Read the Docs Template Documentation

Release

Read the Docs

Jan 22, 2019

Contents

1 Get Started **3**
- 1.1 ESP32-LyraT V4.3 Getting Started Guide . . . 3
- 1.2 ESP32-LyraTD-MSC V2.2 Getting Started Guide . . . 7
- 1.3 About ESP-ADF . . . 12
- 1.4 Setup ESP-IDF . . . 12
- 1.5 Get ESP-ADF . . . 13
- 1.6 Setup Path to ESP-ADF . . . 13
- 1.7 Start a Project . . . 13
- 1.8 Connect and Configure . . . 14
- 1.9 Build, Flash and Monitor . . . 14
- 1.10 Update ESP-ADF . . . 15
- 1.11 Related Documents . . . 16

2 API Reference **33**
- 2.1 Audio Framework . . . 34
- 2.2 Audio Streams . . . 71
- 2.3 Codecs . . . 78
- 2.4 Services . . . 90
- 2.5 Peripherals . . . 95
- 2.6 Abstraction Layer . . . 119
- 2.7 Configuration Options . . . 143

3 Design Guide **145**
- 3.1 Project Design . . . 145
- 3.2 Design Considerations . . . 147
- 3.3 Software Design . . . 150
- 3.4 Development Boards . . . 153
- 3.5 Audio Samples . . . 164

4 Resources **167**

5 Copyrights and Licenses **169**
- 5.1 Software Copyrights . . . 169

6 About **171**

This is the documentation for Espressif Audio Development Framework.

Get Started	API Reference	Design Guide

CHAPTER 1

Get Started

This document is intended to help users set up the software environment for the development of audio applications using hardware based on the ESP32 by Espressif. Through a simple example, we would like to illustrate how to use ESP-ADF (Espressif Audio Development Framework).

To make the start with ESP-ADF quicker, Espressif designed development boards intended to build audio applications with the ESP32. Click the links below to get started.

	
Getting Started with ESP32-LyraT	Getting Started with ESP32-LyraTD-MSC

1.1 ESP32-LyraT V4.3 Getting Started Guide

This guide provides users with functional descriptions, configuration options for ESP32-LyraT V4.3 audio development board, as well as how to get started with the ESP32-LyraT board. Check section *Other Versions of LyraT*, if you have different version of this board.

The ESP32-LyraT is a hardware platform designed for the dual-core ESP32 audio applications, e.g., Wi-Fi or BT audio speakers, speech-based remote controllers, smart-home appliances with audio functionality(ies), etc.

1.1.1 What You Need

- 1 × *ESP32 LyraT V4.3 board*
- 2 x 4-ohm speakers with Dupont female jumper wires or headphones with a 3.5 mm jack
- 2 x Micro-USB 2.0 cables, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

If you like to start using this board right now, go directly to section *Start Application Development*.

Overview

The ESP32-LyraT V4.3 is an audio development board produced by Espressif built around ESP32. It is intended for audio applications, by providing hardware for audio processing and additional RAM on top of what is already onboard of the ESP32 chip. The specific hardware includes:

- **ESP32-WROVER Module**
- **Audio Codec Chip**
- Dual **Microphones** on board
- **Headphone** input
- **2 x 3-watt Speaker** output
- Dual **Auxiliary Input**
- **MicroSD Card** slot (1 line or 4 lines)
- **Six buttons** (2 physical buttons and 4 touch buttons)
- **JTAG** header
- Integrated **USB-UART Bridge Chip**
- Li-ion **Battery-Charge Management**

The block diagram below presents main components of the ESP32-LyraT and interconnections between components.

Components

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT used in this guide. This covers just what is needed now. For detailed technical documentation of this board, please refer to *ESP32-LyraT V4.3 Hardware Reference* and ESP32 LyraT V4.3 schematic (PDF).

ESP32-WROVER Module The ESP32-WROVER module contains ESP32 chip to provide Wi-Fi / BT connectivity and data processing power as well as integrates 32 Mbit SPI flash and 32 Mbit PSRAM for flexible data storage.

Headphone Output Output socket to connect headphones with a 3.5 mm stereo jack.

> **Note:** The socket may be used with mobile phone headsets and is compatible with OMPT standard headsets only. It does work with CTIA headsets. Please refer to Phone connector (audio) on Wikipedia.

Left Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

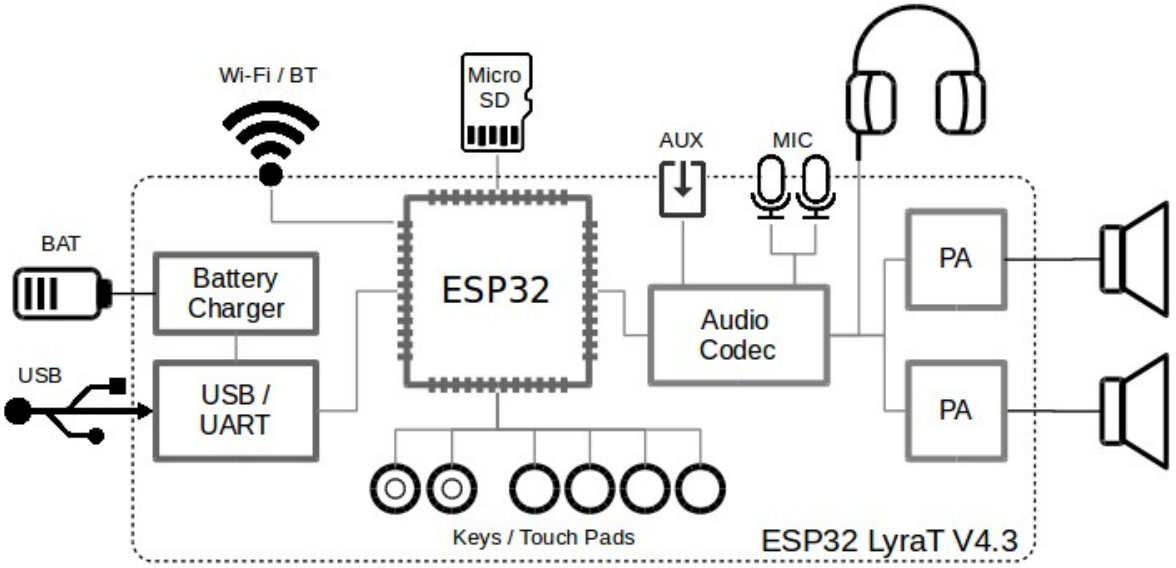


Fig. 1.1: ESP32-LyraT Block Diagram

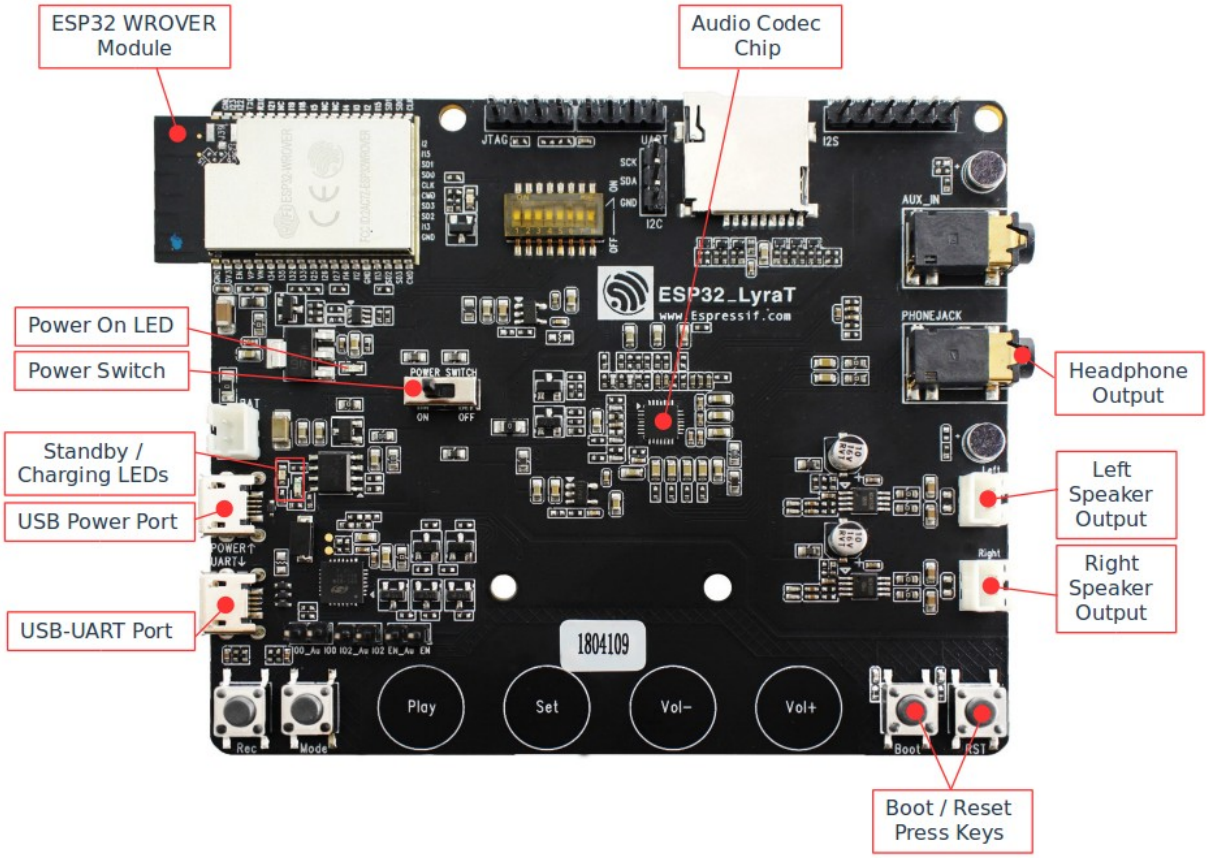


Fig. 1.2: ESP32-LyraT V4.3 Board Layout Overview

Right Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

Boot/Reset Press Keys Boot: holding down the **Boot** button and momentarily pressing the **Reset** button initiates the firmware upload mode. Then user can upload firmware through the serial port. Reset: pressing this button alone resets the system.

Audio Codec Chip The Audio Codec Chip, ES8388, is a low power stereo audio codec with a headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER Module** over I2S and I2S buses to provide audio processing in hardware independently from the audio application.

USB-UART Port Functions as the communication interface between a PC and the ESP32 WROVER module.

USB Power Port Provides the power supply for the board.

Standby / Charging LEDs The **Standby** green LED indicates that power has been applied to the **Micro USB Port**. The **Charging** red LED indicates that a battery connected to the **Battery Socket** is being charged.

Power Switch Power on/off knob: toggling it to the left powers the board on; toggling it to the right powers the board off.

Power On LED Red LED indicating that **Power On Switch** is turned on.

1.1.2 Start Application Development

Before powering up the ESP32-LyraT, please make sure that the board has been received in good condition with no obvious signs of damage.

Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 4-ohm speakers to the **Right** and **Left Speaker Output**. Connecting headphones to the **Headphone Output** is an option.
2. Plug in the Micro-USB cables to the PC and to **both USB ports** of the ESP32 LyraT.
3. The **Standby LED** (green) should turn on. Assuming that a battery is not connected, the **Charging LED** (red) will blink every couple of seconds.
4. Toggle left the **Power On Switch**.
5. The red **Power On LED** should turn on.

If this is what you see on the LEDs, the board should be ready for application upload. Now prepare the PC by loading and configuring development tools what is discussed in the next section.

Develop Applications

If the ESP32 LyraT is initially set up and checked, you can proceed with preparation of the development tools. Go to section *Get Started*, which will walk you through the following steps:

- *Setup ESP-IDF* in your PC that provides a common framework to develop applications for the ESP32 in C language;
- *Get ESP-ADF* to have the API specific for the audio applications;
- *Setup Path to ESP-ADF* to make the framework aware of the audio specific API;
- *Start a Project* that will provide a sample audio application for the ESP32-LyraT board;

- *Connect and Configure* to prepare the application for loading;
- *Build, Flash and Monitor* this will finally run the application and play some music.

1.1.3 Summary of Key Changes from LyraT V4.2

- Removed Red LED indicator light.
- Introduced headphone jack insert detection.
- Replaced single Power Amplifier (PA) chip with two separate chips.
- Updated power management design of several circuits: Battery Charging, ESP32, MicorSD, Codec Chip and PA.
- Updated electrical implementation design of several circuits: UART, Codec Chip, Left and Right Microphones, AUX Input, Headphone Output, MicroSD, Push Buttons and Automatic Upload.

1.1.4 Other Versions of LyraT

- *ESP32-LyraT V4.2 Getting Started Guide*
- *ESP32-LyraT V4 Getting Started Guide*

1.1.5 Related Documents

- *ESP32-LyraT V4.3 Hardware Reference*
- ESP32 LyraT V4.3 schematic (PDF)
- ESP32-LyraT V4.3 Component Layout (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)

1.2 ESP32-LyraTD-MSC V2.2 Getting Started Guide

This guide provides users with functional descriptions, configuration options for ESP32-LyraTD-MSC V2.2 audio development board, as well as how to get started with the ESP32-LyraTD-MSC board.

The ESP32-LyraTD-MSC is a hardware platform designed for smart speakers and AI applications. It supports Acoustic Echo Cancellation (AEC), Automatic Speech Recognition (ASR), Wake-up Interrupt and Voice Interaction.

1.2.1 What You Need

- 1 × *ESP32-LyraTD-MSC V2.2 board*
- 2 x 4-ohm speakers with Dupont female jumper wires or headphones with a 3.5 mm jack
- 2 x Micro-USB 2.0 cables, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

If you like to start using this board right now, go directly to section *Start Application Development*.

Overview

The ESP32-LyraTD-MSC V2.2 is an audio development board produced by Espressif built around ESP32. It is intended for smart speakers and AI applications, by providing hardware for digital signal processing, microphone array and additional RAM on top of what is already onboard of the ESP32 chip.

This audio development board consists of two parts: the upper board (B), which provides a three-microphone array, function keys and LED lights; and the lower board (A), which integrates ESP32-WROVER-B, a MicroSemi Digital Signal Processing (DSP) chip, and a power management module.



Fig. 1.3: ESP32-LyraTD-MSC Side View

The specific hardware includes:

- **ESP32-WROVER-B Module**
- **DSP** (Digital Signal Processing) chip
- Three digital **Microphones** that support far-field voice pick-up
- **2 x 3-watt Speaker** output
- **Headphone** output
- **MicroSD Card** slot (1 line or 4 lines)
- Individually controlled **Twelve LEDs** distributed in a circle on the board's edge
- **Six Function Buttons** that may be assigned user functions
- Several interface ports: **I2S**, **I2C**, **SPI** and **JTAG**
- Integrated **USB-UART Bridge Chip**
- Li-ion **Battery-Charge Management**

The block diagram below presents main components of the ESP32-LyraTD-MSC and interconnections between components.

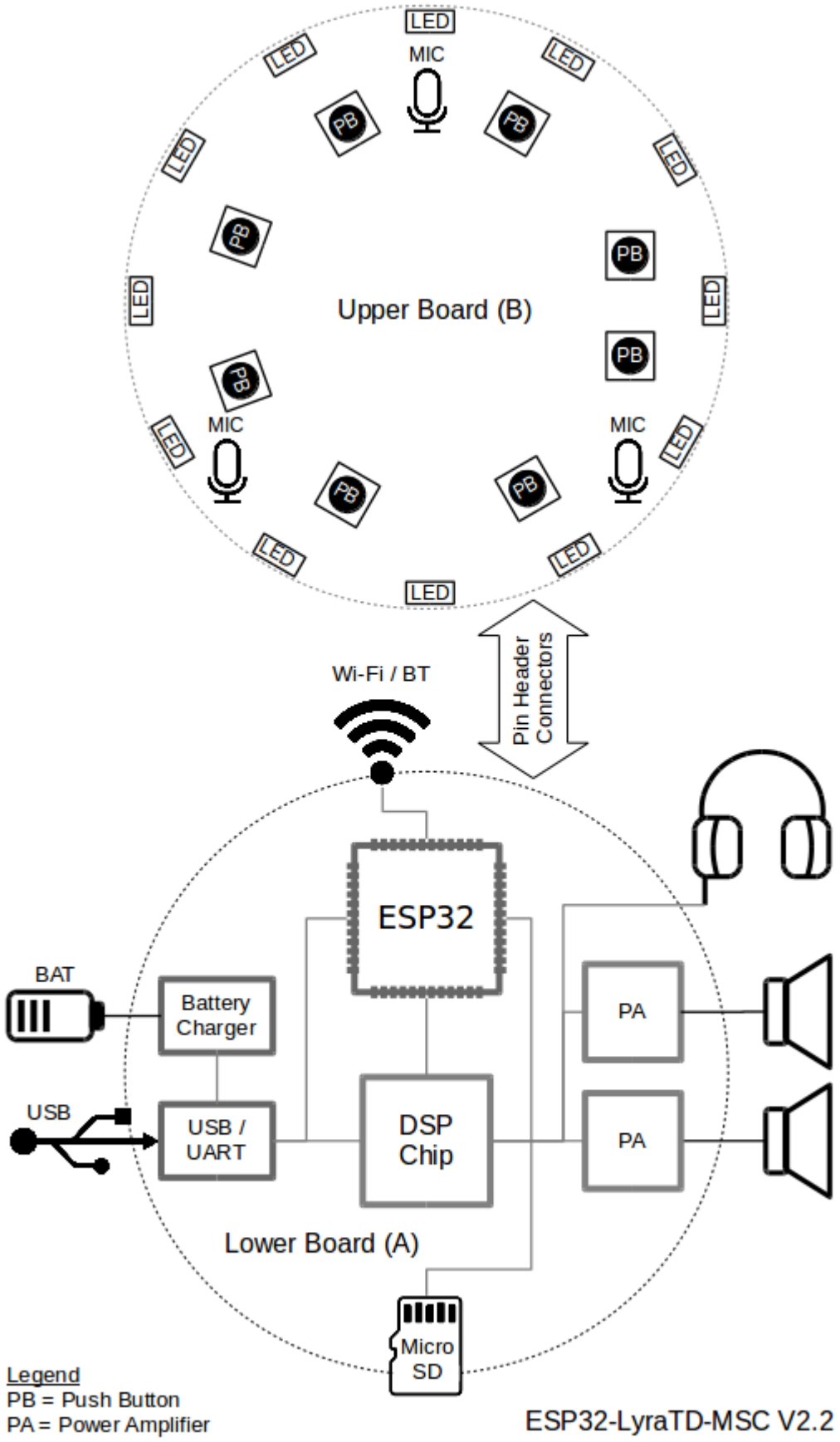


Fig. 1.4: ESP32-LyraTD-MSC Block Diagram

Components

The following list and figure describe key components, interfaces and controls of the ESP32-LyraTD-MSC used in this guide. This covers just what is needed now. For additional details please refer to schematics provided in *Related Documents*.

ESP32-WROVER-B Module The ESP32-WROVER-B module contains ESP32 chip to provide Wi-Fi / BT connectivity and data processing power as well as integrates 32 Mbit SPI flash and 64 Mbit PSRAM for flexible data storage.

DSP Chip The Digital Signal Processing chip ZL38063 is used for Automatic Speech Recognition (ASR) applications. It captures audio data from an external microphone array and outputs audio signals through its Digital-to-Analog-Converter (DAC) port.

Headphone Output Output socket to connect headphones with a 3.5 mm stereo jack.

> **Note:** The socket may be used with mobile phone headsets and is compatible with OMPT standard headsets only. It does work with CTIA headsets. Please refer to Phone connector (audio) on Wikipedia.

Left Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

Right Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

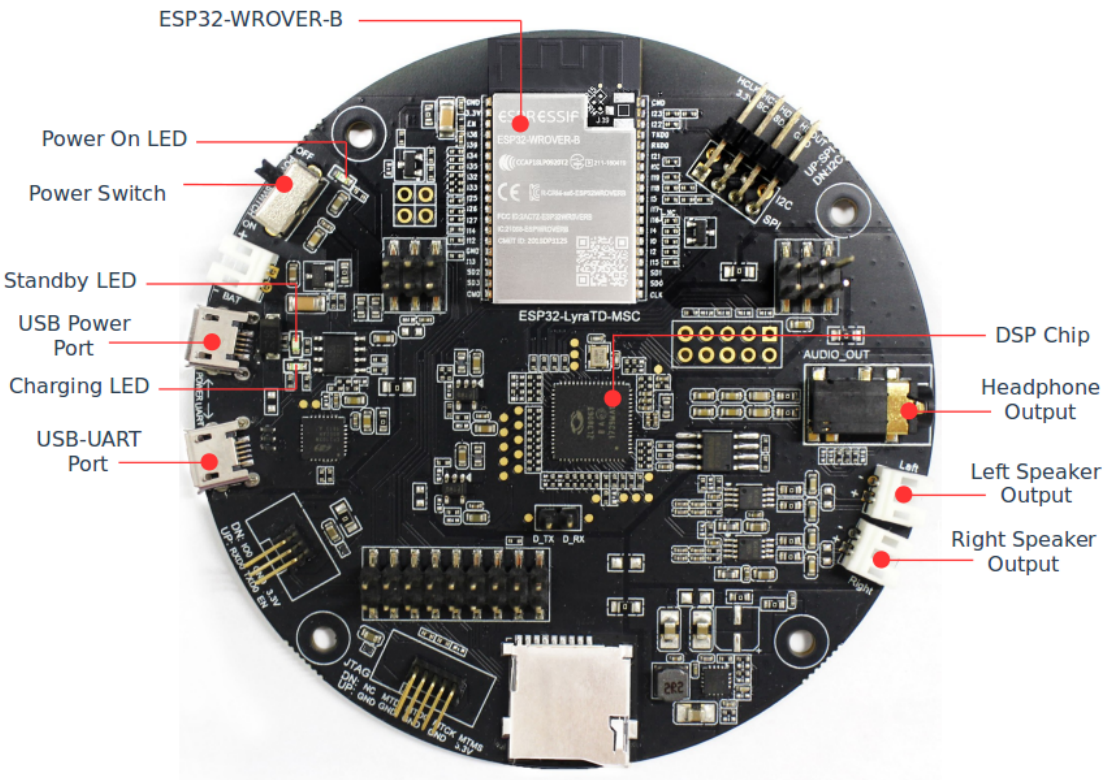


Fig. 1.5: ESP32-LyraTD-MSC V2.2 Lower Board (A) Components

USB-UART Port Functions as the communication interface between a PC and the ESP32 WROVER module.

