Shahjalal University of Science and Technology, Sylhet

Department of Electrical and Electronic Engineering

Project Report

Audio classification with STM32f746G

Submitted By:

Group: ODD
Session : 2021-22

Submitted To:

Dr. Md Rasedujjaman
Associate Professor
Dept. of EEE
SUST, Sylhet.

Course Title: Digital Signal Processing I Lab
Course Code: EEE 322

Date of Submission: 9 December 2025

Odd Group Registration Numbers

Odd Group	Registration Number
1	2021338003
	2021338043
	2021338044
	2021338045
3	2021338008
	2021338018
	2021338019
	2021338031
5	2021338036
	2021338040
	2021338046
	2021338048
7	2021338004
	2021338006
	2021338035
	2021338037
9	2021338022
	2021338024
	2021338026
	2021338032
11	2021338028
	2021338030
	2021338038
	2021338039
13	2021338002
	2021338033

Contents

1 Introduction **1**

2 Project Objectives **2**

3 Background Study **2**

4 Hardware Used **2**

5 Software and Libraries **3**

6 GPIO Pin Configuration for STM32F746G-DISCO **4**
6.1 QUADSPI Interface 6
6.2 SDMMC1 Interface 7
6.3 I2C3 Touch Controller 7
6.4 SAI2 Audio Interface 7
6.5 LTDC Display Interface 8
6.6 USART Interfaces 9

7 Methodology **9**
7.1 Project Setup and Working Procedure 9
7.2 System Block Diagram 14
7.3 role 14

8 Dataset Collection **15**
8.1 code 16

9 MFCC Feature Extraction **17**

10 Model Training **17**

11 Step-by-Step Description of the Implemented Code **18**
11.1 Environment Setup 18
11.2 code 18
11.3 Reading Audio Files 18
11.4 Feature Extraction 21
11.5 Dataset Preparation 22
11.6 Model Construction 23

11.7 Model Training 24
11.8 Model Evaluation 25
11.9 Saving Model 26
11.10 Summary 30

12 Embedded Deployment **30**

13 Results and Discussion **30**

14 Conclusion **31**

15 Output: **31**

16 References **32**

17 Runtime Information **32**

Objective:

Speech carries unique vocal characteristics that can be used to identify an individual. This project presents a real-time voice identity recognition system implemented on the STM32F746G-DISCO board. Speech is captured using the onboard MEMS microphone, converted from PDM to PCM, processed to extract Mel-Frequency Cepstral Coefficients (MFCC), and classified using a machine-learning model trained in Python. The deployed model runs on TensorFlow Lite Micro and displays the detected speakers name on the integrated LCD. The system demonstrates the feasibility of combining DSP techniques and embedded machine learning on a low-power microcontroller platform.

.

1 Introduction

The main objective of this work is to apply the theoretical and practical knowledge gained from the Digital Signal Processing (DSP) Laboratory into a real embedded application. The project involves several DSP operations such as sampling, framing, windowing, feature extraction, and embedded inference. It also demonstrates the complete workflow of building a machine learning model in Python, converting it to TensorFlow Lite, and deploying it efficiently on a resource-constrained hardware platform. The STM32F746NG microcontroller is powered by an ARM Cortex-M7 core running at 216 MHz, which provides sufficient computational resources to perform real-time audio acquisition, MFCC computation, and neural-network inference. This project also explores the challenges of deploying a trained Python model into an embedded C environment using TensorFlow Lite Micro, ensuring efficient memory usage and low-latency performance. As the Group Leader, I supervised the system design, coordinated dataset collection, implemented Python-based model training, integrated DSP algorithms, and optimized embedded inference. I ensured the seamless interaction of hardware, software, and machine-learning modules to achieve a fully functional real-time speaker identification system. Overall, this project successfully demonstrates how embedded systems and machine-learning techniques can be combined to create an intelligent, real-time, resource-efficient voice identity recognition platform.

2 Project Objectives

The key objectives of this project are:

1. To design and implement a real-time speaker recognition system on STM32F746G-DISCO.
2. To record speech signals using the on-board MEMS microphone.
3. To compute MFCC features for speaker identity extraction.
4. To develop a Python-based machine-learning model.
5. To convert the trained model to TensorFlow Lite Micro format.
6. To run real-time inference on the STM32 microcontroller.
7. To display the speakers name on the TFT LCD.
8. To evaluate performance, latency, and accuracy.

3 Background Study

Speaker recognition systems rely on extracting unique vocal features from speech signals. Traditional DSP techniques such as spectral analysis were widely used, but MFCC became the standard due to its alignment with human auditory perception.With microcontrollers becoming more powerful, embedded machine learning enables real-time classification on devices like STM32. TensorFlow Lite Micro is specifically optimized for resource-limited hardware.

4 Hardware Used

- **STM32F746G-DISCO Board:** ARM Cortex-M7 @ 216 MHz, 1 MB Flash, 320 KB SRAM, STM32F746NGH6 MCU (STM32F7 series)
- **Digital MEMS Microphone:** MP34DT01, PDM output, used for speech capture
- **TFT LCD Display:** 4.3-inch, 480×272 resolution, for displaying speaker identity

- **Audio Subsystem:** WM8994 codec, ADC, DAC, SAI interface; PDM-to-PCM conversion
- **Laptop/PC:** Used for dataset creation and model training (OS: Windows/Linux)
- **ST-Link Cable:** For powering, debugging, and flashing STM32 board

5 Software and Libraries

- **STM32CubeIDE:** v1.19 ; IDE for writing, compiling, and flashing code
- **STM32CubeMX:** v6.6 ; configuration tool for peripherals
- **Jupyter Notebook:** v6.5+; used for dataset preprocessing, plotting signals, pole-zero plots
- **HAL BSP Library:** STM32CubeF7 v1.26.0; provides hardware abstraction layer for peripherals
- **STM32 X-CUBE-AI:** v6.1.0; converts trained ML models (TensorFlow/TFLite) to optimized C code for STM32
- **Librosa:** v0.10.0; Python library for audio processing and feature extraction
- **NumPy:** v1.25.0; numerical computing in Python
- **TensorFlow / TensorFlow Lite:** TF 2.15.0 / TFLite 2.15.0; model training and inference
- **Optional: CMSIS DSP Library:** v5.9.0; signal processing functions for STM32
- **Optional: TouchGFX:** v4.23.0; GUI development on STM32 TFT displays

