Translator

Final Report

Table of Contents

Introduction

The goal of the project is to develop a *minimum* viable prototype of a real-time translation system using an ESP32 microcontroller and a host PC. Spoken audio is captured via a microphone module connected to the ESP32 board, sampled through its onboard ADC, and transmitted to the PC over Bluetooth Serial (SPP). A PC-side application receives and processes the audio stream, relying on APIs for converting speech to translated text. The PC then sends the translated text to the ESP32, which it displays on a connected LCD.
The project prioritizes core functionality: Hardware setup, audio capture and sampling, Bluetooth communication, and the LCD display. Any PC-side processing were implemented using existing libraries and models.

Every project component was first implemented and tested independently. Only at the end were they all integrated into the final working prototype.

Project

Components

- Lenovo IdeaPad 5 15IIL05 (Windows PC)
- DOIT ESP32 DEVKIT V1 (Uses ESP32 chip)
  Chosen over other ESP boards for Bluetooth Classic & Continuous Sampling support
- MAX9814
- I2C 20x4 LCD Display Module, with probably a PCF8574 I2C backpack
- A battery pack with VCC & GND "pins"

Programming the ESP32

A program can be uploaded to the ESP32 over USB using the Arduino IDE. The ESP32 supports C/C++, but I used .ino sketches because of familiarity.

At first, the ESP32 was not detected by Windows:

- [Arduino IDE]
  Tools > Port (No port listed)
- [Device Manager]
  Other Devices > CP2102 UWB to UART Bridge Controller (Drivers for this device are not installed. Code 28)

This was solved by installing the **CP210x Universal Window Driver** (v11.4.0) from https://www.silabs.com/developer-tools/usb-to-uart-bridge-vcp-drivers?tab=downloads.

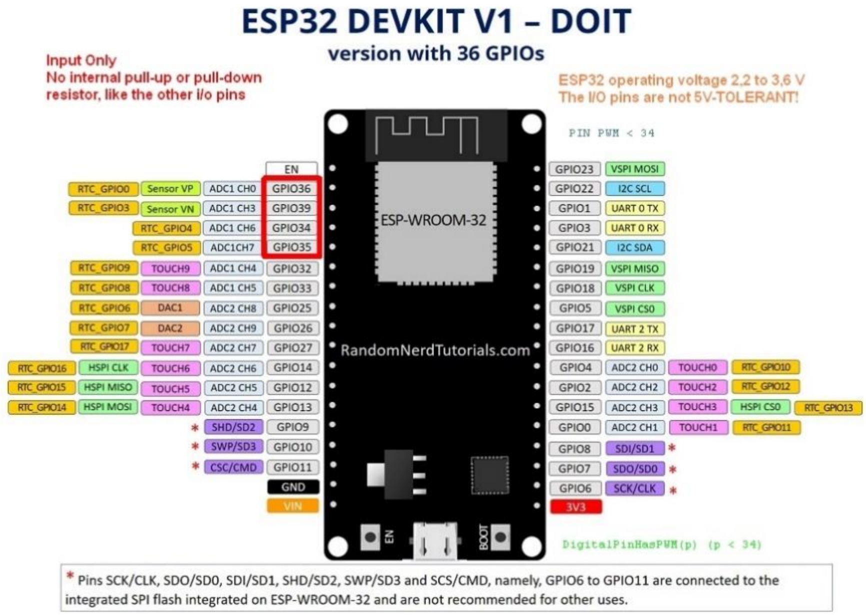
Audio Sampling

MAX9814 - ESP32 Wiring

- VCC – 3.3 V
- GND – GND
- OUT – GPIO35 (ADC1_CH7)

When the audio was replayed on the Windows PC, the volume was normal (a little low, but no background noise) expect for occasional spikes of loud white noise. This was corrected by converting the 12-bits back to 16-bit before transmission.

Setup



[DOIT ESP32 DEVKIT V1 Pinout]

The ESP32 chip has two ADC (Analog-to-Digital) units that support multiple input channels through multiplexing (the ADC has one physical converter, but the MUX can quickly switch to the input pin that is being read at the time).

- **ADC1** (8 channels, GPIOs 32-39)
  - On most ESPs, only ADC1 is available for I2S-based sampling.

- **ADC2** (10 channels, GPIOs 0, 2, 4, 12-15, 25-27)
  - ADC2 is used by the Wi-Fi driver, so it can't be used if the Wi-Fi driver is started. (Not an issue in this project, which uses Bluetooth and not Wi-Fi).

The ADC units have a 12-bit resolution (values 0-4095) but can also be configured to have lower resolutions (9/10/11-bit) for faster conversions, less noise sensitivity and lower power usage, on the downside of lower range and precision.

The SAR (Successive Approximation Register) ADC works like a binary search to find the digital value that best represents the input voltage by (1) comparing the input voltage against a known reference, (2) adjusting the estimate bit-by-bit based on whether the input is higher/lower, (3) until after 12-steps (in 12-bit mode), it settles on a final digital value from 0-4095.

```
adc1_config_width(ADC_WIDTH_BIT_12);  // I used 12-bit resolution (0 – 4095)
```

The ADC isn't very accurate, and calibration is required to handle:

- **High input voltages**: The ADC's default input range is ~0 – 1.1V (tied to the reference voltage). If the mic input has a higher range, the [ESP32 SoC has the following attenuation signals] to scale down the signal before it reaches the ADC:
  - 0 dB: ~ 0.10 – 0.95 V
  - 2.5 dB: ~ 0.10 – 1.25 V
  - 6 dB: ~ 0.15 – 1.75 V
  - 11 dB: ~ 0.15 – 2.45 V

```
adc1_config_channel_atten(ADC1_CHANNEL_7, ADC_ATTEN_DB_11); // Range 150-2450 mV
```

While higher attenuation allows the ADC to measure higher voltages, it lowers the sensitivity to smaller signals (lower sounds can be tuned out). Since my volume in the replay was very quiet, I plan to test lower attenuation levels next.

