Karaoke Machine

Name	Giurgiu Andrei-Ștefan
Group	335CA

Introduction

The project consists of creating a karaoke mechanism, using a speaker for the instrumental and an LCD screen for the lyrics. The user can choose from a preset selection of songs, loaded on a microSD card. The songs which are listed at the beginning on the screen, can be selected using buttons. The speaker volume can also be adjusted using a potentiometer.

The goal of the project is to create a relaxing and fun atmosphere for the users.

General description

The **ESP32** is the main component of the project, since it will be in charge of reading data from the card and send the audio signal to an amplifier using the I2S protocol.

The **MAX88357A amplifier** will take the digital signal from the ESP32 and transform it into a strong analogue signal. After this the sound will be sent to the speaker.

The main storage component is a **16GB microSD card**, formatted as Fat32. The instrumental files are stored in the `.mp3` format and the lyrics files in the `.lrc` format, which is a text format with timestamps before the text, so that syncing with the instrumental is easier.

The user interface is made of three classes of components: the **20×4 LCD display** with an I2C interface, **the volume adapter** and **the buttons**. The display is a simple LCD with an I2C module attached, which at the beginning will show the available songs on the device. Using the buttons, the user can select a song and play it.

Block Scheme

Hardware Design

The list of components used is:

- ESP32-WROOM-32D Microcontroller
- MAX98357A Amplifier
- MB102 Power Module
- LCD Display with PCF8574 I2C module
- microSDHC Card
- 40mm 3W Speaker
- MicroSDHC Reader Module
- Potentiometer
- Buttons
- 2x Breadboard

Component Pinout

Component	Pin Name	Connected To	Reason

ESP32-DEVKITC-32D	3V3	MB102 (VCC2 @ 3.3V) & Potentiometer (Top Pin)	Receives stable 3.3V power from the external supply (standalone mode) and provides the 3.3V reference voltage for the volume potentiometer.
	IO34	Potentiometer (Wiper)	ADC input to read the analog voltage (0-3.3V) to adjust the software volume.
	IO27	PLAY/PAUSE BUTTON	GPIO input with internal pull-up activated to control the playback state.
	IO14	UP BUTTON	GPIO input with internal pull-up activated to navigate up in the song list.
	IO13	DOWN BUTTON	GPIO input with internal pull-up activated to navigate down in the song list.
	GND	Common GND	Essential common ground reference for all power and data signals across the entire system.
	5V	Not Connected	Left disconnected because the ESP32 is powered directly via the 3.3V rail from the MB102.
	IO23	Micro SDHC reader (MOSI)	SPI Master Out Slave In: Sends read commands to the SD card.
	IO22	PCF8574 (SCL)	I2C Clock line to synchronize data transfers to the LCD.
	IO21	PCF8574 (SDA)	I2C Data line to display menus and lyrics on the LCD.
	IO19	Micro SDHC reader (MISO)	SPI Master In Slave Out: Stream audio (.mp3) and lyrics (.lrc) data to the ESP32.
	IO18	Micro SDHC reader (SCK)	SPI Clock line generated by the ESP32 to sync SD card communication.
	IO5	Micro SDHC reader (CS)	SPI Chip Select: Enables the SD card reader during data transfers.
	IO16	MAX98357A (DIN)	I2S Data Line (DOUT): Sends the raw digital audio stream to the amplifier.
	IO26	MAX98357A (BCLK)	I2S Bit Clock: Dictates the timing and speed of individual audio data bits.
	IO25	MAX98357A (LRC)	I2S Left/Right Clock (Word Select): Syncs the audio channels for the DAC.
MB102 Power Supply	VCC1 (Set to 5V)	MAX98357A (Vin), PCF8574 (VCC), Micro SDHC reader (VCC)	Provides heavy-duty 5V power to power-hungry and 5V-logic components to ensure maximum volume, clear LCD contrast, and stable SD card reads.
	VCC2 (Set to 3.3V)	ESP32 (3V3)	Supplies a clean, regulated 3.3V directly to the ESP32 power rail for safe, standalone operation.
	GND	Common GND	Central grounding hub for the breadboard circuit.