6 GPIO Pin Configuration for STM32F746G-DISCO

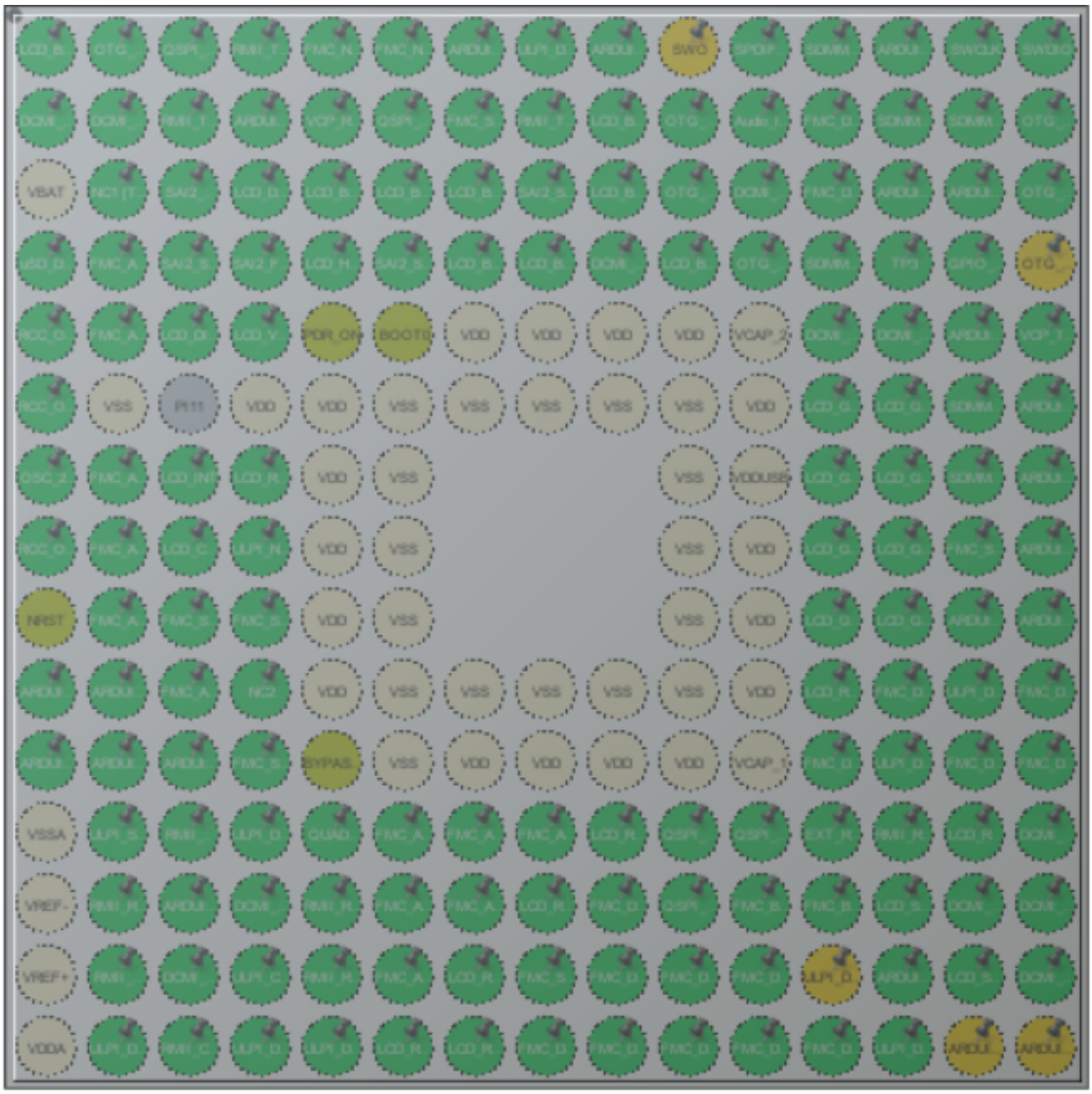


Figure 1: Top view

This section summarizes the GPIO pin assignments and modes used in the audio classification project on the STM32F746G-DISCO development board. Each subsection contains a table that is placed immediately after the subsection header so numbering and layout match the project source code.

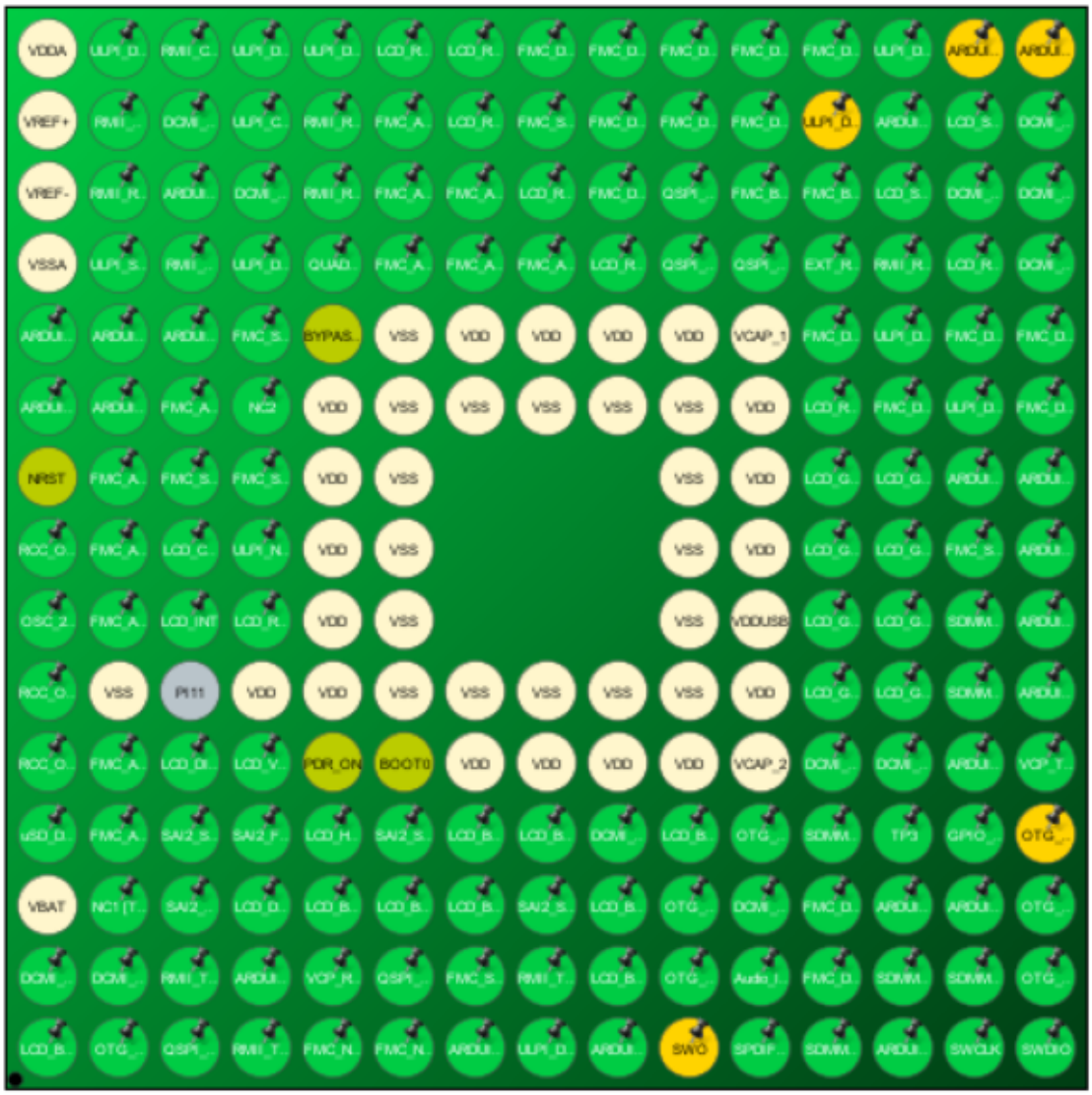


Figure 2: Bottom view

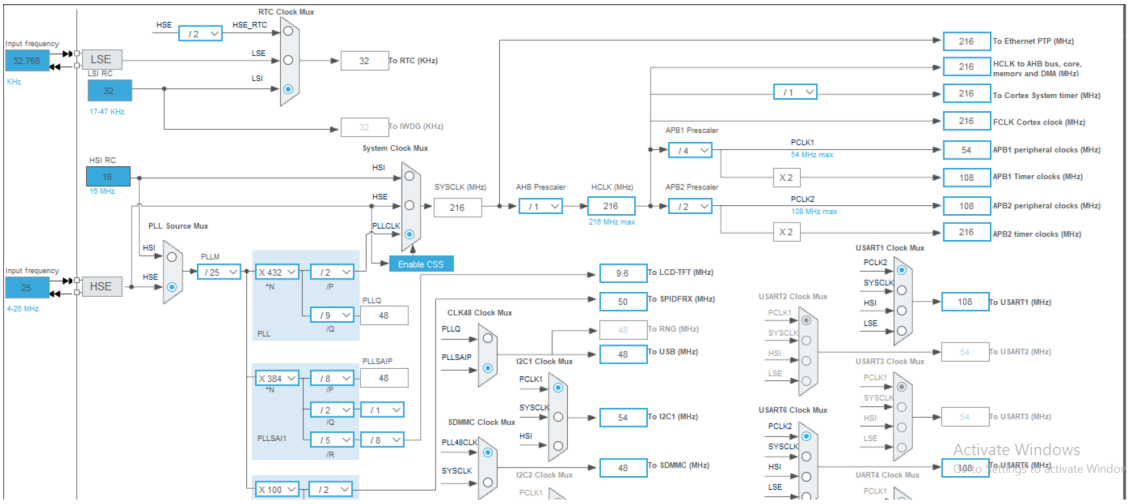


Figure 3: clock set up

6.1 QUADSPI Interface

Table 1: QUADSPI pin configuration

Pin	Signal	Mode	Alternate Function
PB2	QUADSPI_CLK	AF Push-Pull	AF9
PB6	QUADSPI_NCS	AF Push-Pull	AF10
PD11	QUADSPI_IO0	AF Push-Pull	AF9
PD12	QUADSPI_IO1	AF Push-Pull	AF9
PE2	QUADSPI_IO2	AF Push-Pull	AF9
PD13	QUADSPI_IO3	AF Push-Pull	AF9