- **Reference voltage drift**: The [internal reference voltage (1.1V, can be up to 1.2V on some ESPs)]() is generated on-chip and isn't stable (can vary due to small manufacturing differences, supply voltage, or even temperature changes). If it isn't calibrated, the ESP32 might report different values for the same analog voltage. The ESP32 has built-in calibration functions provided in the ESP-IDF (Espressif IoT Dev. Framework) to handle this.

ESP32 Digital SAR ADC Controller Architecture

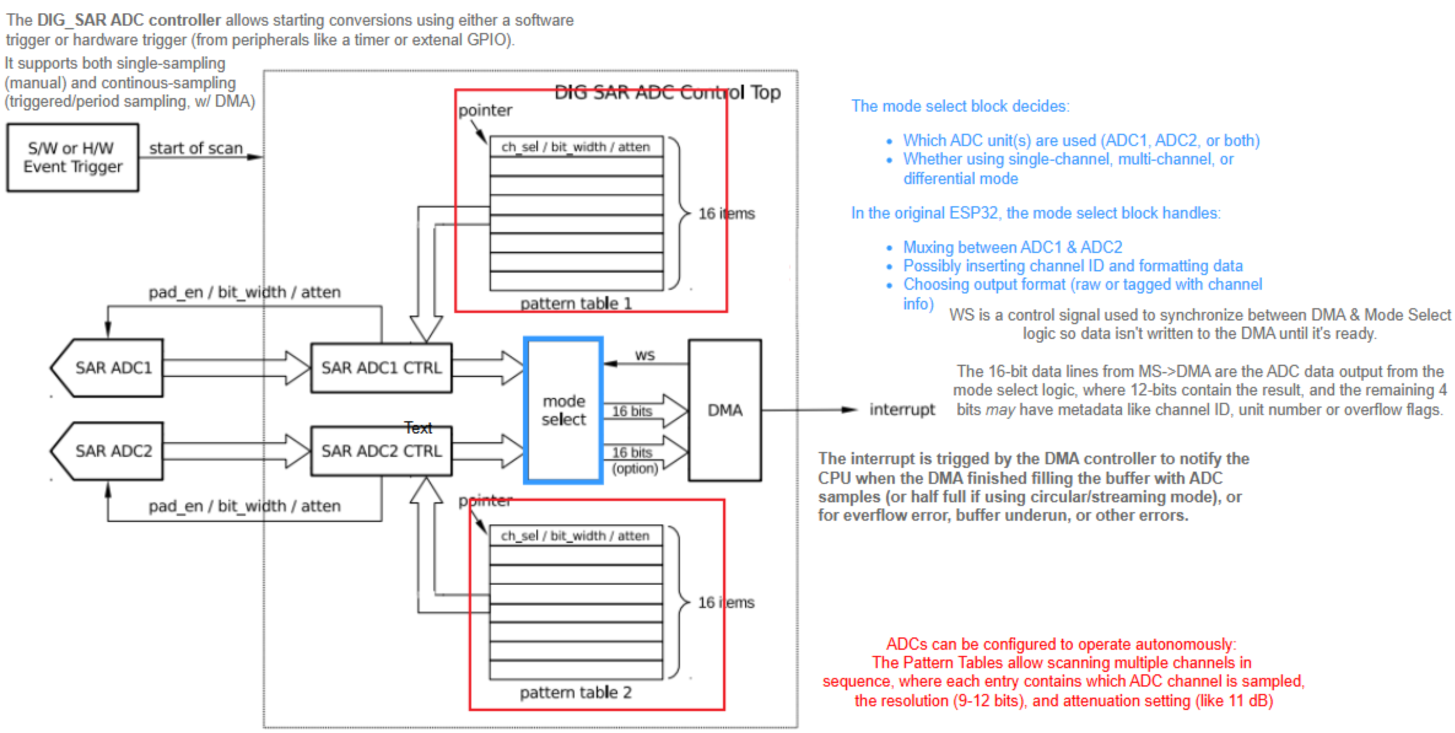


Diagram Source: https://blog.csdn.net/u010261063/article/details/130151411

ADC Sampling model 1: Single Sampling – Too Slow

Single sampling is manually triggering a single ADC read for a GPIO. It reads the voltage once and returns the value. It is available on all ESP variants and simple to use.

```
void loop() {
  int sample = analogRead(35);

  // A target sampling rate f requires a delay of T = 1/f seconds between samples.
  // 1 kHz: T = 1/1000 = 0.001 seconds    = 1000 us
  // 8 kHz: T = 1/8000 = 0.000125 seconds = 125 us
  delayMicroseconds(125); // so delay by 1000 us for ~1kHz, or by 125 us for a 8kHz sample rate
}
```

***Note:** The sample rate here isn't accurate, because the delay time does not account for the time it took to execute analogRead() and anything else*.

Even without the induced delay, [Esp32 analogRead can take ~100 us]() (100 * 10^-6 seconds).
Even assuming no other delays, that means a frequency of 1/(100 * 10^-6) = 10,000 samples per second; a sampling rate of 10 kHz. That is not enough for the [44.1 kHz needed for good-quality audio]().

ADC Sampling model 2: Continuous Sampling

Most ESP32 family chips (including original ESP32 chip used on the DOIT DEVKITV1) support continuous analog sampling using the I2S peripheral in ADC mode. Continuous sampling is more suitable for audio capture because it has a higher sampling rate (theoretically up to 2 MHz for the ESP chip) and lightens the CPU load, leaving the CPU free to handle just Bluetooth and LCD logic:

- **The I2C offloads sampling timing from the CPU:**
  When the I2S peripheral is configured in ADC mode, it generates the clock signals to trigger the ADC conversions, instead of having the CPU manage the sample timing.

- **The CPU doesn't need to poll or copy data with the DMA setup:**
  The Direct Memory Access (DMA) controller is tightly controlled coupled to the I2S peripheral. When the ADC finishes the analog->digital conversion, the result is stored in a dedicated (FIFO) hardware buffer that's 64 words deep (it can hold 64 32-bit values). The DMA automatically grabs that data and writes it directly into a predefined buffer in RAM. The CPU only gets an interrupt when the buffer is full (or half full in continuous circular mode) or when errors occur.