USB Power Port Provides the power supply for the board.

Standby / Charging LEDs The **Standby** green LED indicates that power has been applied to the **Micro USB Port**. The **Charging** red LED indicates that a battery connected to the **Battery Socket** is being charged.

Power Switch Power on/off knob: toggling it right powers the board on; otherwise powers the board off.

Power On LED Red LED indicating that **Power Switch** is turned on.

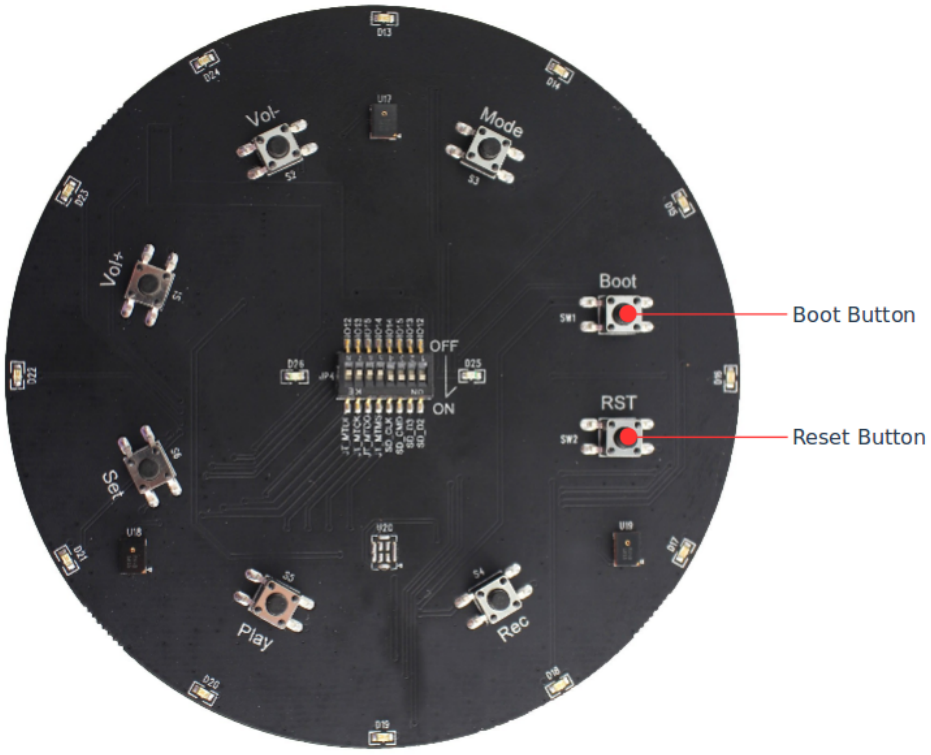


Fig. 1.6: ESP32-LyraTD-MSC V2.2 Upper Board (B) Components

Boot/Reset Buttons Boot: holding down the **Boot** button and momentarily pressing the **Reset** button initiates the firmware upload mode. Then user can upload firmware through the serial port.

Reset: pressing this button alone resets the system.

1.2.2 Start Application Development

Before powering up the ESP32-LyraTD-MSC, please make sure that the board has been received in good condition with no obvious signs of damage. Both the lower A and the upper B board of the ESP32-LyraTD-MSC should be firmly connected together.

Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 4-ohm speakers to the **Right** and **Left Speaker Output**. Connecting headphones to the **Headphone Output** is an option.

2. Plug in the Micro-USB cables to the PC and to **both USB ports** of the ESP32-LyraTD-MSC.
3. The **Standby LED** (green) should turn on. Assuming that a battery is not connected, the **Charging LED** (red) will blink every couple of seconds.
4. Toggle right the **Power Switch**.
5. The red **Power On LED** should turn on.

If this is what you see on the LEDs, the board should be ready for application upload. Now prepare the PC by loading and configuring development tools what is discussed in the next section.

Develop Applications

If the ESP32-LyraTD-MSC is initially set up and checked, you can proceed with preparation of the development tools. Go to section *Get Started*, which will walk you through the following steps:

- *Setup ESP-IDF* in your PC that provides a common framework to develop applications for the ESP32 in C language;
- *Get ESP-ADF* to have the API specific for the audio applications;
- *Setup Path to ESP-ADF* to make the framework aware of the audio specific API;
- *Start a Project* that will provide a sample audio application for the ESP32-LyraTD-MSC board;
- *Connect and Configure* to prepare the application for loading;
- *Build, Flash and Monitor* this will finally run the application and play some music.

1.2.3 Related Documents

- ESP32-LyraTD-MSC V2.2 Schematic Lower Board (A) (PDF)
- ESP32-LyraTD-MSC V2.2 Schematic Upper Board (B) (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER-B Datasheet (PDF)

If you do not have one of the above boards, you can still use ESP-ADF for the ESP32 based audio applications. This is providing your board has a compatible audio codec or DSP chip, or you develop a driver to support communication with your specific chip.

1.3 About ESP-ADF

The ESP-ADF is available as a set of components to extend the functionality already delivered by the ESP-IDF (Espressif IoT Development Framework).

To use ESP-ADF you need set up the ESP-IDF first, and this is described in the next section.

1.4 Setup ESP-IDF

Configure your PC according to ESP32 Documentation. Windows, Linux and Mac OS operating systems are supported.

You have a choice to compile and upload code to the ESP32 by command line with make or using Eclipse IDE.

Note: We are using `~/esp` directory to install the toolchain, ESP-IDF, ESP-ADF and sample applications. You can use a different directory, but need to adjust respective commands.

To make the installation easier and less prone to errors, use the `~/esp` default directory for the installation. Once you get through ESP-IDF setup and move to the ESP-ADF, you will notice that installation of the ESP-ADF follows the similar process. This should make it even easier to get up and running with the ESP-ADF.

If this is your first exposure to the ESP32 and ESP-IDF, then it is recommended to get familiar with hello_world and blink examples first. Once you can build, upload and run these two examples, then you are ready to proceed to the next section.

1.5 Get ESP-ADF

Having the ESP-IDF to compile, build and upload application for ESP32, you can now move to installing audio specific API / libraries. They are provided in ESP-ADF repository. To get it, open terminal, navigate to the directory to put the ESP-ADF, and clone it using `git clone` command:

```
cd ~/esp
git clone --recursive https://github.com/espressif/esp-adf.git
```

ESP-ADF will be downloaded into `~/esp/esp-adf`.

Note: Do not miss the `--recursive` option. If you have already cloned ESP-ADF without this option, run another command to get all the submodules:

```
cd ~/esp/esp-adf
git submodule update --init
```

1.6 Setup Path to ESP-ADF

The toolchain programs access ESP-ADF using `ADF_PATH` environment variable. This variable should be set up on your PC, otherwise the projects will not build. The process to set it up is analogous to setting up the `IDF_PATH` variable, please see instructions in ESP-IDF documentation under Add IDF_PATH to User Profile.

1.7 Start a Project

After initial preparation you are ready to build the first audio application for the ESP32. The process has already been described in ESP-IDF documentation. Now we would like to discuss again the key steps and show how the toolchain is able to access the ESP-ADF components by using the `ADF_PATH` variable.

Note: ESP-ADF is based on a specific release of the ESP-IDF. You will see this release cloned with ESP-ADF as a subdirectory, or more specifically as a submodule e.g. `esp-idf @ ca3faa61` visible on the GitHub. Just follow this instruction and the build scripts will automatically reach ESP-IDF from the submodule.

To demonstrate how to build an application, we will use get-started/play_mp3 project from examples directory in the ADF.

Copy get-started/play_mp3 to ~/esp directory:

```
cd ~/esp
cp -r $ADF_PATH/examples/get-started/play_mp3 .
```

You can also find a range of example projects under the examples directory in the ESP-ADF repository. These example project directories can be copied in the same way as presented above, to begin your own projects.

1.8 Connect and Configure

Connect the audio ESP32 board to the PC, check under what serial port the board is visible and verify, if serial communication works as described in ESP-IDF Documentation.

At the terminal window, go to the directory of play_mp3 application and configure it with menuconfig by selecting the serial port, upload speed and the audio board version:

```
cd ~/esp/play_mp3
make menuconfig
```

Save the configuration.

1.9 Build, Flash and Monitor

Now you can build, upload and check the application. Run:

```
make flash monitor -j5
```

This will build the application including ESP-IDF / ESP-ADF components, upload (flash) binaries to your ESP32 board and start the monitor.

1.9.1 Upload

To upload the binaries, the board should be put into upload mode. To do so, hold down **Boot** button, momentarily press **Reset** button and release the **Boot** button. The upload mode may be initiated anytime during the application build, but no later than "Connecting" message is being displayed:

```
...

esptool.py v2.1
Connecting........_____....
```

Without the upload mode enabled, after showing several `.....`_____, the connection will eventually time out.

Once build and upload is complete, you should see the following:

```
...

Leaving...
Hard resetting...
MONITOR
--- idf_monitor on /dev/ttyUSB0 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
```

1.9.2 Monitor

At this point press the **Reset** button to start the application. Following several lines of start up log, the `play_mp3` application specific messages should be displayed:

```
...

I (303) PLAY_MP3_FLASH: [ 1 ] Start audio codec chip
I (323) PLAY_MP3_FLASH: [ 2 ] Create audio pipeline, add all elements to pipeline,␣
↪and subscribe pipeline event
I (323) PLAY_MP3_FLASH: [2.1] Create mp3 decoder to decode mp3 file and set custom␣
↪read callback
I (333) PLAY_MP3_FLASH: [2.2] Create i2s stream to write data to codec chip
I (343) PLAY_MP3_FLASH: [2.3] Register all elements to audio pipeline
I (353) PLAY_MP3_FLASH: [2.4] Link it together [mp3_music_read_cb]-->mp3_decoder-->
↪i2s_stream-->[codec_chip]
I (363) PLAY_MP3_FLASH: [ 3 ] Setup event listener
I (363) PLAY_MP3_FLASH: [3.1] Listening event from all elements of pipeline
I (373) PLAY_MP3_FLASH: [ 4 ] Start audio_pipeline
W (373) AUDIO_ELEMENT: [mp3] RESUME:Element has not running,state:3,task_run:1
W (393) AUDIO_ELEMENT: [i2s] RESUME:Element has not running,state:3,task_run:1
I (403) PLAY_MP3_FLASH: [ * ] Receive music info from mp3 decoder, sample_rates=44100,
↪ bits=16, ch=2
W (433) AUDIO_ELEMENT: [i2s] RESUME:Element has not running,state:3,task_run:1
I (7183) PLAY_MP3_FLASH: [ 5 ] Stop audio_pipeline
W (7183) AUDIO_PIPELINE: There are no listener registered
```

If there are no issues, besides the above log, you should hear a sound played for about 7 seconds by the speakers or headphones connected to your audio board. Reset the board to hear it again if required.

Now you are ready to try some other examples, or go right to developing your own applications. Check how the examples are made aware of location of the ESP-ADF. Open the get-started/play_mp3/Makefile and you should see

```
PROJECT_NAME := play_mp3
include $(ADF_PATH)/project.mk
```

The second line contains `$ADF_PATH` to point the toolchain to the ESP-ADF. You need similar `Makefile` in your own applications developed with the ESP-ADF.

1.10 Update ESP-ADF

After some time of using ESP-ADF, you may want to update it to take advantage of new features or bug fixes. The simplest way to do so is by deleting existing `esp-adf` folder and cloning it again, which is same as when doing initial installation described in sections *Get ESP-ADF*.

Another solution is to update only what has changed. This method is useful if you have a slow connection to the GitHub. To do the update run the following commands:

```
cd ~/esp/esp-adf
git pull
git submodule update --init --recursive
```

The `git pull` command is fetching and merging changes from ESP-ADF repository on GitHub. Then `git submodule update --init --recursive` is updating existing submodules or getting a fresh copy of new ones. On GitHub the submodules are represented as links to other repositories and require this additional command to get them onto your PC.

1.11 Related Documents

1.11.1 ESP32-LyraT V4.2 Getting Started Guide

This guide provides users with functional descriptions, configuration options for ESP32-LyraT V4.2 audio development board, as well as how to get started with the ESP32-LyraT board.

The ESP32-LyraT development board is a hardware platform designed for the dual-core ESP32 audio applications, e.g., Wi-Fi or BT audio speakers, speech-based remote controllers, smart-home appliances with audio functionality(ies), etc.

If you like to start using this board right now, go directly to section *Start Application Development*.

What You Need

- 1 × *ESP32 LyraT V4.2 board*
- 2 x 4-ohm speakers with Dupont female jumper wires or headphones with a 3.5 mm jack
- 2 x Micro-USB 2.0 cables, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

Overview

The ESP32-LyraT V4.2 is an audio development board produced by Espressif built around ESP32. It is intended for audio applications, by providing hardware for audio processing and additional RAM on top of what is already onboard of the ESP32 chip. The specific hardware includes:

- **ESP32-WROVER Module**
- **Audio Codec Chip**
- Dual **Microphones** on board
- **Headphone** input
- **2 x 3-watt Speaker** output
- Dual **Auxiliary Input**
- **MicroSD Card** slot (1 line or 4 lines)
- **Six buttons** (2 physical buttons and 4 touch buttons)
- **JTAG** header
- Integrated **USB-UART Bridge Chip**
- Li-ion **Battery-Charge Management**

The block diagram below presents main components of the ESP32-LyraT and interconnections between components.

Functional Description

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT board.

ESP32-WROVER Module The ESP32-WROVER module contains ESP32 chip to provide Wi-Fi / BT connectivity and data processing power as well as integrates 32 Mbit SPI flash and 32 Mbit PSRAM for flexible data storage.

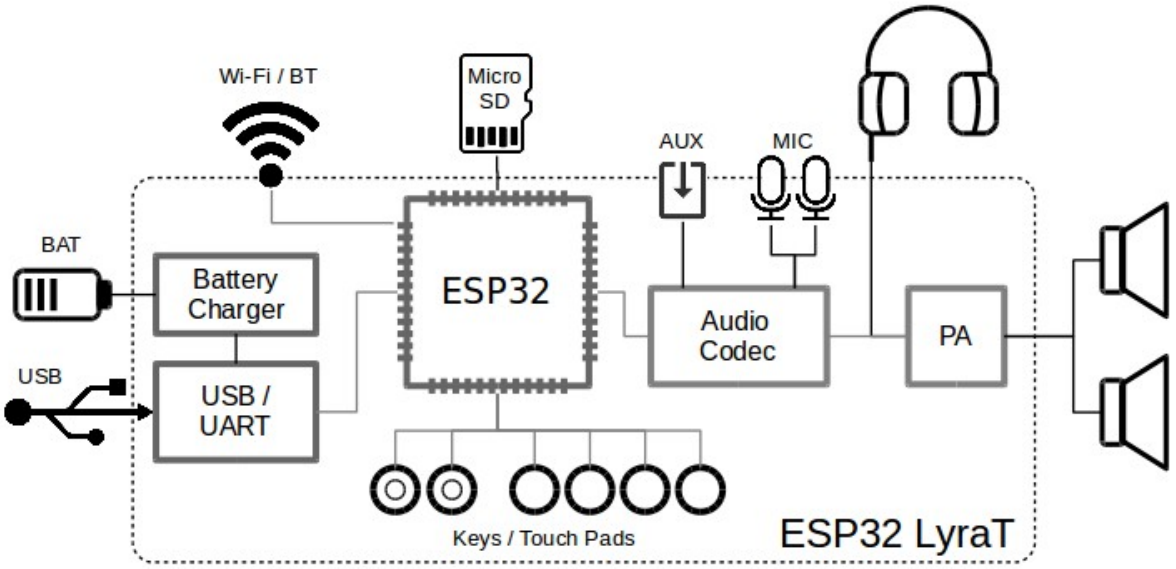


Fig. 1.7: ESP32-LyraT Block Diagram

Green and Red LEDs Two general purpose LEDs controlled by **ESP32-WROVER Module** to indicate certain operation states of the audio application using dedicated API.

Function DIP Switch Used to configure function of GPIO12 to GPIO15 pins that are shared between devices, primarily between **JTAG Header** and **MicroSD Card**. By default, the **MicroSD Card** is enabled with all switches in *OFF* position. To enable the **JTAG Header** instead, switches in positions 3, 4, 5 and 6 should be put *ON*. If **JTAG** is not used and **MicroSD Card** is operated in the one-line mode, then GPIO12 and GPIO13 may be assigned to other functions. Please refer to ESP32 LyraT V4.2 schematic for more details.

JTAG Header Provides access to the **JTAG** interface of **ESP32-WROVER Module**. It may be used for debugging, application upload, as well as implementing several other functions, e.g., Application Level Tracing. See *JTAG Header / JP7* for pinout details. Before using **JTAG** signals to the header, **Function DIP Switch** should be enabled. Please note that when **JTAG** is in operation, **MicroSD Card** cannot be used and should be disconnected because some of JTAG signals are shared by both devices.

UART Header Serial port: provides access to the serial TX/RX signals between **ESP32-WROVER Module** and **USB-UART Bridge Chip**.

I2C Header Provides access to the I2C interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2C Header / JP5* for pinout details.

MicroSD Card The development board supports a MicroSD card in SPI/1-bit/4-bit modes, and can store or play audio files in the MicroSD card. See *MicroSD Card / J5* for pinout details. Note that **JTAG** cannot be used and should be disconnected by setting **Function DIP Switch** when **MicroSD Card** is in operation, because some of signals are shared by both devices.

I2S Header Provides access to the I2S interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2S Header / JP4* for pinout details.

Left Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

AUX Input Auxiliary input socket connected to IN2 (left and right channel) of the **Audio Codec Chip**. Use a 3.5 mm stereo jack to connect to this socket.

Headphone Output Output socket to connect headphones with a 3.5 mm stereo jack.

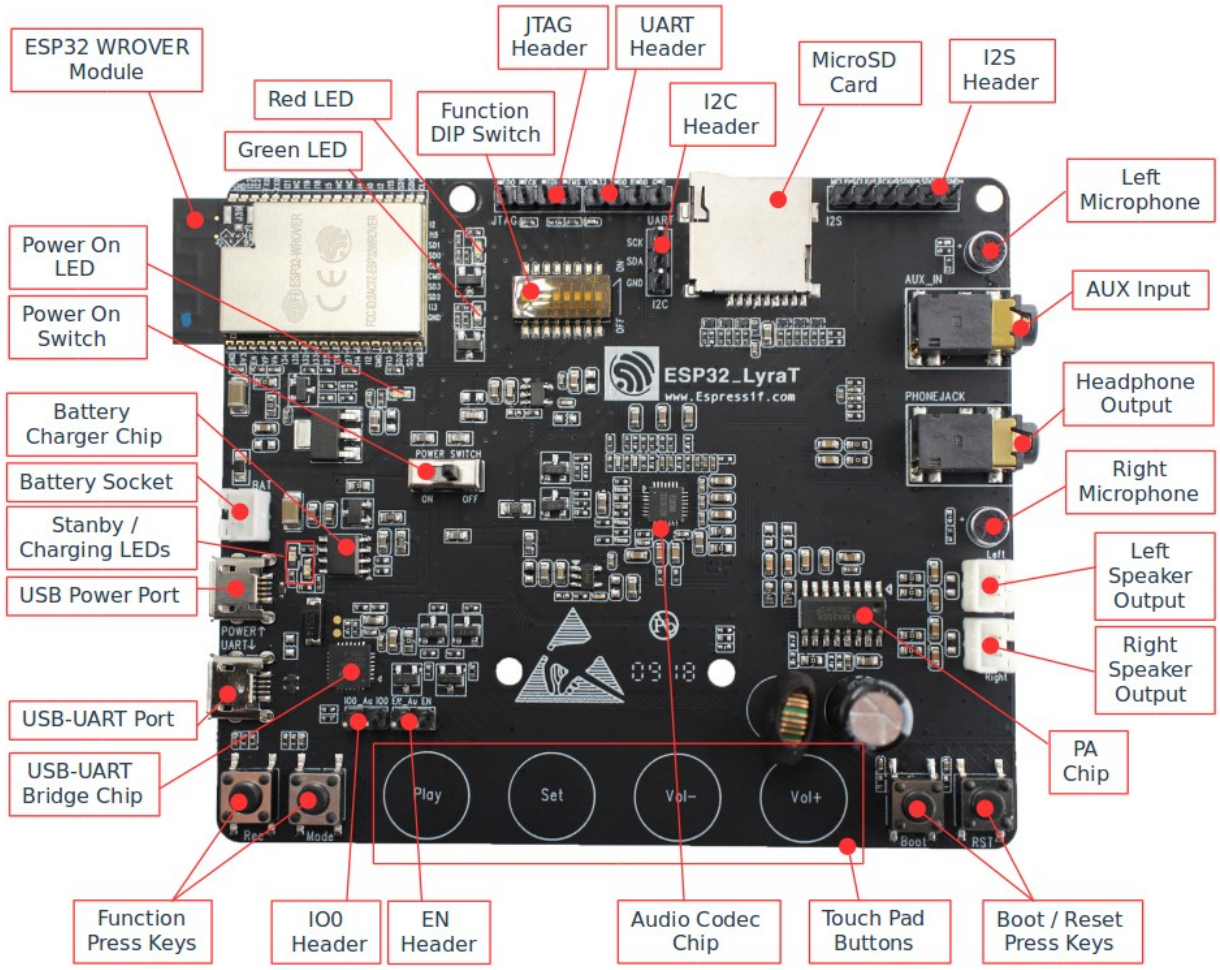


Fig. 1.8: ESP32-LyraT V4.2 Board Layout

Right Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

Left Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

Right Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

PA Chip A power amplifier used to amplify stereo audio signal from the **Audio Codec Chip** for driving two 4-ohm speakers.