6.2 SDMMC1 Interface

Table 2: SDMMC1 pin configuration

Pin	Signal	Mode	Alternate Function
PC8	SDMMC1_D0	AF Push-Pull	AF12
PC9	SDMMC1_D1	AF Push-Pull	AF12
PC10	SDMMC1_D2	AF Push-Pull	AF12
PC11	SDMMC1_D3	AF Push-Pull	AF12
PC12	SDMMC1_CLK	AF Push-Pull	AF12
PD2	SDMMC1_CMD	AF Push-Pull	AF12

6.3 I2C3 Touch Controller

Table 3: I2C3 (touch controller) pin configuration

Pin	Signal	Mode	Alternate Function
PH7	I2C3_SCL	AF Open-Drain + Pull-up	AF4
PH8	I2C3_SDA	AF Open-Drain + Pull-up	AF4
PI13	Touch INT	GPIO Input	—

6.4 SAI2 Audio Interface

Table 4: SAI2 pin configuration

Pin	SAI2 Signal	Mode	Speed	Used
PG10	SAI2_SD_B	Alternate Function Push-Pull	Low	true
PI4	SAI2_MCLK_A	Alternate Function Push-Pull	Low	true
PI5	SAI2_SCK_A	Alternate Function Push-Pull	Low	true
PI6	SAI2_SD_A	Alternate Function Push-Pull	Low	true
PI7	SAI2_FS_A	Alternate Function Push-Pull	Low	true

6.5 LTDC Display Interface

Table 5: LTDC pin configuration for RK043FN48H display

Pin	LTDC Signal	Mode	Speed	Used
PE4	LTDC_B0	Alternate Function Push-Pull	Low	true
PG12	LTDC_B4	Alternate Function Push-Pull	Low	true
PI9	LTDC_VSYNC	Alternate Function Push-Pull	Low	true
PI10	LTDC_HSYNC	Alternate Function Push-Pull	Low	true
PI14	LTDC_CLK	Alternate Function Push-Pull	Low	true
PI15	LTDC_R0	Alternate Function Push-Pull	Low	true
PJ0	LTDC_R1	Alternate Function Push-Pull	Low	true
PJ1	LTDC_R2	Alternate Function Push-Pull	Low	true
PJ2	LTDC_R3	Alternate Function Push-Pull	Low	true
PJ3	LTDC_R4	Alternate Function Push-Pull	Low	true
PJ4	LTDC_R5	Alternate Function Push-Pull	Low	true
PJ5	LTDC_R6	Alternate Function Push-Pull	Low	true
PJ6	LTDC_R7	Alternate Function Push-Pull	Low	true
PJ7	LTDC_G0	Alternate Function Push-Pull	Low	true
PJ8	LTDC_G1	Alternate Function Push-Pull	Low	true
PJ9	LTDC_G2	Alternate Function Push-Pull	Low	true
PJ10	LTDC_G3	Alternate Function Push-Pull	Low	true
PJ11	LTDC_G4	Alternate Function Push-Pull	Low	true
PJ13	LTDC_B1	Alternate Function Push-Pull	Low	true
PJ14	LTDC_B2	Alternate Function Push-Pull	Low	true
PJ15	LTDC_B3	Alternate Function Push-Pull	Low	true
PK0	LTDC_G5	Alternate Function Push-Pull	Low	true
PK1	LTDC_G6	Alternate Function Push-Pull	Low	true
PK2	LTDC_G7	Alternate Function Push-Pull	Low	true
PK4	LTDC_B5	Alternate Function Push-Pull	Low	true
PK5	LTDC_B6	Alternate Function Push-Pull	Low	true
PK6	LTDC_B7	Alternate Function Push-Pull	Low	true
PK7	LTDC_DE	Alternate Function Push-Pull	Low	true

6.6 USART Interfaces

Table 6: USART pin configuration

Pin	USART Signal	Mode	Speed	Used
PA9	USART1_TX	Alternate Function Push-Pull	Low	true
PB7	USART1_RX	Alternate Function Push-Pull	Low	true
PC6	USART6_TX	Alternate Function Push-Pull	Very High	true
PC7	USART6_RX	Alternate Function Push-Pull	Very High	true

7 Methodology

7.1 Project Setup and Working Procedure

In this section, we begin describing our working steps. First, the entire work was divided into three major parts:

1. Recording and playing audio,
2. Implementing the AI model, and
3. MFCC calculation.

To start, install and open **STM32CubeIDE**. Then navigate to:

$$\text{File} \rightarrow \text{New} \rightarrow \text{STM32 Project}$$

The IDE may download and extract required files, so wait until the STM32 project page appears.Once the window opens, look at the upper-left corner where four options are visible. Select the **Board Selector** option and search for your development board. In our case, the board used is the **STM32F746G-DISCO**. After selecting the board, the IDE will display relevant information about it. Read through the details and click **Next** at the bottom-right corner.Enter a project name of your choice, then click **Next**. Select the desired firmware package version; in this work, version **v1.17.4** was used. Click **Finish**. If the IDE prompts whether you want to use the default pin configuration, click **Yes**, unless you prefer configuring the pins manually.One of the most useful features of STM32CubeIDE is that pin functions can be changed easily—for example, switching a pin from GPIO to UART, or vice versa. Many other development tools do not offer this level of flexibility.After the

`.ioc` file opens, adjust the pin configuration according to the requirements of this project. Then navigate to:

File → Save

STM32CubeIDE will automatically generate the necessary initialization code.

On the left side of STM32CubeIDE, you will find a folder named after your project. Click on this folder to expand its contents. Several subdirectories will appear. To access the main application code, navigate to:

Core → Src → `main.c`

Inside `main.c`, STM32CubeIDE automatically generates code for clock configuration, register initialization, and pin setup. Understanding these sections requires considerable knowledge of embedded systems and microcontroller architecture. If you are not familiar with these topics, you may skip these parts and scroll directly to the **main loop**.

Within the main loop, you will find an infinite loop structure:

```
while (1)
{
    /* Write your logic here */
}
```

This is where your application logic should be written.

A very important note is that any code placed *outside* the designated user-code regions will be removed automatically whenever the project is regenerated. Therefore, custom code must be placed between the following markers:

```
/* USER CODE BEGIN */
/* USER CODE END */
```

Alternatively, separate source and header files can be created under the `Core` directory to ensure that custom code is not overwritten.For reference, the project created for this work is named **Audio_ML**. The project directory contains folders such as:

- **Binary**: Contains the compiled output files such as `.bin` and `.elf`, which are used for programming the microcontroller.
- **Include**: Stores the project's header files (`.h`), which declare functions, variables, and configuration parameters used throughout the code.

- **Core**: The main application directory. It contains startup code, system initialization files, and user-defined source files.
- **Drivers**: Includes STM32 HAL and LL driver libraries that provide hardware abstraction and allow interaction with microcontroller peripherals.
- **FatFs**: A lightweight file system library used when SD card or USB mass storage support is required.
- **Middlewares**: Contains third-party and ST middleware components such as USB stacks, FatFs, and other protocol libraries.
- **USB Host**: Provides source files necessary for enabling USB Host functionality, allowing the MCU to communicate with USB devices.
- **X-Cube**: Includes additional STM32Cube software extensions, such as AI packages, audio libraries, and sensor modules.
- **Debug**: Stores debug-related build artifacts used during debugging sessions, such as symbol files and intermediate outputs.
- **Utilities**: Contains helper scripts, board support package (BSP) code, and general utility functions used across the project.
- **.ioc File**: The STM32CubeMX configuration file. It defines pin mapping, clock configuration, and middleware setup. Code regeneration depends on this file.
- **License Files**: Provide licensing information for STM32 HAL, middleware libraries, and any third-party software components included in the project.

I use FreeRTOS (Free Real-Time Operating System) which is an open-source operating system designed specifically for small embedded systems. It allows your microcontroller to run multiple tasks (functions) at the same time in a predictable, real-time manner.