ADC & I2C Setup

```
// ADC resolution & Attenuation
adc1_config_width(ADC_WIDTH_BIT_12);                          // 12-bit resolution (values 0-4095)
adc1_config_channel_atten(ADC1_CHANNEL_7, ADC_ATTEN_DB_11);  // Set atten. to 11 dB (up to 2.45 V for ESP32 chip)
pinMode(35, INPUT);                                           // Program didn't work without this
adc_gpio_init(ADC_UNIT_1, (adc_channel_t)ADC1_CHANNEL_7);  // Required for I2C ADC mode

// I2S setup
i2s_config_t i2s_config = {
   // Master mode (ESP32 generates I2S clock), receive mode, read input from built-in ADC, not external I2S mic.
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX | I2S_MODE_ADC_BUILT_IN),

  .sample_rate = 40000,  // 40 kHz

   // Each sample pushed into DMA buffer will be 16 bits wide (I2S only supports 16/32-bit memory alignment).
  .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,  // The lower 12 bits are the ADC result.

  .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,  // Not using stereo audio, so just need one channel (ignore right)

  .communication_format = I2S_COMM_FORMAT_I2S_LSB, // Data bit alignment during I2S transmission: LSB first.

  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1, // Set intrpt lvl for I2S driver (used by DMA/I2S IRQs) low priority.

   // 2 DMA buffers used in a ring. More buffers -> smoother performance and fewer under/over-runs.
  .dma_buf_count = 2,  // 2 is minimum.

   // Each DMA buffer holds 1024 samples. With 2 buffers, total of 2,048 samples can be buffered at a time.
  .dma_buf_len = 1024, // At 40 kHz, that ~51ms of buffered data before overflow.

   // APLL (Audio Phase-Locked Loop) is more precise clock source. False means I2S clock is derived from
  .use_apll = false, // the general system clock (less accurate but good enough).

  .tx_desc_auto_clear = false, // Transmit (TX) mode set to false because should be in RX mode.

  .fixed_mclk = 0 // Disable the Master Clock, since it's not being used to drive external I2S devices.

}; // i2s_config is for setting format of data stored in DMA buffer. Does NOT configure ADC.

i2s_driver_install(I2S_NUM_0, &i2s_config, 4, &i2s_queue);// Install & start I2S driver for I2S peripheral 0.
i2s_set_adc_mode(ADC_UNIT_1, ADC1_CHANNEL_7);            // Tells I2S peripheral to use ADC input: GPIO35=ADC1_CH7
i2s_adc_enable(I2S_NUM_0);                              // Enables ADC capture mode for the I2S peripheral.
```

Sampling

```
void adc_sampler(void *param) {
  while (true) {
    // In the background:
    // - ADC hardware performs analog->digital conversion
    // - I2S peripheral generates sampling clock
    // - DMA controller transfers ADC samples from I2S's FIFO to a RAM buffer
                                                 // (the RAM buffer is configured during i2s_driver_install())

    size_t bytes_read = 0; // Copy data from DMA buffer in RAM to adcBuffer.
    esp_err_t result = i2s_read(I2S_NUM_0, (void *)adcBuffer, sizeof(adcBuffer), &bytes_read, portMAX_DELAY);

    if (result == ESP_OK && bytes_read > 0 && SerialBT.hasClient()) {
      int sample_count = bytes_read / sizeof(int16_t);
      int16_t *samples = (int16_t *)adcBuffer;

      // i2s_read() returns 16-bit values where only the 12 most significant bits are the audio data.
      // So shift each sample back to 12-bit range (in-place overwrite).
      for (int i = 0; i < sample_count; i++) {
        samples[i] = samples[i] >> 4;
      }

      // Bluetooth logic...
      // Induced delay to avoid overwhelming Bluetooth...

    } else {
      if (result != ESP_OK) { Serial.printf("I2S read failed with error: %d\n", result); }
      // Bluetooth failure logic...
      // Induced delay...
   }
}
```

Sampling Correction

I2s_read() returns 16-bit values, where only the most 12 significant bits are the audio data. The reason for shifting by 4 was to remove those lower bits.
However, audio streams used 8/16/24/32-bit samples, and the PC-end expect 16 bits. The mismatch caused low volume and occasional spikes of white noise.

That was corrected with the following:

```
      for (int i = 0; i < sample_count; i++) {
        samples[i] = samples[i] >> 4;           // Shift each sample back to 12-bit to remove irrelevant bits, then
        samples[i] = (samples[i] - 2048) << 4; // center & scale 12-bit ADC value to a signed 16-bit
      }
```

Bluetooth (ESP32)

Choosing Bluetooth Classic (SPP)

Bluetooth was chosen over Wi-Fi because it would be more reliable for a mobile translation device. Bluetooth Classic was chosen because BLE is not designed for continuous real-time audio, as BLE does not support audio profiles, has lower bandwidth, and uses connection intervals that interrupt streaming.
And although LE Audio (Bluetooth 5.2+) can handle audio, the ESP32 does not support BLE, and "it's complicated" on Windows.

Of the Classic profiles I **settled on Serial Port Profile (SPP)** because it is simple to implement, bi-directional (the ESP needs to transmit audio samples and receive text), and is treated as serial communication (having previously worked with serial communication over USB, it's familiar and easy to get started with).

A2DP (Advanced Audio Distribution Profile) was originally considered as a stretch goal because it specializes in high-quality audio streaming over Bluetooth. However, **the A2DP plan was scratched** because the stream is one-directional. Combining it with another profile so the PC can transmit the translation would introduce unnecessary complexity to the project.