Boot/Reset Press Keys Boot: holding down the **Boot** button and momentarily pressing the **Reset** button initiates the firmware upload mode. Then user can upload firmware through the serial port. Reset: pressing this button alone resets the system.

Touch Pad Buttons Four touch pads labeled *Play*, *Sel*, *Vol+* and *Vol-*. They are routed to **ESP32-WROVER Module** and intended for development and testing of a UI for audio applications using dedicated API.

Audio Codec Chip The Audio Codec Chip, ES8388, is a low power stereo audio codec with a headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER Module** over I2S and I2S buses to provide audio processing in hardware independently from the audio application.

EN Header Install a jumper on this header to enable automatic loading of application to the ESP32. Install or remove jumpers together on both IO0 and EN headers.

IO0 Header Install a jumper on this header to enable automatic loading of application to the ESP32. Install or remove jumpers together on both IO0 and EN headers.

Function Press Keys Two key labeled *Rec* and *Mode*. They are routed to **ESP32-WROVER Module** and intended for developing and testing a UI for audio applications using dedicated API.

USB-UART Bridge Chip A single chip USB-UART bridge provides up to 1 Mbps transfers rate.

USB-UART Port Functions as the communication interface between a PC and the ESP32 module.

USB Power Port Provides the power supply for the board.

Standby / Charging LEDs The **Standby** green LED indicates that power has been applied to the **Micro USB Port**. The **Charging** red LED indicates that a battery connected to the **Battery Socket** is being charged.

Battery Charger Chip Constant current & constant voltage linear charger for single cell lithium-ion batteries AP5056. Used for charging of a battery connected to the **Battery Socket** over the **Micro USB Port**.

Power On Switch Power on/off knob: toggling it to the left powers the board on; toggling it to the right powers the board off.

Battery Socket Two pins socket to connect a single cell Li-ion battery.

Power On LED Red LED indicating that **Power On Switch** is turned on.

Note: The **Power On Switch** does not affect / disconnect the Li-ion battery charging.

Hardware Setup Options

There are a couple of options to change the hardware configuration of the ESP32-LyraT board. The options are selectable with the **Function DIP Switch**.

Enable MicroSD Card in 1-wire Mode

DIP SW	Position
1	OFF
2	OFF
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF [1]
8	n/a

1. **AUX Input** detection may be enabled by toggling the DIP SW 7 *ON*

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is available for use with the API

Enable MicroSD Card in 4-wire Mode

DIP SW	Position
1	ON
2	ON
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF
8	n/a

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Enable JTAG

DIP SW	Position
1	OFF
2	OFF
3	ON
4	ON
5	ON
6	ON
7	ON
8	n/a

In this mode:

- **MicroSD Card** functionality is not available, remove the card from the slot
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Allocation of ESP32 Pins

Several pins / terminals of ESP32 modules are allocated to the on board hardware. Some of them, like GPIO0 or GPIO2, have multiple functions. Please refer to the tables below or ESP32 LyraT V4.2 schematic for specific details.

Red / Green LEDs

	ESP32 Pin	LED Color
1	GPIO19	Red LED
2	GPIO22	Green LED

Touch Pads

	ESP32 Pin	Touch Pad Function
1	GPIO33	Play
2	GPIO32	Set
3	GPIO13	Vol- [1]
4	GPIO27	Vol+

1. *Vol-* function is not available if **JTAG** is used. It is also not available for the **MicroSD Card** configured to operate in 4-wire mode.

MicroSD Card / J5

	ESP32 Pin	MicroSD Signal
1	MTDI / GPIO12	DATA2
2	MTCK / GPIO13	CD / DATA3
3	MTDO / GPIO15	CMD
4	MTMS / GPIO14	CLK
5	GPIO2	DATA0
6	GPIO4	DATA1
7	GPIO21	CD

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

EN and IO0 Headers / JP23 and J24

	ESP32 Pin	Header Pin
1	n/a	EN_Auto
2	EN	EN

	ESP32 Pin	Header Pin
1	n/a	IO0_Auto
2	GPIO0	IO0

I2S Header / JP4

	I2C Header Pin	ESP32 Pin
1	MCLK	GPI0
2	SCLK	GPIO5
1	LRCK	GPIO25
2	DSDIN	GPIO26
3	ASDOUT	GPIO35
3	GND	GND

I2C Header / JP5

	I2C Header Pin	ESP32 Pin
1	SCL	GPIO23
2	SDA	GPIO18
3	GND	GND

JTAG Header / JP7

	ESP32 Pin	JTAG Signal
1	MTDO / GPIO15	TDO
2	MTCK / GPIO13	TCK
3	MTDI / GPIO12	TDI
4	MTMS / GPIO14	TMS

Function DIP Switch / JP8

	Switch OFF	Switch ON
1	GPIO12 not allocated	MicroSD Card 4-wire
2	Touch *Vol-* enabled	MicroSD Card 4-wire
3	MicroSD Card	JTAG
4	MicroSD Card	JTAG
5	MicroSD Card	JTAG
6	MicroSD Card	JTAG
7	MicroSD Card 4-wire	AUX IN detect [1]
8	not used	not used

1. The **AUX Input** signal pin should not be be plugged in when the system powers up. Otherwise the ESP32 may not be able to boot correctly.

Start Application Development

Before powering up the ESP32-LyraT, please make sure that the board has been received in good condition with no obvious signs of damage.

Initial Setup

Prepare the board for loading of the first sample application:

1. Install jumpers on **IO0** and **EN** headers to enable automatic application upload. If there are no jumpers then upload may be triggered using **Boot / RST** buttons.
2. Connect 4-ohm speakers to the **Right** and **Left Speaker Output**.Connecting headphones to the **Headphone Output** is an option.
3. Plug in the Micro-USB cables to the PC and to **both USB ports** of the ESP32 LyraT.
4. The **Standby LED** (green) should turn on. Assuming that a battery is not connected, the **Charging LED** will blink every couple of seconds.
5. Toggle left the **Power On Switch**.
6. The red **Power On LED** should turn on.

If this is what you see on the LEDs, the board should be ready for application upload. Now prepare the PC by loading and configuring development tools what is discussed in the next section.

Develop Applications

If the ESP32 LyraT is initially set up and checked, you can proceed with preparation of the development tools. Go to section *Get Started*, which will walk you through the following steps:

- *Setup ESP-IDF* in your PC that provides a common framework to develop applications for the ESP32 in C language;
- *Get ESP-ADF* to have the API specific for the audio applications;
- *Setup Path to ESP-ADF* to make the framework aware of the audio specific API;
- *Start a Project* that will provide a sample audio application for the ESP32-LyraT board;

- *Connect and Configure* to prepare the application for loading;
- *Build, Flash and Monitor* this will finally run the application and play some music.

Related Documents

- ESP32 LyraT V4.2 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging
- *ESP32-LyraT V4 Getting Started Guide*

1.11.2 ESP32-LyraT V4 Getting Started Guide

This guide provide users with functional descriptions, configuration options for ESP32-LyraT V4 audio development board, as well as how to get started with ESP32-LyraT board.

The ESP32-LyraT development board is a hardware platform specifically designed for the dual-core ESP32 audio applications, e.g., Wi-Fi or BT audio speakers, speech-based remote controllers, smart-home appliances with audio functionality(ies), etc.

If you like to start using this board right now, go directly to section *Start Application Development*.

What You Need

- 1 × *ESP32-LyraT V4 board*
- 2 x 4-ohm speakers with Dupont female jumper wires or headphones with a 3.5 mm jack
- 1 x Micro USB 2.0 Cable, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

Overview

The ESP32-LyraT V4 is an audio development board produced by Espressif built around ESP32. It is intended for audio applications, by providing hardware for audio processing and additional RAM on top of what is already onboard of the ESP32 chip. The specific hardware includes:

- **ESP32-WROVER Module**
- **Audio Codec Chip**
- Dual **Microphones** on board
- **Headphone** input
- **2 x 3 Watt Speaker** output
- Dual **Auxiliary Input**
- **MicroSD Card** slot (1 line or 4 lines)
- **6 buttons** (2 physical buttons and 4 touch buttons)
- **JTAG** header

- Integrated **USB-UART Bridge Chip**
- Li-ion **Battery-Charge Management**

Block diagram below presents main components of the ESP32-LyraT and interconnections between components.

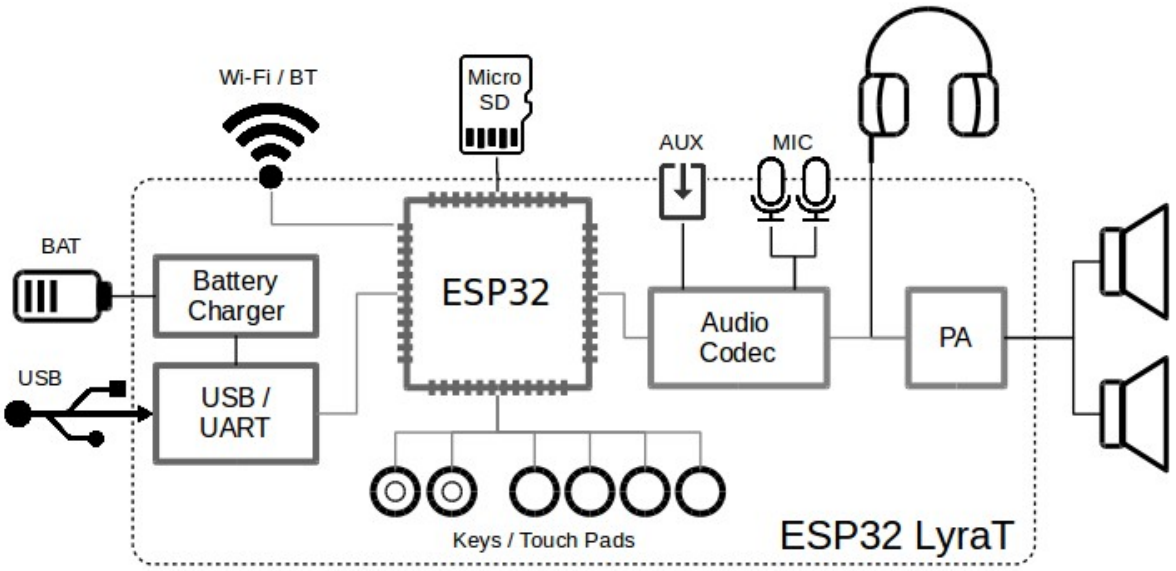


Fig. 1.9: ESP32-LyraT block diagram

Functional Description

The following list and figure below describe key components, interfaces and controls of the ESP32-LyraT board.

ESP32-WROVER Module The ESP32-WROVER module contains ESP32 chip to provide Wi-Fi / BT connectivity and data processing power as well as integrates 32 Mbit SPI flash and 32 Mbit PSRAM for flexible data storage.

Green and Red LEDs Two general purpose LEDs controlled by **ESP32-WROVER Module** to indicate certain operation states of the audio application using dedicated API.

Function DIP Switch Used to configure function of GPIO12 to GPIO15 pins that are shared between devices, primarily between **JTAG Header** and **MicroSD Card**. By default **MicroSD Card** is enabled with all switches in *OFF* position. To enable **JTAG Header** instead, switches in positions 3, 4, 5 and 6 should be put *ON*. If **JTAG** is not used and **MicroSD Card** is operated in one-line mode, then GPIO12 and GPIO13 may be assigned to other functions. Please refer to ESP32 LyraT V4 schematic for more details.

JTAG Header Provides access to the **JTAG** interface of **ESP32-WROVER Module**. May be used for debugging, application upload, as well as implementing several other functions, e.g., Application Level Tracing. See *JTAG Header / JP7* for pinout details. Before using **JTAG** signals to the header, **Function DIP Switch** should be enabled. Please note that when **JTAG** is in operation, **MicroSD Card** cannot be used and should be disconnected because some of JTAG signals are shared by both devices.

UART Header Serial port provides access to the serial TX/RX signals between **ESP32-WROVER Module** and **USB-UART Bridge Chip**.

I2C Header Provides access to the I2C interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2C Header / JP5* for pinout details.

MicroSD Card The development board supports a MicroSD card in SPI/1-bit/4-bit modes, and can store or play audio files in the MicroSD card. See *MicroSD Card / J5* for pinout details. Note that **JTAG** cannot be used and should be disconnected by setting **Function DIP Switch** when **MicroSD Card** is in operation, because some of the signals are shared by both devices.

I2S Header Provides access to the I2S interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2S Header / JP4* for pinout details.

Left Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

AUX Input Auxiliary input socket connected to IN2 (left and right channels) of the **Audio Codec Chip**. Use a 3.5 mm stereo jack to connect to this socket.

Headphone Output Output socket to connect headphones with a 3.5 mm stereo jack.

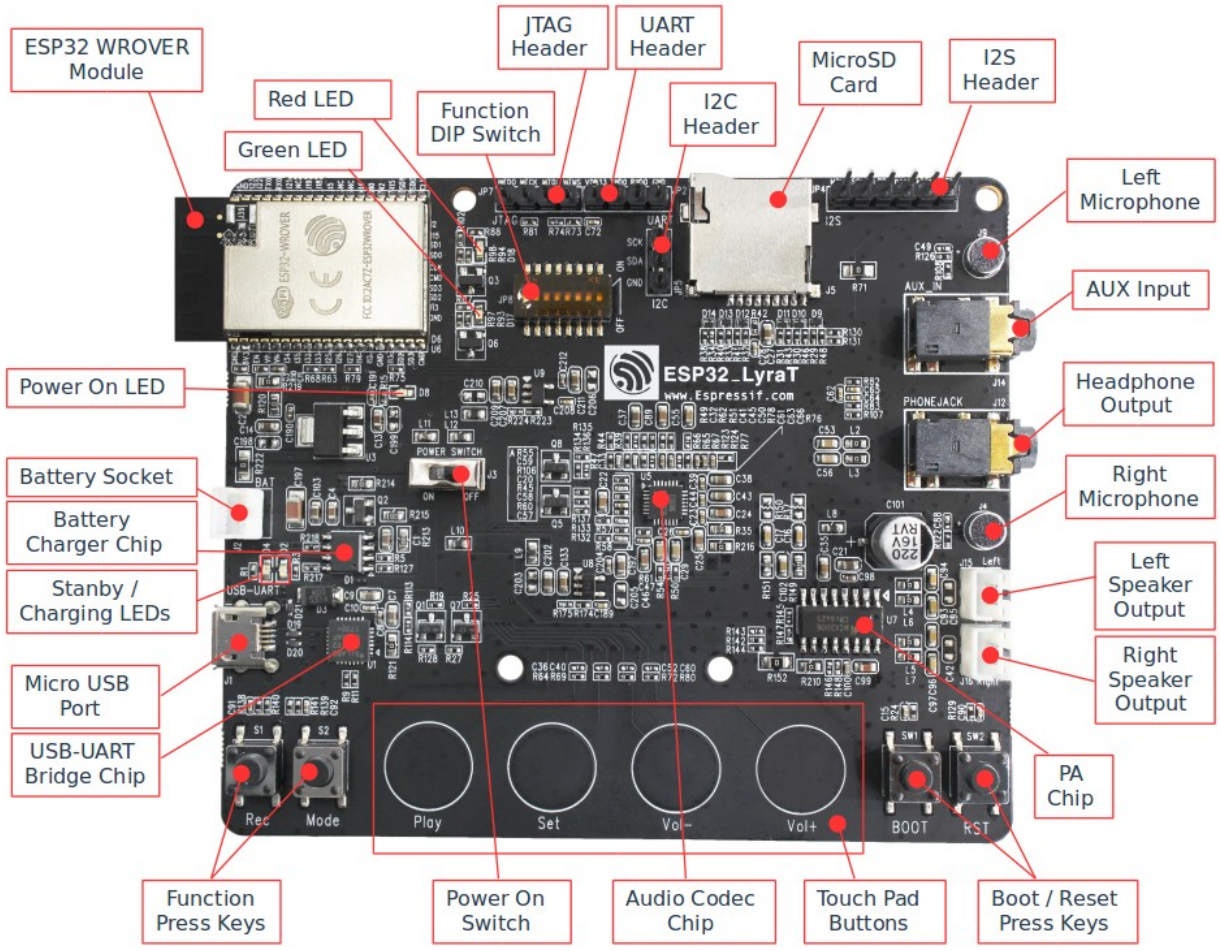


Fig. 1.10: ESP32 LyraT V4 board layout

Right Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

Left Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

Right Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

PA Chip A power amplifier used to amplify stereo audio signal from the **Audio Codec Chip** for driving two 4-ohm speakers.

Boot/Reset Press Keys Boot: holding down the **Boot** button and momentarily pressing the **Reset** button initiates the firmware upload mode. Then user can upload firmware through the serial port. Reset: pressing this button alone resets the system.

Touch Pad Buttons Four touch pads labeled *Play*, *Sel*, *Vol+* and *Vol-*. They are routed to **ESP32-WROVER Module** and intended for development and testing of a UI for audio applications using dedicated API.

Audio Codec Chip The Audio Codec Chip, ES8388, is a low-power stereo audio codec with headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER Module** over I2S and I2S buses to provide audio processing in hardware independently from the audio application.

Function Press Keys Two key labeled *Rec* and *Mode*. They are routed to **ESP32-WROVER Module** and intended for developing and testing a UI for audio applications using dedicated API.

USB-UART Bridge Chip A single chip USB-UART bridge provides up to 1 Mbps transfer rate.

Micro USB Port USB interface. It functions as the power supply for the board and the communication interface between a PC and the ESP32 module.

Standby / Charging LEDs The **Standby** green LED indicates that power has been applied to the **Micro USB Port**. The **Charging** red LED indicates that a battery connected to the **Battery Socket** is being charged.

Battery Charger Chip Constant current & constant voltage linear charger for single cell lithium-ion batteries AP5056. Used for charging of a battery connected to the **Battery Socket** over the **Micro USB Port**.

Power On Switch Power on/off knob: toggling it to the left powers the board on; toggling it to the right powers the board off.

Battery Socket Two pins socket to connect a single cell Li-ion battery.

Power On LED Red LED indicating that **Power On Switch** is turned on.

Note: The **Power On Switch** does not affect / disconnect the Li-ion battery charging.

Hardware Setup Options

There are couple of options to change the hardware configuration of the ESP32-LyraT board. The options are selectable with the **Function DIP Switch**.

Enable MicroSD Card in 1-wire Mode

DIP SW	Position
1	OFF
2	OFF
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF [1]
8	n/a

1. **AUX Input** detection may be enabled by toggling the DIP SW 7 *ON*

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is available for use with the API

Enable MicroSD Card in 4-wire Mode

DIP SW	Position
1	ON
2	ON
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF
8	n/a

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Enable JTAG

DIP SW	Position
1	OFF
2	OFF
3	ON
4	ON
5	ON
6	ON
7	ON
8	n/a

In this mode:

- **MicroSD Card** functionality is not available, remove the card from the slot
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Allocation of ESP32 Pins

Several pins / terminals of ESP32 modules are allocated to the onboard hardware. Some of them, like GPIO0 or GPIO2, have multiple functions. Please refer to tables below or ESP32 LyraT V4 schematic for specific details.