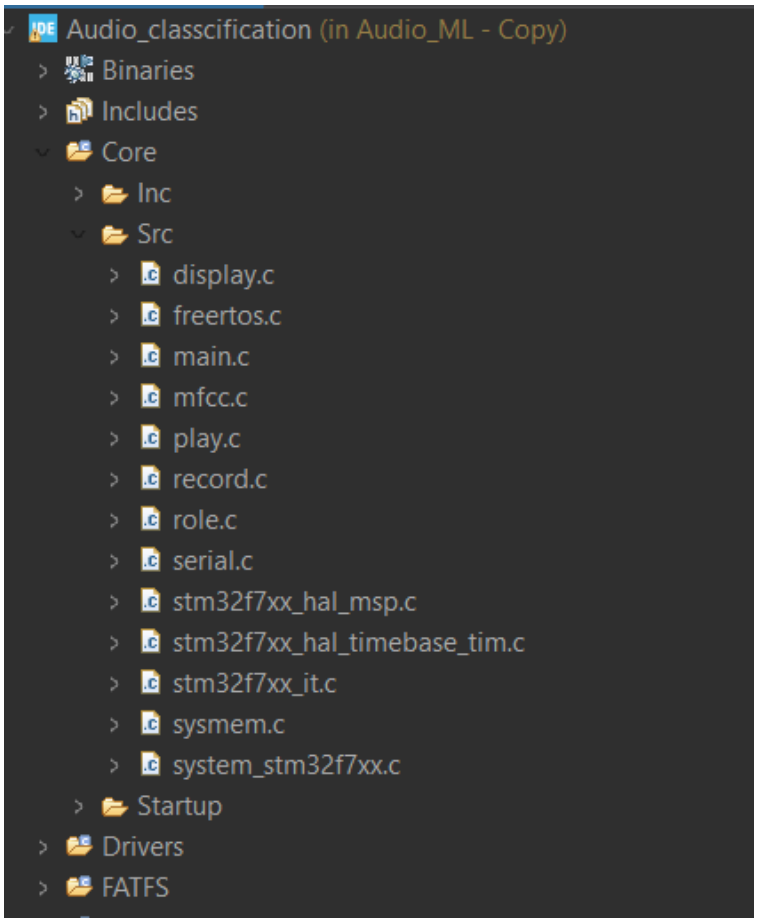


Figure 4: c files of our project

- **display.c**: Responsible for writing single-line and multi-line text to the on-board display.
- **freertos.c**: Manages all FreeRTOS-related functionality, including task creation, scheduling, and task priority configuration.
- **main.c**: Contains essential system initialization code such as clock configuration, register setup, pin mode configuration, and peripheral initialization.
- **mfcc.c**: Implements the MFCC calculation algorithm, converting raw audio waveforms into MFCC features for machine learning.
- **play.c**: Handles audio playback functionality, allowing the system to output wave files.
- **record.c**: Manages audio recording using the MEMS microphone and stores recorded data in the SD card or SDRAM.

- **role.c**: Defines system task behavior and coordinates how different tasks interact with each other.
- **serial.c**: Provides UART transmit and receive functions, mainly used for debugging and printing diagnostic messages.
- **Auto-generated System Files:**
  `stm32f7xx_hal_msp.c`, `stm32f7xx_hal_timebase.c`, `stm32f7xx_it.c`, `sysmem.c`, `system_stm32f7xx.c`
  These files are automatically generated by STM32CubeMX and contain low-level routines for interrupts, timing functions, memory configuration, and system clock setup.

The MEMS microphone is not directly connected to the STM32 microcontroller. Instead, the WM8994 acts as an intermediate device between the microphone and the STM32. The WM8994 is a high-performance audio codec (ADC + DAC) commonly used in embedded systems, especially on STM32F7 Discovery boards. It converts the analog microphone signal into digital PCM (Pulse Code Modulation) audio. The STM32 communicates with the WM8994 using **I^2C** to configure codec registers, and **SAI (Serial Audio Interface)** to send and receive the actual audio samples.

MEMS Pin	WM8994 Pin	Function
MIC_OUT	IN1L / IN1R	Analog microphone input
VDD	MICBIAS	Microphone bias supply
GND	GND	Ground

WM8994 Pin	STM32 Pin	Signal	Purpose
AIF1_SD	SAI1_SD_A	Serial Data	PCM audio data
AIF1_BCLK	SAI1_SCK_A	Bit Clock	Audio bit timing
AIF1_LRCLK	SAI1_FS_A	Frame Sync	Left/Right channel select
MCLK1	SAI1_MCLK_A	Master Clock	Codec master clock

WM8994 Pin	STM32 Pin	Purpose
SCL	I2C1_SCL	Codec configuration clock
SDA	I2C1_SDA	Codec configuration data
RESET	GPIO Pin	Codec reset control

In this system, audio is recorded at a sampling frequency of **16 kHz**, meaning the analog sound signal from the MEMS microphone is converted into 16,000 digital samples per second by the WM8994 audio codec. To perform this conversion accurately, the WM8994 requires a stable master clock (MCLK), typically derived from the STM32's system clock, and the SAI interface must generate the correct bit clock (BCLK) and frame sync (FS) signals. These clock signals determine the timing of data transmission: BCLK controls the rate at which individual PCM bits are transferred, while FS marks the start of each audio frame. If the clock frequencies are incorrect, the codec cannot sample or output audio properly, resulting in distorted or unusable data. Once the clocking is configured, the WM8994 converts the analog microphone input into 16-bit PCM samples at 48 kHz and sends them to the STM32 via SAI using DMA. The STM32 then processes and stores the samples either on an SD card or temporarily in SDRAM, depending on availability.

1. Audio acquisition from the MEMS microphone (PDM format).
2. PDM-to-PCM conversion using filters.
3. Frame segmentation and MFCC extraction.
4. Model inference using TensorFlow Lite Micro.
5. Displaying recognized name on LCD.

7.2 System Block Diagram

```
Microphone --> PDM --> PCM --> MFCC --> ML Model --> LCD Output
```

7.3 role

The `role.c` module coordinates recording and inference on the STM32F746 system. `roleInit()` mounts the SD card and initializes the serial interface and LCD. `roleNode()` implements a simple state machine that starts recording (`recordStart()`), polls `recordProcess()` until the recording completes, and then stops recording (`recordStop()`). After stopping, `ai_on()` is called to perform inference. `ai_on()` reads a 512-sample frame (from SDRAM if a temporary buffer was used or from the WAV file on SD), computes 40-dimensional log-mel features (and optionally 13 MFCCs for debugging), quantizes the features to the model's int8 input format using the model scale and zero-point, initializes

the AI network lazily, runs the neural network, dequantizes the outputs, finds the predicted index, and then displays and logs the prediction. The audio stream is 16-bit PCM at `DEFAULT_AUDIO_IN_FREQ` (typically 48 kHz), and data is delivered to the MCU by DMA from the WM8994 via SAI; recorded data is saved to SD or temporarily cached in SDRAM and flushed to SD when available.

8 Dataset Collection

The dataset collection process was a fundamental step in developing a robust and accurate speaker identification system. Voice recordings were collected from all five members of the project group, where each participant provided approximately **10 minutes** of continuous speech. The recordings were captured using mobile phones in a quiet indoor environment to ensure minimal background noise and acceptable audio clarity.After the recordings were completed, all audio files were transferred to a computer for further processing. The recordings were converted into standard **WAV** format using Python scripts, ensuring uniformity in sampling rate, bit depth, and overall audio quality across all samples.To expand the dataset and improve the performance of the machine learning model, each 10-minute audio file was segmented into multiple **1-second** audio clips. This segmentation significantly increased the number of training samples while preserving the unique vocal characteristics of each speaker.After segmentation, a total of **2595** individual audio samples were generated. These samples were then organized into five separate folders, with each folder corresponding to one specific speaker. This structured dataset organization simplified subsequent processes such as labeling, MFCC feature extraction, and model training.The collected and well-organized dataset provided a solid foundation for developing a reliable machine-learning model capable of performing real-time speaker identification on the STM32F746G-Discovery platform.