Setup

The **Single-Sampling** sketch was much simpler; the 12-bit sample was split into two bytes (big-endian) and send over Bluetooth SPP using SerialBT.write(), which transmits only 8-bit values at a time.

```
#include "BluetoothSerial.h"
BluetoothSerial SerialBT;

#define MIC_PIN 35

void setup() {
  Serial.begin(115200);
  Serial.println("Setting up...");
  delay(1000);

  if (SerialBT.begin("ESP32SPP", false)) { Serial.println("Bluetooth started as ESP32SPP"); SerialBT.enableSSP(); }
  else                                   { Serial.println("Bluetooth failed to start"); }

  if (SerialBT.isReady()) { Serial.println("Bluetooth is ready.");
  } else                  { Serial.println("Bluetooth not ready.");}


  analogReadResolution(12);        // 12-bit ADC resolution (0-4095)
  analogSetAttenuation(ADC_11db); // Allow 0-2.45V signal (for ESP32 chip. May differ for other boards)
}

void loop() {
  int sample = analogRead(35);                // Ex value: 3017 = 0b00001011  11011001
  uint8_t high_byte = (sample >> 8) & 0xFF;  //    = 11            ^high bits
  uint8_t low_byte = sample & 0xFF;         //    = 217                        ^low bits
  // https://stackoverflow.com/a/68716108

  /*
  analogRead won't return negative values, but 'sample' is still a signed int.
  When shifting signed values, the compiler might perform arithmetic (sign-extending) shifts.
  Using '& 0xFF' can mask off any sign-extended bits in that case.
  Even if it isn't an issue here, it doesn't hurt.
  */

  /*
  Send as two separate bytes because Bluetooth Serial (SPP) (& many serial comm. protocols) transmit data in bytes
  (8 bits). The 12-bit ADC values need to split.

  The above uses a big-endian format: [high byte][low byte]
  Because it's easier to reconstruct. Using a 4-8 split means less bit-shifting than a 6-6 split.
  */


  Serial.printf("Sending: %d = [%02X %02X]\n", sample, high_byte, low_byte);
  SerialBT.write(high_byte);
  SerialBT.write(low_byte);

  // Receiver would reconstruct as:
  // uint16_t sample = ((uint16_t)high_byte << 8) | low_byte;

  delayMicroseconds(125);
}
```

On the other hand, the **Continuous-Sampling** sketch uses DMA to capture multiple samples into a buffer. For better efficiency, it sends multiple bytes (2 bytes per sample * sample count) in one call with:

```
SerialBT.write((uint8_t *)samples, sample_count * sizeof(int16_t));
```

- *samples* is a pointer to an array of 16-bit signed ints
- It casts the pointer to a *uint8_t** so data can be transmitted byte-by-byte.

Bluetooth Connectivity (PC-side)

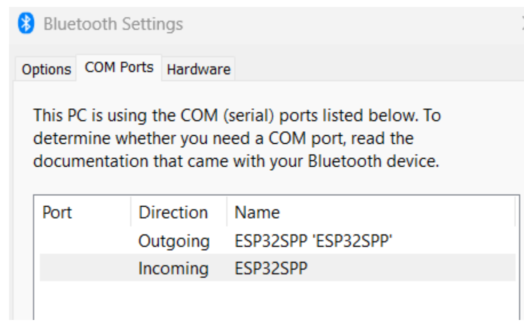
Bluetooth SPP emulates a serial connection over Bluetooth. The ESP32 uses BluetoothSerial to create “SerialBT”, a virtual interface similar to UART.

On the PC, it appears as a COM port, which is used like a USB port.
Windows specifically creates two virtual COM ports when pairing to Bluetooth devices:

- **Incoming**: For the PC to accept incoming connection requests, it’s exposed as the SerialPort service to the ESP
- **Outgoing**: For the PC to initiate a connection to the device. This is the one used to connect to the ESP
  This port can be established by viewing the stream over a serial terminal emulator like PuTTY.

Debugging

During early tests, the Bluetooth name for the ESP32 was set to ‘ESP32SPP’.
When the name was changed, Windows wouldn’t forget the old service name (despite removing device from Bluetooth list, restarting Bluetooth stack via services.msc, deleting the old registry, and restarting the PC), and the port fields were empty.



Thus, the ESP’s Bluetooth name stuck for the remainder of the project.

Other issues were solved on the ESP-side, by:

- Uninstalling the esp32 libraries, and reinstalling version 1.0.4
  (per https://github.com/espressif/arduino-esp32/issues/5164#issuecomment-838509946)
- Setting the ESP as a server with the false flag in `SerialBT.begin(NAME, false);` .

Data Receival & Audio Replay (PC-side)

A python script handles the receival on the PC-end, using the pyserial library to open the outgoing virtual port, and pyaudio to reconstruct the audio.

```
import pyaudio
import serial
import time

#  Serial settings need to match the virtual serial COM port the PC creates
SERIAL_PORT = "COM4"  # Outgoing port
BAUD_RATE =  192000
PORT_TIMEOUT = 1         # Independent of PC. Limiting how many sec(s) blocking read/write operations take

# PyAudio settings
SAMPLE_RATE = 12000 # Hz = samples/second. Must match ESP32’s I2D audio output config.
CHUNK_SIZE = 512       # Instead of continuous data, it stores in a buffer (commonly sized 512 or 1024). Smaller->faster for real-time applications, but more reads.
```

```
FORMAT = pyaudio.paInt16   # 16-bit signed integers
NUM_CHANNELS = 1           # 1 for mono, 2 for stereo (left & right)

def main():
  # Open the virtual port
  ser = serial.Serial(SERIAL_PORT, baudrate=BAUD_RATE, timeout=PORT_TIMEOUT)
  time.sleep(2)      # Wait for ESP32 to init connection
  ser.flushInput()  # Discard noise

  # Setup PyAudio as output stream to allow playback
  p = pyaudio.PyAudio()
  stream = p.open(format=FORMAT, channels= NUM_CHANNELS, rate=SAMPLE_RATE, output=True)

  try:
    while True:                       # PLAYBACK LOOP:
      data = ser.read(CHUNK_SIZE) # (1) Read a chunk data from serial port
      if data:
        stream.write(data)            # (2) Play the audio

  except KeyboardInterrupt:
    print("User terminated program.")

  finally:
    stream.stop_stream()
    stream.close()
    p.terminate()
    ser.close()

if __name__ == "__main__":
  main()
```