Red / Green LEDs

	ESP32 Pin	LED Color
1	GPIO19	Red LED
2	GPIO22	Green LED

Touch Pads

	ESP32 Pin	Touch Pad Function
1	GPIO33	Play
2	GPIO32	Set
3	GPIO13	Vol- [1]
4	GPIO27	Vol+

1. *Vol-* function is not available if **JTAG** is used. It is also not available for the **MicroSD Card** configured to operate in 4-wire mode.

MicroSD Card / J5

	ESP32 Pin	MicroSD Signal
1	MTDI / GPIO12	DATA2
2	MTCK / GPIO13	CD / DATA3
3	MTDO / GPIO15	CMD
4	MTMS / GPIO14	CLK
5	GPIO2	DATA0
6	GPIO4	DATA1
7	GPIO21	CD

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

I2S Header / JP4

	I2C Header Pin	ESP32 Pin
1	MCLK	GPI0
2	SCLK	GPIO5
1	LRCK	GPIO25
2	DSDIN	GPIO26
3	ASDOUT	GPIO35
3	GND	GND

I2C Header / JP5

	I2C Header Pin	ESP32 Pin
1	SCL	GPIO23
2	SDA	GPIO18
3	GND	GND

JTAG Header / JP7

	ESP32 Pin	JTAG Signal
1	MTDO / GPIO15	TDO
2	MTCK / GPIO13	TCK
3	MTDI / GPIO12	TDI
4	MTMS / GPIO14	TMS

Function DIP Switch / JP8

	Switch OFF	Switch ON
1	GPIO12 not allocated	MicroSD Card 4-wire
2	Touch *Vol-* enabled	MicroSD Card 4-wire
3	MicroSD Card	JTAG
4	MicroSD Card	JTAG
5	MicroSD Card	JTAG
6	MicroSD Card	JTAG
7	MicroSD Card 4-wire	AUX IN detect [1]
8	not used	not used

1. The **AUX Input** signal pin should not be plugged in when the system powers up. Otherwise the ESP32 may not be able to boot correctly.

Start Application Development

Before powering up the ESP32-LyraT, please make sure that the board has been received in good condition with no obvious signs of damage.

Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 4-ohm speakers to the **Right** and **Left Speaker Output**. Optionally connect headphones to the **Headphone Output**.
2. Plug in the Micro-USB cable to the PC and to the **Micro USB Port** of the ESP32-LyraT.
3. The **Standby LED** (green) should turn on. Assuming that a battery is not connected, the **Charging LED** will momentarily blink every couple of seconds.
4. Toggle left the **Power On Switch**.

5. The red **Power On LED** should turn on.

If this is what you see on the LEDs, the board should be ready for application upload. Now prepare the PC by loading and configuring development tools what is discussed in the next section.

Develop Applications

If the ESP32-LyraT is initially set up and checked, you can proceed with preparation of the development tools. Go to section *Get Started*, which will walk you through the following steps:

- *Setup ESP-IDF* in your PC that provides a common framework to develop applications for the ESP32 in C language;
- *Get ESP-ADF* to have the API specific for the audio applications;
- *Setup Path to ESP-ADF* to make the framework aware of the audio specific API;
- *Start a Project* that will provide a sample audio application for the ESP32-LyraT board;
- *Connect and Configure* to prepare the application for loading;
- *Build, Flash and Monitor* this will finally run the application and play some music.

Related Documents

- ESP32 LyraT V4 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging

CHAPTER 2

API Reference

This API provides a way to develop audio applications using *Elements* like *Codecs*, *Streams* or *Filters*.

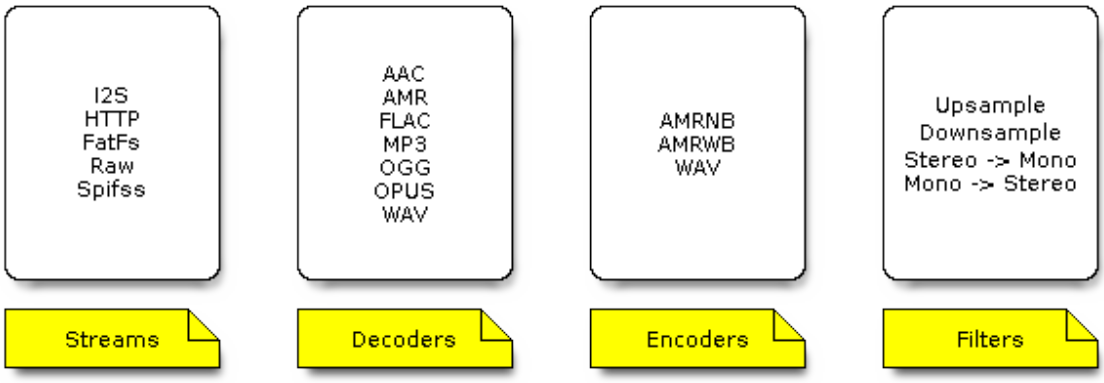


Fig. 2.1: **Elements** of the Audio Development Framework

The application is developed by combining the *Elements* into a *Pipeline*. A diagram below presents organization of two elements, MP3 decoder and I2S stream, in the Audio Pipeline, that has been used in get-started/play_mp3 example.

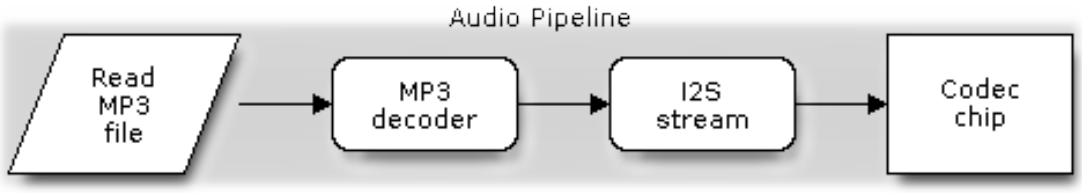


Fig. 2.2: Sample Organization of Elements in Audio Pipeline

The audio data is typically acquired using an input *Stream*, processed with *Codecs* and *Filters*, and finally output with another *Stream*. There is an *Event Interface* to facilitate communication of the application events. Interfacing with specific hardware is done using *Peripherals*.

See a table of contents below with links to description of all the above components.

2.1 Audio Framework

2.1.1 Audio Element

The basic building block for the application programmer developing with ADF is the `audio_element` object. Every decoder, encoder, filter, input stream, or output stream is in fact an Audio Element.

This API has been designed and then used to implement Audio Elements provided by ADF.

The general functionality of an Element is to take some data on input, processes it, and output to a the next. Each Element is run as a separate task. To enable control on particular stages of the data lifecycle from the input, during processing and up to the output, the `audio_element` object provides possibility to trigger callbacks per stage. There are seven types of available callback functions: open, seek, process, close, destroy, read and write, and they are defined in `audio_element_cfg_t`. Particular Elements typically use a subset of all avialable callbacks. For instance the *MP3 Decoder* is using open, process, close and destroy callback functions.

The available Audio Element types intended for development with this API are listed in description of *audio_common.h* header file under `audio_element_type_t` enumerator.

API Reference

Header File

- audio_pipeline/include/audio_element.h

Functions

audio_element_handle_t **audio_element_init** (*audio_element_cfg_t *config*)
Initialize audio element with config.

Return

- audio_elemenent handle object
- NULL

Parameters

- `config`: The configuration

esp_err_t **audio_element_deinit** (*audio_element_handle_t el*)
Destroy audio element handle object, stop, clear, deletel all.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_setdata** (*audio_element_handle_t el*, void **data*)
Set context data to element handle object, it can be retrieve by getdata.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `data`: The data pointer

void ***audio_element_getdata** (*audio_element_handle_t el*)
Get context data from element handle object.

Return data pointer

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_set_tag** (*audio_element_handle_t el*, **const** char **tag*)
Set elemenet tag name, or clear if tag = NULL.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `tag`: The tag name pointer

char ***audio_element_get_tag** (*audio_element_handle_t el*)
Get element tag name.

Return Element tag name pointer

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_setinfo** (*audio_element_handle_t el*, *audio_element_info_t *info*)
Set audio element infomation.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `info`: The information pointer

esp_err_t **audio_element_getinfo** (*audio_element_handle_t el*, *audio_element_info_t *info*)
Get audio element infomation.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `info`: The information pointer

esp_err_t **audio_element_set_uri** (*audio_element_handle_t el*, **const** char *uri)
Set audio element URI.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `uri`: The uri pointer

char ***audio_element_get_uri** (*audio_element_handle_t el*)
Get audio element URI.

Return URI pointer

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_run** (*audio_element_handle_t el*)
Start Audio Element, with this function, audio_element will start as freeRTOS task, And put the task into 'PAUSED' state. Note: Element does not actually start when this function returns.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_terminate** (*audio_element_handle_t el*)
Start Audio Element, with this function, audio_element will start as freeRTOS task, and put the task into 'PAUSED' state. Note: Element does not actually start when this function returns.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_stop**(*audio_element_handle_t el*)

Request stop of the Audio Element. After receiving the stop request, the element will ignore the actions being performed (read/write, wait for the ringbuffer ...) and close the task, reset the state variables. Note: this API only sends requests, Element does not actually stop when this function returns.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_wait_for_stop**(*audio_element_handle_t el*)

After the `audio_element_stop` function is called, the Element task will perform some abort procedures. This function will be blocked until Element Task has done and exit.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_pause**(*audio_element_handle_t el*)

Request audio Element enter 'PAUSE' state, in this state, the task will wait for any event.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_resume**(*audio_element_handle_t el*, float *wait_for_rb_threshold*, TickType_t *timeout*)

Request audio Element enter 'RUNNING' state, in this state, the task listens to event and invokes the callback functions. At the same time it will wait until the size/total_size of the output ringbuffer is greater than or equal to `wait_for_rb_threshold` If the timeout period has been exceeded and ringbuffer output has not yet reached `wait_for_rb_threshold` then the function will return.

- ESP_FAIL

Return

- ESP_OK

Parameters

- `el`: The audio element handle
- `wait_for_rb_threshold`: The wait for rb threshold (0 .. 1)
- `timeout`: The timeout

esp_err_t **audio_element_msg_set_listener**(*audio_element_handle_t el*, *audio_event_iface_handle_t listener*)

This function will add a `listener` to listen to all events from audio element `el` Any event from el->external_event will be send to the `listener`.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `listener`: The event will be listen to

esp_err_t **audio_element_msg_remove_listener**(*audio_element_handle_t el*, *audio_event_iface_handle_t listener*)

Remove listener out of el, no new events will be sent to the listene.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `listener`: The listener

esp_err_t **audio_element_set_input_ringbuf**(*audio_element_handle_t el*, *ringbuf_handle_t rb*)

Set Element input ringbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `rb`: The ringbuffer handle

ringbuf_handle_t **audio_element_get_input_ringbuf**(*audio_element_handle_t el*)

Get Element input ringbuffer.

Return ringbuf_handle_t

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_set_output_ringbuf** (*audio_element_handle_t el*, *ringbuf_handle_t rb*)
Set Element output ringbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `rb`: The ringbuffer handle

ringbuf_handle_t **audio_element_get_output_ringbuf** (*audio_element_handle_t el*)
Get Element output ringbuffer.

Return ringbuf_handle_t

Parameters

- `el`: The audio element handle

audio_element_state_t **audio_element_get_state** (*audio_element_handle_t el*)
Get current Element state.

Return audio_element_state_t

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_abort_input_ringbuf** (*audio_element_handle_t el*)
If the element is requesting data from the input ringbuffer, this function forces it to abort.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_abort_output_ringbuf** (*audio_element_handle_t el*)
If the element is waiting to write data to the ringbuffer output, this function forces it to abort.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_wait_for_buffer** (*audio_element_handle_t el*, int *size_expect*, TickType_t *timeout*)

This function will wait until the sizeof the output ringbuffer is greater than or equal to `size_expect` If the timeout period has been exceeded and ringbuffer output has not yet reached `size_expect` then the function will return `ESP_FAIL`

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `size_expect`: The size expect
- `timeout`: The timeout

esp_err_t **audio_element_report_status** (*audio_element_handle_t el*, *audio_element_status_t status*)

Element will sendout event (status) to event by this function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `status`: The status

esp_err_t **audio_element_report_info** (*audio_element_handle_t el*)

Element will sendout event (information) to event by this function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_report_codec_fmt** (*audio_element_handle_t el*)

Element will sendout event (codec format) to event by this function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_set_input_timeout** (*audio_element_handle_t el*, TickType_t *timeout*)

Set input read timeout (default is `portMAX_DELAY`)

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `timeout`: The timeout

esp_err_t **audio_element_set_output_timeout** (*audio_element_handle_t* *el*, TickType_t *timeout*)
Set output read timeout (default is `portMAX_DELAY`)

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `timeout`: The timeout

esp_err_t **audio_element_reset_input_ringbuf** (*audio_element_handle_t* *el*)
Reset inputbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_reset_output_ringbuf** (*audio_element_handle_t* *el*)
Reset outputbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

audio_element_err_t **audio_element_input** (*audio_element_handle_t* *el*, char **buffer*, int *wanted_size*)
Call this function to provice Element input data. Depending on setup using ringbuffer or function callback, Element invokes read ringbuffer, or calls read callback funtion.

Return

- > 0 number of bytes produced
- <=0 audio_element_err_t

Parameters

- `el`: The audio element handle
- `buffer`: The buffer pointer
- `wanted_size`: The wanted size

audio_element_err_t **audio_element_output** (*audio_element_handle_t el*, char **buffer*, int *write_size*)

Call this function to sendout Element output data. Depending on setup using ringbuffer or function callback, Element will invoke write to ringbuffer, or call write callback funtion.

Return

- > 0 number of bytes written
- <=0 audio_element_err_t

Parameters

- `el`: The audio element handle
- `buffer`: The buffer pointer
- `write_size`: The write size

esp_err_t **audio_element_set_read_cb** (*audio_element_handle_t el*, *stream_func fn*, void **context*)

This API allows the application to set a read callback for the first audio_element in the pipeline for allowing the pipeline to interface with other systems. The callback is invoked every time the audio element requires data to be processed.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `fn`: Callback read function. The callback function should return number of bytes read or -1 in case of error in reading. Note that the callback function may decide to block and that may block the entire pipeline.
- `context`: An optional context which will be passed to callback function on every invocation

esp_err_t **audio_element_set_write_cb** (*audio_element_handle_t el*, *stream_func fn*, void **context*)

This API allows the application to set a write callback for the last audio_element in the pipeline for allowing the pipeline to interface with other systems. The callback is invoked every time the audio element has a processed data that needs to be passed forward.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element
- `fn`: Callback write function. The callback function should return number of bytes written or -1 in case of error in writing. Note that the callback function may decide to block and that may block the entire pipeline.

- `context`: An optional context which will be passed to callback function on every invocation

QueueHandle_t **audio_element_get_event_queue** (*audio_element_handle_t el*)
Get External queue of Emitter, we can read any event that has been send out of Element from this `QueueHandle_t`

Return QueueHandle_t

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_set_ringbuf_done** (*audio_element_handle_t el*)
Set inputbuffer and outputbuffer have finished.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_reset_state** (*audio_element_handle_t el*)
Enforce 'AEL_STATE_INIT' state.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

int **audio_element_get_output_ringbuf_size** (*audio_element_handle_t el*)
Get Element output ringbuffer size.

Return

- =0: Parameter NULL
- >0: Size of ringbuffer

Parameters

- `el`: The audio element handle

esp_err_t **audio_element_multi_input** (*audio_element_handle_t el*, char **buffer*, int *wanted_size*, int *index*, TickType_t *ticks_to_wait*)
Call this function to read data from multi input ringbuffer by given index.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- `el`: The audio element handle
- `buffer`: The buffer pointer
- `wanted_size`: The wanted size
- `index`: The index of multi input ringbuffer, start from `0`, should be less than `NUMBER_OF_MULTI_RINGBUF`
- `ticks_to_wait`: Timeout of ringbuffer

esp_err_t **audio_element_multi_output**(*audio_element_handle_t el*, char **buffer*, int *wanted_size*, TickType_t *ticks_to_wait*)

Call this function write data by multi output ringbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `buffer`: The buffer pointer
- `wanted_size`: The wanted size
- `ticks_to_wait`: Timeout of ringbuffer

esp_err_t **audio_element_set_multi_input_ringbuf**(*audio_element_handle_t el*, *ringbuf_handle_t rb*, int *index*)

Set multi input ringbuffer Element.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle
- `rb`: The ringbuffer handle
- `index`: Index of multi ringbuffer, starts from `0`, should be less than `NUMBER_OF_MULTI_RINGBUF`

esp_err_t **audio_element_set_multi_output_ringbuf**(*audio_element_handle_t el*, *ringbuf_handle_t rb*, int *index*)

Set multi output ringbuffer Element.

Return

- ESP_OK
- ESP_ERR_INVALID_ARG

Parameters

- `el`: The audio element handle
- `rb`: The ringbuffer handle

- `index`: Index of multi ringbuffer, starts from `0`, should be less than `NUMBER_OF_MULTI_RINGBUF`

ringbuf_handle_t **audio_element_get_multi_input_ringbuf**(*audio_element_handle_t* *el*, int *index*)

Get handle of multi input ringbuffer Element by index.

Return

- NULL Error
- Others ringbuf_handle_t

Parameters

- `el`: The audio element handle
- `index`: Index of multi ringbuffer, starts from `0`, should be less than `NUMBER_OF_MULTI_RINGBUF`

ringbuf_handle_t **audio_element_get_multi_output_ringbuf**(*audio_element_handle_t* *el*, int *index*)

Get handle of multi output ringbuffer Element by index.

Return

- NULL Error
- Others ringbuf_handle_t

Parameters

- `el`: The audio element handle
- `index`: Index of multi ringbuffer, starts from `0`, should be less than `NUMBER_OF_MULTI_RINGBUF`

Structures

struct audio_element_reserve_data_t

Audio Element user reserved data.

Public Members

int **user_data_0**

user data 0

int **user_data_1**

user data 1

struct audio_element_info_t

Audio Element informations.

Public Members

int **sample_rates**

Sample rates in Hz

int **channels**
Number of audio channel, mono is 1, stereo is 2

int **bits**
Bit wide (8, 16, 24, 32 bits)

int **volume**
Volume in percent

bool **mute**
Mute

int64_t **byte_pos**
The current position (in bytes) being processed for an element

int64_t **total_bytes**
The total bytes for an element

char ***uri**
URI (optional)

audio_codec_t **codec_fmt**
Music format (optional)

audio_element_reserve_data_t **reserve_data**
This value is reserved for user use

struct audio_element_cfg_t
Audio Element configurations Each Element at startup will be a self-running task. These tasks will execute the callback open -> [loop: read -> process -> write] -> close These callback functions are provided by the user corresponding to this configuration.

Public Members

io_func **open**
Open callback function

io_func **seek**
Seek callback function

process_func **process**
Process callback function

io_func **close**
Close callback function

io_func **destroy**
Destroy callback function

stream_func **read**
Read callback function

stream_func **write**
Write callback function

int **buffer_len**
Buffer length use for an Element

int **task_stack**
Element task stack

int **task_prio**
Element task priority (based on freeRTOS priority)

int **task_core**
Element task running in core (0 or 1)

int **out_rb_size**
Output ringbuffer size

void ***data**
User context

char ***tag**
Element tag

bool **enable_multi_io**
Enable multi input and output ringbuffer

Macros

AUDIO_ELEMENT_INFO_DEFAULT

DEFAULT_ELEMENT_RINGBUF_SIZE

DEFAULT_ELEMENT_BUFFER_LENGTH

DEFAULT_ELEMENT_STACK_SIZE

DEFAULT_ELEMENT_TASK_PRIO

DEFAULT_ELEMENT_TASK_CORE

DEFAULT_AUDIO_ELEMENT_CONFIG

Type Definitions

typedef struct audio_element ***audio_element_handle_t**

typedef esp_err_t (***io_func**) (*audio_element_handle_t* self)

typedef *audio_element_err_t* (***process_func**) (*audio_element_handle_t* self, char *el_buffer, int el_buf_len)

typedef *audio_element_err_t* (***stream_func**) (*audio_element_handle_t* self, char *buffer, int len, TickType_t ticks_to_wait, void *context)

Enumerations

enum audio_element_err_t
Values:

AEL_IO_OK = ESP_OK

AEL_IO_FAIL = ESP_FAIL

AEL_IO_DONE = -2

AEL_IO_ABORT = -3

AEL_IO_TIMEOUT = -4

AEL_PROCESS_FAIL = -5

enum audio_element_state_t
Audio element state.

