buttons used to check for bluetooth devices and connect to our bluetooth devices, noted as connectToDevice and searchDevices. The search devices will check that our device is compatible with bluetooth communication, then will check to see if bluetooth is currently enabled and then search for a bluetooth device and display it on our textView.

```
if (pairedDevices.size() > 0) {
    for (BluetoothDevice device : pairedDevices) {
        String deviceName = device.getName();
        String deviceHardwareAddress = device.getAddress();
        btDevicesString += deviceName + " || " + deviceHardwareAddress + "\n";

        if (deviceName.equals("HC-05")) { // If HC-05 is found
            Log.d(TAG, msg: "HC-05 found, saving device.");
            if (device.getUuids() != null && device.getUuids().length > 0) {
                arduinoUUID = device.getUuids()[0].getUuid();
            } else {
                Log.e(TAG, msg: "No UUIDs found for device: " + device.getName());
            }
            arduinoBTModule = device;
            connectToDevice.setEnabled(true); // Enable the button once the device is found
        }
    }
    btDevices.setText(btDevicesString);
    btReadings.setText("Found Devices...");
} else {
    Log.d(TAG, msg: "No paired devices found.");
}
```

Figure 92: Buttons to search for bluetooth devices.

The other button is used to connect our device to our bluetooth module, the HC-05. It will call upon our connectThread class and pass our bluetooth devices UUID to those threads to enable bluetooth communication. We check that our connectThread socket is connected and then we can start our connectedthread. We then set a 3000 ms buffer to ensure stability, as there is a physical delay associated with pairing a device that cannot be easily circumvented due to the quality of our bluetooth module and using Bluetooth Classic 2.0. We will also display text to show whether our device has successfully connected or if it failed to connect.

```
239                 if (arduinoBTModule != null) {
240
241                     // Check if connectThread already exists
242                     if (connectThread == null) {
243                         // Create a new connectThread only if it's not already created
244                         connectThread = new ConnectThread(arduinoBTModule, arduinoUUID, handler);
245                         connectThread.start();  // Start the connection thread
246
247                         // Use a handler or delay to check if the socket is connected
248                         new Handler().postDelayed(() -> {
249                             // Ensure the socket is connected before starting the ConnectedThread
250                             if (connectThread.getMmSocket() != null && connectThread.getMmSocket().isConnected()) {
251                                 if (connectedThread == null) {
252                                     connectedThread = new ConnectedThread(connectThread.getMmSocket());
253                                     connectedThread.start();  // Start the communication thread
254                                     btReadings.setText("Connected...");
255                                 }
256                             } else {
257                                 Log.e(TAG,  msg: "Connection failed: Socket not connected.");
258                                 btReadings.setText("Failed to Connect...");
259                             }
260                         },  delayMillis: 3000); // Adjust the delay if needed
261                     }
```

Figure 93: Button to connect the smartphone to our microcontroller.

The final button is the speak button, which uses the speak(View view) method. This button is used to initialize our speechRecognizer, which uses google speeches smartphone's built-in speech to text api. There are various methods within a speechRecognizer, such as onEndofSpeech(), onError(), onResults(), and onPartialResults().

The onEndofSpeech() method is used when the device detects silence and ends speech recognition. This will then call our speech recognizer again, using recursion.

```
@Override
public void onEndOfSpeech() {
    Log.d(TAG,  msg: "Speech input ended");
    if (isListening && !isStartListeningPending) {
        isStartListeningPending = true;
        new Handler().postDelayed(() -> {
            startListening();
            isStartListeningPending = false;
        },  delayMillis: 5); // Increased delay to 500ms
    }
}
```

Figure 94: This method is used to handle end of speech recognition.

The onError() method is used when our device is unable to transcribe the audio it receives, either due to poor clarity or ambient noise, it will conduct error handling and then call our speech recognizer.

```
@Override
public void onError(int error) {
    Log.e(TAG,  msg: "Speech recognition error: " + error);
    if (isListening && !isStartListeningPending) {
        isStartListeningPending = true;
        new Handler().postDelayed(() -> {
            startListening();
            isStartListeningPending = false;
        },  delayMillis: 50); // Increased delay to 500ms
    }
}
```

Figure 95: This method is error handling during speech recognition.

The next method is our onResults() method, this will take our successfully recorded and processed speech and then provide it in a usable form to then be used in our transcription screen as a string. We then create an array using the data and then condition check the data to see if it is listed as null or an empty string. This is then formatted using the timeformatter library and is then added to our textView textToShow which is displayed in our second screen view.

```
no usages
@Override
public void onResults(Bundle results) {
    // Get the results and convert them into text
    //if (hasFinalResult) return; // Prevent duplicate processing
    //hasFinalResult = true;
    String stopgap = "";
    ArrayList<String> data = results.getStringArrayList(SpeechRecognizer.RESULTS_RECOGNITION);
    if (data != null && !data.isEmpty()) {
        String recognizedText = data.get(0);
        textView.setText(recognizedText);
        previousText = "";

        stopgap = recognizedText;
        LocalDateTime current = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd, HH:mm:ss");
        String formatted = current.format(formatter);

        textToShow = textToShow + formatted + ":   " + recognizedText + '\n';

        Log.d(TAG, msg: "Recognized text: " + recognizedText);
        // Process recognized text here (e.g., translate or send via Bluetooth)
        String character = recognizedText;
        //while(!character.equals("\n")){
       // processRecognizedText(" \n");
        //    processRecognizedText(recognizedText);
         //}
    }
}
```

Figure 96: This method processes our speech recognition.

The final method within our speechRecognizer is our onPartialResults. This allows our app to send data very close to real-time, as it will process speech to text, while the speech recognizer is working. It will call upon another method, processRecognizedText(newPart), where newPart is our data we wish to send through bluetooth.

```
if (!newPart.isEmpty()) {
    //Log.d(TAG, "New words: " + newPart);
    processRecognizedText(newPart);
}
```

Figure 97: This method does the real-time transcription.

A key method is our startListening() method, which controls our speechRecognizer. We create an intent, as we wish to use the built-in speech recognizer in our smartphone and then enable partial results for real-time capability. This is why it is important to have permission handling earlier, as it is essential that the user enables permission of their smartphone's microphone and bluetooth communication, because it would otherwise cause the app to crash due to their built-in phones security system

```
private void startListening() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL, RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    intent.putExtra(RecognizerIntent.EXTRA_PARTIAL_RESULTS, value: true);
    intent.putExtra( name: "android.speech.extra.DICTATION_MODE", value: true);
```

Figure 98: This method initializes the speech recognizer.

The processRecognizedText(String text) is a method used to send our data to our connectedThread, that was initialized earlier to then send the data over bluetooth to our HC-05 bluetooth module. We first have a nested if statement, in which we check to see that bluetooth is enabled and that our connectedThread is active and then we write to our connectedThread in bytes. At this point, our speech recognizer would continuously listen and then our background threads would handle the communication aspect of sending data to our bluetooth module serially.

```
private void processRecognizedText(String text) {
    // This function will process recognized text
    // For example, send it over Bluetooth or do some processing
    Log.d(TAG, msg: "Processing recognized text: " + text);
    //btReadings.setText("Processing Speech...");
    // If you want to send this data over Bluetooth, use your existing method here
    if (arduinoBTModule != null && !text.isEmpty()) {
        if (connectedThread != null && connectThread.getMmSocket().isConnected()) {
            text = text + " \n";

            connectedThread.write(text.getBytes());
            Log.d(TAG, msg: "Sent to HC-05: " + text);
        } else {
            Log.e(TAG, msg: "No active Bluetooth connection.");
        }
    }

}
```

Figure 99: This method sends data to our threads

The final methods are onResume() and onDestroy() which are both used to keep our app running while our device is turned off and the destroy method is used to release the speechRecognizer resources when the app is closed to ensure there are no issues with our smartphones functionality after the app is closed. These are used to ensure all resources are handled properly to prevent automatic restarts of our app and any unexpected crashes.

```java
@Override
protected void onResume() {
    super.onResume();

    if (speechRecognizer == null) {
        Log.d(TAG, msg: "Reinitializing SpeechRecognizer on resume.");
        speechRecognizer = SpeechRecognizer.createSpeechRecognizer( context: this);
    }

}

@Override
protected void onDestroy() {
    super.onDestroy();
    if (speechRecognizer != null) {
        speechRecognizer.destroy();  // Release the SpeechRecognizer resources
    }
}
```

Figure 100: These methods allow use of our app while the app is closed.

ConnectThread.java

This is a class that will open the bluetooth socket to our HC-05 bluetooth module. It uses the device's UUID to set the results. We first initialize a variable mmSocket of type BluetoothSocket and have a run() method, to then get the bluetooth socket to connect with our bluetooth device. It will then call mmSocket.connect() to connect to our device through the socket. This socket is then passed onto the connectedThread.

```java
@SuppressLint("MissingPermission")
public void run() {

    try {
        // Connect to the remote device through the socket. This call blocks
        // until it succeeds or throws an exception.
        mmSocket.connect();
    } catch (IOException connectException) {
        // Unable to connect; close the socket and return.
        handler.obtainMessage(ERROR_READ, obj: "Unable to connect to the BT device").sendToTarget();
        Log.e(TAG, msg: "connectException: " + connectException);
        try {
            mmSocket.close();
        } catch (IOException closeException) {
            Log.e(TAG, msg: "Could not close the client socket", closeException);
        }
        return;
    }
```

Figure 101: This method connects our smartphone to our bluetoothsocket.