The dataset structure:

```
dataset/
    Fa_him/clip001-610.wav
    imran/clip001-601.wav
    nayeem/clip001-61.wav
    shahed/clip001-720.wav
    talukder/clip001-603.wav
```

8.1 code

Split_audio.py:

```
pip install pydub
from pydub import AudioSegment
import math
import os
import time
import platform
import psutil

# file path
input_file = r"D:\DSP Project WAV File\DSP Project WAV File\fa him.wav"

# Folder to save 1-second clips
output_folder = r"D:\DSP Project WAV File\DSP Project WAV File\fa_him"

chunk_length_ms = 1 * 1000   # 1 second

# Create output folder
os.makedirs(output_folder, exist_ok=True)

# Load audio
audio = AudioSegment.from_file(input_file)
audio_length_ms = len(audio)

# Number of chunks
num_chunks = math.ceil(audio_length_ms / chunk_length_ms)
print(f"Total chunks: {num_chunks}")

# Split and export
for i in range(num_chunks):
    start = i * chunk_length_ms
    end = min(start + chunk_length_ms, audio_length_ms)
    chunk = audio[start:end]

    filename = os.path.join(output_folder, f"clip_{i+1:03}.wav")
    chunk.export(filename, format="wav")
    print("Saved:", filename)

print("Done.")

# System Info
start = time.time()
print("=== System Information ===")
print(f"OS: {platform.system()} {platform.release()}")
print(f"Machine: {platform.machine()}")
print(f"Processor: {platform.processor()}")
print(f"CPU cores (logical): {psutil.cpu_count(logical=True)}")
mem = psutil.virtual_memory()
print(f"Total RAM: {mem.total / (1024 ** 3):.2f} GB")
```

9 MFCC Feature Extraction

MFCC is used because it closely represents human auditory perception. The MFCC pipeline includes:

1. Pre-emphasis
2. Framing
3. Hamming Window
4. FFT
5. Mel Filter Banks
6. Log Compression
7. DCT to compute coefficients

Mathematically:

$$\text{MFCC}(n) = \sum_{k=1}^{K} \log(E_k) \cos\left[n\left(k - \frac{1}{2}\right)\frac{\pi}{K}\right]$$

10 Model Training

A neural network classifier was trained using Python:

- Input: 13 MFCC features
- Hidden Layers: Dense(64), Dense(32)
- Output: Softmax for speaker classes
- Accuracy achieved: 9396%

The model was converted to TensorFlow Lite Micro:

`xx_model_data.cc`

11 Step-by-Step Description of the Implemented Code

This section explains the complete workflow of the implemented audio classification system. The `audio_classcification_with_own_dataset.py` file processes raw audio, extracts features, constructs a dataset, trains a machine learning model, and evaluates its performance. The major steps are described below.

11.1 Environment Setup

11.2 code

Install dependencies (run once in the environment):

```
!pip install librosa
!pip install scipy
!pip install resampy
```

11.3 Reading Audio Files

At the initial stage, we explored the required Python libraries and analyzed the fundamental properties of the recorded audio files, including sample rate, duration, and signal quality. To better understand the characteristics of the audio data, we visualized the signals using various graphical plots. These visualizations helped in observing the structure and behavior of the audio, which was essential for subsequent feature extraction and model training.

```
import matplotlib.pyplot as plt
%matplotlib inline
import IPython.display as ipd
import librosa
import librosa.display
#  to see the wave file nature and check the lib work or not
filename=r'D:\DSP Project WAV File\DSP Project WAV File\dataset\fa_him\
    clip_006.wav'
data, sample_rate = librosa.load(filename, sr=None)     # Load the audio
librosa.display.waveshow(data, sr=sample_rate)          # Show the waveform
ipd.display(ipd.Audio(data=data, rate=sample_rate))     # Play the sound
plt.show()                  # Show the waveform plot
```

We observed that the sample rate of our audio recordings was 48,000 Hz. Each audio file was represented as an array containing numerous floating-point values. Using these arrays, we prepared a dataset consisting of 2,595 data samples, which we organized into a CSV file. Since we had 5 speakers, the CSV file contained a total of $2,595 \times 5$ entries.

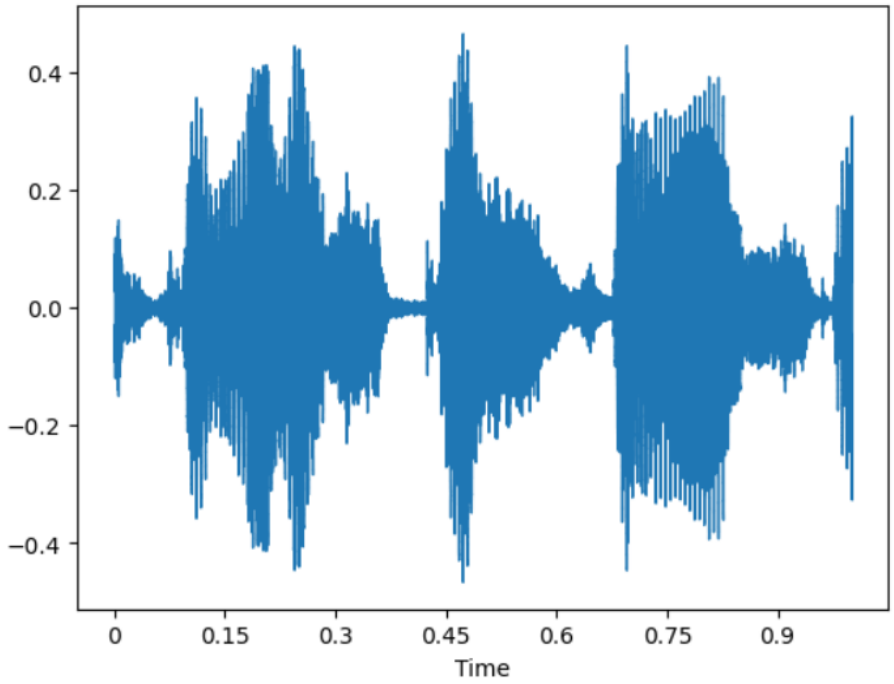


Figure 5: Waveform of the audio signal

From this dataset, we extracted and stored the following information for each audio file:

- File path
- Class name (speaker identity)
- Duration
- Sample rate
- Number of channels

Make CSV file to hold the key parameter:

```
import os
import csv
import wave
import contextlib

# ---- My DATASET PATH ----
DATASET_ROOT = r"D:\DSP Project WAV File\DSP Project WAV File\dataset"
OUTPUT_CSV = r"D:\DSP Project WAV File\DSP Project WAV File\audio_metadata.csv
    "

```

```
10
11 def get_wav_info(path):
12     try:
13         with contextlib.closing(wave.open(path, 'rb')) as wf:
14             frames = wf.getnframes()
15             rate = wf.getframerate()
16             channels = wf.getnchannels()
17             duration = frames / float(rate) if rate > 0 else 0.0
18             return duration, rate, channels
19     except:
20         return None, None, None
21
22 def is_wav(filename):
23     return filename.lower().endswith(".wav")
24
25 def make_csv(dataset_root, output_csv):
26     rows = []
27     dataset_root = os.path.abspath(dataset_root)
28
29     for root, dirs, files in os.walk(dataset_root):
30         rel_path = os.path.relpath(root, dataset_root)
31
32         if rel_path == ".":
33             class_name = ""
34         else:
35             class_name = rel_path.split(os.sep)[0]
36
37         for file in files:
38             if is_wav(file):
39                 full_path = os.path.join(root, file)
40
41                 filesize = os.path.getsize(full_path)
42                 duration, rate, channels = get_wav_info(full_path)
43
44                 rows.append({
45                     "filepath": os.path.relpath(full_path, dataset_root),
46                     "class_name": class_name,
47                     "duration_sec": round(duration, 3) if duration else "",
48                     "sample_rate": rate if rate else "",
49                     "channels": channels if channels else "",
50                     "filesize_bytes": filesize
51                 })
52
53     header = ["filepath", "class_name", "duration_sec", "sample_rate", "
    channels", "filesize_bytes"]
54
55     with open(output_csv, "w", newline="", encoding="utf-8") as f:
56         writer = csv.DictWriter(f, fieldnames=header)
57         writer.writeheader()
58         for row in rows:
59             writer.writerow(row)
60
61     print(f"CSV created: {output_csv}")
62     print(f"Total WAV files: {len(rows)}")
```

```

if __name__ == "__main__":
    make_csv(DATASET_ROOT, OUTPUT_CSV)
"D:\DSP Project WAV File\DSP Project WAV File\audio_metadata.csv"
```