Values:

AEL_STATE_NONE = 0

AEL_STATE_INIT

AEL_STATE_RUNNING

AEL_STATE_PAUSED

AEL_STATE_STOPPED

AEL_STATE_FINISHED

AEL_STATE_ERROR

enum audio_element_msg_cmd_t
Audio element action command, process on dispatcher

Values:

AEL_MSG_CMD_NONE = 0

AEL_MSG_CMD_ERROR = 1

AEL_MSG_CMD_FINISH = 2

AEL_MSG_CMD_STOP = 3

AEL_MSG_CMD_PAUSE = 4

AEL_MSG_CMD_RESUME = 5

AEL_MSG_CMD_DESTROY = 6

AEL_MSG_CMD_REPORT_STATUS = 8

AEL_MSG_CMD_REPORT_MUSIC_INFO = 9

AEL_MSG_CMD_REPORT_CODEC_FMT = 10

enum audio_element_status_t
Audio element status report

Values:

AEL_STATUS_NONE = 0

AEL_STATUS_ERROR_OPEN = 1

AEL_STATUS_ERROR_INPUT = 2

AEL_STATUS_ERROR_PROCESS = 3

AEL_STATUS_ERROR_OUTPUT = 4

AEL_STATUS_ERROR_CLOSE = 5

AEL_STATUS_ERROR_TIMEOUT = 6

AEL_STATUS_ERROR_UNKNOWN = 7

AEL_STATUS_INPUT_DONE = 8

AEL_STATUS_INPUT_BUFFERING = 9

AEL_STATUS_OUTPUT_DONE = 10

AEL_STATUS_OUTPUT_BUFFERING = 11

AEL_STATUS_STATE_RUNNING = 12

AEL_STATUS_STATE_PAUSED = 13

AEL_STATUS_STATE_STOPPED = 14

AEL_STATUS_MOUNTED = 15

AEL_STATUS_UNMOUNTED = 16

2.1.2 Audio Pipeline

Dynamic combination of a group of linked *Elements* is done using the Audio Pipeline. You do not deal with the individual elements but with just one audio pipeline. Every element is connected by a ringbuffer. The Audio Pipeline also takes care of forwarding messages from the element tasks to an application.

A diagram below presents organization of three elements, HTTP reader stream, MP3 decoder and I2S writer stream, in the Audio Pipeline, that has been used in player/pipeline_http_mp3 example.

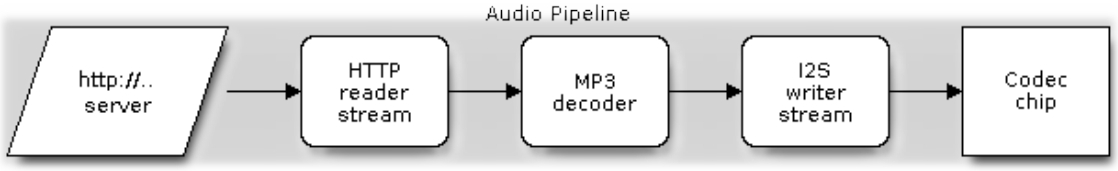


Fig. 2.3: Sample Organization of Elements in Audio Pipeline

API Reference

Header File

- audio_pipeline/include/audio_pipeline.h

Functions

audio_pipeline_handle_t **audio_pipeline_init** (*audio_pipeline_cfg_t *config*)

Initialize audio_pipeline_handle_t object audio_pipeline is responsible for controlling the audio data stream and connecting the audio elements with the ringbuffer It will connect and start the audio element in order, responsible for retrieving the data from the previous element and passing it to the element after it. Also get events from each element, process events or pass it to a higher layer.

Return

- audio_pipeline_handle_t on success
- NULL when any errors

Parameters

- `config`: The configuration - audio_pipeline_cfg_t

esp_err_t **audio_pipeline_deinit** (*audio_pipeline_handle_t pipeline*)

This function removes all of the element's links in audio_pipeline, cancels the registration of all events, invokes the destroy functions of the registered elements, and frees the memory allocated by the init function. Briefly, frees all memory.

Return ESP_OK

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_register** (*audio_pipeline_handle_t pipeline*, *audio_element_handle_t el*, **const** char ***name*)

Registering an element for audio_pipeline, each element can be registered multiple times, but `name` (as String) must be unique in audio_pipeline, which is used to identify the element for link creation mentioned in the `audio_pipeline_link`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle
- `el`: The Audio Element Handle
- `name`: The name identifier of the audio_element in this audio_pipeline

esp_err_t **audio_pipeline_unregister** (*audio_pipeline_handle_t pipeline*, *audio_element_handle_t el*)

Unregister the audio_element in audio_pipeline, remove it from the list.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle
- `el`: The Audio Element Handle

esp_err_t **audio_pipeline_run** (*audio_pipeline_handle_t pipeline*)

Start Audio Pipeline.

With this function audio_pipeline will create tasks for all elements, that have been linked using the linking functions.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_terminate**(*audio_pipeline_handle_t pipeline*)

Stop Audio Pipeline.

With this function audio_pipeline will destroy tasks of all elements, that have been linked using the linking functions.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_resume**(*audio_pipeline_handle_t pipeline*)

This function will set all the elements to the `RUNNING` state and process the audio data as an inherent feature of audio_pipeline.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_pause**(*audio_pipeline_handle_t pipeline*)

This function will set all the elements to the `PAUSED` state. Everything remains the same except the data processing is stopped.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_stop**(*audio_pipeline_handle_t pipeline*)

Stop all elements and clear information of items. Free up memory for all task items. The link state of the elements in the pipeline is kept, events are still registered, but the `audio_pipeline_pause` and `audio_pipeline_resume` functions have no effect. To restart audio_pipeline, use the `audio_pipeline_resume` function.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_wait_for_stop**(*audio_pipeline_handle_t pipeline*)

The `audio_pipeline_stop` function sends requests to the elements and exits. But they need time to get rid of time-blocking tasks. This function will wait until all the Elements in the pipeline actually stop.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_link**(*audio_pipeline_handle_t pipeline*, **const** char **link_tag*[], int *link_num*)

The audio_element added to audio_pipeline will be unconnected before it is called by this function. Based on element's `name` already registered by `audio_pipeline_register`, the path of the data will be linked in the order of the link_tag. Element at index 0 is first, and index `link_num -1` is final. As well as audio_pipeline will subscribe all element's events.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle
- `link_tag`: Array of element `name` was registered by `audio_pipeline_register`
- `link_num`: Total number of elements of the `link_tag` array

esp_err_t **audio_pipeline_unlink**(*audio_pipeline_handle_t pipeline*)

Removes the connection of the elements, as well as unsubscribe events.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

audio_element_handle_t **audio_pipeline_get_el_by_tag**(*audio_pipeline_handle_t pipeline*, **const** char **tag*)

Find element from registered pipeline by tag.

Return

- NULL when any errors
- Others on success

Parameters

- `pipeline`: The Audio Pipeline Handle
- `tag`: A char pointer

esp_err_t **audio_pipeline_remove_listener**(*audio_pipeline_handle_t pipeline*)
Remove event listener from this audio_pipeline.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_set_listener**(*audio_pipeline_handle_t pipeline*, *audio_event_iface_handle_t evt*)
Set event listner for this audio_pipeline, any event from this pipeline can be listen to by `evt`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle
- `evt`: The Event Handle

audio_event_iface_handle_t **audio_pipeline_get_event_iface**(*audio_pipeline_handle_t pipeline*)
Get the event iface using by this pipeline.

Return The Event Handle

Parameters

- `pipeline`: The pipeline

esp_err_t **audio_pipeline_link_insert**(*audio_pipeline_handle_t pipeline*, bool *first*, *audio_element_handle_t prev*, *ringbuf_handle_t conect_rb*, *audio_element_handle_t next*)
Insert the specific audio_element to audio_pipeline, previous element connect to the next element by ring buffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pipeline`: The audio pipeline handle
- `first`: Previous element is first input element, need to set `true`
- `prev`: Previous element
- `conect_rb`: Connect ring buffer
- `next`: Next element

esp_err_t **audio_pipeline_register_more**(*audio_pipeline_handle_t pipeline*, *audio_element_handle_t element_1*, ...)
Register a NULL-terminated list of elements to audio_pipeline.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pipeline`: The audio pipeline handle
- `element_1`: The element to add to the audio_pipeline.
- `...`: Additional elements to add to the audio_pipeline.

esp_err_t **audio_pipeline_unregister_more**(*audio_pipeline_handle_t* *pipeline*, *audio_element_handle_t* *element_1*, ...)

Unregister a NULL-terminated list of elements to audio_pipeline.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pipeline`: The audio pipeline handle
- `element_1`: The element to add to the audio_pipeline.
- `...`: Additional elements to add to the audio_pipeline.

esp_err_t **audio_pipeline_link_more**(*audio_pipeline_handle_t* *pipeline*, *audio_element_handle_t* *element_1*, ...)

Adds a NULL-terminated list of elements to audio_pipeline.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pipeline`: The audio pipeline handle
- `element_1`: The element to add to the audio_pipeline.
- `...`: Additional elements to add to the audio_pipeline.

esp_err_t **audio_pipeline_listen_more**(*audio_pipeline_handle_t* *pipeline*, *audio_element_handle_t* *element_1*, ...)

Subscribe a NULL-terminated list of element's events to audio_pipeline.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `pipeline`: The audio pipeline handle
- `element_1`: The element event to subscribe to the audio_pipeline.
- `...`: Additional elements event to subscribe to the audio_pipeline.

esp_err_t **audio_pipeline_check_items_state**(*audio_pipeline_handle_t pipeline*, *audio_element_handle_t dest_el*, *audio_element_status_t status*)

Update the destination element state and check the all of linked elements state are same.

Return

- ESP_OK All linked elements state are same.
- ESP_FAIL All linked elements state are not same.

Parameters

- `pipeline`: The audio pipeline handle
- `dest_el`: Destination element
- `status`: The new status

esp_err_t **audio_pipeline_reset_items_state**(*audio_pipeline_handle_t pipeline*)

Reset pipeline element items state to `AEL_STATUS_NONE`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_reset_ringbuffer**(*audio_pipeline_handle_t pipeline*)

Reset pipeline element ringbuffer.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_breakup_elements**(*audio_pipeline_handle_t pipeline*, *audio_element_handle_t kept_ctx_el*)

Break up all the linked elements of specific `pipeline`. The include and before `kept_ctx_el` elements and connected ringbuffer will be reserved.

Note There is no element reserved when `kept_ctx_el` is NULL. This function will unsubscribe all element's events.

Return

- ESP_OK All linked elements state are same.
- ESP_ERR_INVALID_ARG Invalid parameters.

Parameters

- `pipeline`: The audio pipeline handle
- `kept_ctx_el`: Destination keep elements

esp_err_t **audio_pipeline_relink**(*audio_pipeline_handle_t* *pipeline*, **const** char **link_tag*[], int *link_num*)

Basing on element's `name` already registered by `audio_pipeline_register`, relink the pipeline following the order of `names` in the 'link_tag.

Note If the ringbuffer is not enough to connect the new pipeline will create new ringbuffer.

Return

- ESP_OK All linked elements state are same.
- ESP_FAIL Error.
- ESP_ERR_INVALID_ARG Invalid parameters.

Parameters

- `pipeline`: The Audio Pipeline Handle
- `link_tag`: Array of elements `name` that was registered by `audio_pipeline_register`
- `link_num`: Total number of elements of the `link_tag` array

Structures

struct audio_pipeline_cfg

Audio Pipeline configurations.

Public Members

int **rb_size**

Audio Pipeline ringbuffer size

Macros

DEFAULT_PIPELINE_RINGBUF_SIZE

DEFAULT_AUDIO_PIPELINE_CONFIG

Type Definitions

typedef struct audio_pipeline ***audio_pipeline_handle_t**

typedef struct *audio_pipeline_cfg* **audio_pipeline_cfg_t**

Audio Pipeline configurations.

2.1.3 Event Interface

The ADF provides the Event Interface API to establish communication between Audio Elements in a pipeline. The API is built around FreeRTOS queue. It implements 'listeners' to watch for incoming messages and inform about them with a callback function.

Application Examples

Implementation of this API is demonstrated in couple of examples including get-started/play_mp3.

API Reference

Header File

- audio_pipeline/include/audio_event_iface.h

Functions

audio_event_iface_handle_t **audio_event_iface_init** (*audio_event_iface_cfg_t* **config*)
Initialize audio event.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `config`: The configurations

esp_err_t **audio_event_iface_destroy** (*audio_event_iface_handle_t evt*)
Cleanup event, it doesn't free evt pointer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

esp_err_t **audio_event_iface_set_listener** (*audio_event_iface_handle_t evt*, *audio_event_iface_handle_t listener*)
Add audio event `evt` to the listener, then we can listen `evt` event from `listen`

Return

- ESP_OK
- ESP_FAIL

Parameters

- `listener`: The event can listen another event
- `evt`: The event to be added to

esp_err_t **audio_event_iface_remove_listener** (*audio_event_iface_handle_t listener*, *audio_event_iface_handle_t evt*)
Remove audio event `evt` from the listener.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `listener`: The event listener
- `evt`: The event to be removed from

esp_err_t **audio_event_iface_set_cmd_waiting_timeout**(*audio_event_iface_handle_t evt*, TickType_t *wait_time*)

Set current queue wait time for the event.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event
- `wait_time`: The wait time

esp_err_t **audio_event_iface_waiting_cmd_msg**(*audio_event_iface_handle_t evt*)

Waiting internal queue message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

esp_err_t **audio_event_iface_cmd**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)

Trigger an event for internal queue with a message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event
- `msg`: The message

esp_err_t **audio_event_iface_cmd_from_isr**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)

It's same with `audio_event_iface_cmd`, but can send a message from ISR.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event
- `msg`: The message

esp_err_t **audio_event_iface_sendout**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)

Trigger and event out with a message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event
- `msg`: The message

esp_err_t **audio_event_iface_discard**(*audio_event_iface_handle_t evt*)

Discard all ongoing event message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

esp_err_t **audio_event_iface_listen**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*, TickType_t *wait_time*)

Listening and invoke callback function if there are any event are comming.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event
- `msg`: The message
- `wait_time`: The wait time

QueueHandle_t **audio_event_iface_get_queue_handle**(*audio_event_iface_handle_t evt*)

Get External queue handle of Emmitter.

Return External QueueHandle_t

Parameters

- `evt`: The external queue

esp_err_t **audio_event_iface_read**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*, TickType_t *wait_time*)

Read the event from all the registered event emitters in the queue set of the interface.

Return

- ESP_OK On successful receiving of event
- ESP_FAIL In case of a timeout or invalid parameter passed

Parameters

- `evt`: The event interface
- `msg`: The pointer to structure in which event is to be received
- `wait_time`: Timeout for receiving event

QueueHandle_t **audio_event_iface_get_msg_queue_handle**(*audio_event_iface_handle_t evt*)

Get Internal queue handle of Emmitter.

Return Internal QueueHandle_t

Parameters

- `evt`: The Internal queue

esp_err_t **audio_event_iface_set_msg_listener**(*audio_event_iface_handle_t evt*, *audio_event_iface_handle_t listener*)

Add audio internal event `evt` to the listener, then we can listen `evt` event from `listen`

Return

- ESP_OK
- ESP_FAIL

Parameters

- `listener`: The event can listen another event
- `evt`: The event to be added to

Structures

struct audio_event_iface_msg_t

Event message

Public Members

int **cmd**

Command id

void ***data**

Data pointer

int **data_len**

Data length

void ***source**

Source event

int **source_type**

Source type (To know where it came from)

bool **need_free_data**
Need to free data pointer after the event has been processed

struct audio_event_iface_cfg_t
Event interface configurations

Public Members

int **internal_queue_size**
It's optional, Queue size for event `internal_queue`

int **external_queue_size**
It's optional, Queue size for event `external_queue`

int **queue_set_size**
It's optional, QueueSet size for event `queue_set`

on_event_iface_func **on_cmd**
Function callback for listener when any event arrived

void ***context**
Context will pass to callback function

TickType_t **wait_time**
Timeout to check for event queue

int **type**
it will pass to *audio_event_iface_msg_t* source_type (To know where it came from)

Macros

DEFAULT_AUDIO_EVENT_IFACE_SIZE

AUDIO_EVENT_IFACE_DEFAULT_CFG

Type Definitions

typedef esp_err_t (***on_event_iface_func**) (*audio_event_iface_msg_t* *, void *)

typedef struct audio_event_iface ***audio_event_iface_handle_t**

2.1.4 Audio Common

Enumerations that define type of *Audio Elements*, type and format of *Codecs* and type of *Streams*.

API Reference

Header File

- audio_pipeline/include/audio_common.h

Macros

ELEMENT_SUB_TYPE_OFFSET

mem_assert(x)

Enumerations

enum audio_element_type_t

Values:

AUDIO_ELEMENT_TYPE_UNKNOW = 0x01<<ELEMENT_SUB_TYPE_OFFSET

AUDIO_ELEMENT_TYPE_ELEMENT = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+1)

AUDIO_ELEMENT_TYPE_PLAYER = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+2)

AUDIO_ELEMENT_TYPE_SERVICE = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+3)

AUDIO_ELEMENT_TYPE_PERIPH = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+4)

enum audio_stream_type_t

Values:

AUDIO_STREAM_NONE = 0

AUDIO_STREAM_READER

AUDIO_STREAM_WRITER

enum audio_codec_type_t

Values:

AUDIO_CODEC_TYPE_NONE = 0

AUDIO_CODEC_TYPE_DECODER

AUDIO_CODEC_TYPE_ENCODER

enum audio_codec_t

Values:

AUDIO_CODEC_NONE = 0

AUDIO_CODEC_RAW

AUDIO_CODEC_WAV

AUDIO_CODEC_MP3

AUDIO_CODEC_AAC

AUDIO_CODEC_OPUS

AUDIO_PLAYLIST

2.1.5 ESP Audio

This component provides several simple high level APIs. It is intended for quick implementation of audio applications based on typical interconnections of standardized audio elements.

API Reference

Header File

- esp-adf-libs/esp_audio/include/audio_def.h

Structures

struct esp_audio_state_t
esp_audio status information parameters

Public Members

esp_audio_status_t **status**
Status of esp_audio

audio_err_t **err_msg**
Status is AUDIO_STATUS_ERROR,err_msg will be setup

Macros

ESP_ERR_AUDIO_BASE
Starting number of ESP audio error codes

Type Definitions

typedef enum *audio_err_t* **audio_err_t**

typedef enum *esp_audio_status_t* **esp_audio_status_t**

Enumerations

enum audio_err_t
Values:

ESP_ERR_AUDIO_NO_ERROR = ESP_OK

ESP_ERR_AUDIO_FAIL = ESP_FAIL

ESP_ERR_AUDIO_NO_INPUT_STREAM = ESP_ERR_AUDIO_BASE + 1

ESP_ERR_AUDIO_NO_OUTPUT_STREAM = ESP_ERR_AUDIO_BASE + 2

ESP_ERR_AUDIO_NO_CODEC = ESP_ERR_AUDIO_BASE + 3

ESP_ERR_AUDIO_HAL_FAIL = ESP_ERR_AUDIO_BASE + 4

ESP_ERR_AUDIO_MEMORY_LACK = ESP_ERR_AUDIO_BASE + 5

ESP_ERR_AUDIO_INVALID_URI = ESP_ERR_AUDIO_BASE + 6

ESP_ERR_AUDIO_INVALID_PATH = ESP_ERR_AUDIO_BASE + 7

ESP_ERR_AUDIO_INVALID_PARAMETER = ESP_ERR_AUDIO_BASE + 8

ESP_ERR_AUDIO_NOT_READY = ESP_ERR_AUDIO_BASE + 9

ESP_ERR_AUDIO_NOT_SUPPORT = ESP_ERR_AUDIO_BASE + 10

ESP_ERR_AUDIO_TIMEOUT = ESP_ERR_AUDIO_BASE + 11

ESP_ERR_AUDIO_ALREADY_EXISTS = ESP_ERR_AUDIO_BASE + 12

ESP_ERR_AUDIO_UNKNOWN = ESP_ERR_AUDIO_BASE + 13

ESP_ERR_AUDIO_OPEN = ESP_ERR_AUDIO_BASE + 0x100

ESP_ERR_AUDIO_INPUT = ESP_ERR_AUDIO_BASE + 0x101

ESP_ERR_AUDIO_PROCESS = ESP_ERR_AUDIO_BASE + 0x102

ESP_ERR_AUDIO_OUTPUT = ESP_ERR_AUDIO_BASE + 0x103

ESP_ERR_AUDIO_CLOSE = ESP_ERR_AUDIO_BASE + 0x104

enum esp_audio_status_t

Values:

AUDIO_STATUS_UNKNOWN = 0

AUDIO_STATUS_RUNNING = 1

AUDIO_STATUS_PAUSED = 2

AUDIO_STATUS_STOPED = 3

AUDIO_STATUS_ERROR = 4

enum audio_termination_type_t

Values:

TERMINATION_TYPE_NOW = 0

Audio operation will be terminated immediately

TERMINATION_TYPE_DONE = 1

Audio operation will be stopped untill is finished

TERMINATION_TYPE_MAX

Header File

- esp-adf-libs/esp_audio/include/esp_audio.h

Functions

esp_audio_handle_t **esp_audio_create**(**const** *esp_audio_cfg_t* **cfg*)

Create esp_audio instance according to 'cfg' parameter.

This function create an esp_audio instance, at the specified configuration.

Return

- NULL: Error
- Others: esp_audio instance fully certifying

Parameters

- `cfg`: Provide esp_audio initialization configuration

audio_err_t **esp_audio_destroy**(*esp_audio_handle_t handle*)

Specific esp_audio instance will be destroyed.