MAX98357A	LRC	ESP32 (IO25)	Receives the audio channel sync timing.
	BCLK	ESP32 (IO26)	Receives the digital audio bit timing.
	DIN	ESP32 (IO16)	Receives the digital I2S audio data stream.
	GAIN	Common GND	Anchors the pin to GND to set a 12dB gain and prevents it from floating like an antenna, **eliminating background hiss**.
	SD	Not Connected	Left floating to keep the amplifier always active.
	GND	Common GND	Completes the power loop and provides audio ground reference.
	Vin	MB102 (VCC1 @ 5V)	Power input to drive the audio speaker at full efficiency.
	+	Speaker (+)	Positive terminal for the amplified analog audio output.
	-	Speaker (-)	Negative terminal for the amplified analog audio output.
PCF8574	VCC	MB102 (VCC1 @ 5V)	Power supply for the I2C backpack and the LCD backlight.
	GND	Common GND	Completes the power circuit for the display interface.
	SDA	ESP32 (IO21)	I2C data interface for text communication.
	SCL	ESP32 (IO22)	I2C clock interface for text communication.
Micro SDHC reader	CS	ESP32 (IO5)	Listens for the SPI activation signal from the microcontroller.
	SCK	ESP32 (IO18)	Receives the SPI bus clock signal.
	MOSI	ESP32 (IO23)	Receives instructions from the ESP32.
	MISO	ESP32 (IO19)	Transmits the requested files back to the processor.
	VCC	MB102 (VCC1 @ 5V)	Safe 5V source to handle transient current spikes during SD card read operations.
	GND	Common GND	Completes the power loop for the reader module.
Potentiometer	Top Pin	ESP32 (3V3)	Connects to the safe 3.3V rail to set the upper limit of the voltage divider.
	Wiper	ESP32 (IO34)	Delivers a variable voltage (0-3.3V) to the ADC pin based on the knob position.
	Bottom Pin	Common GND	Connects to ground to set the lower limit of the voltage divider.
PLAY/PAUSE BUTTON	Pin 1	ESP32 (IO27)	Interconnects the button to the hardware interrupt pin on the ESP32.
	Pin 2	Common GND	Pulls the GPIO to LOW when pressed, fighting the internal pull-up resistor to register a click (active-LOW logic).
UP BUTTON	Pin 1	ESP32 (IO14)	Interconnects the button to the track navigation system.
	Pin 2	Common GND	Pulls the GPIO to LOW when pressed, fighting the internal pull-up resistor (active-LOW logic).

DOWN BUTTON	Pin 1	ESP32 (IO13)	Interconnects the button to the track navigation system.
	Pin 2	Common GND	Pulls the GPIO to LOW when pressed, fighting the internal pull-up resistor (active-LOW logic).
Speaker	+	MAX98357A (+)	Connects to the positive terminal of the audio transducer.
	-	MAX98357A (-)	Connects to the negative terminal of the audio transducer.

Electrical Scheme

Software Design

This project will be developed using the PlatformIO extension in Visual Studio Code. I will use Arduino for the main framework. Other libraries that will be included are:

- **ESP32-AudioI2S**, this library abstracts the interaction with the amplifier via I2S and the decoding of the `.mp3` files
- **LiquidCrystal_I2C**, this library abstracts the interaction with the LCD display's control module, PCF8574

In order to not have glitching or interrupting music, I decided to use **FreeRTOS** tasks for my project. There will be three main tasks:

- The UI task, where the user can interact with the system by choosing a song, pausing, playing a stop and adjusting the volume.
- The lyric task, where the text is sent to the LCD at the correct timestamps.
- The audio task, which will be the task with the highest priority, since sound quality is crucial.

Since some parts of the system that are handled by the UI task affect the audio and lyric tasks, I decided to facilitate inter-task communication using **FreeRTOS queues**. These are thread-safe buffers where I store information like if the track must be paused, the volume must be changed or what song to play. Since performance is a huge factor, each thread checks the queue in a non-blocking manner to see whether the other threads pushed any updates.

The program starts by setting up the Serial (which will not be used in this project, but still there for debugging), I2C for the LCD and the SPI for the SD card. After that, the SD card is scanned to see which songs are available. This is done in the `scan_sd_for_songs()` function.