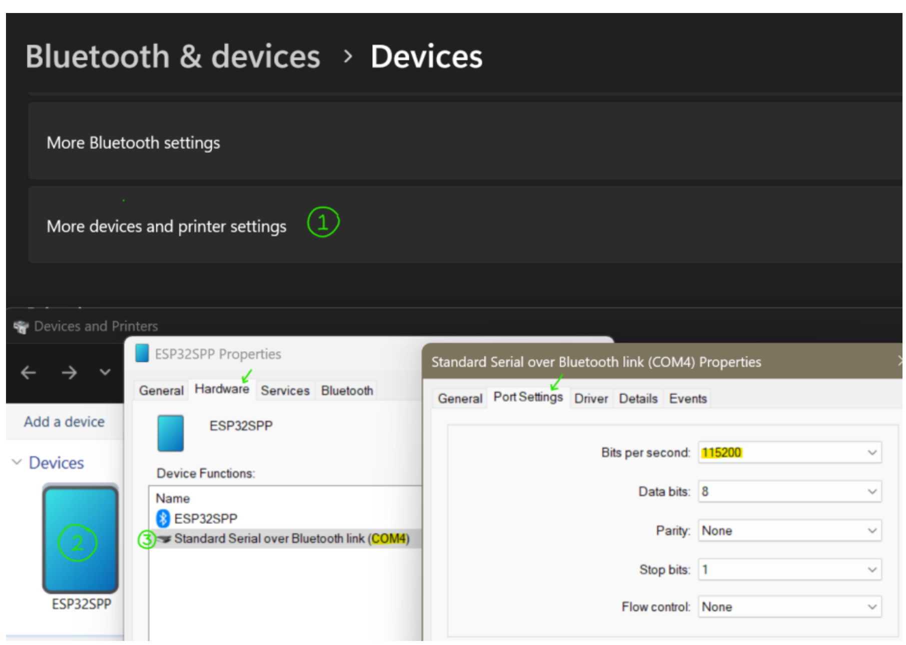
Finding Serial's Parameters:
Settings > Bluetooth
> View More Devices
> More Devices & Printer Settings

Double-click <ESP-name>
A properties window will pop up.

> Hardware
> Device Functions:
*Standard Serial over Bluetooth link (**COM#**)*

Double click that function.
Its properties window will pop up.

> Port Settings
*> (See **Bits per second**)*



Translation (PC-side)

There were a few priorities in finding the API:

- A free model without credit limits for freedom in testing and demonstration.
- A model that is accurate *enough*.
- A model that can translate to English *at least*.
- A model that can translate from Arabic *at least* (for demonstration purposes).

Initially, the APIs were all tested on WAV files: First of a voice recorded by the PC's mic first, then the audio transmitted by the ESP32 and reconstructed on the PC end.

Vosk was first because it's supposedly lightweight, works offline, and supports multiple languages. However, the Arabic model failed to load, so I moved on. I tried **Klaam** next, which provided transcription for two dialectal forms MSA and EYG). However, it didn't handle repeat text and wasn't fast enough, so I used **Whisper**'s small model. It provided speech-to-English translation in one step but was very inaccurate. Combining Whisper's transcription with MarianMT translation was attempted to improve accuracy, but the output was still garbage. Switching to the large-v3 improved it significantly. Since Klaam wasn't significantly faster and still required translation, **Whisper large-v3** was chosen because the project time was running out.

The model satisfied the minimum requirements, but still had a very slow loading time and a slow translation time, especially on the CPU. This made attempts at live streaming or silence-detection-based recording challenging because of constant processing and awkward wait times.
For the program to be useable, it was changed to a prompt-based approach instead, where audio is buffered only between manual start and stop prompts from the user. The final setup isn't real-time, but good enough for an MVP.

Model Results:

Expected Output (good old alphabet rhyme):
Alif-un (the letter) rabbit runs and it plays, eating carrots so it doesn't get tired.
Ba-un (the letter) duck jumped a jump. It fell and the cat laughed at it.
Alif-un (the letter) rabbit runs and it plays, eating carrots so it doesn't get tired.

Model Outputs:

Whisper (small model)	A thousand ants, oh Lord, eat a seed so it can grow. It's a plant, a plant, a plant, and I've found a seed from this plant. A thousand ants, oh Lord, eat a seed so it can grow.
Whisper (small model) (cleaned input audio)	A thousand ants, what a pity! They eat a tree to make it sit, they cover it, cover it, cover it and cover it, and cover it from this cover. A thousand ants, what a pity! They eat a tree to make it sit.
Whisper (small) + **Marian**	A thousand ambers eat carrots so they don't get some bounced potatoes and give a little bit of this cat.
Whisper (small) + **Marian** (cleaned input audio)	A thousand bunnies running the goat eat a button so that they don't have some rubber potatoes that fell out of this cat, and a thousand rabbits that go through the goat eat a button so that they don't.
Klaam (MSA)	ألف أرنب يجري يلعب يأكل تزراً كيلا يتعب بعء بطونطط نطى و اع (Alif-un rabbit runs and plays, eating carrots so it doesn't get tired. Ba'a ? jumped and bounces, falling)
Klaam (EYG)	أليفون ارنا بياجري يلعبي كله سذدا كه لا يتعب بأن بطا ونط نطة و (Alif-un, we want to play everything lightly and without getting tired by jumping and hopping.)

LCD Display (ESP-side)

Wiring

The ESP32 board couldn't supply enough current to power the LCD, especially with the backlight on, so a battery pack was used to supply the LCD separately. The LCD's VCC and GND were connected to the battery pack, and the ground was shared between the ESP32 and the battery pack.