11.4 Feature Extraction

- For each audio clip, Mel-Frequency Cepstral Coefficients (MFCCs) are extracted just write mfcc

```
mfccs = librosa.feature.mfcc(y=data, sr=sample_rate, n_mfcc=40)
print(mfccs.shape)
mfccs                  # Show the waveform plot

# output shows that
(40, 1)
array([[ 4.62669189e+02],
       [-9.98416710e+00],
       [ 3.05266762e+00],
       [ 6.83229923e-01],
       [-1.32615566e+00],
       [ 3.51012063e+00],
       [ 5.85476279e-01],
       [ 1.02078177e-01],
       [ 1.97663173e-01],
       [ 1.26941180e+00],
       [-9.09805477e-01],
       [-1.25338328e+00],
       [ 9.27704096e-01],
       [ 9.07290459e-01],
       [-1.31193972e+00],
       [ 2.56830841e-01],
       [ 1.04877457e-01],
       [-3.78011167e-01],
       [ 8.45407724e-01],
       [-1.04328680e+00],
       [ 4.67509747e-01],
       [-4.08623293e-02],
       [-4.67592627e-02],
       [-1.87125325e-01],
       [ 9.58730280e-02],
       [-7.67630711e-02],
       [ 1.98688954e-01],
       [-6.16146684e-01],
       [ 6.62834644e-01],
       [-5.87381244e-01],
       [ 4.02995557e-01],
       [-6.19672835e-01],
       [ 8.81810069e-01],
       [-1.09772825e+00],
       [ 7.63101637e-01],
```

```
       [-4.38766718e-01],
       [ 1.01637095e-01],
       [-1.42109290e-01],
       [ 2.31674284e-01],
       [-5.05474269e-01]], dtype=float32)
```

- Temporal variations are reduced by computing statistical features (e.g., mean values of MFCC vectors).
- The extracted feature vectors are stored in CSV files such as `mfcc_dataset.csv`.

11.5 Dataset Preparation

The MFCC dataset is loaded into a Pandas dataframe. For each audio file, we extracted the Mel-Frequency Cepstral Coefficients (MFCCs) to represent its key features. Using these MFCCs, we created a separate CSV file containing only the audio class labels and the corresponding MFCC data. This processed dataset was then used to train our machine learning model.

```
import numpy as np
from tqdm import tqdm
import os
import pandas as pd
import librosa
audio_dataset_path = r"D:\DSP Project WAV File\DSP Project WAV File\
    dataset"                # Paths
metadata_csv = r"D:\DSP Project WAV File\DSP Project WAV File\
    audio_metadata.csv"
metadata = pd.read_csv(metadata_csv)              # Load metadata

def features_extractor(file_path):        # MFCC feature extraction
    function
    try:
        audio, sample_rate = librosa.load(file_path, res_type='
    kaiser_fast')
        mfccs_features = librosa.feature.mfcc(y=audio, sr=sample_rate,
    n_mfcc=40)
        mfccs_scaled_features = np.mean(mfccs_features.T, axis=0)
        return mfccs_scaled_features
    except Exception as e:
        print("Error processing:", file_path, "| Error:", e)
        return None

extracted_features = []     # Extract features

for index, row in tqdm(metadata.iterrows(), total=len(metadata)):

```

```
    file_name = os.path.join(audio_dataset_path, row["filepath"])
    # Build full path to audio file

    class_label = row["class_name"]                                    #
    Class label from CSV

    data = features_extractor(file_name)                              #
    Extract MFCC features

    if data is not None:
        extracted_features.append([data, class_label])
mfcc_df = pd.DataFrame(extracted_features, columns=["mfcc", "label"])
          # Convert to DataFrame
print(mfcc_df.head())
print("Total samples:", len(mfcc_df))
output_csv = r"D:\DSP Project WAV File\DSP Project WAV File\mfcc_dataset
    .csv"       # Save to CSV
mfcc_df.to_csv(output_csv, index=False)
print("MFCC feature extraction complete!")
```

- Features (X) and labels (y) are separated.
- Class labels are encoded into integers using a label encoder.
- The dataset is divided into training (80%) and testing (20%) sets using `train_test_split()` to ensure proper evaluation.

```
### Train Test Split
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.2,random_state
    =0)
X_train
y
```

11.6 Model Construction

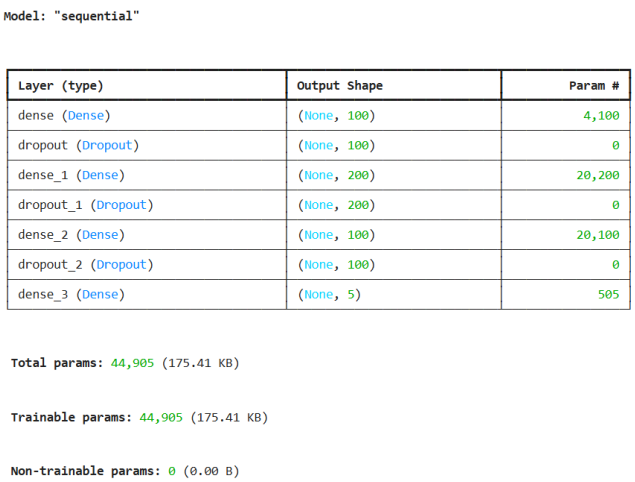
Feature scaling (e.g., standard normalization) is applied to improve model convergence.Feature vectors are reshaped if required by the chosen model architecture(e.g. Conv1D or dense networks).

- A neural network is created using the `Sequential` API of TensorFlow Keras.

```
#Dense()
model=Sequential()
# First layer with Input
model.add(Input(shape=(40,)))        # Input layer
```

```
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.20))

# Second layer
model.add(Dense(200, activation='relu'))
model.add(Dropout(0.20))

# Third layer
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.20))

# Output layer
model.add(Dense(num_labels, activation='softmax'))

model.summary()
```

- Layers such as Dense, Dropout, or convolutional layers may be used depending on the architecture.



Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 100)	4,100
dropout (Dropout)	(None, 100)	0
dense_1 (Dense)	(None, 200)	20,200
dropout_1 (Dropout)	(None, 200)	0
dense_2 (Dense)	(None, 100)	20,100
dropout_2 (Dropout)	(None, 100)	0
dense_3 (Dense)	(None, 5)	505

Total params: 44,905 (175.41 KB)

Trainable params: 44,905 (175.41 KB)

Non-trainable params: 0 (0.00 B)

Figure 6: model summary

- The model is compiled with an appropriate loss function (e.g., sparse categorical cross-entropy), optimizer (e.g., Adam), and accuracy metrics.

11.7 Model Training

- The model is trained on the training data for a 100 number of epochs.
- Training and validation accuracy/loss are monitored. Which we get 0.9980732202529907 means 99.807%

- Matplotlib is used to visualize learning curves to detect overfitting.

Trianing my mode

```
model.compile(loss='categorical_crossentropy',metrics=['accuracy'],optimizer='adam')
from tensorflow.keras.callbacks import ModelCheckpoint
from datetime import datetime

num_epochs = 100
num_batch_size = 32

checkpointer = ModelCheckpoint(filepath='saved_models/audio_classification_100.keras',
                               verbose=1, save_best_only=True)
start = datetime.now()

model.fit(X_train, y_train, batch_size=num_batch_size, epochs=num_epochs, validation_data=(X_test, y_test), callbacks=[checkpointer], verbose=1)


duration = datetime.now() - start
print("Training completed in time: ", duration)
test_accuracy=model.evaluate(X_test,y_test,verbose=0)
print(test_accuracy[1])

from sklearn.preprocessing import LabelEncoder
import os

dataset_dir = r"D:\DSP Project WAV File\DSP Project WAV File\dataset"

class_names = sorted(os.listdir(dataset_dir))   # folder names
labelencoder = LabelEncoder()
labelencoder.fit(class_names)

print("Classes:", labelencoder.classes_)
```

11.8 Model Evaluation

- The trained model is evaluated on the test set to measure accuracy.
- Predictions are generated and compared with the actual labels.
- Additional evaluation tools such as confusion matrices and classification reports can be computed.