Return

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no instance to free, call esp_audio_init first

Parameters

- `handle`: The esp_audio instance

audio_err_t **esp_audio_input_stream_add**(*esp_audio_handle_t handle*, *audio_element_handle_t in_stream*)

Add audio input stream to specific esp_audio instance.

Return

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_ALREADY_EXISTS: in_stream has already exist or have the same stream tag.

Parameters

- `handle`: The esp_audio instance
- `in_stream`: Audio stream instance

audio_err_t **esp_audio_output_stream_add**(*esp_audio_handle_t handle*, *audio_element_handle_t out_stream*)

Add audio output stream to specific esp_audio instance.

Return

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_ALREADY_EXISTS: out_stream has already exist or have the same stream tag.

Parameters

- `handle`: The esp_audio instance
- `out_stream`: The audio stream element instance

audio_err_t **esp_audio_codec_lib_add**(*esp_audio_handle_t handle*, *audio_codec_type_t type*, *audio_element_handle_t lib*)

Add a new codec lib that can decode or encode a music file.

Return

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_ALREADY_EXISTS: lib has already exist or have the same extension.

Parameters

- `handle`: The esp_audio instance
- `type`: The audio codec type(encoder or decoder)

- `lib`: To provide audio stream element

audio_err_t **esp_audio_codec_lib_query**(*esp_audio_handle_t* *handle*, *audio_codec_type_t* *type*, **const** char **extension*)

Check if this kind of music extension is supported or not.

Note This function just query the codec which has already add by esp_audio_codec_lib_add. The max length of extension is 6.

Return

- ESP_ERR_AUDIO_NO_ERROR: supported
- ESP_ERR_AUDIO_NOT_SUPPORT: not support
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

Parameters

- `handle`: The esp_audio instance
- `type`: The CODEC_ENCODER or CODEC_DECODER
- `extension`: Such as "mp3", "wav", "aac"

audio_err_t **esp_audio_play**(*esp_audio_handle_t* *handle*, *audio_codec_type_t* *type*, **const** char **uri*, int *pos*)

Play the given uri.

The esp_audio_play have follow activity, setup inputstream, outputstream and codec by uri, start all of them. There is a rule that esp_audio will select input stream, codec and output stream by URI field.

Rule of URI field are as follow.

- `UF_SCHEMA` field of URI for choose input stream from existing streams. e.g:"http","file"
- `UF_PATH` field of URI for choose codec from existing codecs. e.g:"/audio/mp3_music.mp3"
- `UF_FRAGMENT` field of URI for choose output stream from existing streams, output stream is I2S by default.
- `UF_USERINFO` field of URI for specific sample rate and channels at encode mode.

The format "user:password" in the userinfo field, "user" is sample rate, "password" is channels.

Now esp_audio_play support follow URIs.

- "https://dl.espressif.com/dl/audio/mp3_music.mp3"
- "http://media-ice.musicradio.com/ClassicFMMP3"
- "file://sdcard/test.mp3"
- "iis://16000:2@from.pcm/rec.wav#file"
- "iis://16000:1@record.pcm/record.wav#raw"

Note

- The URI parse by `http_parser_parse_url`,any illegal string will be return `ESP_ERR_AUDIO_INVALID_URI`.
- If the esp_decoder codec is added to `handle`, then the `handle` of esp_decoder will be set as the default decoder, even if other decoders are added.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_TIMEOUT: timeout the play activity.
- ESP_ERR_AUDIO_INVALID_URI: URI is illegal
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

Parameters

- `handle`: The esp_audio_handle_t instance
- `uri`: Such as "file://sdcard/test.wav" or "http://iot.espressif.com/file/example.mp3",
- `type`: Specific handle type decoder or encoder
- `pos`: Specific starting position by bytes

audio_err_t **esp_audio_stop** (*esp_audio_handle_t handle*, *audio_termination_type_t type*)

Stop the esp_audio.

Note If user queue has been registered by evt_que, AUDIO_STATUS_STOPED event for success or AUDIO_STATUS_ERROR event for error will be received. `TERMINATION_TYPE_DONE` only works with input stream which can't stoped by itself, e.g. `raw read/write stream`, others streams are no effect.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_TIMEOUT: timeout the stop activity.

Parameters

- `handle`: The esp_audio instance
- `type`: Stop immediately or done

audio_err_t **esp_audio_pause** (*esp_audio_handle_t handle*)

Pause the esp_audio.

Note Only support music and without live stream. If user queue has been registered by evt_que, AUDIO_STATUS_PAUSED event for success or AUDIO_STATUS_ERROR event for error will be received.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_TIMEOUT: timeout the pause activity.

Parameters

- `handle`: The esp_audio instance

audio_err_t **esp_audio_resume** (*esp_audio_handle_t handle*)

Resume the music paused.

Note Only support music and without live stream. If user queue has been registered by evt_que, AUDIO_STATUS_PLAYING event for success or AUDIO_STATUS_ERROR event for error will be received.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_TIMEOUT: timeout the resume activity.

Parameters

- `handle`: The esp_audio instance

audio_err_t **esp_audio_vol_set** (*esp_audio_handle_t handle*, int *vol*)
Setting esp_audio volume.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_CTRL_HAL_FAIL: error with hardware.
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

Parameters

- `handle`: The esp_audio instance
- `vol`: Specific volume will be set. 0-100 is legal. 0 will be mute.

audio_err_t **esp_audio_vol_get** (*esp_audio_handle_t handle*, int **vol*)
Get esp_audio volume.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_CTRL_HAL_FAIL: error with hardware.
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

Parameters

- `handle`: The esp_audio instance
- `vol`: A pointer to int that indicates esp_audio volume.

audio_err_t **esp_audio_state_get** (*esp_audio_handle_t handle*, *esp_audio_state_t* **state*)
Get esp_audio status.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance or esp_audio does not playing

Parameters

- `handle`: The esp_audio instance
- `state`: A pointer to *esp_audio_state_t* that indicates esp_audio status.

audio_err_t **esp_audio_pos_get** (*esp_audio_handle_t handle*, int **pos*)
Get the position in bytes of currently played music.

Note This function works only with decoding music.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_NOT_READY:no out stream.

Parameters

- `handle`: The esp_audio instance
- `pos`: A pointer to int that indicates esp_audio decoding position.

audio_err_t **esp_audio_time_get**(*esp_audio_handle_t handle*, int **time*)
Get the position in microseconds of currently played music.

Note This function works only with decoding music.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_NOT_READY:no out stream.

Parameters

- `handle`: The esp_audio instance
- `time`: A pointer to int that indicates esp_audio decoding position.

audio_err_t **esp_audio_setup**(*esp_audio_handle_t handle*, *esp_audio_setup_t *sets*)
Choose the `in_stream`, `codec` and `out_stream` definitely, and set `uri`.

Note This function provide a manual way to select in/out stream and codec, should be called before the `esp_audio_play`, then ignore the `esp_audio_play` URI parameter only one time.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_MEMORY_LACK: allocate memory fail

Parameters

- `handle`: The esp_audio instance
- `sets`: A pointer to *esp_audio_setup_t*.

Structures

struct esp_audio_cfg_t
esp_audio configuration parameters

Public Members

int **in_stream_buf_size**
Input buffer size

int **out_stream_buf_size**
Output buffer size

int **resample_rate**
Destination sample rate,0:disable rsample;others:44.1K,48K,32K,16K,8K has supported

audio_hal_handle_t **hal**
Codec IC hardware handle

QueueHandle_t **evt_que**
For received esp_audio events (optional)

int **task_prio**
esp_audio task priority

struct esp_audio_setup_t
esp_audio setup parameters by manual

Public Members

audio_codec_type_t **set_type**
Set codec type

int **set_sample_rate**
Set music sample rate

int **set_channel**
Set music channels

int **set_pos**
Set starting position

char ***set_path**
Set out stream path

char ***set_uri**
Set URI

char ***set_in_stream**
Tag of in_stream

char ***set_codec**
Tag of the codec

char ***set_out_stream**
Tag of out_stream

Type Definitions

typedef void ***esp_audio_handle_t**

typedef struct *esp_audio_cfg_t* **esp_audio_cfg_t**
esp_audio configuration parameters

typedef struct *esp_audio_setup_t* **esp_audio_setup_t**
esp_audio setup parameters by manual

2.2 Audio Streams

An *Audio Element* responsible for acquiring of audio data and then sending the data out after processing, is called the Audio Stream.

The following stream types are supported:

- *I2S Stream*
- *HTTP Stream*
- *FatFs Stream*
- *Raw Stream*
- *Spiffs Stream*

To set the stream type, use provided structure, e.g. `i2s_stream_cfg_t` for I2S stream, together with `audio_stream_type_t` enumerator.

See description below for the API details.

2.2.1 I2S Stream

When the I2S stream type is "writer", the data may be sent either to a codec chip or to the internal DAC of ESP32. To simplify configuration, two macros are provided to cover each case:

- `I2S_STREAM_CFG_DEFAULT` - the I2S stream is communicating with a codec chip
- `I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT` - the stream data are sent to the DAC

Each macro configures several other stream parameters such as sample rate, bits per sample, DMA buffer length, etc.

Header File

- audio_stream/include/i2s_stream.h

Functions

audio_element_handle_t **i2s_stream_init** (*i2s_stream_cfg_t* **config*)

Create a handle to an Audio Element to stream data from I2S to another Element or get data from other elements sent to I2S, depending on the configuration of stream type is AUDIO_STREAM_READER or AUDIO_STREAM_WRITER.

Note If I2S stream is enabled with built-in DAC mode, please don't use I2S_NUM_1. The built-in DAC functions are only supported on I2S0 for the current ESP32 chip.

Return The Audio Element handle

Parameters

- `config`: The configuration

esp_err_t **i2s_stream_set_clk** (*audio_element_handle_t* *i2s_stream*, int *rate*, int *bits*, int *ch*)

Setup clock for I2S Stream, this function is only used with handle created by `i2s_stream_init`

Return

- ESP_OK

- ESP_FAIL

Parameters

- `i2s_stream`: The i2s element handle
- `rate`: Clock rate (in Hz)
- `bits`: Audio bit width (8, 16, 24, 32)
- `ch`: Number of Audio channels (1: Mono, 2: Stereo)

Structures

struct i2s_stream_cfg_t

I2S Stream configurations Default value will be used if any entry is zero.

Public Members

audio_stream_type_t **type**
Type of stream

i2s_config_t **i2s_config**
I2S driver configurations

i2s_pin_config_t **i2s_pin_config**
I2S driver hardware pin configurations

i2s_port_t **i2s_port**
I2S driver hardware port

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

I2S_STREAM_TASK_STACK

I2S_STREAM_BUF_SIZE

I2S_STREAM_TASK_PRIO

I2S_STREAM_TASK_CORE

I2S_STREAM_RINGBUFFER_SIZE

I2S_STREAM_CFG_DEFAULT

I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT

2.2.2 HTTP Stream

Header File

- audio_stream/include/http_stream.h

Functions

audio_element_handle_t **http_stream_init** (*http_stream_cfg_t *config*)

Create a handle to an Audio Element to stream data from HTTP to another Element or get data from other elements sent to HTTP, depending on the configuration the stream type, either AUDIO_STREAM_READER or AUDIO_STREAM_WRITER.

Return The Audio Element handle

Parameters

- `config`: The configuration

esp_err_t **http_stream_next_track** (*audio_element_handle_t el*)

esp_err_t **http_stream_restart** (*audio_element_handle_t el*)

Structures

struct http_stream_event_msg_t

Stream event message.

Public Members

http_stream_event_id_t **event_id**

Event ID

void ***http_client**

Reference to HTTP Client using by this HTTP Stream

void ***buffer**

Reference to Buffer using by the Audio Element

int **buffer_len**

Length of buffer

void ***user_data**

User data context, from *`http_stream_cfg_t`*

audio_element_handle_t **el**

Audio element context

struct http_stream_cfg_t

HTTP Stream configurations Default value will be used if any entry is zero.

Public Members

audio_stream_type_t **type**
Type of stream

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

http_stream_event_handle_t **event_handle**
The hook function for HTTP Stream

void ***user_data**
User data context

bool **enable_playlist_parser**
Enable playlist parser

Macros

HTTP_STREAM_TASK_STACK

HTTP_STREAM_TASK_CORE

HTTP_STREAM_TASK_PRIO

HTTP_STREAM_RINGBUFFER_SIZE

HTTP_STREAM_CFG_DEFAULT

Type Definitions

typedef int (***http_stream_event_handle_t**) (*http_stream_event_msg_t* *msg)

Enumerations

enum http_stream_event_id_t
HTTP Stream hook type.

Values:

HTTP_STREAM_PRE_REQUEST = 0x01
The event handler will be called before HTTP Client making the connection to the server

HTTP_STREAM_ON_REQUEST
The event handler will be called when HTTP Client is requesting data, If the fucntion return the value (-1: ESP_FAIL), HTTP Client will be stopped If the fucntion return the value > 0, HTTP Stream will ignore the post_field If the fucntion return the value = 0, HTTP Stream continue send data from post_field (if any)

HTTP_STREAM_ON_RESPONSE
The event handler will be called when HTTP Client is receiving data If the fucntion return the value (-1: ESP_FAIL), HTTP Client will be stopped If the fucntion return the value > 0, HTTP Stream will ignore the read function If the fucntion return the value = 0, HTTP Stream continue read data from HTTP Server

HTTP_STREAM_POST_REQUEST
The event handler will be called after HTTP Client send header and body to the server, before fetching the headers

HTTP_STREAM_FINISH_REQUEST
The event handler will be called after HTTP Client fetch the header and ready to read HTTP body

HTTP_STREAM_RESOLVE_ALL_TRACKS

HTTP_STREAM_FINISH_TRACK

HTTP_STREAM_FINISH_PLAYLIST

2.2.3 FatFs Stream

Header File

- audio_stream/include/fatfs_stream.h

Functions

audio_element_handle_t **fatfs_stream_init** (*fatfs_stream_cfg_t *config*)
Create a handle to an Audio Element to stream data from FatFs to another Element or get data from other elements written to FatFs, depending on the configuration the stream type, either AUDIO_STREAM_READER or AUDIO_STREAM_WRITER.

Return The Audio Element handle

Parameters

- `config`: The configuration

Structures

struct fatfs_stream_cfg_t
FATFS Stream configurations, if any entry is zero then the configuration will be set to default values.

Public Members

audio_stream_type_t **type**
Stream type

int **buf_sz**
Audio Element Buffer size

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

FATFS_STREAM_BUF_SIZE

FATFS_STREAM_TASK_STACK

FATFS_STREAM_TASK_CORE

FATFS_STREAM_TASK_PRIO

FATFS_STREAM_RINGBUFFER_SIZE

FATFS_STREAM_CFG_DEFAULT

2.2.4 Raw Stream

Header File

- audio_stream/include/raw_stream.h

Functions

audio_element_handle_t **raw_stream_init**(*raw_stream_cfg_t* **cfg*)
Initialize RAW stream.

Return The audio element handle

Parameters

- `cfg`: The RAW Stream configuration

int **raw_stream_read**(*audio_element_handle_t pipeline*, char **buffer*, int *buf_size*)
Read data from Stream.

Return Number of bytes actually read.

Parameters

- `pipeline`: The audio pipeline handle
- `buffer`: The buffer
- `buf_size`: Maximum number of bytes to be read.

int **raw_stream_write**(*audio_element_handle_t pipeline*, char **buffer*, int *buf_size*)
Write data to Stream.

Return Number of bytes written

Parameters

- `pipeline`: The audio pipeline handle

- `buffer`: The buffer
- `buf_size`: Number of bytes to write

Structures

struct raw_stream_cfg_t

Raw stream provides APIs to obtain the pipeline data without output stream or fill the pipeline data without input stream. The stream has two types / modes, reader and writer:

- AUDIO_STREAM_READER, e.g. [i2s]->[filter]->[raw],[i2s]->[codec-amr]->[raw]
- AUDIO_STREAM_WRITER, e.g. [raw]->[codec-mp3]->[i2s] Raw Stream configurations

Public Members

audio_stream_type_t **type**

Type of stream

int **out_rb_size**

Size of output ringbuffer

Macros

RAW_STREAM_RINGBUFFER_SIZE

RAW_STREAM_CFG_DEFAULT

2.2.5 Spiffs Stream

Header File

- audio_stream/include/spiffs_stream.h

Functions

audio_element_handle_t **spiffs_stream_init** (*spiffs_stream_cfg_t *config*)

Create a handle to an Audio Element to stream data from SPIFFS to another Element or get data from other elements written to SPIFFS, depending on the configuration the stream type, either AUDIO_STREAM_READER or AUDIO_STREAM_WRITER.

Return The Audio Element handle

Parameters

- `config`: The configuration

Structures

struct spiffs_stream_cfg_t

SPIFFS Stream configuration, if any entry is zero then the configuration will be set to default values.

Public Members

audio_stream_type_t **type**
Stream type

int **buf_sz**
Audio Element Buffer size

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

SPIFFS_STREAM_BUF_SIZE

SPIFFS_STREAM_TASK_STACK

SPIFFS_STREAM_TASK_CORE

SPIFFS_STREAM_TASK_PRIO

SPIFFS_STREAM_RINGBUFFER_SIZE

SPIFFS_STREAM_CFG_DEFAULT

2.3 Codecs

2.3.1 AAC Decoder

Decode an audio data stream provided in AAC format.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/aac_decoder.h

Functions

audio_element_handle_t **aac_decoder_init**(*aac_decoder_cfg_t* **config*)
Create an Audio Element handle to decode incoming AAC data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct aac_decoder_cfg_t
AAC Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

AAC_DECODER_TASK_STACK_SIZE

AAC_DECODER_TASK_CORE

AAC_DECODER_TASK_PRIO

AAC_DECODER_RINGBUFFER_SIZE

DEFAULT_AAC_DECODER_CONFIG

2.3.2 AMR Decoder and Encoder

Decode and encode an audio data stream from / to AMR format. Encoders cover both AMRNB and AMRWB formats.

Application Examples

Implementation of this API is demonstrated in the following examples:

- player/element_sdcard_amr
- recorder/pipeline_amr_sdcard

API Reference - Decoder

Header File

- esp-adf-libs/esp_codec/include/codec/amr_decoder.h

Functions

audio_element_handle_t **amr_decoder_init** (*amr_decoder_cfg_t *config*)
Create an Audio Element handle to decode incoming AMR data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct amr_decoder_cfg_t
AMR Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

AMR_DECODER_TASK_STACK_SIZE

AMR_DECODER_TASK_CORE

AMR_DECODER_TASK_PRIO

AMR_DECODER_RINGBUFFER_SIZE

DEFAULT_AMR_DECODER_CONFIG

API Reference - AMRNB Encoder

Header File

- esp-adf-libs/esp_codec/include/codec/amrnb_encoder.h

Functions

audio_element_handle_t **amrnb_encoder_init** (*amrnb_encoder_cfg_t *config*)
Create an Audio Element handle to encode incoming AMRNB data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct amrnb_encoder_cfg_t
AMRNB Encoder configurations.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

AMRNB_ENCODER_TASK_STACK

AMRNB_ENCODER_TASK_CORE

AMRNB_ENCODER_TASK_PRIO

AMRNB_ENCODER_RINGBUFFER_SIZE

DEFAULT_AMRNB_ENCODER_CONFIG

API Reference - AMRWB Encoder

Header File

- esp-adf-libs/esp_codec/include/codec/amrwb_encoder.h

Functions

audio_element_handle_t **amrwb_encoder_init**(*amrwb_encoder_cfg_t* **config*)
Create an Audio Element handle to encode incoming amrwb data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct amrwb_encoder_cfg_t
AMRWB Encoder configurations.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

AMRWB_ENCODER_TASK_STACK

AMRWB_ENCODER_TASK_CORE

AMRWB_ENCODER_TASK_PRIO

AMRWB_ENCODER_RINGBUFFER_SIZE

DEFAULT_AMRWB_ENCODER_CONFIG

2.3.3 FLAC Decoder

Decode an audio data stream provided in FLAC format.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/flac_decoder.h

Functions

audio_element_handle_t **flac_decoder_init** (*flac_decoder_cfg_t* **config*)
Create an Audio Element handle to decode incoming FLAC data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct flac_decoder_cfg_t
FLAC Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

FLAC_DECODER_TASK_STACK_SIZE

FLAC_DECODER_TASK_CORE

FLAC_DECODER_TASK_PRIO

FLAC_DECODER_RINGBUFFER_SIZE

DEFAULT_FLAC_DECODER_CONFIG

2.3.4 MP3 Decoder

Decode an audio data stream provided in MP3 format.