ConnectedThread.java

This is a public class that, given an open bluetooth socket, will open, manage and close the data stream from the HC-05 bluetooth module. First we will initialize the input and output streams and then use a run() method, to continue to listen to the inputstream until an exception is thrown. The new method is our write(byte[] bytes) method, in which we set a bluetooth buffer size to 64 bytes and initialize our offset to 0. We then track the position at which we are sending data to our bluetooth module, by incrementing our offset by the number of bytes that were sent. We run a while(){} until the offset surpasses the total input streamed data. This is then sent through mmOutStream.write(bytes, offset, bytesToSend);.

```
public void write(byte[] bytes) {
    try {
        int chunkSize = 64; // Bluetooth HC-05 buffer size (adjust if needed)
        int offset = 0; //
        // Track current position

        while (offset < bytes.length) {
            int bytesToSend = Math.min(chunkSize, bytes.length - offset);
            mmOutStream.write(bytes, offset, bytesToSend);
            mmOutStream.flush(); // Ensure data is sent immediately
            offset += bytesToSend;

            Log.d(TAG, msg: "Sent chunk: " + new String(bytes, offset: offset - bytesToSend, bytesToSend));
            Thread.sleep( millis: 250); // Short delay to prevent buffer overflow
        }

        Log.d(TAG, msg: "Finished sending full message.");
    } catch (IOException | InterruptedException e) {
        Log.e(TAG, msg: "Error writing to output stream", e);
    }
}
```

Figure 102: This method sends data to our microcontroller.

SecondActivity.java

Similarly to the MainActivity.java, we have to initialize buttons, textviews and have the following methods: onCreate(), and findLinesContainingWord(). This activity is connected to our activity_second.xml file, in which we link our buttons seen there to our second activity button, so they will activate certain lines of codes.

The onCreate() method will initialize all of our buttons, and textView's. The buttons include buttonSearch, buttonReset and buttonBack. The textView includes a textViewDisplay and an editable text view, known as EditText called editTextSearch.

Once we are on the second screen, it will automatically pull all transcribed data that was sent through our intent and will display the received text within our textViewDisplay.

```
// Get the text from the Intent
receivedText = getIntent().getStringExtra( name: "text_key");

// Display the received text
textViewDisplay.setText(receivedText);
```

Figure 103: This method displays transcribed text on our app.

The editTextSearch is used in tandem with the buttonSearch. In this onClick() method we ask the user to type in a set of characters they would like to search through their transcribed text and then once the search button is pressed, it will check to see if the if statement is empty and if it is not, it will call the findLinesContainingWord() method. This will then compare our entire transcribed conversation with the typed in set of characters and then return a results string that will then be displayed in our textViewDisplay for an entire sentence that includes the word.

```
// Search Button Click Listener
buttonSearch.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String searchText = editTextSearch.getText().toString().trim();
        if (!searchText.isEmpty()) {
            String result = findLinesContainingWord(receivedText, searchText);
            textViewDisplay.setText(result);
        } else {
            textViewDisplay.setText("No input provided");
        }

        // Hide the keyboard after clicking search
        hideKeyboard();
    }
});
```

Figure 104: This method allows the user to enter text.

The next button is the buttonReset, in which it simply restores the textViewDisplay, to its original view and clears the editTextSearch to an empty string (“”).

```
// Reset Button Click Listener
buttonReset.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        textViewDisplay.setText(receivedText); // Reset to original text
        editTextSearch.setText(""); // Clear the search input
        hideKeyboard(); // Hide keyboard when resetting
    }
});
```

Figure 105: This method resets the text view to its original state.

The final button is the buttonBack and it will close the second Activity and return back to the mainActivity screen using finish();.

The last method within our second screen is the findLinesContainingWord(), in which we use an if statement to check for null text and then split our current text by the new line character (“\n”) and then iterate through the newly built string, using a for loop. We then compare the searched word with the trimmed set of lines and append all lines that include our searched word to the results string. Ultimately, it will then return the string to the user and identify if the searched word is seen or not. In the case there does not exist a sentence that includes the searched word or time stamp, it will return “No text available”.

```
// Function to find ALL lines containing the searched word
1 usage
private String findLinesContainingWord(String text, String word) {
    if (text != null) {
        String[] lines = text.split( regex: "\n"); // Split text into lines
        StringBuilder result = new StringBuilder();

        for (String line : lines) {
            if (line.contains(word)) {
                result.append(line).append("\n"); // Append matching lines
            }
        }

        return result.length() > 0 ? "Found:\n" + result.toString().trim() : "Word not found!";
    }
    return "No text available!";
}
```

Figure 106: This method searches for lines of text that match the searched word.

User Interface Design

The user interface is designed in Android Studio. It utilizes xml to create various buttons, textView’s, image buttons, scrollable textviews and display images on our app. There are two

screens to our app, one that is our initial screen and then a second screen that displays the transcribed text.

activity_main.xml

The main components of this screen include our buttons that search for bluetooth devices, our speak, stop, transcription and clear buttons, our power on and off button and lastly our textView's. The layout width and height were constrained to match the parent, as in it matches the width and height of whichever android device is used. All the buttons are then constrained appropriately to each other, to ensure it can easily be used on different devices of different scale and resolutions. This prevents situations where part of a button will appear off screen for one device and ensures uniformity. The size of text is controlled by sp and dp, dp is density-independent pixels and sp is scale-independent pixels, they both are used in our app. The scale-independent pixels are used based on the user's smartphone and will adjust automatically based on screen density. The density-independent pixels are dependent on the physical density of the screen and will automatically adjust depending on the scale of the smartphone.

A sample button is seen with the stop button:

```
<!-- Stop Button -->
<Button
    android:id="@+id/stopButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="         Stop         "
    app:layout_constraintBottom_toTopOf="@id/refresh"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="26dp" />
```

Figure 107: Sample xml code for a button.

The key features of all the buttons are the unique id used to connect our .java files with the xml files, the text displayed on the button to let the user know which button is which and the size of the button which is based on the size of the text. All buttons are made to be rectangular in nature and are constrained by other buttons with slight offsets seen by layout_marginBottom and layout_marginEnd and constraining the button to the unique id of another button seen by "@id/refresh".

A unique button is our image button, which is initially created in the off position, but will change depending on whether the OLED screen is on or off. This is controlled by the app and the user would click the button as they would a normal button.

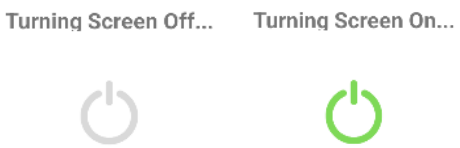


Figure 108: The on and off images.

The image button has the same constraints and unique id that a normal button would have, however, it will change based on the image png used from our MainActivity.java file. The two power icons above are created as off_1.jpg and on_1.jpg, which will alternate depending on which state we had pressed the button and the state we want our OLED screen to be in.

```
<ImageButton
    android:id="@+id/on_off"
    android:layout_width="150dp"
    android:layout_height="150dp"
    android:layout_marginBottom="65dp"
    android:background="?android:colorBackground"
    android:contentDescription="ON_OFF button"
    android:scaleType="fitCenter"
    android:src="@drawable/off_1"
    app:layout_constraintBottom_toTopOf="@id/refresh"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />
```

Figure 109: The xml code used for the image button.

The last type of asset used in the activity_main.xml file is the textview, where similarly to the button, they have a unique id and are constrained to match the parent theme. The only difference is that they cannot be edited by the user and are strictly used in this case to display text.

```
<!-- Linked Bluetooth devices: Text -->
<TextView
    android:id="@+id/btDevicesText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Linked Bluetooth devices:"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginTop="32dp" />
```

Figure 110: The xml code used for the textview.

The final screen will have all our buttons, textviews and image buttons that can be viewed and interacted with the user, as seen below.

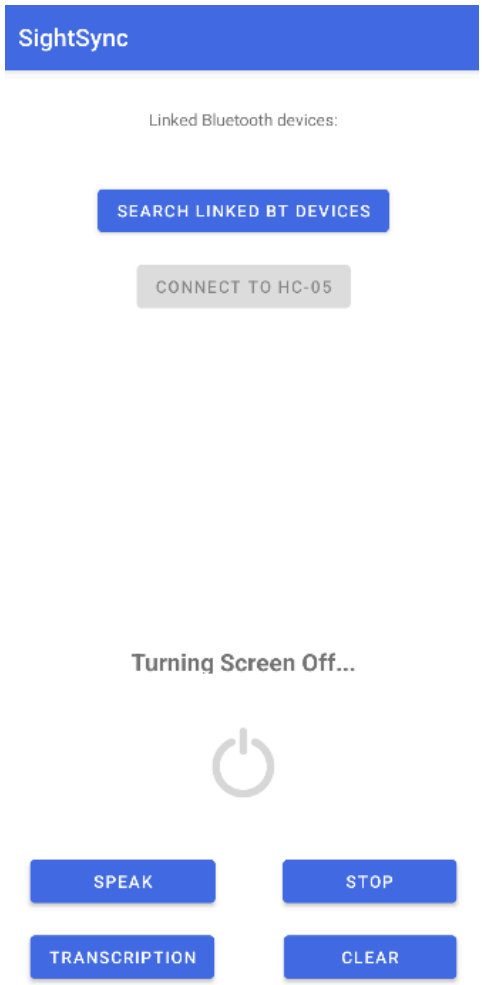


Figure 111: This image is our first screen of our app.

activity_second.xml

The second screen is very similar in nature to the main activity xml file, as it constrains the app to match whichever android device it is using, by constraining the width and height to "match_parent" and the size of text is controlled by sp and dp. It has buttons and textView features, however, the main difference is the additional EditText and the ScrollView assets.

The EditText asset allows the user to enter in any phrase, as a text box. It is connected to the SecondActivity.java file, through its unique id "editTextSearch" and it is similarly constrained as all other assets within the xml file, to match its parent. Due to the varying lengths of the text that can be written, it will also wrap the text, to not miss any characters on the app.

```
<EditText
    android:id="@+id/editTextSearch"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter phrase to search"/>
```

Figure 112: The xml code used for the edit text asset.