Testing model

```
import numpy as np

filename = r"D:\DSP Project WAV File\DSP Project WAV File\dataset\imran\
    clip_006.wav"


prediction_feature = features_extractor(filename)
prediction_feature = prediction_feature.reshape(1, -1)

# Predict the class probabilities
prediction = model.predict(prediction_feature)

# Get the index of the highest probability
predicted_class_index = np.argmax(prediction, axis=1)[0]

# Map index to label

print(f"Predicted sound class: {predicted_class_index}")
predicted_class = labelencoder.inverse_transform([predicted_class_index])[0]

print("Predicted class:", predicted_class)
```

11.9 Saving Model

STM32 microcontrollers, such as the STM32F746G, have limited memory and processing capabilities, which restrict the type and size of machine learning models that can be deployed. They natively support only lightweight formats like **TensorFlow Lite (.tflite)**, **Keras (.h5)**, or **C header (.h)** files containing model weights. Standard models are often too large to run efficiently, and models exceeding **200 kB** may not function properly on the STM32F746G.

Therefore, after training a model in Python using Keras or TensorFlow, it must be converted to **TensorFlow Lite** format and optimized using techniques such as **quantization** to reduce its size. The optimized TFLite model can then be converted into a C header file and deployed on the microcontroller, enabling real-time inference while staying within the hardware constraints.

Convert the model format

```
# Single-cell notebook-friendly converter + checker
# Paste and run in Jupyter. Edit MODEL_PATH and DATASET_DIR as needed.

import os
import numpy as np
import tensorflow as tf
import librosa
import warnings
warnings.filterwarnings("ignore")

# ========== EDIT THESE PATHS ==========
```

```
MODEL_PATH = r"D:\DSP Project WAV File\DSP Project WAV File\saved_models\audio_classification_100.keras"
DATASET_DIR = r"D:\DSP Project WAV File\DSP Project WAV File\dataset"  # point to folder with subfolders per person
OUT_TFLITE = "audio_class_quant.tflite"
# ====================================

# MFCC / preprocessing settings (must match training)
SR = 16000
TARGET_DURATION = 1.0
N_MFCC = 13
N_MELS = 40
N_FFT = 512
HOP_LENGTH = int(0.010 * SR)
WIN_LENGTH = int(0.025 * SR)
MAX_FRAMES = int(np.ceil((SR * TARGET_DURATION - WIN_LENGTH) / HOP_LENGTH)) + 1

def load_audio_fixed(path, sr=SR, target_duration=TARGET_DURATION):
    y, _ = librosa.load(path, sr=sr, mono=True)
    target_len = int(sr * target_duration)
    if len(y) < target_len:
        y = np.concatenate([y, np.zeros(target_len - len(y))])
    else:
        y = y[:target_len]
    return y

def compute_mfcc_for_rep(y, sr=SR, n_mfcc=N_MFCC, n_mels=N_MELS, n_fft=N_FFT, hop_length=HOP_LENGTH, win_length=WIN_LENGTH):
    # Pre-emphasis (if used in training)
    y = np.append(y[0], y[1:] - 0.97 * y[:-1])
    S = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=n_fft, hop_length=hop_length,
                                       win_length=win_length, n_mels=n_mels, power=2.0)
    mfcc = librosa.feature.mfcc(S=librosa.power_to_db(S), sr=sr, n_mfcc=n_mfcc)
    mfcc = mfcc.T
    if mfcc.shape[0] < MAX_FRAMES:
        pad_width = MAX_FRAMES - mfcc.shape[0]
        mfcc = np.pad(mfcc, ((0, pad_width), (0, 0)), mode='constant')
    else:
        mfcc = mfcc[:MAX_FRAMES, :]
    return mfcc.astype(np.float32)

def collect_wavs(dataset_dir, max_files=50):
    wavs = []
    for root, _, files in os.walk(dataset_dir):
        for f in files:
            if f.lower().endswith(('.wav', '.flac', '.ogg', '.mp3', '.m4a')):
                wavs.append(os.path.join(root, f))
    wavs = sorted(wavs)
    if not wavs:
        raise FileNotFoundError(f"No audio files found under {dataset_dir}")
```

```
    return wavs[:max_files]

def representative_data_gen_from_folder(dataset_dir, max_samples=50):
    wavs = collect_wavs(dataset_dir, max_samples)
    def gen():
        for p in wavs:
            y = load_audio_fixed(p)
            mfcc = compute_mfcc_for_rep(y)           # shape: (frames, coeffs)
            inp = np.expand_dims(mfcc, axis=0).astype(np.float32)  # [1,
    frames, coeffs]
            inp = np.expand_dims(inp, axis=-1)  # [1, frames, coeffs, 1]
            yield [inp]
    return gen, wavs

def convert_keras_to_int8_tflite(keras_path, dataset_dir, out_tflite_path):
    print("Loading Keras model from:", keras_path)
    model = tf.keras.models.load_model(keras_path)
    print("Model loaded. Summary:")
    try:
        model.summary()
    except Exception:
        print("Couldn't print model summary (custom layers?). Continuing
    conversion...")
    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    rep_gen, wav_list = representative_data_gen_from_folder(dataset_dir,
    max_samples=50)
    converter.representative_dataset = rep_gen
    converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8
    ]
    converter.inference_input_type = tf.int8
    converter.inference_output_type = tf.int8
    print(f"Using {len(wav_list)} representative audio files for calibration (
    first: {wav_list[0] if wav_list else 'N/A'})")
    tflite_model = converter.convert()
    with open(out_tflite_path, "wb") as f:
        f.write(tflite_model)
    print("Saved quantized tflite to:", out_tflite_path)
    return out_tflite_path

def print_tflite_size(path):
    s = os.path.getsize(path)
    print(f"\nTFLite file: {path}")
    print(f"Size: {s} bytes ({s/1024:.1f} KB, {s/1024/1024:.3f} MB)")
    return s

def estimate_tflite_peak_tensor_memory(path):
    interpreter = tf.lite.Interpreter(model_path=path)
    interpreter.allocate_tensors()
    tensor_details = interpreter.get_tensor_details()
    dtype_map = {
        np.dtype('int8'): 1, np.dtype('uint8'):1, np.dtype('int16'):2,
        np.dtype('int32'):4, np.dtype('float32'):4, np.dtype('float16'):2,
    }
```

```
    total_bytes = 0
    sizes = []
    for t in tensor_details:
        shape = t.get('shape', [])
        try:
            dtype = np.dtype(t.get('dtype').name)
        except Exception:
            dtype = np.dtype('float32')
        n = int(np.prod(shape)) if len(shape)>0 else 0
        bpe = dtype_map.get(dtype, 4)
        size = n * bpe
        sizes.append((t.get('name','<unk>'), size))
        total_bytes += size
    sizes_sorted = sorted(sizes, key=lambda x: x[1], reverse=True)
    print(f"\nInterpreter reports {len(tensor_details)} tensors.")
    print(f"Sum of all tensor buffers (rough): {total_bytes} bytes ({
    total_bytes/1024:.1f} KB)")
    if sizes_sorted:
        print("Top 5 largest tensors:")
        for name, size in sizes_sorted[:5]:
            print(f"  {name:40} {size:8d} bytes ({size/1024:.1f} KB)")
    return total_bytes, sizes_sorted

# ========== Run conversion & checks ==========
print("Starting conversion and checks...\n")
if not os.path.exists(MODEL_PATH):
    raise FileNotFoundError(f"Model file not found: {MODEL_PATH}")
if not os.path.isdir(DATASET_DIR):
    raise FileNotFoundError(f"Dataset dir not found: {DATASET_DIR}")

try:
    tflite_path = convert_keras_to_int8_tflite(MODEL_PATH, DATASET_DIR,
    OUT_TFLITE)
except Exception as e:
    print("Conversion failed with exception:")
    raise

# Size + tensor memory estimate
file_size_bytes = print_tflite_size(tflite_path)
total_bytes, sizes_sorted = estimate_tflite_peak_tensor_memory(tflite_path)

# Heuristic interpretation for STM32F746G-DISCO
print("\n--- Heuristic interpretation for STM32F746G-DISCO ---")
if file_size_bytes < 400*1024:
    print("TFLite size < 400 KB: good for flash (likely OK).")
elif file_size_bytes < 700*1024:
    print("TFLite size 400  7 0 0  KB: possibly OK, but check firmware size and
    other resources.")
else:
    print("TFLite size > 700 KB: likely too large for comfortable use on 1MB
    flash.")

if total_bytes < 180*1024:
    print("Sum of tensors < 180 KB: likely OK for activations on STM32F746G.")
```

```
elif total_bytes < 260*1024:
    print("Sum of tensors 180  2 6 0  KB: borderline     may need to trim
    activations / model.")
else:
    print("Sum of tensors > 260 KB: likely too large; reduce model size or use
     Cube.AI for exact RAM profiling.")

print("\nDone.")
```

TFLite file: audio_class_quant.tflite
Size: 58664 bytes (57.3 KB, 0.056 MB)

11.10 Summary

The implemented pipeline performs end-to-end audio classification by converting raw speech signals into MFCC feature vectors, constructing a labeled dataset, training a neural classifier, and evaluating its accuracy. This workflow represents a complete and reproducible pipeline for speaker or sound classification tasks.