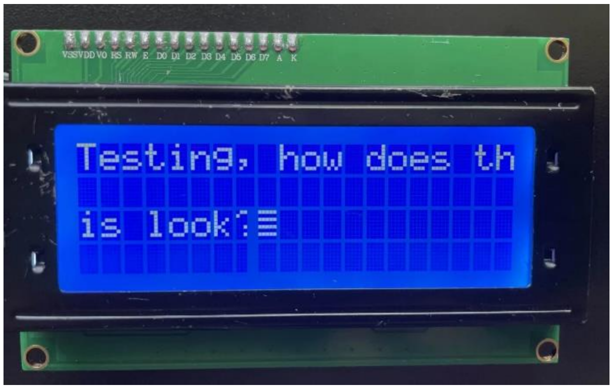
LCD Pins

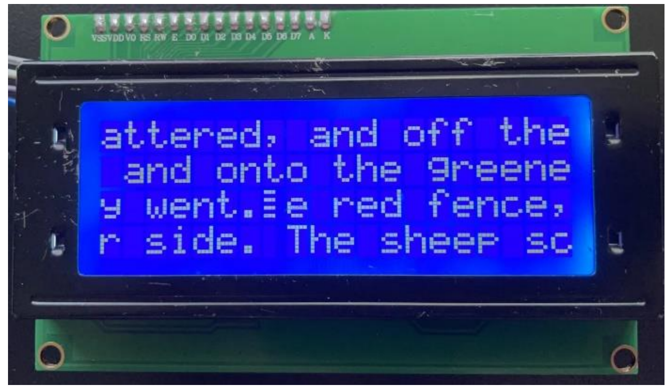
- SDA – GPIO21 (ESP32)
- SCL – GPIO22 (ESP32)
- VCC – Battery VCC
- GND – Shared between the ESP32 GND and the Battery GND

Program

As was done for all components, the LCD was first tested stand-alone:

```
#include <Wire.h>
#include "<LiquidCrystal_I2C.h>"

#define SPA_GPIO_PIN 21
#define SCL_GPIO_PIN 22

#define NUM_COLS 20
#define NUM_ROWS 4
#define I2C_ADDR 0x27

LiquidCrystal_I2C lcd(I2C_ADDR, NUM_COLS, NUM_ROWS);

void setup() {
  Wire.begin(SDA_GPIO_PIN, SCL_GPIO_PIN);
  Serial.begin(115200);

  lcd.init();
  lcd.backlight();
  lcd.clear();

  Wire.begin("\nType something and press Enter:");
}

void loop() {
   if (Serial.available() {
      String input = Serial.readStringUntil('\n');

       lcd.clear();
       lcd.setCursor(0, 0);
       lcd.print(input);

       Serial.print("Displayed on LCD: ");
       Serial.println(input);
   }
}
```





When there are large amounts of text, the LCD screen has an overflow issue, so scrolling logic was implemented:

```
#include <Wire.h>
#include "<LiquidCrystal_I2C.h>"

#define SCROLL_DELAY_MS 650

#define SPA_GPIO_PIN 21
#define SCL_GPIO_PIN 22

#define NUM_COLS 20
#define NUM_ROWS 4
#define I2C_ADDR 0x27

LiquidCrystal_I2C lcd(I2C_ADDR, NUM_COLS, NUM_ROWS);
String lcdBuffer[NUM_ROWS];

std::vector<String> wrapText(const String& text);
void scrollAndAddLine(const String& newLine);
void refreshLCD();

void setup() {
  Wire.begin(SDA_GPIO_PIN, SCL_GPIO_PIN);
  Serial.begin(115200);

  lcd.init();
  lcd.backlight();
  lcd.clear();

  Wire.begin("\nType something and press Enter:");
}

void loop() {
   if (Serial.available() {
      String input = Serial.readStringUntil('\n');
      input.trim();

      // Split input into wrapped lines
      auto wrappedLines = wrapText(input);
      for (String& line : wrappedLines) {
        scrollAndAddLine(line);
        refreshLCD();
        delay(SCROLL_DELAY_MS); // Give time for user to read.
      }

      Serial.print("Displayed on LCD:\n");  // Display on Serial monitor for debugging purposes.
      for (int i = 0; i < NUM_ROWS; i++) {
        Serial.println(lcdBuffer[i]);
      }
   }
}

// Wraps text into lines with word boundary respecting screen width
```

```
std::vector<String> wrapText(const String& text) {
  std::vector<String> lines;
  int start = 0;
  int end;
  int space_idx;

  while (start < text.length()) {
   end = start + NUM_COLS;

    if (end >= text.length()) {
      lines.push_back(text.substring(start));
      break;
    }

    space_idx = text.lastIndexOf(' ', end);    // Break on last space within screen width.
    if (space_idx <= start) { space_idx = end; } // If the word is too long, force it.

    lines.push_back(text.substring(start, space_idx));
    start = space_idx;

    // Skip any spaces at the start of the next line
    while (start < text.length() && text[start] == ' ') { start++; }
  }

  return lines;
}

// Scrolls lines up and appends a new one at the bottom
void scrollAndAddLine(const String& newLine) {
  for (int i = 0; i < NUM_ROWS - 1; i++) {
    lcdBuffer[i] = lcdBuffer[i + 1];
  }
  lcdBuffer[NUM_ROWS - 1] = newLine;
}

// Writes entire buffer to the LCD
void refreshLCD() {
  lcd.clear();
  for (int i = 0; i < NUM_ROWS; i++) {
    lcd.setCursor(0, i);
    lcd.print(lcdBuffer[i]);
  }
}
```

Final Demonstration

Pre-recorded demonstration (minor edits for some video stabilization):

Demo.mp4

Lab Demonstration Results:

```
Translated:
 A deer is playing and eating carrots so that he doesn't get tired.
```

```
Translated:
 "'No God save my daddy, as he is free'
```

What Worked

- Thank you for emphasizing sticking to the MVP and starting with the board-side instead of the "fun" stuff. It would not have been possible to have a working demo otherwise. The minimum requirements were satisfied (using the reduced goal of Arabic-input-only to English-output-only to speed up demo time).

What Could Have Been Improved

- The LCD backlight was turned on for better visibility, but it draws a lot of power. I connected the battery pack only during testing, yet it still died on demo-day. I was lucky some students lent me some.
  - On that note, students probably still couldn't read from the small ceiling-facing LCD on the table.
- The demo system doesn't support multi-language input/output. The scope was narrows to Arabic input and English translation.
  - The whisper translation model can only translate to English. This can be fixed by using the Whisper model for transcribing the audio to text and using another API for the translation.
- The demo system is not real-time. It uses a prompted approach instead of continuous streaming, mostly because the Whisper large-v3 model was very slow to load and translate.