The ScrollView asset uses a TextView asset in combination with the scroll feature, to allow the user to read their transcribed text and scroll through the many lines of texts that would have been seen. This allows seamless and easy reading of our text, which is convenient for the user. It has similar constraints, as the widths of the textbox are static, however, like the EditText asset, it will wrap lines of text, to make it fit within the static width, without missing out on other characters.

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="430dp"
    android:layout_marginTop="16dp">

    <TextView
        android:id="@+id/textViewDisplay"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="18sp" />

</ScrollView>
```

Figure 113: The xml code used for the scroll view asset.

This provides the user with a second screen that is accessed once the Transcription button is clicked and can then return the user back to the original screen when the back button is clicked.

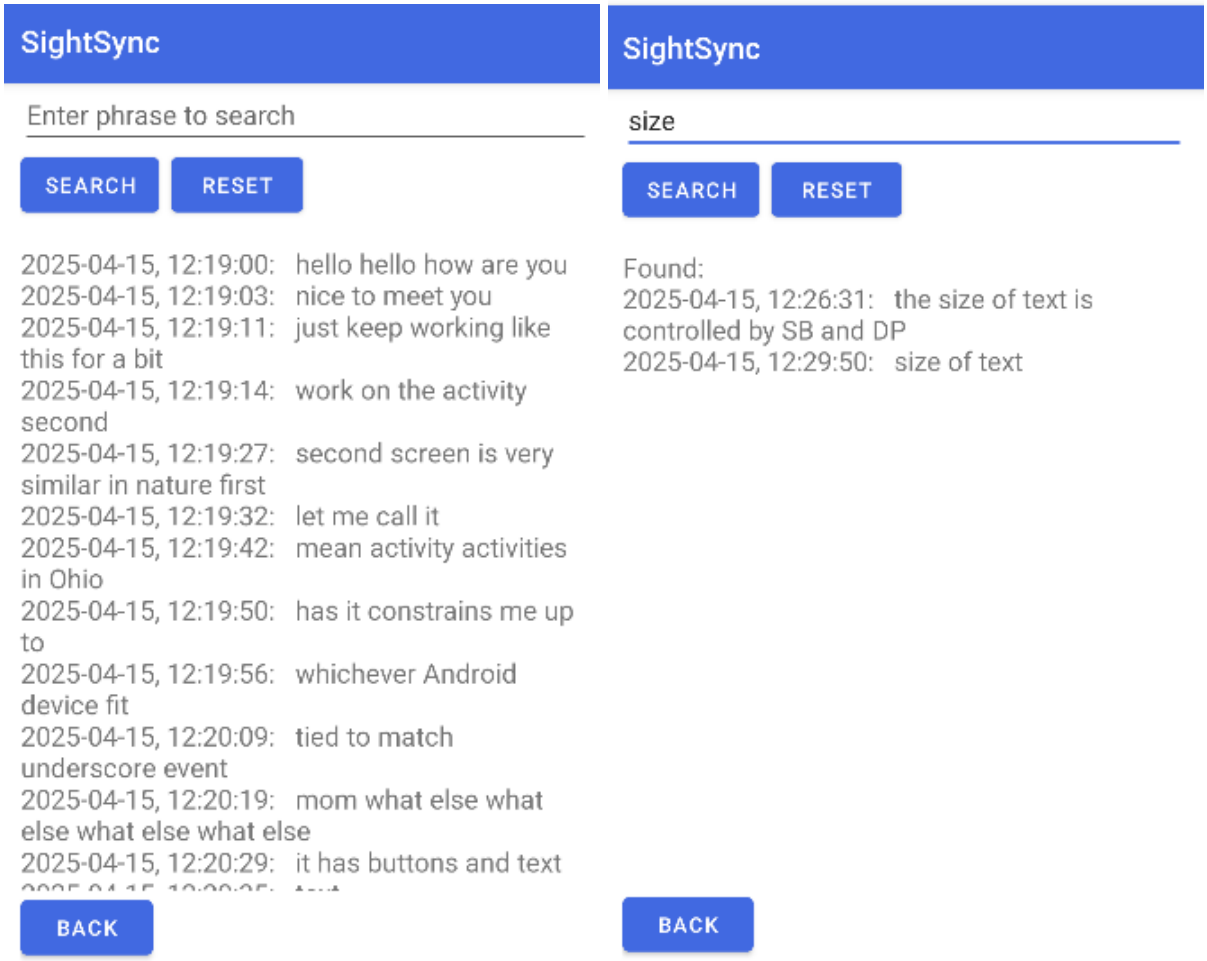


Figure 114: This is our transcription screen view.

Comprehensive Testing

Android Studio - Logcat Debugging

In Android Studio, there is a debugging feature, where one can use LogCat to throw errors, to see where we are within our code. They are written in the format of Log.d(TAG, "Begin Execution"); and have a variety of indicators, from Log.e for error, Log.d for debugging, Log.W for warning and Log.I for information. These messages appear in the LogCat view and allow the user to see all processes within our app. These lines of code are written by the user to allow them to know which parts of the code are being used, while we are using the app. Using a USB-C wire and the Android Studio development platform, we are able to see various logs from our code below to help in understanding what is occurring and what errors appear.

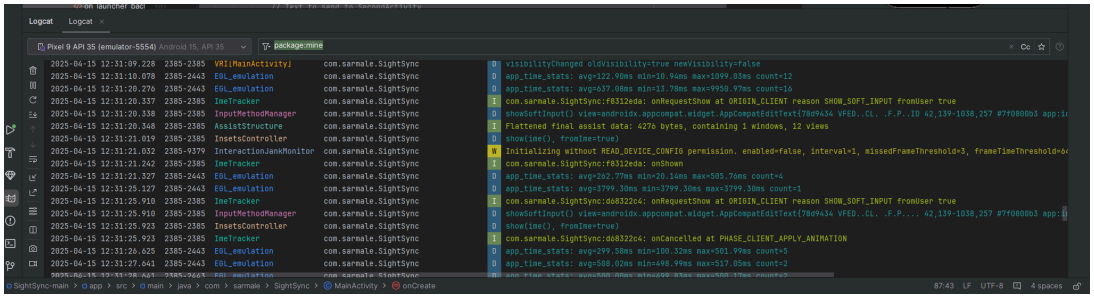


Figure 115: A screen capture of the LogCat debugger.

It was particularly helpful in trying to see what text was processed and what text was actually sent to the HC-05 bluetooth module. This was because we had come into an issue where text was not being sent to our microcontroller and we needed to understand whether the problem was on the smartphone app or the STM32's code for receiving data serially.

Functional Testing

The functional testing of our glasses involved testing all aspects of our project, from smartphone processing, communication between our devices, our projection system and our power management system. Once all these integral components were assembled, we had tested to see if text displays on our OLED screen from beginning to end of our final product. This was done utilizing different sentences of varying lengths and then analysing the results.

Table #: This is functional testing data of our complete prototype.

Sentences	# of Words	Trial 1	Trial 2	Trial 3	Total Accuracy
How are you	3	3	3	3	100.00%
How is the weather	4	4	4	4	100.00%
How are you today Laura	5	4	4	4	80.00%
How are you feeling today Ahmed	6	5	5	5	83.33%
The young boy ran towards the park	7	7	6	6	90.48%
They went to the beach and made food	8	8	7	8	95.83%
Today I need to work on my final project	9	9	9	8	96.30%
We will graduate soon after we complete this project today	10	9	9	10	93.33%
This bottle is blue and very cool and doesn't spill water	11	10	10	10	90.91%
Hello Ahmed did you finish your project today and did it work	12	12	10	12	94.44%
If this project does well we will do very well in this course	13	13	13	12	97.44%

The Average accuracy from our results had shown that our product was 92.91% accurate.

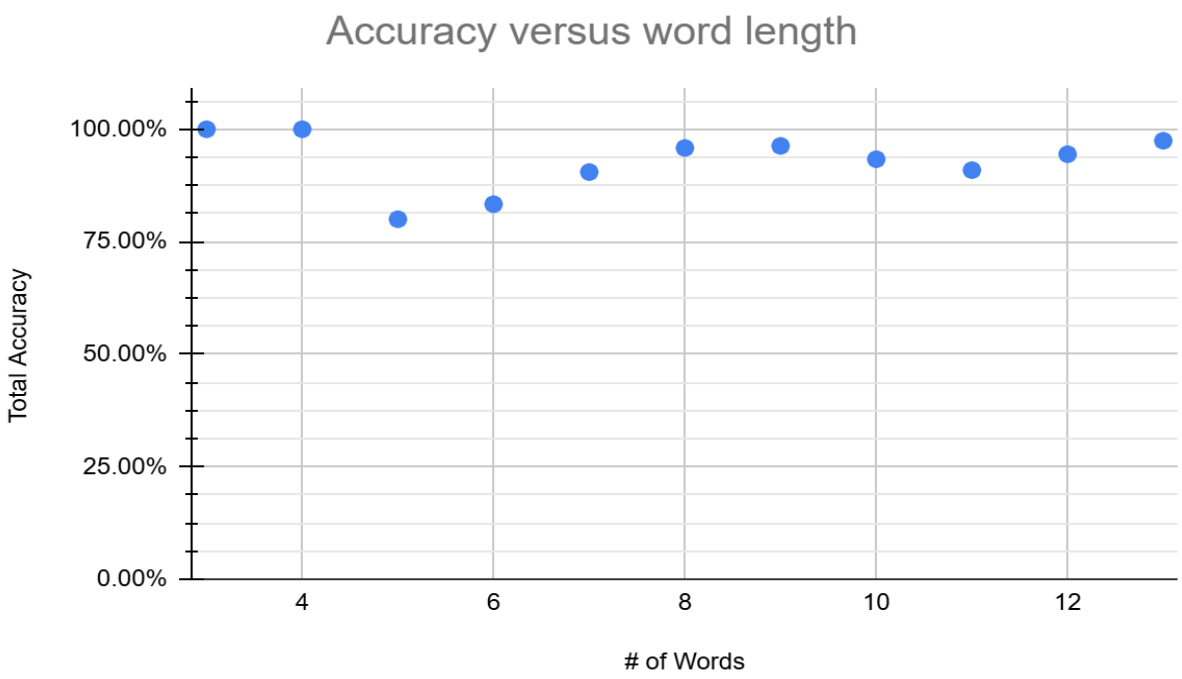


Figure 116: A scatter plot of the results of our glasses.

Our results show that the accuracy increases the longer a sentence is, due to the way in which accuracy is calculated. Accuracy was calculated by the number of words that are correctly displayed on the screen over the total number of words. A systematic error was seen, where the last word would often get missed and would not be sent. In shorter sentences, it reduces our accuracy a lot, however, in larger sentences, this turned out to not be as large of an issue. Another issue we had seen is when it comes to names, common names like Bob, Ahmed, Laura, or Sally are easily recognized, however, names such as Hashim, Makeish and Hira were not recognized at all. This is expected, as speech to text API's are Large Language Models and these models do not have much data, if any, on the more complex names. The last issue has to do with how quickly someone speaks. It is nearly 100% accurate if the user speaks slower, louder and clearer, in comparison to when one speaks too quickly or quietly resulting in incorrect speech to text conversion. Most words that were long, such as "intended" would end up transcribed as "in the end", which is an unavoidable issue due to the speech to text api used.

Critical Problems Solved [4.0]

Porting the OLED Driver Code for the STM32 Nucleo Family

We encountered several challenges while integrating the microcontroller (µC) with the OLED drivers. Initially, the OLED drivers were designed specifically for the STM32F103RBT6 µC. However, the µC used in this project was the STM32F042K6 Nucleo, necessitating the porting of the drivers to this particular model. When running the code prior to the porting process, approximately 143 errors were encountered, as detailed below:

```
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in5.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_0in96_rgb.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_0in95_rgb.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in5_b.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in3_c.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_0in96.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in51.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in54.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in27_rgb.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in5_rgb.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_2in42.o] Error 1
make: *** [Core/User/OLED/subdir.mk:61: Core/User/OLED/OLED_1in3.o] Error 1
"make -j20 all" terminated with exit code 2. Build might be incomplete.

16:47:58 Build Failed. 143 errors, 4 warnings. (took 1s.70ms)
```

Figure 117: Total Errors Encountered When Compiling

Many of these issues were related to incorrect GPIO pin initializations, improper linking of header files to the source code, and multiple declarations of functions. These problems were addressed by correctly linking the header files, removing duplicate declarations, and properly initializing the GPIO pins, as outlined below:

```
/* Private defines -----------------------------------------------------------*/
#define MCO_Pin GPIO_PIN_0
#define MCO_GPIO_Port GPIOF
#define OLED_CS_Pin GPIO_PIN_5
#define OLED_CS_GPIO_Port GPIOA
#define OLED_DC_Pin GPIO_PIN_0
#define OLED_DC_GPIO_Port GPIOB
#define OLED_RST_Pin GPIO_PIN_1
#define OLED_RST_GPIO_Port GPIOB
#define SWDIO_Pin GPIO_PIN_13
#define SWDIO_GPIO_Port GPIOA
#define SWCLK_Pin GPIO_PIN_14
#define SWCLK_GPIO_Port GPIOA
#define IIC_SCL_SOFT_Pin GPIO_PIN_4
#define IIC_SCL_SOFT_GPIO_Port GPIOB
#define IIC_SDA_SOFT_Pin GPIO_PIN_7
#define IIC_SDA_SOFT_GPIO_Port GPIOB

/* USER CODE BEGIN Private defines */
/* Private defines -----------------------------------------------------------*/
#define HC05_EN_PIN    GPIO_PIN_3
#define HC05_EN_PORT   GPIOA //A2
#define HC05_STATE_PIN GPIO_PIN_6
#define HC05_STATE_PORT GPIOA //A5
```

Figure 118: Correctly Initialized GPIO Ports

A major issue that was tricky to debug involved several key variables required for the OLED drivers to function, which were declared an extern in the header file. Since they were declared as extern, the compiler recognized that these variables were declared somewhere but did not allocate memory for them until they were defined. However, they were not defined anywhere in the code. This issue was resolved by defining the following variables in the main.c file:

```
SMBUS_HandleTypeDef hsmbus1;

SPI_HandleTypeDef hspi1;

UART_HandleTypeDef huart2;
```

Figure 119: hsmbus, spi, and uart handle variables

Another issue we encountered was that our µC could support only up to 32 KB of flash memory. However, due to the size of our driver files, we exceeded the available flash memory by 5,492 bytes, as shown below:

```
/arm-none-eabi/bin/ld.exe: region `FLASH' overflowed by 5492
```

Figure 120: Flash Overflow Compilation Error

This was resolved by enabling the -Os compiler optimization in the project configuration settings, which optimizes the code for size.

Establishing I2C Communication between the 1.51-inch Transparent OLED and STM32

Initially, we were using the 1.51-inch transparent OLED and encountered issues establishing SPI communication between the OLED and the STM32. After porting the drivers to work with the STM32 Nucleo, we expected the OLED to turn on and display the correct text (based on the OLED test function), but nothing appeared. While we initially suspected an issue with the code, it seemed correct. Consequently, we referred to the OLED μC manual, where we found the following information:

8.1.3 MCU Serial Interface (4-wire SPI)

The 4-wire serial interface consists of serial clock: SCLK, serial data: SDIN, D/C#, CS#. In 4-wire SPI mode, D0 acts as SCLK, D1 acts as SDIN. For the unused data pins, D2 should be left open. The pins from D3 to D7, E and R/W# (WR#)# can be connected to an external ground.

Table 8-4 : Control pins of 4-wire Serial interface

Function	E	R/W#	CS#	D/C#	D0
Write command	Tie LOW	Tie LOW	L	L	↑
Write data	Tie LOW	Tie LOW	L	H	↑

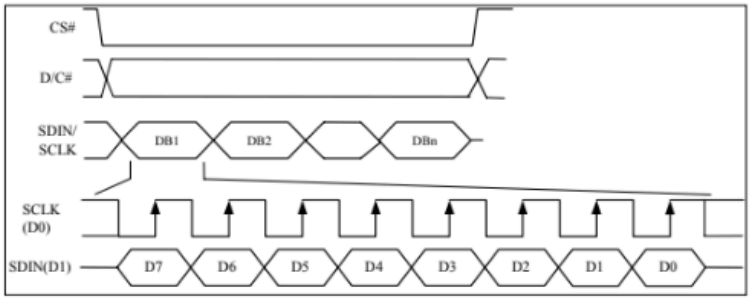


Figure 121: SPI Communication Protocol Datasheet for OLED

We realized that we hadn't correctly set the CS, DC, and RST pins in the GPIO initialization function. This was resolved by setting all the GPIO pin states to low, as shown below:

```c
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* USER CODE BEGIN MX_GPIO_Init_1 */
  /* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_3|OLED_CS_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, OLED_DC_Pin|OLED_RST_Pin|IIC_SCL_SOFT_Pin|IIC_SDA_SOFT_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pins : PA3 OLED_CS_Pin */
  GPIO_InitStruct.Pin = GPIO_PIN_3|OLED_CS_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pins : OLED_DC_Pin OLED_RST_Pin IIC_SCL_SOFT_Pin IIC_SDA_SOFT_Pin */
  GPIO_InitStruct.Pin = OLED_DC_Pin|OLED_RST_Pin|IIC_SCL_SOFT_Pin|IIC_SDA_SOFT_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* USER CODE BEGIN MX_GPIO_Init_2 */
  /* USER CODE END MX_GPIO_Init_2 */
}
```

Figure 122: STM32 GPIO Initialization Function

We then realized that the SPI communication configuration was also incorrect. The data size was set to 4 bits, with the first bit being the least significant bit. However, according to the OLED manual, the data size should be 8 bits, with the first bit as the most significant bit. The SPI settings were adjusted accordingly to resolve these issues:

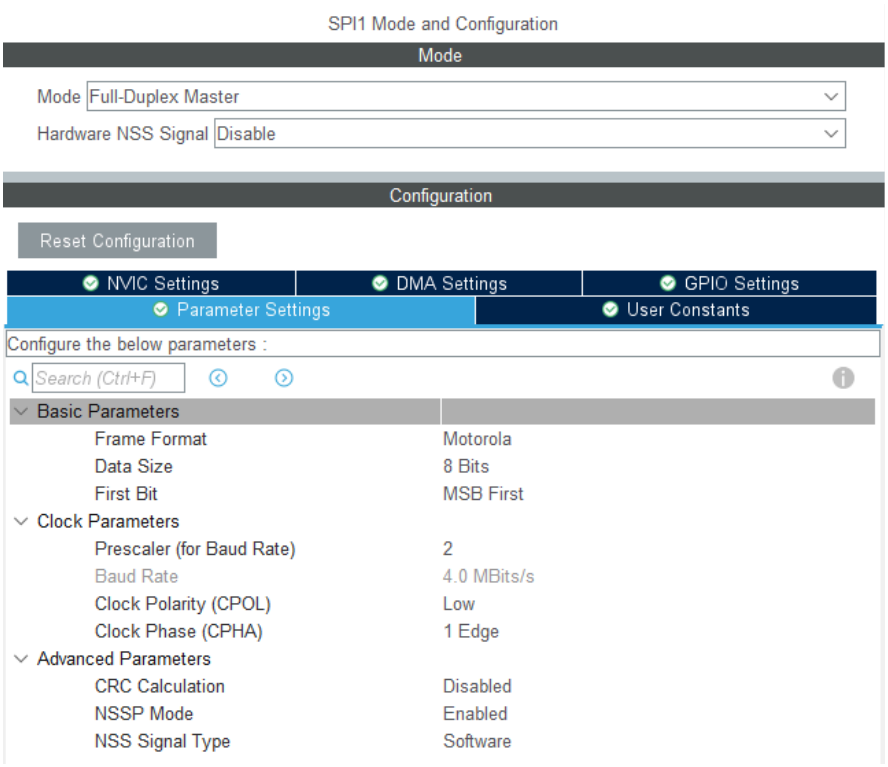


Figure 123: STM32CubeMX IDE SPI Configuration Settings

After adjusting the SPI communication configuration to the correct settings, the µC finally powered on, as shown below:

Figure 124: 1.51 Inch OLED Screen Successfully Turning On

Establishing Software I2C Communication between the 0.96-inch OLED and STM32

The 1.51-inch transparent OLED turned out to be too large for our project, so we decided to switch to the 0.96-inch OLED. However, this presented a new set of challenges. For hours, we tried to establish hardware I2C communication between the 0.96-inch OLED module and the STM32, only to discover that this wasn't possible. The driver for the OLED module only supports software I2C, as indicated in the code provided by the manufacturer, as shown below:

```
97  //  OLED_0in91_test();  //  Only IIC !!!
98
99  //  OLED_0in95_rgb_test();  //  Only SPI !!!
100
101 //  OLED_0in96_test();    //  IIC must USE_IIC_SOFT
102
103 //  OLED_lin3_test();   //  IIC must USE_IIC_SOFT
104
105 //  OLED_lin3_c_test();
106
107 //  OLED_lin5_test();
108
109 //  OLED_lin5_rgb_test(); //  Only SPI !!!
```

Figure 125: Commented Code Given to us my the OLED 0.96 Inch Screen Manufacturers

As shown, calling the OLED_0in96_test() function using I2C would not work because it requires the use of I2C_SOFT instead. This meant we had to modify a few files, as explained in the "**0.96-inch Waveshare OLED Display Overview**" section of this report. To reiterate, we made changes to several lines of code in the DEV_Config.h, Soft_IIC.h, and OLED_0in96.c files.

While making these changes, we uncovered something we're particularly proud of—a major bug introduced by actual engineers! This bug was found in the function shown below

```
static void OLED_0in96_WriteReg(uint8_t Reg)
{
#if USE_SPI_4W

#elif USE_IIC_SOFT
    iic_start();
    iic_write_byte(I2C_ADR << 1); // Changed code
    iic_wait_for_ack();
    iic_write_byte(0x00);
    iic_wait_for_ack();
    iic_write_byte(Reg);
    iic_wait_for_ack();
    iic_stop();
#endif
}
```

Figure 126: WriteReg function used to write data to the OLED Screen from the STM32 using software I2C

Originally, the iic_write_byte function had a hardcoded address for the OLED screen, but this is incorrect. It should use the address set by the user in the DEV_Config.h file, specifically the I2C_ADR variable, bit-shifted to the left by 1. This is a bug because the manufacturers instruct users to set the I2C_ADR to the desired address for communication. However, this doesn't work since the address in OLED_0in96_WriteReg is hardcoded. The issue was fixed by changing the address to I2C_ADR << 1, making it dynamic and allowing it to change based on the user's configuration settings.

Establishing I2S Communication between the ICS 43434 MEMS Microphone and STM32

This was a minor issue we encountered. For some reason, port PA5 on the STM32 was outputting 1.3V as high, which was not being recognized by the ICS 43434 microphone. As a result, it wasn't being properly driven with a clock. This port corresponds to I2S1_CK, which is the clock for the ICS 43434. The issue was resolved by switching to a different port for I2S1_CK, specifically port PB3.

Troubleshooting the HC-05 Bluetooth Module

When attempting to transmit audio data from the STM32 to the SightSync smartphone app via Bluetooth, we encountered an issue where nothing was being transmitted properly. The Bluetooth module would glitch and turn off, requiring us to unplug it and plug it back in to reset it. After various trials and errors, including code adjustments, we discovered that the HC-05 Bluetooth module can only transmit 5 times per second. This was determined by manually adding a 200ms delay in the I2S DMA callback function, as shown in the code below:

```
57 void HAL_I2S_RxCpltCallback(I2S_HandleTypeDef *hi2s) {
58     HAL_UART_Transmit(&huart1, (uint8_t*)i2sBuffer, BUFFER_SIZE * 4, HAL_MAX_DELAY);
59     for(int i = 0; i < 600000; i++);
70 }
71
```

Figure 127: HAL I2S DMA Callback Function which gets called when the buffer is full

This delay resulted in the following TX signal from the STM32 on the oscilloscope:

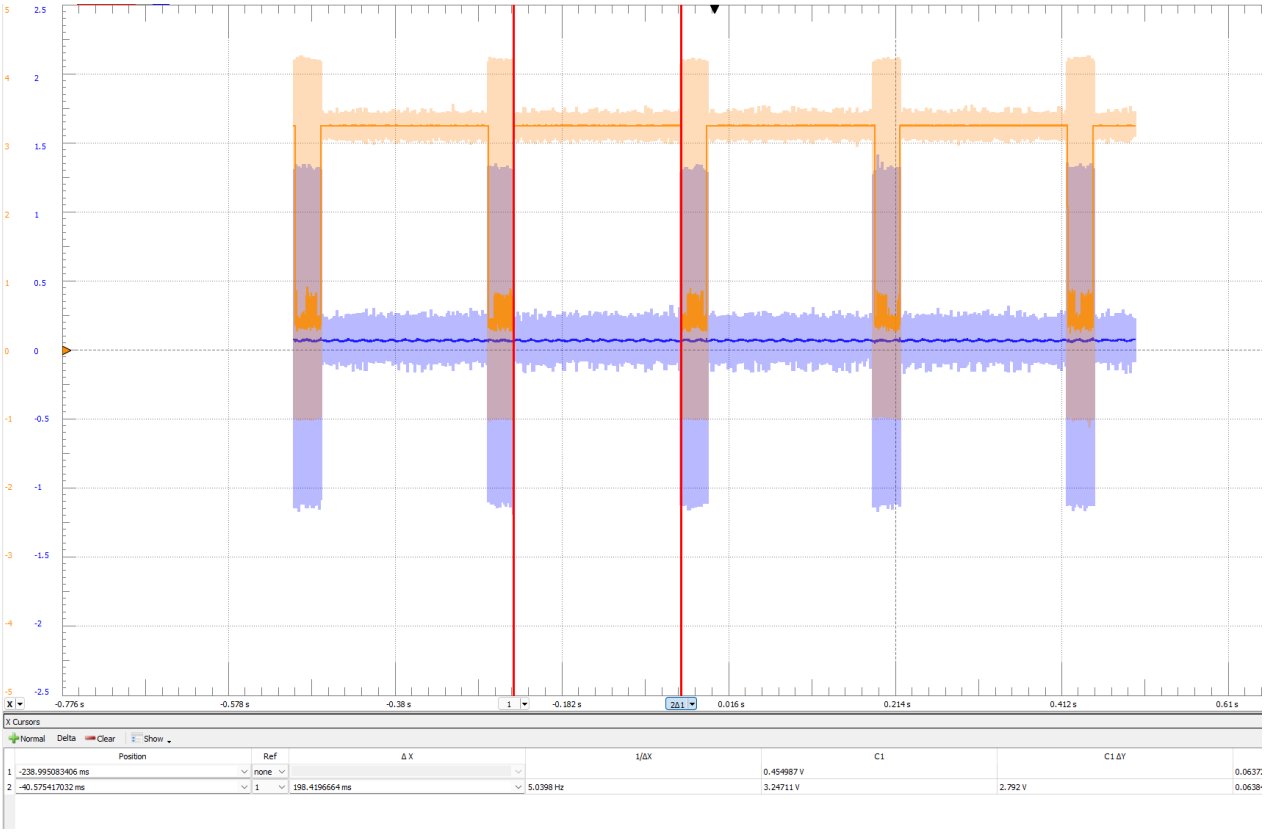


Figure 128: Analog Discovery 2 Oscilloscope Output

As shown, the delay occurs between the two red cursors (with the orange signal at 3.3V). Since there is a delay between each transmission and each transmission lasts around 50 ms, this means the STM32 is transmitting only about 200 ms of voice audio data per second, which is insufficient for speech recognition.

We tried several fixes, such as increasing the baud rate, but this didn't resolve the issue. We also attempted to increase the buffer size to reduce the required delay (allowing the audio buffer to take longer to fill up, which would introduce a natural delay that doesn't actually impact the audio), but we hit a physical limit due to the limited SRAM (only 6 KB). This means the issue couldn't be fixed unless we switched to a different Bluetooth module. Unfortunately, we were too far into the project to make such a drastic change, so we had to stick with the HC-05 Bluetooth module, which can only transmit 5 times per second. As a result, we decided to capture audio through the smartphone app instead of using the built-in MEMS mic.

One Bluetooth module we researched was the BM83 V2.0 from Microchip. If we had more time, we would have opted for this module. An image of the module is shown below:

Figure 129: BM83 Bluetooth Module

Firmware Issues Related to DMA

We faced many issues when trying to establish DMA. First, we attempted to use half-buffer callbacks, where functions would be called when the buffer is half full and when the buffer is completely full. This approach worked, but due to the Bluetooth issues discussed in the "Troubleshooting the HC-05 Bluetooth Module" section, we limited the callbacks to only trigger when the buffer is full to avoid transmitting too fast and overloading the HC-05.

After switching to full-buffer callbacks for DMA, we encountered another issue: I2S DMA seems to stop working if the buffer size exceeds 700. Although we're not entirely sure why, we identified 700 as the maximum buffer size for DMA and decided to stick with it.

Additionally, we ran into issues with the arguments passed into the `HAL_I2S_Receive_DMA` function, which was explained in the ICS 43434 microphone section.

We also faced difficulties with UART DMA, as it simply didn't work, and we couldn't pinpoint the cause. As a result, we decided to use classical UART transmissions instead, which execute CPU instructions rather than bypassing them like in DMA.

App Crashed after 5 minutes of use

The original revisions of our app had not been tested for long term use, thus, during the testing phase, it was discovered that the app would crash and the security features of our smartphone would automatically restart our device. In order to debug this issue, we had utilized LogCat, in which we had discovered there was a security issue related to the overuse of the speechRecognizer library, in which the Speech Recognizer was called frequently. Using two log statements on the onError() method `Log.e(TAG, "Speech recognition error: " + error);` and the onEndofSpeech() method `Log.d(TAG, "Speech input ended");`, we were able to identify that multiple times within 0.1 ms, the speech recognizer would be called multiple times. Essentially, when the speech recognizer would end, it would call upon the speech recognizer, but due to the logic not changing as quickly as needed, it would end up throwing an onEndofSpeech() prematurely and call the speech recognizer again. This created 1, 2, 4, 8, and then 16 instances of the speech recognizer, causing 2^{n-1} occurrences of the speech recognizer.

This was resolved by placing flags (`isStartListeningPending = true;`) that would prevent quick succession calls of the startListening() method and adding in a 5ms delay for the onEndofSpeech() and a 50ms delay on the onError() method. This resulted in our app no longer crashing and was tested to work past 45 minutes, without the phone showing any warnings, heating up or restarting itself due to its own security features.

Communication between Smartphone and STM32

Another key problem was connecting the smartphone to our STM32 and OLED screen. We were able to simulate activities using an Arduino and another HC-05 bluetooth module and had shown that it would work as intended.

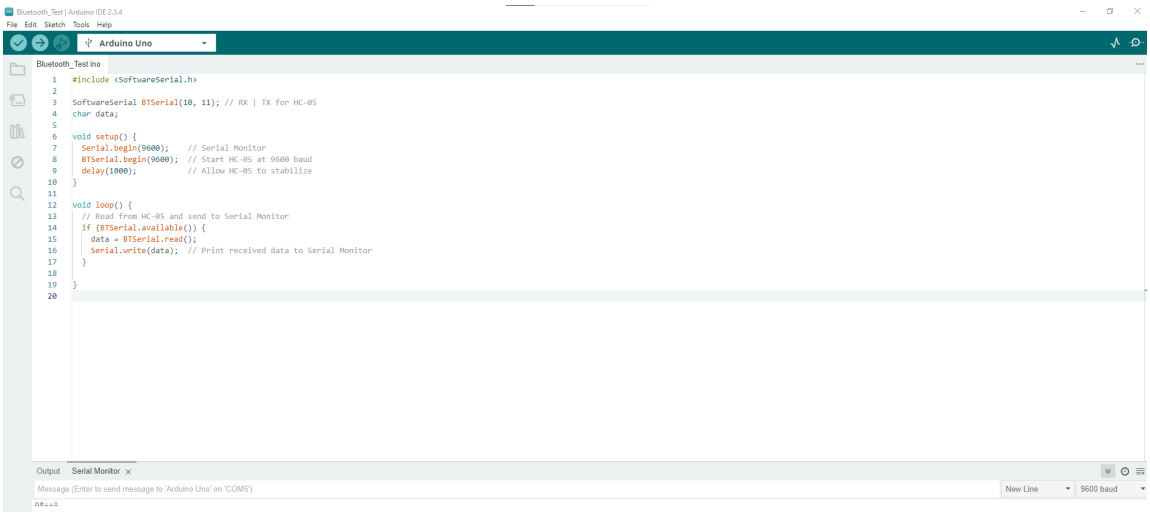


Figure 130: Screen capture of testing the communication using an Arduino

However, due to the differences in coding an Arduino versus a standard microcontroller, such as a STM32, we had difficulty sending data through bluetooth to our STM 32. It would send sometimes as null character boxes, sometimes it would not send, it would overflow and would freeze the OLED screen and it was not real-time.

App Development Side:

The changes from the app development side were to decrease the buffer, to have less amounts of data be sent for every time it sends data, to give the microcontroller more time to send the data to the OLED screen. This was done by decreasing the buffer from 128 bytes to 64 bytes and setting an internal delay of 250ms.