UI Task

- This task initialises the buttons, that will be on active-low logic, and handles all the user interface

facilities.
- Those are:
  - Listing available songs and playing the selected one.
  - Allowing a circular iteration between the available songs that can be played.
  - Checking if the potentiometer was actioned to modify the volume.
  - Pausing/Resuming the current playing track.
- For debouncing, I decided to use the FreeRTOS delay mechanism, which basically yields the core for the specified interval of time (30 ms).
- The LCD will have > cursor in front of the song that is currently selected to be played.
- In order to optimize reading the analog pin where the potentiometer is connected, I check the potentiometer value once every 100ms (the UI task runs once every 10ms and I read it once in 10 runs).
- Each time NEXT and PREV button are pressed, the LCD updates the cursor and the menu.
- When the user hits the SELECT button, the lyric file is fully loaded in memory and parsed. This is done so that the SPI bus won t be a bottleneck when reading the audio file.

LRC parser

- In order to sync the lyrics and the audio, I need to know at which timestamp a lyric needs to be pushed to the display.
- I created a struct LyricLine, which contains an unsigned int which keeps the timestamp of the lyric in ms and a char array where the actual lyric is stored.
- The main logic of the parser is in the `parse_lrc` function, which firstly checks if there is an `[offset:]` attribute at the start. If there is, it means that the lyric timestamps from the file aren't fully in sync with the instrumental, so it needs to be edited.
- The format of a timestamp for all the files is [mm:ss.ms].
- The parser reads this format and converts everything to ms, adding the offset as well and saving everything in an array of type LyricLine.

Lyric Task

- This task will run once every 50ms.
- If a new song just started, all the lyric state variables are reset.
- If the user still browses the menu, only the lyric positions are reset, to not show any lyrics while the user decides what song to listen to.
- One smart feature of this task is how synchronization is done. I rely on two time functions: one from the audio library (`getAudioCurrentTime`) and one from the FreeRTOS system tick for that specific iteration, using the `xTaskGetTickCount` function.
- getAudioCurrentTime only returns the time in seconds and for a nice UX I need millisecond precision. For this specific reason, I keep the moment of time when a full second passed, in the `last_sync_tick` variable. This is where the `now_tick` variable comes to play. This variable is the key to have milisecond precision, since it tells me how many milliseconds passed since the last update. This is because it uses the FreeRTOS clock. In order to see how much time has passed since the last full second of the song started, I substract from the `now_tick` variable the

`last_sync_tick` variable.
- After that, I sequentially iterate through the `current_lyrics` array to see if I need to update the text on the LCD.
- When the track is paused, the reference second time is equal to the current time to not lose the synchronization.
- When the song ends, the lyric state variables are reset to show the menu UI.

Audio Task

- The audio task, since it is very important, it runs once every 2ms.
- This task reads from the Queues to see if the audio needs to be paused, if the track needs to be changed or if the volume needs to be adjusted.
- When it's not paused, I call the `loop()` method from the audio object to play the music. This handles behind the curtain mp3 decoding and sending input to the amplifier via I2S.

Testing

- When the parts came, I assembled the circuit and checked if the audio plays correctly on the speaker. I had to solder the wires so that was a bit stressful, since it was the first time for me doing this. The result can be seen in the first demo in the demos section below.
- Before the lab for the software milestone, I implemented a demo synchronization between the speaker and the LCD on a single song. The UI was tested only by myself, with the synchronization being validated by the laboratory assisted.
- I tested the UI, by keeping the buttons pressed for a very long time, pressing them in fast succesion. Also, the pause/resume button was tested in the same way and the volume regulator played a big role to see if the speaker behaved correctly at both low and high volume.

Optimizations and elements of originality

- This project doesn't rely on dynamic memory allocations, using only stack allocated char arrays instead of String objects for the text processing. This stops fragmentation of the RAM memory and brings stability to the system.
- The functionality separation two different cores, means that the main players of this architecture won't fight for resources, eliminating glitches in the music stream.
- Using FreeRTOS timers as high-precision synchronization mechanisms.

Conclusions

- This project was a really fun experience, where I experimented with many new concepts, like Real Time Operating Systems and audio processing API's for embedding systems.
- Also learning how to solder is a really valuable skill that I will definitely use in the future.