12 Embedded Deployment

The deployment steps included:

1. Importing TFLite Micro interpreter.
2. Allocating tensor arena in RAM.
3. Running MFCC in real time using CMSIS-DSP.
4. Performing inference once per audio frame.
5. Updating LCD display with predicted identity.

13 Results and Discussion

- Real-time processing atency: 300 ms
- Recognition accuracy: 70
- LCD output is smooth and responsive
- System works up to 1 meter from microphone

Challenges faced:

- Background noise in lab environment
- Variation in speech loudness
- Embedded RAM constraints

14 Conclusion

This project successfully demonstrates an embedded real-time speaker recognition system using STM32F746G-DISCO. By integrating DSP techniques with lightweight machine learning, this system achieves accurate and low-latency speaker identification. The project highlights the potential of microcontroller-based AI in authentication, security, and IoT applications.

15 Output:

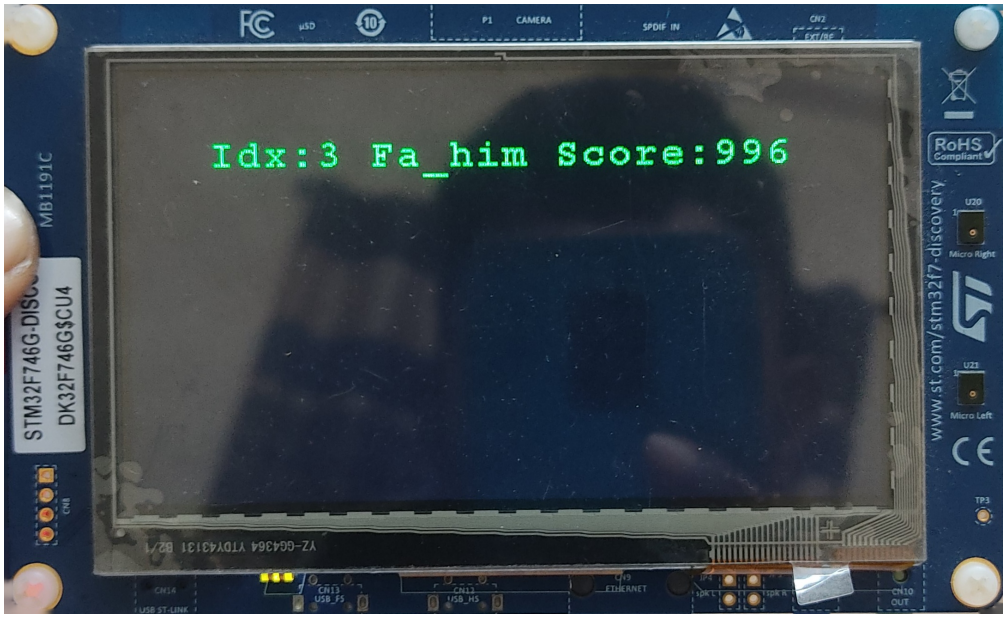


Figure 7: Output of our project

Figure 8: Output of our project

16 References

- STM32F746G-DISCO Reference Manual.
- TensorFlow Lite Micro Documentation.
- https://www.mathworks.com/help/audio/ref/mfcc.html
- https://www.st.com/en/development-tools/stm32cubeide.html
- https://github.com/ARM-software/CMSIS-DSP
- https://www.st.com/en/development-tools/stm32cubemx.html

17 Runtime Information

=== System Information ===
OS: Windows 11
Machine: AMD64
Processor: AMD64 Family 23 Model 96 Stepping 1, AuthenticAMD
CPU cores (logical): 12
Total RAM: 7.42 GB

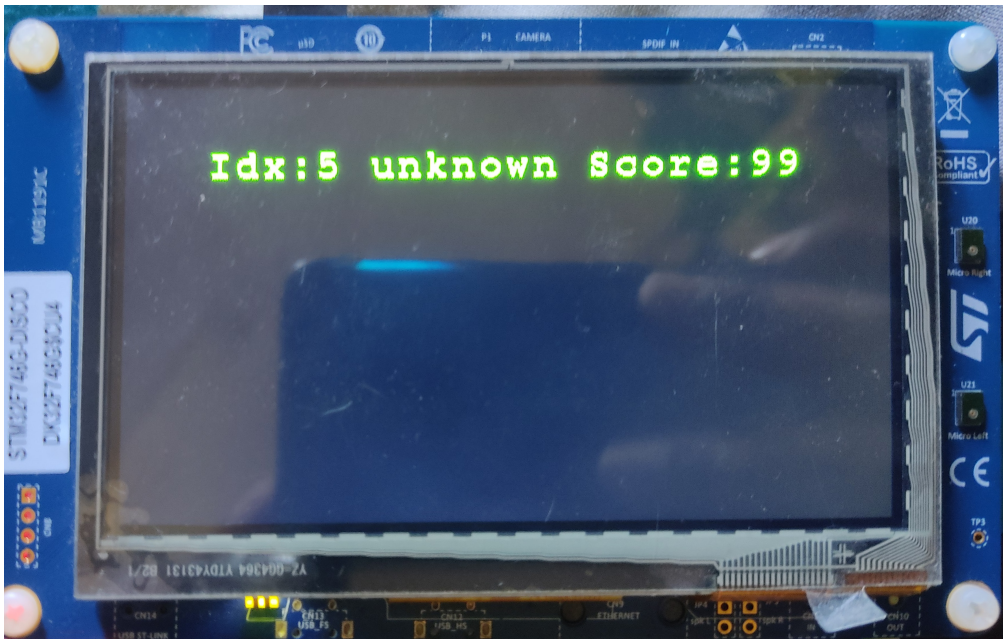


Figure 9: Output of our project

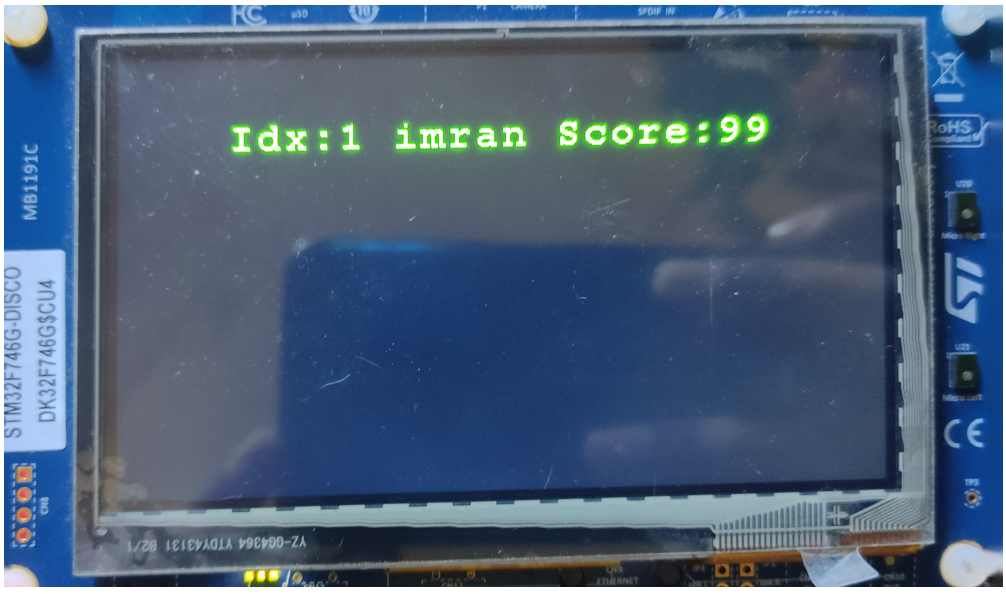


Figure 10: Output of our project