```
Thread.sleep(250);
```

This provided a short delay to prevent buffer overflow.

Another change was to send processed text in real-time, through partial transcription, instead of waiting till the speech recognition ended and sending the entire text at once. This not only utilized the threads better, having it be able to process speech and send text over bluetooth in parallel, but also allowed our product to reach a communication time between speech to our OLED screen to around 300 - 400ms, depending on the number of bytes in word. This was deemed to be a comfortable pace for the user to read the words from the screen.

STM32 Firmware Side:

The changes made to the STM firmware included increasing the buffer size to support longer words and implementing a communication protocol where a `\n` newline character indicates the end of a word. This helps the firmware identify word boundaries correctly. We also switched from blocking UART sockets to non-blocking sockets with very short timeouts. With low timeouts, the socket repeatedly checks its receive buffer, allowing it to continuously receive data without blocking incoming messages from the SightSync Android app. This approach worked—previously, words were not being displayed on the STM32, which we believed was due to socket blocking. Once we switched to a non-blocking setup using a do-while loop, the issue was resolved, and the words were displayed almost in real time (With a delay of around 0.4 seconds).

Conclusion [5.0]

SightSync has successfully demonstrated a functional wearable assistive device that bridges communication gaps for the deaf and hard-of-hearing by providing real-time speech-to-text transcription. The final prototype achieves 92.91% transcription accuracy with a low-latency

display of 300-400ms, surpassing initial performance targets. This was accomplished through a hardware-software co-design approach: the system integrates an STM32 microcontroller, and OLED display into an ergonomic 3D-printed frame, while leveraging Google's Speech-to-Text API via Bluetooth. Key achievements include the development of a compact optical projection system, a modular frame design, and seamless integration of off-the-shelf components with custom PCBs, and an Android app that leverages Google's Speech-to-Text API for reliable transcription.

User testing revealed two critical insights: first, the hands-free design significantly reduced the cognitive load compared to smartphone-dependent alternatives, and second, the 10-second auto-clear functionality proved essential for maintaining situational awareness. While the current reliance on smartphone processing diverges from the original vision of a fully self-contained device, this compromise enabled rapid iteration and higher accuracy. The project's success lies in its ability to deliver a practical, discreet, hands-free communication tool that empowers users without requiring external devices beyond a paired smartphone. SightSync represents a meaningful step toward accessible, discreet assistive technology, with a strong foundation for future commercialization.

Figure 131: Hashim wearing the SightSync Captioning Glasses

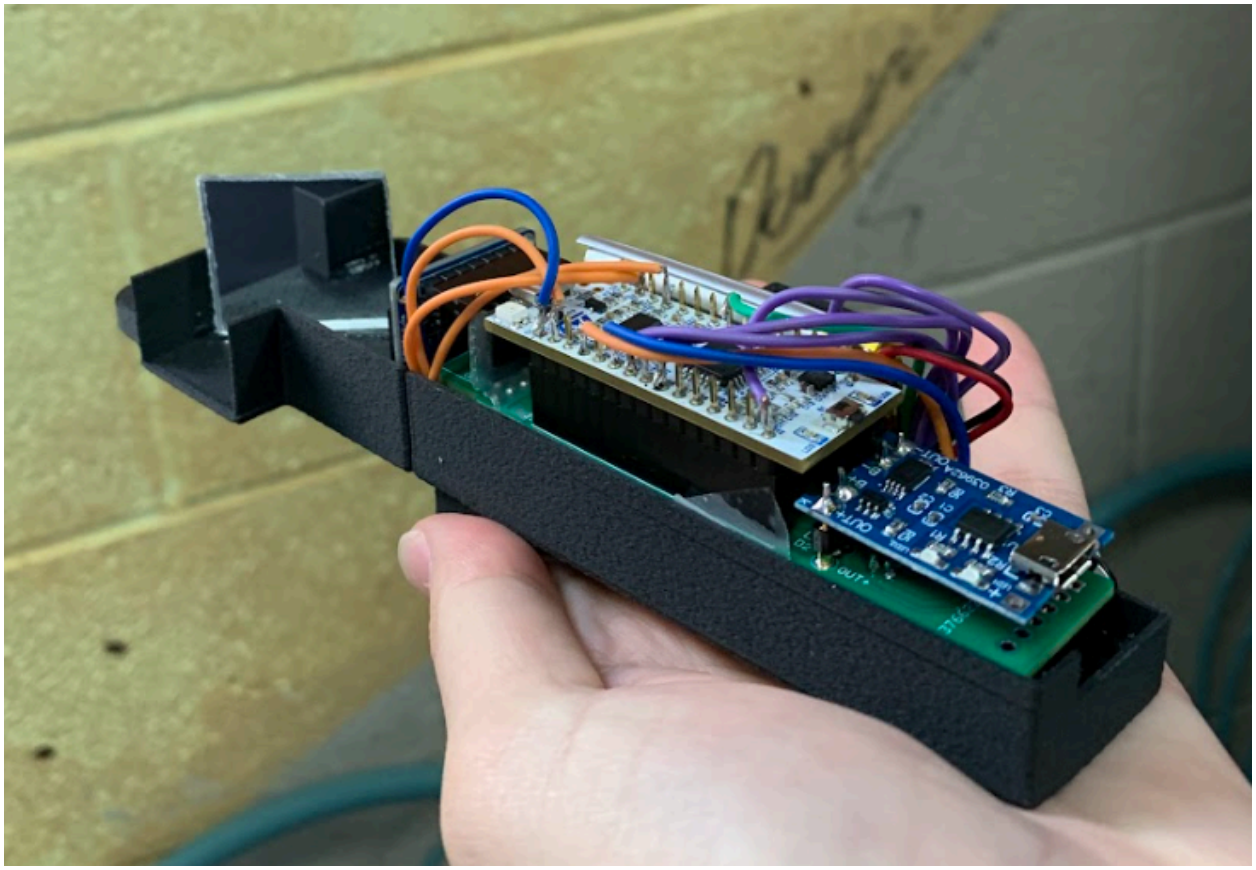


Figure 132: Our SightSync System stored within our housing compartment

Future Plan [6.0]

To transition SightSync from a proof-of-concept to a market-ready product, three parallel development tracks are proposed. Technically, the highest priority is eliminating smartphone dependency by migrating to an ESP32-S3 microcontroller with on-device edge-based speech recognition, which would enable offline operation while reducing latency. Concurrently, the optical system should be upgraded to waveguide-based AR displays to expand the field of view beyond the current projection, while reducing bulk. For enhanced usability, implementing multi-speaker differentiation through voice fingerprinting algorithms would address a key limitation identified during group testing scenarios.

Commercialization efforts will require strategic partnerships with accessibility organizations to conduct large-scale beta testing. Cost reduction will focus on design-for-manufacturing principles—replacing 3D-printed frames with injection-molded components and transitioning from custom PCBs to a system-on-module architecture could reduce unit costs to approximately $200 price point that is competitive against alternatives like XRAI Glasses.

Long-term research directions include AI personalization for accent adaptation and exploring energy-harvesting techniques such as flexible solar cells integrated into the frame. The team also envisions expanding language support through a cloud-based translation layer, which could position SightSync as a tool for both accessibility and cross-linguistic communication. These initiatives will be guided by continued engagement with the deaf and hard-of-hearing community to ensure technological advancements translate into genuine quality-of-life improvements.

Individual Self-Reflection and Assessment [7.0]

Overall Evaluation [Ahmed] - Excellent - Total Contribution 25%

Contribution Table - Ahmed

Task	Description	Percentage Contribution
Brainstorm Ideas	I recommended that the group pivot from the original braille machine concept to using smart glasses to caption the user's voice. While the initial idea was for real-time translation, transcription for deaf and hard-of-hearing users seemed like a more impactful application.	20%
Proposal	Completed my section on **Deliverables**, outlining all project milestones along with their **bronze**, **silver**, and **gold** achievement levels.	20%
Report A	Completed my sections on **hardware and firmware**, which included porting OLED driver code to the STM32, selecting a suitable IDE for STM32 development, learning to use the ST-LINK debugger, and establishing I2C communication between the OLED and STM32. I also documented all challenges encountered throughout these tasks.	20%
Report B	Similar to Report A, I was responsible for integrating the hardware components and writing the firmware to ensure that all parts of the system worked together smoothly.	20%
Research on Speech to text API	Researched various APIs including OpenAI's Whisper, Google's Speech-to-Text API, Vosk's speech-to-text API, and Google's cloud-based Speech-to-Text API.	20%
Research on Bluetooth Modules	Researched various Bluetooth modules that support Bluetooth Classic 2.0 and are capable of data transmission, including the HC-05, HC-06, and BM83S.	30%
Research on Microcontrollers	Researched various microcontrollers including the Arduino Nano, Raspberry Pi Pico, ESP32, and STM32. Compared the pros and cons of each and ultimately chose the STM32.	80%
Research on Mics	Researched various microphones and their benefits, including both analog and digital options, and eventually	50%

	decided on the ICS 43434 microphone.	
Research on OLEDS	Researched different OLED displays with the group, but eventually, Hashim and I took the lead and found the most suitable option—the 0.96-inch OLED.	40%
Writing Firmware to control OLED	Wrote the firmware for the STM32 to control the OLED screen. This included initializing software I2C for communication with the STM32, setting up the correct GPIO ports, fixing bugs in the OLED drivers, adapting the OLED drivers to work with the STM32, and physically soldering resistors off the OLED hardware. I also wrote the code to receive words and display them on the screen. Essentially, I handled all the tasks related to controlling the OLED from the STM32.	100%
Writing Firmware to Control Bluetooth Module	Wrote firmware to control the HC05 Bluetooth module. This involved initializing UART for communication with the STM32, configuring GPIO ports for several HC05 pins, and implementing UART functions to transmit audio data and receive text data. Essentially, I handled all tasks related to controlling the Bluetooth module from the STM32.	100%
Writing Firmware to Control Mic	Wrote firmware to control the ICS 43434 microphone. This included establishing I2S communication by enabling I2S and configuring the necessary registers. I also set up DMA to automatically trigger callback functions when the audio buffer was full. Additionally, I implemented firmware to transmit the audio data to the HC05 during the I2S DMA callback functions. Essentially, I handled all tasks related to controlling the ICS 43434 microphone from the STM32.	100%