PC-side program (.py):

```
import time
import serial
import numpy as np
import whisper
import threading
from scipy.signal import resample

# --- Serial config ---
SERIAL_PORT = "COM4"
BAUD_RATE = 115200
PORT_TIMEOUT = 1

# --- Audio config ---
INPUT_SAMPLE_RATE = 12000
TARGET_SAMPLE_RATE = 16000
FORMAT_WIDTH = 2
CHUNK_SIZE = 512
MAX_DURATION_SEC = 10
MAX_SAMPLES = int(MAX_DURATION_SEC * INPUT_SAMPLE_RATE)

# --- Whisper config ---
LANG = "ar"
MODEL_SIZE = "large-v3"

# --- Recording control ---
stop_recording = False

def decode_audio(buffer):
    audio_int16 = np.frombuffer(buffer, dtype=np.int16).astype(np.float32)
```

```
    audio_int16 /= 32768.0
    audio_int16 -= np.mean(audio_int16)
    peak = np.max(np.abs(audio_int16))
    if peak > 0:
        audio_int16 /= peak

    num_samples = int(len(audio_int16) * TARGET_SAMPLE_RATE / INPUT_SAMPLE_RATE)
    return resample(audio_int16, num_samples)

def wait_for_enter():
    global stop_recording
    input()  # Block until Enter is pressed
    stop_recording = True

def main():
    global stop_recording

    print("Loading Whisper model...")
    model = whisper.load_model(MODEL_SIZE)

    print("Opening serial port...")
    ser = serial.Serial(SERIAL_PORT, baudrate=BAUD_RATE, timeout=PORT_TIMEOUT)
    time.sleep(2)
    ser.flushInput()

    print("\nReady. You'll be prompted to press Enter to start and stop recording.\n")

    try:
        while True:
            input("Press Enter to START recording...")

            stop_recording = False
            buffer = bytearray()

            # Launch background thread to wait for user to stop
            threading.Thread(target=wait_for_enter, daemon=True).start()

            print("Recording... Press Enter again to STOP.")

            while not stop_recording:
                if ser.in_waiting:
                    chunk = ser.read(min(CHUNK_SIZE, ser.in_waiting))
                    buffer += chunk

                    if len(buffer) > MAX_SAMPLES * FORMAT_WIDTH:
                        print("Reached max duration.")
                        break

                time.sleep(0.01)

            print(f"\nCaptured {len(buffer)} bytes. Transcribing...")

            audio = decode_audio(buffer)
```

```
            result = model.transcribe(audio, language=LANG, task="translate")
            translated_text = result["text"].strip()
            print("\nTranslated:\n", translated_text)

            if translated_text:
                ser.write(translated_text.encode("utf-8") + b"\n")

            print("\n--- Done ---\n")

    except KeyboardInterrupt:
        print("User interrupted.")
    finally:
        ser.close()

if __name__ == "__main__":
    main()
```