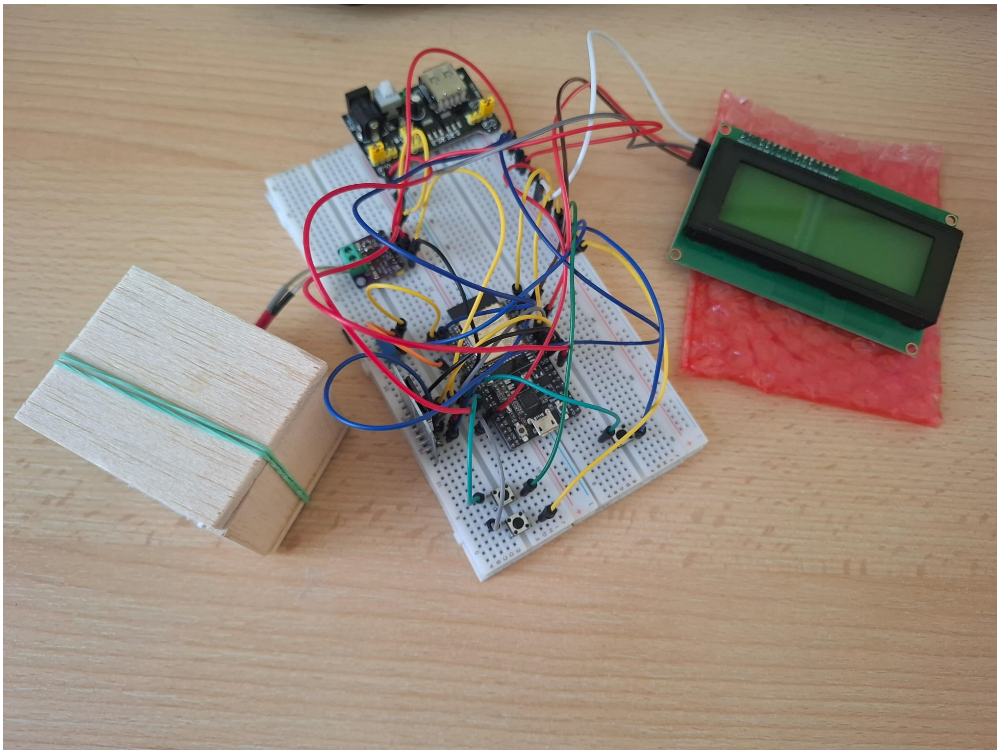
Download

Click here.

Journal

- 15.04.2026: First discussion with the assistant regarding the project theme.
- 05.05.2026: Partial creation of the technical documentation, which contains the project theme, block diagram and list of components.
- 08.05.2026: Add the electrical scheme of the project.
- 09.05.2026: Add the component pinout and refined the general description section.
- 12.05.2026: Refine the electrical scheme and add a short description of the software.
- 16.05.2026: Add simple demo.
- 23.05.2026: 90% of software implementation ready.
- 24.05.2026: Fine tuned the implementation.

Demos



Demo video

I did a small demo, where I just played the audio to see if the ESP32 can read the microSD card and if the speaker works. Also, in the video, you can see that the I2C module is connected and gets the text from the ESP32 that says “SD card mounted successfully”.

Below, you can see the final version of the project demo:

Final demo

Resources

All the lyric files were originally taken from lrclib.net and preprocessed using Audacity in order to sync the timestamps with the instrumental.

Datasheets

Component	Link Datasheet
ESP32-WROOM-32D	Datasheet microcontroller
Amplifier MAX98357A	Datasheet MAX98357A
LCD 20×4	Datasheet display
MB102 Power supply module	Datasheet MB102
MicroSDHC reader module	Datasheet MicroSDHC

FreeRTOS documentation (link here) .

AudioI2S documentation (link here) .

Export to PDF

From:
http://ocw.cs.pub.ro/courses/ - **CS Open CourseWare**

Permanent link:
http://ocw.cs.pub.ro/courses/pm/prj2026/vlad.radulescu2901/andrei.giurgiu0801

Last update: **2026/05/24 18:44**