Writing Firmware for STM32	Overall, I wrote all the firmware for the STM32, which included configuring I2S, software I2C, UART, communication lines, GPIO ports, DMA, PLL values, and clock configurations. I was also responsible for understanding and using STM32CubeIDE. Additionally, I wrote the firmware to control all the peripherals (mic, OLED, Bluetooth) through the STM32.	100%
Functional Testing: Testing the Frame	I helped my group members in testing by holding the lens when they shined a light through it. I also assisted with troubleshooting, ensuring everything was aligned properly for accurate testing.	20%

Functional Testing: Testing the Complete Project	I completed functional testing to ensure that our app worked seamlessly with the entire project. I tested the entire flow, starting from the STM32 side by connecting to Bluetooth and receiving data. Then, I verified that the data was correctly displayed on the OLED screen, and finally, I checked to see if it was reflected on the mirror as intended.	50%
Capstone Video: Recording	I was featured in one of the scenes of the capstone video recordings, which ended up becoming a blooper. I also had the opportunity to explain the high-level system design of our project, where I discussed how everything worked on the hardware and firmware side.	20%
Capstone Video: Editing	I was responsible for a decent amount of editing on the videos and adding the finishing touches.	20%
Final Report	I was responsible for the "System Hardware & Firmware" section, which included the "Adafruit ICS 43434 MEMS Microphone Overview," "0.96-inch Waveshare OLED Display Driver Overview," "Robojax HC-05 Bluetooth Module Overview," and "STM32 F042K6 Nucleo Microcontroller Overview." I also handled the parts in the "Critical Problems Solved" section related to the OLED, STM32, Microphone, and Bluetooth modules, along with their corresponding firmware.	20%

Overall Evaluation [Hashim] - Excellent - Total Contribution 15%

Contribution Table - Hashim

Task	Description	Percentage Contribution
Brainstorm Ideas	Initial meetings with Makeish and Laura to brainstorm ideas.	20%
Proposal	Conducted further research on the project topic with a focus on assumptions and risks.	20%
Report A	Completed my section for the physical implementation of the device along with the hardware components list section.	20%
Report B	Completed my section for the physical implementation of the device with a focus on new research and an alternative design form factor for the device.	20%
Research on OLEDS	Researched different OLED displays with the group, but eventually, Ahmed and I took the lead and found the most suitable option—the 0.96-inch OLED.	40%
Frame Design	Used Autodesk Fusion360 to design multiple variations of the frame from scratch. The designs were tested and iterated after each print.	100%
Research on Lenses	I conducted research on types of lenses online, and through discussions with Dr. Li. I performed calculations to determine which lens specifications would work best for our project and placed an order for one from Newport Corporation.	50%
Research on Projection System	Researched mirror types (glass, acrylic, front surface, back surface) that would work best for our use case. I also came across different projection methods that I then looked into to narrow our design options down to just two potential solutions.	30%
Cutting Acrylic Mirror	Along with Makeish, I cut an acrylic mirror to various dimensions to fit within the frame and within the mirror holder.	50%
Functional Testing: Testing	I was involved in testing the frame through projection testing with a light source and experimentation with	25%

the Frame	various lenses. I also focused on component packaging and measurements for any changes in the frame design.	
Functional Testing: Testing the Complete Project	I was heavily involved in the final packaging of the device. I also assisted in testing to make sure that the device was functioning correctly and made final alterations.	20%
Capstone Video: Recording	I was acting in most scenes of the video and helped set up the scenes.	20%
Capstone Video: Editing	Collaborated with the rest of the group on editing the video. Recorded and added a robotic voice to add to a scene.	20%
Final Report	Completed my section for the physical design of the device.	20%

Overall Evaluation [Hira] - Excellent - Total Contribution 15%

Contribution Table - Hira

Task	Description	Percentage Contribution
Brainstorm Ideas	Contributed to the conceptual development of the assistive AI-Glasses system, and consensus toward the final project proposal.	20%
Proposal	Completed my section on the project description, clearly defining the communication accessibility problem, our technical solution, and long-term societal impact.	20%
Report A	Completed sections on projects key objectives and progress overview, establishing measurable success criteria for development phases.	20%
Report B	Outlined predictions on final achievements and deliverables, validating achievement benchmarks against the initial project goals.	20%
Initial Research on Microcontroller and OLED	Researched various microcontrollers and OLED displays, conducting beginning initial testing with Raspberry Pi 4 to display text with Waveshare 1.51” screen.	30%
Research on Lenses	Conducted research on convex vs plano-convex lenses, and collimation of light from various sources.	15%
Research on Projection System	Research projection system designs and optical configurations to optimize text visibility within the users field of vision.	25%
Functional Testing: Testing the Frame/ Projection System	Assisted in testing the OLED screen with our lenses and prototypes. Conducting iterative optical testing to identify adjustments to improve clarity, reduce glare, and enhance mechanical alignment to enhance readability.	25%
Functional Testing: Testing the Complete Project	Assisting in validating system performance, implementing critical refinements such as improving readability through adjustments to lens system, correcting projection alignment in users view, and minimizing reflections and distortions during projection.	20%

Capstone Poster Design	Selected and organized project visuals from all project phases, including prototyping images and technical diagrams, to create an engaging and informative poster.	50%
Capstone Video: Recording	Co-directed and appeared in the demonstration video while coordinating with the team in setting up scenes.	20%
Capstone Video: Editing	Established the video's narrative structure during initial editing, ensuring alignment with our planned script and technical demonstration goals.	20%
Final Report	Completed the conclusion and future plans section, analyzing project achievements and outlining actionable next steps for technical and commercial development.	20%

Overall Evaluation [Laura] - Excellent - Total Contribution 20%

Contribution Table - Laura

Task	Description	Percentage Contribution
Brainstorm Ideas	Contributed to idea generation, decision matrices, and meetings with the professors, all as a group to come to a final project idea.	20%
Proposal	Completed my section on the project description, clearly defining the communication accessibility problem, our technical solution, and long-term societal impact.	20%
Report A	Completed sections on projects key objectives and progress overview, establishing measurable success criteria for development phases.	20%
Report B	Completed my section for the electrical implementation of the device with a focus on power delivery system and visual slide organization.	20%
Early Research: Microphone	Researched different types of microphones that could surface mount to a PCB, pick up frequencies in our desired range, and have good noise suppression, while being cheap and compatible with our microcontroller.	25%
Early Research: Lens Manufacturing	Researched lens manufacturers who could manufacture custom lenses, reached out and looked into acrylic printing options.	20%
Early Research: Projection system	Researched possible projection systems that would be more feasible than our original design. Came across youtube videos and instructables that offered a much cheaper way to project text into a user's vision.	25%
System Circuitry: Research	Many months of research went into investigating battery protection circuits, integrated chips that could be used to monitor battery charge level, temperature, charge speed, batteries that could be used, how to step up and step down voltages to match core component power requirements.	100%