Application Examples

Implementation of this API is demonstrated in the following examples:

- get-started/play_mp3
- player/pipeline_sdcard_mp3

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/mp3_decoder.h

Functions

audio_element_handle_t **mp3_decoder_init** (*mp3_decoder_cfg_t *config*)
Create an Audio Element handle to decode incoming MP3 data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct mp3_decoder_cfg_t
Mp3 Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

MP3_DECODER_TASK_STACK_SIZE

MP3_DECODER_TASK_CORE

MP3_DECODER_TASK_PRIO

MP3_DECODER_RINGBUFFER_SIZE

DEFAULT_MP3_DECODER_CONFIG

2.3.5 OGG Decoder

Decode an audio data stream provided in OGG format.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/ogg_decoder.h

Functions

audio_element_handle_t **ogg_decoder_init** (*ogg_decoder_cfg_t *config*)
Create an Audio Element handle to decode incoming OGG data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct ogg_decoder_cfg_t
OGG Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

OGG_DECODER_TASK_STACK_SIZE

OGG_DECODER_TASK_CORE

OGG_DECODER_TASK_PRIO

OGG_DECODER_RINGBUFFER_SIZE

DEFAULT_OGG_DECODER_CONFIG

2.3.6 OPUS Decoder

Decode an audio data stream provided in OPUS format.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/opus_decoder.h

Functions

audio_element_handle_t **decoder_opus_init**(*opus_decoder_cfg_t* **config*)
Create an Audio Element handle to decode incoming OPUS data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct opus_decoder_cfg_t
OPUS Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
CPU core number (0 or 1) where decoder task in running

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

OPUS_DECODER_TASK_STACK_SIZE

OPUS_DECODER_TASK_CORE

OPUS_DECODER_TASK_PRIO

OPUS_DECODER_RINGBUFFER_SIZE

DEFAULT_OPUS_DECODER_CONFIG

2.3.7 WAV Decoder and Encoder

Decode and encode an audio data stream from / to WAV format.

Application Examples

Implementation of this API is demonstrated in the following examples:

- player/pipeline_sdcard_wav
- recorder/pipeline_wav_sdcard

API Reference - Decoder

Header File

- esp-adf-libs/esp_codec/include/codec/wav_decoder.h

Functions

audio_element_handle_t **wav_decoder_init** (*wav_decoder_cfg_t *config*)
Create an Audio Element handle to decode incoming WAV data.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct wav_decoder_cfg_t
brief WAV Decoder configurations

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

WAV_DECODER_TASK_STACK

WAV_DECODER_TASK_CORE

WAV_DECODER_TASK_PRIO

WAV_DECODER_RINGBUFFER_SIZE

DEFAULT_WAV_DECODER_CONFIG

API Reference - Encoder

Header File

- esp-adf-libs/esp_codec/include/codec/wav_encoder.h

Functions

audio_element_handle_t **wav_encoder_init** (*wav_encoder_cfg_t *config*)
Create a handle to an Audio Element to encode incoming data using WAV format.

Return The audio element handle

Parameters

- `config`: The configuration

Structures

struct wav_encoder_cfg_t
WAV Encoder configurations.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

int **task_core**
Task running in core (0 or 1)

int **task_prio**
Task priority (based on freeRTOS priority)

Macros

WAV_ENCODER_TASK_STACK

WAV_ENCODER_TASK_CORE

WAV_ENCODER_TASK_PRIO

WAV_ENCODER_RINGBUFFER_SIZE

DEFAULT_WAV_ENCODER_CONFIG

2.3.8 Resample Filter

The Resample Filter is an *Audio Element* designed to downsample or upsample the incoming data stream as well as to convert the data between stereo and mono.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codec/filter_resample.h

Functions

esp_err_t **rsp_filter_set_src_info**(*audio_element_handle_t self*, int *src_rate*, int *src_ch*)
Set the source audio sample rate and the number of channels to be processed by the resample.

Return ESP_OK ESP_FAIL

Parameters

- `self`: Audio element handle
- `src_rate`: The sample rate of stream data
- `src_ch`: The number channels of stream data

audio_element_handle_t **rsp_filter_init**(*rsp_filter_cfg_t* **config*)
Create an Audio Element handle to resample incoming data.

Depending on configuration, there are upsampling, downsampling, as well as converting data between mono and dual.

- If the audio_codec_type_t is `AUDIO_CODEC_TYPE_DECODER`, `src_rate` and `src_ch` will be fetched from `audio_element_getinfo`.
- If the audio_codec_type_t is `AUDIO_CODEC_TYPE_ENCODER`, `src_rate`, `src_ch`, `dest_rate` and `dest_ch` must be configured.

Return The audio element handler

Parameters

- `config`: The configuration

Structures

struct rsp_filter_cfg_t
Resample Filter Configuration.

Public Members

int **src_rate**
The sampling rate of the source PCM file (in Hz)

int **src_ch**
The number of channel(s) of the source PCM file (Mono=1, Dual=2)

int **dest_rate**
The sampling rate of the destination PCM file (in Hz)

int **dest_ch**
The number of channel(s) of the destination PCM file (Mono=1, Dual=2)

int **sample_bits**
The bit width of the PCM file. Currently, the only supported bit width is 16 bits.

int **mode**
The resampling mode (the encoding mode or the decoding mode). For decoding mode, input PCM length is constant; for encoding mode, output PCM length is constant.

int **max_indata_bytes**
The maximum buffer size of the input PCM (in bytes)

int **out_len_bytes**
The buffer length of the output stream data. This parameter must be configured in encoding mode.

int **type**
The resampling type (Automatic, Upsampling and Downsampling)

int **complexity**
Indicates the complexity of the resampling. This parameter is only valid when a FIR filter is used. Range: 0~4; O indicates the lowest complexity, which means the accuracy is the lowest and the speed is the fastest; Meanwhile, 4 indicates the highest complexity, which means the accuracy is the highest and the speed is the slowest.

int **down_ch_idx**
Indicates the channel that is selected (the right channel or the left channel). This parameter is only valid when the complexity parameter is set to 0 and the number of channel(s) of the input file has changed from dual to mono.

int **out_rb_size**
Output ringbuffer size

int **task_stack**
Task stack size

int **task_core**
Task running on core

int **task_prio**
Task priority

Macros

RSP_FILTER_BUFFER_BYTE

RSP_FILTER_TASK_STACK

RSP_FILTER_TASK_CORE

RSP_FILTER_TASK_PRIO

RSP_FILTER_RINGBUFFER_SIZE

DEFAULT_RESAMPLE_FILTER_CONFIG

2.4 Services

To interface an ESP32 based audio device with external physical or virtual devices, like a Bluetooth speaker or a cloud server, the ADF provides services. A service is a software implementation of specific protocol to facilitate communication between devices. Usually it also covers a set of functionalities to execute specific operations that involve either one or both devices, e.g. muting a Bluetooth speaker during playback or recognizing voice commands to adjust the color temperature of light in a room. The service may also provide polices to allow device operation by specific user or application.

For details please refer to descriptions listed below.

2.4.1 Bluetooth Service

This service is dedicated to interface with Bluetooth devices and provides:

- A2DP (Advanced Audio Distribution Profile), that implements streaming of multimedia audio using a Bluetooth connection;
- AVRCP (Audio/Video Remote Control Profile) used together with A2DP for remote control of devices such as headphones, car audio systems, or speakers.

Application Example

Implementation of this API is demonstrated in the following example:

- player/pipeline_bt_sink

Header File

- audio_service/include/bluetooth_service.h

Functions

esp_err_t **bluetooth_service_start** (*bluetooth_service_cfg_t* *config)

Initialize and start the Bluetooth service. This function can only be called for one time, and `bluetooth_service_destroy` must be called after use.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `config`: The configuration

audio_element_handle_t **bluetooth_service_create_stream** ()

Create Bluetooth stream, it is valid when Bluetooth service has started. The returned audio stream compatible with existing audio streams and can be used with the Audio Pipeline.

Return The Audio Element handle

esp_periph_handle_t **bluetooth_service_create_periph** ()

Create Bluetooth peripheral, it is valid when Bluetooth service has started. The returned bluetooth peripheral compatible with existing peripherals and can be used with the ESP Peripherals.

Return The Peripheral handle

esp_err_t **periph_bluetooth_play** (*esp_periph_handle_t periph*)

Send the AVRC passthrough command (PLAY) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_pause**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (PAUSE) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_stop**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (STOP) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_next**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (NEXT) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_prev**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (PREV) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_rewind**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (REWIND) to the Bluetooth device.

Return

- ESP_OK

- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_fast_forward**(*esp_periph_handle_t periph*)
Send the AVRC passthrough command (FAST FORWARD) to the Bluetooth device.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_discover**(*esp_periph_handle_t periph*)
Start device discovery.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_ERR_INVALID_ARG: if invalid parameters are provided
- ESP_FAIL: others

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_cancel_discover**(*esp_periph_handle_t periph*)
Cancel device discovery.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `periph`: The periph

esp_err_t **periph_bluetooth_connect**(*esp_periph_handle_t periph*, *bluetooth_addr_t remote_bda*)
Connect remote Device.

Return

- ESP_OK : Succeed
- ESP_ERR_INVALID_STATE: if bluetooth stack is not yet enabled
- ESP_FAIL: others

Parameters

- `periph`: The periph

- `remote_bda`: remote Bluetooth device address

esp_err_t **bluetooth_service_destroy**()
Destroy and cleanup bluetooth service, this function must be called after destroying the Bluetoth Stream and Bluetooth Peripheral created by `bluetooth_service_create_stream` and `bluetooth_service_create_periph`

Return

- ESP_OK
- ESP_FAIL

Structures

struct bluetooth_service_cfg_t
brief Bluetooth service configuration

Public Members

const char ***device_name**
Bluetooth local device name

const char ***remote_name**
Bluetooth remote device name

bluetooth_service_mode_t **mode**
Bluetooth working mode

Macros

BLUETOOTH_ADDR_LEN
brief Bluetooth address length

Type Definitions

typedef uint8_t **bluetooth_addr_t**[BLUETOOTH_ADDR_LEN]
brief Bluetooth device address

Enumerations

enum bluetooth_service_mode_t
brief Bluetooth service working mode

Values:

BLUETOOTH_A2DP_SINK
A2DP Bluetooth sink audio, ESP32 will receive audio data from other bluetooth devices

BLUETOOTH_A2DP_SOURCE
A2DP Bluetooth source audio, ESP32 can send audio data to other bluetooth devices

enum periph_bluetooth_event_id_t
brief Bluetooth peripheral event id

Values:

PERIPH_BLUETOOTH_UNKNOWN = 0
No event

PERIPH_BLUETOOTH_CONNECTED
A bluetooth device was connected

PERIPH_BLUETOOTH_DISCONNECTED
Last connection was disconnected

PERIPH_BLUETOOTH_AUDIO_STARTED
The audio session has been started

PERIPH_BLUETOOTH_AUDIO_SUSPENDED
The audio session has been suspended

PERIPH_BLUETOOTH_AUDIO_STOPPED
The audio session has been stopped

Header File

- audio_service/include/bt_keycontrol.h

2.5 Peripherals

There are several peripherals available in the ESP-ADF, ranging from buttons and LEDs to SD Card or Wi-Fi. The peripherals are implemented using common API that is then expanded with peripheral specific functionality. The following description covers common functionality.

2.5.1 ESP Peripherals

This library simplifies the management of peripherals, by pooling and monitoring in a single task, adding basic functions to send and receive events. And it also provides APIs to easily integrate new peripherals.

Note: Note that if you do not intend to integrate new peripherals into esp_peripherals, you are only interested in simple api `esp_periph_init`, `esp_periph_start`, `esp_periph_stop` and `esp_periph_destroy`. If you want to integrate new peripherals, please refer to *Periph Button* source code

---

Examples

```
#include "esp_log.h"
#include "esp_peripherals.h"
#include "periph_sdcard.h"
#include "periph_button.h"
#include "periph_touch.h"

static const char *TAG = "ESP_PERIPH_TEST";
```

```
static esp_err_t _periph_event_handle(audio_event_iface_msg_t *event, void *context)
{
    switch ((int)event->source_type) {
        case PERIPH_ID_BUTTON:
            ESP_LOGI(TAG, "BUTTON[%d], event->event_id=%d", (int)event->data, event->
↪cmd);
            break;
        case PERIPH_ID_SDCARD:
            ESP_LOGI(TAG, "SDCARD status, event->event_id=%d", event->cmd);
            break;
        case PERIPH_ID_TOUCH:
            ESP_LOGI(TAG, "TOUCH[%d], event->event_id=%d", (int)event->data, event->
↪cmd);
            break;
        case PERIPH_ID_WIFI:
            ESP_LOGI(TAG, "WIFI, event->event_id=%d", event->cmd);
            break;
    }
    return ESP_OK;
}

void app_main(void)
{
    // Initialize Peripherals pool
    esp_periph_config_t periph_cfg = {
        .event_handle = _periph_event_handle,
        .user_context = NULL,
    };
    esp_periph_init(&periph_cfg);

    // Setup SDCARD peripheral
    periph_sdcard_cfg_t sdcard_cfg = {
        .root = "/sdcard",
        .card_detect_pin = GPIO_NUM_34,
    };
    esp_periph_handle_t sdcard_handle = periph_sdcard_init(&sdcard_cfg);

    // Setup BUTTON peripheral
    periph_button_cfg_t btn_cfg = {
        .gpio_mask = GPIO_SEL_36 | GPIO_SEL_39
    };
    esp_periph_handle_t button_handle = periph_button_init(&btn_cfg);

    // Setup TOUCH peripheral
    periph_touch_cfg_t touch_cfg = {
        .touch_mask = TOUCH_PAD_SEL4 | TOUCH_PAD_SEL7 | TOUCH_PAD_SEL8 | TOUCH_PAD_
↪SEL9,
        .tap_threshold_percent = 70,
    };
    esp_periph_handle_t touch_handle = periph_touch_init(&touch_cfg);

    // Start all peripheral
    esp_periph_start(button_handle);
    esp_periph_start(sdcard_handle);
    esp_periph_start(touch_handle);
    vTaskDelay(10*1000/portTICK_RATE_MS);

    //Stop button peripheral
```

```
    esp_periph_stop(button_handle);
    vTaskDelay(10*1000/portTICK_RATE_MS);

    //Start button again
    esp_periph_start(button_handle);
    vTaskDelay(10*1000/portTICK_RATE_MS);

    //Stop & destroy all peripherals
    esp_periph_destroy();
}
```

API Reference

Header File

- esp_peripherals/include/esp_peripherals.h

Functions

esp_err_t **esp_periph_init**(*esp_periph_config_t* *config)

brief Initialize esp_peripherals, create empty peripherals list. Call this function only 1 time, before `esp_periph_start` any peripherals. This call will initialize the data needed for esp_peripherals to work, but does not actually create the task. The `event_handle` is optional if you want to receive events from this callback function. The esp_peripherals task will send all events out to event_iface, can be listen by event_iface by `esp_periph_get_event_iface`. The `user_context` will sent `esp_periph_event_handle_t` as *context parameter.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `config`: The configurations

esp_err_t **esp_periph_start**(*esp_periph_handle_t periph*)

Add the Peripheral to peripherals list, enable and start monitor task (if not)

Note This peripheral must be created by `esp_periph_create`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `periph`: The peripheral

esp_err_t **esp_periph_stop**(*esp_periph_handle_t periph*)

Stop monitor the peripheral, the peripheral state still kept. This funciton only temporary disables the peripheral.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

esp_err_t **esp_periph_stop_all**()
Stop monitor all peripherals, the peripheral state still kept. This funciton only temporary disables the peripheral.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_destroy**()
This function will stop and kill the monitor task, calling all destroy callback functions of peripheral (so you do not need to destroy the peripheral object manually). And also to remove all memory allocated to the peripherals list, you need to call the `esp_periph_init` function again if you want to use it.

Return

- ESP_OK
- ESP_FAIL

esp_periph_state_t **esp_periph_get_state**(*esp_periph_handle_t periph*)
Get the current state of peripheral.

Return The peripharal working state

Parameters

- `periph`: The handle of peripheral

esp_periph_handle_t **esp_periph_get_by_id**(int *periph_id*)
Get the peripheral handle by Peripheral ID.

Return The esp_periph_handle_t

Parameters

- `periph_id`: as esp_periph_id_t, or any ID you use when call `esp_periph_create`

esp_err_t **esp_periph_send_event**(*esp_periph_handle_t periph*, int *event_id*, void **data*, int *data_len*)
In addition to sending an event via event_iface, this function will dispatch the `event_handle` callback if the event_handle callback is provided at `esp_periph_init`

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `event_id`: The event identifier

- `data`: The data
- `data_len`: The data length

esp_err_t **esp_periph_send_cmd**(*esp_periph_handle_t periph*, int *cmd*, void **data*, int *data_len*)

When this function is called, the command is passed to the event_iface command queue, and the `esp_periph_run_func` of this peripheral will be executed in the main peripheral task. This function can be called from any task, basically it only sends a queue to the main peripheral task.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `cmd`: The command
- `data`: The data
- `data_len`: The data length

esp_err_t **esp_periph_send_cmd_from_isr**(*esp_periph_handle_t periph*, int *cmd*, void **data*, int *data_len*)

Similar to `esp_periph_send_cmd`, but it can be called in the hardware interrupt handle.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `cmd`: The command
- `data`: The data
- `data_len`: The data length

esp_err_t **esp_periph_set_data**(*esp_periph_handle_t periph*, void **data*)

Set the user data.

Note Make sure the `data` lifetime is sufficient, this function does not copy all data, it only stores the data pointer

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `data`: The data

void ****esp_periph_get_data**(*esp_periph_handle_t periph*)

Get the userdata stored in the peripheral.

Return Peripheral data pointer

Parameters

- `periph`: The periph

esp_err_t **esp_periph_start_timer**(*esp_periph_handle_t periph*, TickType_t *interval_tick*, *timer_callback callback*)

Each peripheral can initialize a timer, which is by default NULL. When this function is called, the timer for the peripheral is created and it invokes the callback function every interval tick.

Note - You do not need to stop or destroy the timer, when the `esp_periph_destroy` function is called, it will stop and destroy all

- This timer using FreeRTOS Timer, with autoreload = true

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `interval_tick`: The interval tick
- `callback`: The callback

esp_err_t **esp_periph_stop_timer**(*esp_periph_handle_t periph*)

Stop peripheral timer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph

QueueHandle_t **esp_periph_get_queue**()

Peripheral using event_iface to control the event, all events send out to event_iface queue. But in case we want to read events directly from the event_iface queue, this function will be useful.

Return The queue handle

esp_periph_id_t **esp_periph_get_id**(*esp_periph_handle_t periph*)

Get Peripheral Identify.

Return The peripheral identify

Parameters

- `periph`: The periph

esp_err_t **esp_periph_set_id**(*esp_periph_handle_t periph*, *esp_periph_id_t periph_id*)

Set Peripheral identify.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `periph_id`: The periph identifier

audio_event_iface_handle_t **esp_periph_get_event_iface**()
Return the event_iface used by this esp_peripherals.

Return The audio event iface handle

esp_periph_handle_t **esp_periph_create**(int *periph_id*, **const** char **tag*)
Call this function to initialize a new peripheral.

Return The peripheral handle

Parameters

- `periph_id`: The periph identifier
- `tag`: The tag name, we named it easy to get in debug logs

esp_err_t **esp_periph_set_function**(*esp_periph_handle_t periph*, *esp_periph_func init*, *esp_periph_run_func run*, *esp_periph_func destroy*)
Each peripheral has a cycle of sequential operation from initialization, execution of commands to destroy, functions assigned by this function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `init`: The initialize
- `run`: The run
- `destroy`: The destroy

long long **esp_periph_tick_get**()
Get tick in milliseconds (from ESP32 boot)

Return Tick in milliseconds

esp_err_t **esp_periph_set_callback**(*esp_periph_event_handle_t cb*)
Set event callback function.

Return

- ESP_OK
- ESP_FAIL

Structures

struct esp_periph_config_t
Common peripherals configurations.

Public Members

void ***user_context**
peripherals context data

esp_periph_event_handle_t **event_handle**
The Peripheral events handle

Macros

periph_tick_get

Type Definitions

typedef struct esp_periph ***esp_periph_handle_t**

typedef esp_err_t (***esp_periph_func**) (*esp_periph_handle_t* periph)

typedef esp_err_t (***esp_periph_run_func**) (*esp_periph_handle_t* periph, *audio_event_iface_msg_t* *msg)

typedef esp_err_t (***esp_periph_event_handle_t**) (*audio_event_iface_msg_t* *event, void *context)

typedef void (***timer_callback**) (xTimerHandle tmr)

Enumerations

enum esp_periph_id_t
Peripheral Identify, this must be unique for each peripheral added to the peripherals list.

Values:

PERIPH_ID_BUTTON = AUDIO_ELEMENT_TYPE_PERIPH + 1

PERIPH_ID_TOUCH = AUDIO_ELEMENT_TYPE_PERIPH + 2

PERIPH_ID_SDCARD = AUDIO_ELEMENT_TYPE_PERIPH + 3

PERIPH_ID_WIFI = AUDIO_ELEMENT_TYPE_PERIPH + 4

PERIPH_ID_FLASH = AUDIO_ELEMENT_TYPE_PERIPH + 5

PERIPH_ID_AUXIN = AUDIO_ELEMENT_TYPE_PERIPH + 6

PERIPH_ID_ADC = AUDIO_ELEMENT_TYPE_PERIPH + 7

PERIPH_ID_CONSOLE = AUDIO_ELEMENT_TYPE_PERIPH + 8

PERIPH_ID_BLUETOOTH = AUDIO_ELEMENT_TYPE_PERIPH + 9

PERIPH_ID_LED = AUDIO_ELEMENT_TYPE_PERIPH + 10

PERIPH_ID_SPIFFS = AUDIO_ELEMENT_TYPE_PERIPH + 11

PERIPH_ID_ADC_BTN = AUDIO_ELEMENT_TYPE_PERIPH + 12

PERIPH_ID_IS31FL3216 = AUDIO_ELEMENT_TYPE_PERIPH + 13

enum esp_periph_state_t

Peripheral working state.