ESP-side sketch (.ino):

```
#include <Arduino.h>
#include <BluetoothSerial.h>
#include <LiquidCrystal_I2C.h>
#include <Wire.h>
#include <vector>
#include "driver/adc.h"
#include "driver/i2s.h"

#define BT_DEVICE_NAME "ESP32SPP"
#define MAX_CONNECTION_ATTEMPTS 10

#define I2S_SAMPLE_RATE 12000
#define I2S_DMA_BUF_LEN 512
#define I2S_DMA_BUF_COUNT 2
#define ADC_BUFFER_SIZE I2S_DMA_BUF_LEN

#define SDA_GPIO_PIN 21
#define SCL_GPIO_PIN 22

#define NUM_COLS 20
#define NUM_ROWS 4
#define I2C_ADDR 0x27
#define SCROLL_DELAY_MS 650

BluetoothSerial SerialBT;
LiquidCrystal_I2C lcd(I2C_ADDR, NUM_COLS, NUM_ROWS);
String lcdBuffer[NUM_ROWS];

uint8_t adcBuffer[ADC_BUFFER_SIZE];
static QueueHandle_t i2s_queue;

void adc_sampler(void *param);
```

```
std::vector<String> wrapText(const String &text);
void scrollAndAddLine(const String &newLine);
void refreshLCD();
void displayRecievedText();

void setup() {
 Serial.begin(115200);
 Serial.println("\nBooting...");

 // Bluetooth setup
 int attempt_count = 0;
 while (!SerialBT.begin(BT_DEVICE_NAME, false) && attempt_count < MAX_CONNECTION_ATTEMPTS) {
  Serial.printf("Retrying Bluetooth (%d)...\n", ++attempt_count);
  delay(2000);
 }

 if (attempt_count >= MAX_CONNECTION_ATTEMPTS) {
  Serial.println("Bluetooth failed. Restarting...");
  delay(1000);
  esp_restart();
 }
 Serial.print("Bluetooth has been setup (Bluetooth name: ");
 Serial.print(BT_DEVICE_NAME);
 Serial.println("). ");

 // ADC + I2S Setup
 Serial.println("Configuring ADC & setting up I2S...");
 adc1_config_width(ADC_WIDTH_BIT_12);                                   // 12-bit resolution (0-4095)
 adc1_config_channel_atten(ADC1_CHANNEL_7, ADC_ATTEN_DB_11);  //  Set attenuation to 11 dB
 pinMode(35, INPUT);                                                    // Program didn't work without this
 adc_gpio_init(ADC_UNIT_1, (adc_channel_t)ADC1_CHANNEL_7);         // Required for I2S ADC mode

 i2s_config_t i2s_config = {
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_RX | I2S_MODE_ADC_BUILT_IN),
  .sample_rate = I2S_SAMPLE_RATE,
  .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
  .channel_format = I2S_CHANNEL_FMT_ONLY_LEFT,
  .communication_format = I2S_COMM_FORMAT_I2S_LSB,
  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
  .dma_buf_count = I2S_DMA_BUF_COUNT,
  .dma_buf_len = I2S_DMA_BUF_LEN,
  .use_apll = false,
  .tx_desc_auto_clear = false,
  .fixed_mclk = 0
 };

 i2s_driver_install(I2S_NUM_0, &i2s_config, 4, &i2s_queue);
 i2s_set_adc_mode(ADC_UNIT_1, ADC1_CHANNEL_7); // GPIO35 = ADC1_CH7
 i2s_adc_enable(I2S_NUM_0);

 Serial.println("ADC and I2S have been setup.");
```

```
  // LCD Setup
  Wire.begin(SDA_GPIO_PIN, SCL_GPIO_PIN);
  lcd.init();
  lcd.backlight();
  lcd.clear();
  lcd.setCursor(0, 0);

  lcd.print("Ready.");
  Serial.println("LCD has been setup.");

  // Start ADC task.
  Serial.println("Starting ADC task...");
  xTaskCreatePinnedToCore(adc_sampler, "ADC Reader", 4096, NULL, 1, NULL, 0);

  Serial.println("Setup complete!\n");
}

void loop() {
  if (!SerialBT.connected()) { Serial.println("[Bluetooth] Not connected."); delay(2000); }
  if (!SerialBT.hasClient()) { SerialBT.println("Ping from ESP32"); delay(1000); return; }

  displayRecievedText();  // Continuously check for incoming text from PC
}

void adc_sampler(void *param) {
  while (true) {
    size_t bytes_read = 0;
    esp_err_t result = i2s_read(I2S_NUM_0, (void *)adcBuffer, sizeof(adcBuffer), &bytes_read, portMAX_DELAY);

    if (result == ESP_OK && bytes_read > 0 && SerialBT.hasClient()) {
      int sample_count = bytes_read / sizeof(int16_t);
      int16_t *samples = (int16_t *)adcBuffer;

      for (int i = 0; i < sample_count; i++) {
        samples[i] = (samples[i] >> 4);                  // Cleaned 12-bit
        samples[i] = (samples[i] - 2048) << 4;       // Signed 16-bit
      }

      //Serial.println("Sending samples. ");
      SerialBT.write((uint8_t *)samples, sample_count * sizeof(int16_t));
      vTaskDelay(10 / portTICK_PERIOD_MS); // Slow down slightly to avoid overwhelming Bluetooth

    } else {
      if (result != ESP_OK) { Serial.printf("I2S read failed with error: %d\n", result); }
      else if (!SerialBT.hasClient()) { Serial.println("[Bluetooth] No client."); delay(1000); }
      else if (!bytes_read > 0)      { Serial.println("Buffer empty."); }

      vTaskDelay(20 / portTICK_PERIOD_MS); // Yield to other tasks
    }
  }
}
```

```
void displayRecievedText() {
  static String buffer;

  while (SerialBT.available()) {
    char c = SerialBT.read();

    if (c == '\n') {
      buffer.trim();

      if (buffer.length() > 0) {
        Serial.println("[Received] " + buffer);

        auto lines = wrapText(buffer);
        for (String &line : lines) {
          scrollAndAddLine(line);
          refreshLCD();
          delay(SCROLL_DELAY_MS);
        }
      }
      buffer = "";  // Clear buffer after newline

    } else {   buffer += c; }
  }
}

// LCD UTILS

std::vector<String> wrapText(const String &text) {
  std::vector<String> lines;
  int start = 0;

  while (start < text.length()) {
    int end = start + NUM_COLS;
    if (end >= text.length()) {
      lines.push_back(text.substring(start));
      break;
    }

    int spaceIndex = text.lastIndexOf(' ', end);
    if (spaceIndex <= start) { spaceIndex = end; }
    lines.push_back(text.substring(start, spaceIndex));
    start = spaceIndex;
    while (start < text.length() && text[start] == ' ') { start++ };
  }
  return lines;
}

void scrollAndAddLine(const String &newLine) {
  for (int i = 0; i < NUM_ROWS - 1; i++) {
    lcdBuffer[i] = lcdBuffer[i + 1];
  }
  lcdBuffer[NUM_ROWS - 1] = newLine;
}
```

```
void refreshLCD() {
  lcd.clear();
  for (int i = 0; i < NUM_ROWS; i++) {
    lcd.setCursor(0, i);
    lcd.print(lcdBuffer[i]);
  }
}
```

References

Bluetooth:

- https://www.ezurio.com/resources/blog/bluetooth-low-energy-vs-bluetooth-classic-what-s-the-difference
- https://www.ezurio.com/support/faqs/how-does-le-audio-quality-compare-classic-audio-quality
- https://www.espressif.com/sites/default/files/documentation/esp32_bluetooth_architecture_en.pdf
- https://stackoverflow.com/a/45241019

Setup:

- https://www.reddit.com/r/esp32/comments/122mgn3/can_an_esp32_dev_board_provide_5v/

Installing the Drivers CP2102 for the USB Bridge Chip

- https://techexplorations.com/guides/esp32/begin/cp21xxx/

Recommended sampling rates for audio:

- https://www.productlondon.com/what-sample-rate-should-i-use/
- https://majormixing.com/audio-sample-rate-and-bit-depth-complete-guide/
- https://www.reddit.com/r/audiophile/comments/1d79o4l/comment/l6yc5n2/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1&utm_content=share_button

ADC calibration:

- https://docs.espressif.com/projects/esp-idf/en/v4.3.2/esp32/api-reference/peripherals/adc.html
- https://docs.espressif.com/projects/esp-idf/en/release-v4.4/esp32/api-reference/peripherals/adc.html

I2S and DMA for high-speed ADC sampling:

- https://www.atomic14.com/2020/09/12/esp32-audio-input
- https://docs.espressif.com/projects/esp-faq/en/latest/software-framework/peripherals/adc.html#:~:text=The%20ESP32%20ADC%20has%2018%20channels.%20If%20you,internal%20significant%20digit%20of%20ADC%20is%2012%20bits.
- https://circuitdigest.com/microcontroller-projects/i2s-communication-on-esp32-to-transmit-and-receive-audio-data-using-max98357a
- https://docs.espressif.com/projects/arduino-esp32/en/latest/api/i2s.html

[PC] PyAudio and Serial

- https://people.csail.mit.edu/hubert/pyaudio/
- https://people.csail.mit.edu/hubert/pyaudio/docs/#example-blocking-mode-audio-i-o
- https://www.headphonesty.com/2019/07/sample-rate-bit-depth-bit-rate/