System Circuitry: PCB Part Sourcing	Spent another couple of months adapting the circuit with different ICs, capacitors, resistors, LEDs by looking through datasheets and ensuring the power requirements are met without having too much current, causing temp	100%

	rises and damage. This also includes finding the proper headers for each core component that would mount to the PCB (OLED, bluetooth, mic, charging IC, STM, battery connector, manual ON/OFF switch), and ensuring they were the proper pitch, orientation, and angle to fit within our frames and seat them properly without misconnections.	
System Circuitry: PCB Design	Designed all wiring schematics and imported to PCB layout. Designed 2D PCB layout while considering the 3D space to package our frames as small as possible for user comfortability.	100%
System Circuitry: PCB Parts and Board Procurement	Ordered all components used on the PCB as well as the PCB itself from JLCPCB. Reached out to tens of local PCB manufacturers for quotes and lead times, which ultimately resulted too costly as a backup plan in case the online shop order did not arrive in time.	100%
Building prototypes: printing	Facilitated the printing of our 10 prototype prints with access to 3D Prusa printers at work.	100%
Functional Testing: Testing the Frame/ Projection System	Assisted in testing the OLED screen with our lenses and prototypes. Conducting iterative optical testing to identify adjustments to improve clarity, reduce glare, and enhance mechanical alignment to enhance readability.	25%
Functional Testing: Testing the Complete Project	Assisting in validating system performance, implementing critical refinements such as improving readability through adjustments to lens system, correcting projection alignment in users view, and minimizing reflections and distortions during projection.	20%
Capstone Poster	Collaborated with Hira to complete the poster for our capstone. Printed the 4ftx3ft poster at the Burlington library, along with the QR code for the capstone video.	50%
Capstone Video: Story and Recording	Outlined all scenes with Makeish before filming. Co-directed and appeared in video while coordinating with the team in setting up scenes.	20%
Capstone Video: Editing	Edited recorded scenes in Canva, ensuring alignment with our planned script, narrative, and technical goals.	40%

Final Report	Completed the introduction, system circuitry, and my own individual self reflection sections. General grammar and sentence structure review.	20%

Overall Evaluation [Makeish] - Excellent - Total Contribution 25%

Contribution Table - Makeish

Task	Description	Percentage Contribution
Brainstorming Ideas	Initial meetings with Laura and Hashim on deciding project topics.	20%
Proposal	Completed my section on Deliverables, detailing all milestones and their bronze, silver and gold level	20%
Report A	Completed my sections on User Interface, Backend Development, Bluetooth Communication, Speech To Text Conversion and the challenges I had faced.	20%
Report B	Similar to Report A, I was responsible for the App Development, bluetooth communication and the speech to text recognition aspects.	20%
Research on Speech to text API	Researched various different API's, Open AI's Whisper, Google's speech to text API, Vosk speech to text API and Google's speech to text cloud based API.	80%
Research on App Development Platforms	Researched various methods to create an app, google colab research, Kivy, Flutter, pycharm, Android Studio, web based applications.	90%
Research on Bluetooth Modules	Researched various different bluetooth modules that can use Bluetooth Classic 2.0, that would be able to transmit data, such as the HC-05, HC-06, and BM83S.	50%
Research on Projection System	Researched various instructables and youtube videos to find a working projection system that could house a small OLED screen.	25%
Researched lenses	Researched plano-convex lenses on edmund optics and looks at reading glasses at 3.5+ strength.	25%
Research on Microcontrollers	Primarily researched Arduino's and utilized an Arduino Uno SMRAZA to conduct testing of the app, while the STM 32 was being worked on by Ahmed.	20%
App Development:	Created the speech recognizer, multi-threaded programming for bluetooth communication, error handling,	100%

Backend	bluetooth permission checking and transcription capabilities. Coded the app side of sending data over bluetooth to the HC-05 Bluetooth Module	
App Development: User Interface	Created the two different screens, all buttons, textViews, ImageButtons, Scrollable textViews and editable textviews. Connected the backend and front end together	100%
Functional Testing: Testing the Frame	I helped bring in materials like a kleenex box and held up the light source, to see if we were able to collimate light. I also assisted with seeing if the light would be bright enough to reflect off an acrylic pane.	20%
Cutting Acrylic Mirror	Using the maker space studio at McMaster University, I had cut various sized rectangles from an acrylic based mirror to be used in our projection system.	50%
Functional Testing: Testing the Complete Project	I had completed functional testing to see if our app would work with our complete project. I had tested it from the app side, connecting to bluetooth and sending data, to then be checked to see if it was displayed on the OLED screen and then the mirror.	25%
Analyzing Results	I had taken our product to analyze how accurate the bluetooth data communication was and to see how fast the real-time speech to text conversion was.	100%
Capstone Video: StoryBoard	I, along with Laura, had created a set of 7 scenes that were used to create our video. I had created a high level of what would happen with some key phrases.	50%
Capstone Video: Recording	I had recorded scenes 4 and 6 and was a part of the first 3 scenes in our video. I was also the one to explain how the app worked in the project in the video and submitted the video on Mac Video.	20%
Capstone Poster	I had gathered key terms, phrases and images from testing, my app and past reports to provide to our team for the poster. I also did research on the cost of printing the poster at Staples and at MPS.	10%
Final Report	I was responsible for the app development sections, from design, creating, testing and problem solving	20%

References [8.0]

1. A. Alduuyher, “SightSync AI Glasses Capstone Project,” GitHub. [Online]. Available: https://github.com/ahmedalduuyher/SightSync-AI-Glasses-Capstone-Project-. [Accessed: Apr. 21, 2025].

2. AlainsProjects, “Arduino Data Glasses for My Multimeter,” *Instructables*. [Online]. Available: https://www.instructables.com/Arduino-Data-Glasses-for-My-Multimeter/. [Accessed: Apr. 21, 2025].

3. MKS Instruments, “KPX079AR.14 Plano-Convex Lens,” *Newport*. [Online]. Available: https://www.newport.com/p/KPX079AR.14. [Accessed: Apr. 21, 2025].

4. STMicroelectronics, *UM1956: STM32 Nucleo-32 Boards (MB1180) User Manual*, Rev. 6, Mar. 2025. [Online]. Available: https://www.st.com/resource/en/user_manual/um1956-stm32-nucleo32-boards-mb1180-stmicroelectronics.pdf. [Accessed: Apr. 21, 2025].

5. Adafruit Industries, “Adafruit I2S MEMS Microphone Breakout - ICS-43434,” *Adafruit*, Product ID: 6049. [Online]. Available: https://www.adafruit.com/product/6049#technical-details. [Accessed: Apr. 21, 2025].

6. Waveshare, “0.96inch OLED Module,” *Waveshare Wiki*. [Online]. Available: https://www.waveshare.com/wiki/0.96inch_OLED_Module#Overview. [Accessed: Apr. 21, 2025].

7. Components101, “HC-05 Bluetooth Module Pinout, Specifications, Default Settings, Replacements & Datasheet,” *Components101*. [Online]. Available: https://components101.com/wireless/hc-05-bluetooth-module. [Accessed: Apr. 21, 2025].

8. The-Frugal-Engineer, “ArduinoBTExamplev3,” GitHub. [Online]. Available: https://github.com/The-Frugal-Engineer/ArduinoBTExamplev3.git. [Accessed: Apr. 21, 2025].

9. Gironbel, “SightSync,” GitHub. [Online]. Available: https://github.com/gironbel/SightSync.git. [Accessed: Apr. 21, 2025].

10. D. Nedelkovski, “Arduino and HC-05 Bluetooth Module Complete Tutorial,” *HowToMechatronics*. [Online]. Available: https://howtomechatronics.com/tutorials/arduino/arduino-and-hc-05-bluetooth-module-tutorial/. [Accessed: Apr. 21, 2025].

11. STMicroelectronics, “NUCLEO-F042K6 - STM32 Nucleo-32 Development Board with STM32F042K6 MCU, Supports Arduino Nano Connectivity,” *STMicroelectronics*.

[Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-f042k6.html. [Accessed: Apr. 21, 2025].