Values:

PERIPH_STATE_NULL

PERIPH_STATE_INIT

PERIPH_STATE_RUNNING

PERIPH_STATE_PAUSE

PERIPH_STATE_STOPPING

PERIPH_STATE_ERROR

PERIPH_STATE_STATUS_MAX

The peripheral specific functionality is available by calling dedicated functions described below. Some peripherals are available on both *ESP32-LyraT* and *ESP32-LyraTD-MSC* development boards, some on a specific board only. The following table provides all implemented peripherals broken down by development board.

2.5.2 Wi-Fi Peripheral

The Wi-Fi Peripheral is used to configure Wi-Fi connections, provide APIs to control Wi-Fi connection configuration, as well as monitor the status of Wi-Fi networks.

Application Example

Implementation of this API is demonstrated in player/pipeline_http_mp3 example.

API Reference

Header File

- esp_peripherals/include/periph_wifi.h

Functions

esp_periph_handle_t **periph_wifi_init**(*periph_wifi_cfg_t* **config*)

Create the wifi peripheral handle for esp_peripherals.

Note The handle was created by this function automatically destroy when `esp_periph_destroy` is called

Return The esp peripheral handle

Parameters

- `config`: The configuration

esp_err_t **periph_wifi_wait_for_connected**(*esp_periph_handle_t periph*, TickType_t *tick_to_wait*)

This function will block current thread (in `tick_to_wait` tick) and wait until ESP32 connected to the Wi-Fi network, and got ip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `tick_to_wait`: The tick to wait

periph_wifi_state_t **periph_wifi_is_connected**(*esp_periph_handle_t periph*)

Check the Wi-Fi connection status.

Return Wi-Fi network status

Parameters

- `periph`: The periph

esp_err_t **periph_wifi_config_start**(*esp_periph_handle_t periph*, *periph_wifi_config_mode_t mode*)

Start Wi-Fi network setup in `mode`

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `mode`: The mode

esp_err_t **periph_wifi_config_wait_done**(*esp_periph_handle_t periph*, TickType_t *tick_to_wait*)

Wait for Wi-Fi setup done.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `tick_to_wait`: The tick to wait

Structures

struct periph_wifi_cfg_t

The Wi-Fi peripheral configuration.

Public Members

bool **disable_auto_reconnect**
Disable Wi-Fi auto reconnect

int **reconnect_timeout_ms**
The reconnect timeout after disconnected from Wi-Fi network

const char ***ssid**
SSID of target AP

const char ***password**
password of target AP

Enumerations

enum periph_wifi_state_t
Peripheral Wi-Fi event id.

Values:

PERIPH_WIFI_UNCHANGE = 0

PERIPH_WIFI_CONNECTING

PERIPH_WIFI_CONNECTED

PERIPH_WIFI_DISCONNECTED

PERIPH_WIFI_SETTING

PERIPH_WIFI_CONFIG_DONE

PERIPH_WIFI_CONFIG_ERROR

PERIPH_WIFI_ERROR

enum periph_wifi_config_mode_t
Wi-Fi setup mode type.

Values:

WIFI_CONFIG_ESPTOUCH
Using smartconfig with ESPTOUCH protocol

WIFI_CONFIG_AIRKISS
Using smartconfig with AIRKISS protocol

WIFI_CONFIG_ESPTOUCH_AIRKISS
Using smartconfig with ESPTOUCH_AIRKISS protocol

WIFI_CONFIG_WPS
Using WPS (not support)

WIFI_CONFIG_BLUEFI
Using BLUEFI

WIFI_CONFIG_WEB
Using HTTP Server (not support)

2.5.3 SD Card Peripheral

If your board has a SD card connected, use this API to initialize, mount and unmount the card, see functions *periph_sdcard_init()*, periph_sdcard_mount() and periph_sdcard_unmount(). The data reading / writing is implemented in a separate API described in *FatFs Stream*.

Application Examples

Implementation of this API is demonstrated in couple of examples:

- player/pipeline_sdcard_mp3
- player/pipeline_sdcard_wav
- recorder/pipeline_wav_sdcard

API Reference

Header File

- esp_peripherals/include/periph_sdcard.h

Functions

esp_periph_handle_t **periph_sdcard_init**(*periph_sdcard_cfg_t *sdcard_config*)
Create the sdcard peripheral handle for esp_peripherals.

Note The handle was created by this function automatically destroy when esp_periph_destroy is called

Return The esp peripheral handle

Parameters

- sdcard_config: The sdcard configuration

bool **periph_sdcard_is_mounted**(*esp_periph_handle_t periph*)
Check the sdcard is mounted or not.

Return SDCARD mounted state

Parameters

- periph: The periph

Structures

struct periph_sdcard_cfg_t
The SD Card Peripheral configuration.

Public Members

int **card_detect_pin**
Card detect gpio number

const char ***root**
Base path for vfs

Enumerations

enum periph_sdcard_event_id_t
Peripheral sdcard event id.

Values:

SDCARD_STATUS_UNKNOWN
No event

SDCARD_STATUS_CARD_DETECT_CHANGE
Detect changes in the card_detect pin

SDCARD_STATUS_MOUNTED
SDCARD mounted successfully

SDCARD_STATUS_UNMOUNTED
SDCARD unmounted successfully

SDCARD_STATUS_MOUNT_ERROR
SDCARD mount error

SDCARD_STATUS_UNMOUNT_ERROR
SDCARD unmount error

2.5.4 Spiffs Peripheral

Use this API to initialize, mount and unmount spiffs partition, see functions *periph_spiffs_init()*, `periph_spiffs_mount()` and `periph_spiffs_unmount()`. The data reading / writing is implemented in a separate API described in *Spiffs Stream*.

Application Example

Implementation of this API is demonstrated in filter/pipeline_spiffs_amr_resample example.

API Reference

Header File

- esp_peripherals/include/periph_spiffs.h

Functions

esp_periph_handle_t **periph_spiffs_init**(*periph_spiffs_cfg_t *spiffs_config*)
Create the spiffs peripheral handle for esp_peripherals.

Note The handle created by this function will be automatically destroyed when `esp_periph_destroy` is called

Return The esp peripheral handle

Parameters

- `spiffs_config`: The spiffs configuration

bool **periph_spiffs_is_mounted**(*esp_periph_handle_t periph*)
Check if the SPIFFS is mounted or not.

Return SPIFFS mounted state

Parameters

- `periph`: The periph

Structures

struct periph_spiffs_cfg_t
The SPIFFS Peripheral configuration.

Public Members

const char ***root**
Base path for vfs

const char ***partition_label**
Optional, label of SPIFFS partition to use. If set to NULL, first partition with subtype=spiffs will be used.

size_t **max_files**
Maximum number of files that could be open at the same time.

bool **format_if_mount_failed**
If true, it will format the file system if it fails to mount.

Enumerations

enum periph_spiffs_event_id_t
Peripheral spiffs event id.

Values:

SPIFFS_STATUS_UNKNOWN
No event

SPIFFS_STATUS_MOUNTED
SPIFFS mounted successfully

SPIFFS_STATUS_UNMOUNTED
SPIFFS unmounted successfully

SPIFFS_STATUS_MOUNT_ERROR
SPIFFS mount error

SPIFFS_STATUS_UNMOUNT_ERROR
SPIFFS unmount error

2.5.5 Console Peripheral

Console Peripheral is used to control the Audio application from the terminal screen. It provides 2 ways do execute command, one sends an event to esp_peripherals (for a command without parameters), another calls a callback function (need parameters). If there is a callback function, no event will be sent.

Please make sure that the lifetime of *periph_console_cmd_t* must be ensured during console operation, *periph_console_init()* only reference, does not make a copy.

Code example

```
#include "freertos/FreeRTOS.h"
#include "esp_log.h"
#include "esp_peripherals.h"
#include "periph_console.h"

static const char *TAG = "ESP_PERIPH_TEST";

static esp_err_t _periph_event_handle(audio_event_iface_msg_t *event, void *context)
{
    switch ((int)event->source_type) {
        case PERIPH_ID_CONSOLE:
            ESP_LOGI(TAG, "CONSOLE, command id=%d", event->cmd);
            break;
    }
    return ESP_OK;
}

esp_err_t console_test_cb(esp_periph_handle_t periph, int argc, char *argv[])
{
    int i;
    ESP_LOGI(TAG, "CONSOLE Callback, argc=%d", argc);
    for (i=0; i<argc; i++) {
        ESP_LOGI(TAG, "CONSOLE Args[%d] %s", i, argv[i]);
    }
    return ESP_OK;
}

void app_main(void)
{
    // Initialize Peripherals pool
    esp_periph_config_t periph_cfg = {
        .event_handle = _periph_event_handle,
        .user_context = NULL,
    };
    esp_periph_init(&periph_cfg);

    const periph_console_cmd_t cmd[]= {
        { .cmd = "play", .id = 1, .help = "Play audio" },
        { .cmd = "stop", .id = 2, .help = "Stop audio" },
```

```
        { .cmd = "test", .help = "test console", .func = console_test_cb },
    };

    periph_console_cfg_t console_cfg = {
        .command_num = sizeof(cmd)/sizeof(periph_console_cmd_t),
        .commands = cmd,
    };
    esp_periph_handle_t console_handle = periph_console_init(&console_cfg);
    esp_periph_start(console_handle);
    vTaskDelay(30000/portTICK_RATE_MS);
    ESP_LOGI(TAG, "Stopped");
    esp_periph_destroy();

}
```

API Reference

Header File

- esp_peripherals/include/periph_console.h

Functions

esp_periph_handle_t **periph_console_init** (*periph_console_cfg_t *config*)
Initialize Console Peripheral.

Return The esp peripheral handle

Parameters

- `config`: The configuration

Structures

struct periph_console_cmd_t
Command structure.

Public Members

const char ***cmd**
Name of command, must be unique

int **id**
Command ID will be sent together when the command is matched

const char ***help**
Explanation of the command

console_cmd_callback_t **func**
Function callback for the command

struct periph_console_cfg_t
Console Peripheral configuration.

Public Members

int **command_num**
Total number of commands

const *periph_console_cmd_t* ***commands**
Pointer to array of commands

int **task_stack**
Console task stack, using default if the value is zero

int **task_prio**
Console task priority (based on freeRTOS priority), using default if the value is zero

const char ***prompt_string**
Console prompt string, using default CONSOLE_PROMPT_STRING if the pointer is NULL

Macros

CONSOLE_DEFAULT_TASK_PRIO

CONSOLE_DEFAULT_TASK_STACK

CONSOLE_DEFAULT_PROMPT_STRING

Type Definitions

typedef esp_err_t (***console_cmd_callback_t**) (*esp_periph_handle_t* periph, int argc, char *argv[])

2.5.6 Touch Peripheral

Initialize ESP32 touchpad peripheral and retrieve information from the touch sensors.

Application Example

Implementation of this API is demonstrated in get-started/play_mp3_control example.

API Reference

Header File

- esp_peripherals/include/periph_touch.h

Functions

esp_periph_handle_t **periph_touch_init** (*periph_touch_cfg_t* *config)
Create the touch peripheral handle for esp_peripherals.

Note The handle was created by this function automatically destroy when `esp_periph_destroy` is called

Return The esp peripheral handle

Parameters

- `config`: The configuration

Structures

struct periph_touch_cfg_t

The Touch peripheral configuration.

Public Members

int **touch_mask**

Touch pad mask using for this Touch peripheral, ex: TOUCH_PAD_SEL0 | TOUCH_PAD_SEL1

int **tap_threshold_percent**

Tap threshold percent, Tap event will be determined if the percentage value is less than the non-touch value

int **long_tap_time_ms**

Long tap duration in milliseconds, default is 2000ms, PERIPH_TOUCH_LONG_TAP will be occurred if TAP and time hold longer than this value

Enumerations

enum esp_touch_pad_sel_t

Touch pad selection.

Values:

TOUCH_PAD_SEL0 = BIT(0)

TOUCH_PAD_SEL1 = BIT(1)

TOUCH_PAD_SEL2 = BIT(2)

TOUCH_PAD_SEL3 = BIT(3)

TOUCH_PAD_SEL4 = BIT(4)

TOUCH_PAD_SEL5 = BIT(5)

TOUCH_PAD_SEL6 = BIT(6)

TOUCH_PAD_SEL7 = BIT(7)

TOUCH_PAD_SEL8 = BIT(8)

TOUCH_PAD_SEL9 = BIT(9)

enum periph_touch_event_id_t

Peripheral touch event id.

Values:

PERIPH_TOUCH_UNCHANGE = 0

No event

PERIPH_TOUCH_TAP

When touch pad is tapped

PERIPH_TOUCH_RELEASE
When touch pad is released after tap

PERIPH_TOUCH_LONG_TAP
When touch pad is tapped and held after `long_tap_time_ms` time

PERIPH_TOUCH_LONG_RELEASE
When touch pad is released after long tap

2.5.7 Button Peripheral

To control application flow you may use buttons connected and read through the ESP32 GPIOs. This API provides functions to initialize sepecifc GPIOs and obtain information on button events such as when it has been pressed, when released, when pressed for a long time and released after long press. To get information on particular event, establish a callback function with `button_dev_add_tap_cb()` or `button_dev_add_press_cb()`.

Application Example

Implementation of this API is demonstrated in recorder/pipeline_raw_http example.

API Reference

Header File

- esp_peripherals/include/periph_button.h

Functions

esp_periph_handle_t **periph_button_init** (*periph_button_cfg_t *but_cfg*)
Create the button peripheral handle for esp_peripherals.

Note The handle was created by this function automatically destroy when `esp_periph_destroy` is called

Return The esp peripheral handle

Parameters

- `but_cfg`: The but configuration

Structures

struct periph_button_cfg_t
The Button peripheral configuration.

Public Members

uint64_t **gpio_mask**
GPIO Mask using for this Button peripheral, it is BIT(GPIO_NUM), ex: GPIO_SEL_36 | GPIO_SEL_36

int **long_press_time_ms**
Long press duration in milliseconds, default is 2000ms

Enumerations

enum periph_button_event_id_t
Peripheral button event id.

Values:

PERIPH_BUTTON_UNCHANGE = 0
No event

PERIPH_BUTTON_PRESSED
When button is pressed

PERIPH_BUTTON_RELEASE
When button is released

PERIPH_BUTTON_LONG_PRESSED
When button is pressed and kept for more than `long_press_time_ms`

PERIPH_BUTTON_LONG_RELEASE
When button is released and event PERIPH_BUTTON_LONG_PRESSED happened

2.5.8 LED Peripheral

Blink or fade a LED connected to a GPIO with configurable On and Off times.

Application Examples

Implementation of this API is demonstrated in couple of examples:

- /cloud_services/google_translate_device
- /dueros

API Reference

Header File

- esp_peripherals/include/periph_led.h

Functions

esp_periph_handle_t **periph_led_init** (*periph_led_cfg_t *config*)
Create the LED peripheral handle for esp_peripherals.

Note The handle was created by this function automatically destroy when `esp_periph_destroy` is called

Return The esp peripheral handle

Parameters

- `config`: The configuration

esp_err_t **periph_led_blink**(*esp_periph_handle_t periph*, int *gpio_num*, int *time_on_ms*, int *time_off_ms*, bool *fade*, int *loop*)

Bink LED Peripheral, this function will automatically configure the gpio_num to control the LED, with `time_on_ms` as the time (in milliseconds) switch from OFF to ON (or ON if fade is disabled), and `time_off_ms` as the time (in milliseconds) switch from ON to OFF (or OFF if fade is disabled). When switching from ON -> OFF and vice versa, the loop decreases once, and will turn off the effect when the loop is 0. With a loop value less than 0, the LED effect will loop endlessly. PERIPH_LED_BLINK_FINISH events will be sent at each end of loop.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The LED periph
- `gpio_num`: The gpio number
- `time_on_ms`: The time on milliseconds
- `time_off_ms`: The time off milliseconds
- `fade`: Fading enabled
- `loop`: Loop

esp_err_t **periph_led_stop**(*esp_periph_handle_t periph*, int *gpio_num*)

Stop Blink the LED.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `periph`: The periph
- `gpio_num`: The gpio number

Structures

struct periph_led_cfg_t

The LED peripheral configuration.

Public Members

ledc_mode_t **led_speed_mode**

LEDC speed speed_mode, high-speed mode or low-speed mode

ledc_timer_bit_t **led_duty_resolution**

LEDC channel duty resolution

ledc_timer_t **led_timer_num**

Select the timer source of channel (0 - 3)

uint32_t **led_freq_hz**
LEDC timer frequency (Hz)

Enumerations

enum periph_led_event_id_t
Peripheral LED event id.

Values:

PERIPH_LED_UNCHANGE = 0
No event

PERIPH_LED_BLINK_FINISH
When LED blink is finished

2.5.9 ADC Button Peripheral

Read status of buttons connected to an ADC input using a resistor ladder. Configuration provides for more than a singe ADC input to read several sets of buttons. For an example hardware implementation please refer to schematic of ESP32-LyraTD-MSC V2.2 Upper Board (PDF).

Application Examples

Implementation of this API is demonstrated in the following example:

- checks/check_msc_adc_button

API Reference

Header File

- esp_peripherals/include/periph_adc_button.h

Functions

esp_periph_handle_t **periph_adc_button_init** (*periph_adc_button_cfg_t *btn_cfg*)
Create the button peripheral handle for esp_peripherals.

Note The handle created by this function is automatically destroyed when esp_periph_destroy is called.

Return The esp peripheral handle.

Parameters

- `btn_cfg`: The button configuration.

Structures

struct periph_adc_button_cfg_t
The configuration of ADC Button.

Public Members

adc_arr_t ***arr**
An array with configuration of buttons

int **arr_size**
The array size

Macros

ADC_DEFAULT_ARR

Enumerations

enum periph_adc_button_event_id_t
Values:

PERIPH_ADC_BUTTON_IDLE

PERIPH_ADC_BUTTON_CLICK

PERIPH_ADC_BUTTON_PRESS

PERIPH_ADC_BUTTON_RELEASE

2.5.10 LED Controller Peripheral

This peripheral is applicable to IS31Fl3216 chip that is a light LED controller with an audio modulation mode. It can store data of 8 Frames with internal RAM to play small animations automatically. You can also use it to control a number of LEDs connected to GPIOs. If you want to use the IS31Fl3216, see functions *periph_is31fl3216_init()*, *periph_is31fl3216_set_blink_pattern()*, *periph_is31fl3216_set_duty()*, *periph_is31fl3216_set_state()*.

Application Examples

Implementation of this API is demonstrated in checks/check_msc_leds example.

API Reference

Header File

- esp_peripherals/include/periph_is31fl3216.h

Functions

esp_periph_handle_t **periph_is31fl3216_init** (*periph_is31fl3216_cfg_t *is31fl3216_config*)
Initializate the is31fl3216.

Return

- ESP_OK Success

- ESP_FAIL Fail

Parameters

- `is31fl3216_config`:

esp_err_t **periph_is31fl3216_set_blink_pattern**(*esp_periph_handle_t periph*, uint16_t *blink_pattern*)

Set the current enable channels.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `periph`: The is31fl3216 handle
- `blink_pattern`: The bit pattern of enabled channels

esp_err_t **periph_is31fl3216_set_duty**(*esp_periph_handle_t periph*, uint8_t *index*, uint8_t *value*)

Set the duty of the channel.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `periph`: The is31fl3216 handle
- `index`: The channel number
- `value`: The value of the channel's duty to be set

esp_err_t **periph_is31fl3216_set_state**(*esp_periph_handle_t periph*, *periph_is31fl3216_state_t state*)

Set the state of all the channels.

Return

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- `periph`: The is31fl3216 handle
- `state`: The state of all channels

Structures

struct periph_is31fl3216_cfg_t

The configuration of is31fl3216.

Public Members

uint32_t **duty**[IS31FL3216_CH_NUM]
An array of the is31fl3216's duty

uint16_t **is31fl3216_pattern**
Current enable channel

periph_is31fl3216_state_t **state**
The state of all the channels

Macros

IS31FL3216_CH_NUM

Enumerations

enum periph_is31fl3216_state_t
Values:

IS31FL3216_STATE_OFF

IS31FL3216_STATE_ON

IS31FL3216_STATE_FLASH

IS31FL3216_STATE_BY_AUDIO

Name of Peripheral	ESP32-LyraT	ESP32-LyraTD-MSC
Wi-Fi	✓	✓
SD Card	✓	✓
Spiffs	✓	✓
Console	✓	✓
Touch	✓	
Button	✓	
LED	✓	
ADC Button		✓
LED Controller		✓

2.6 Abstraction Layer

2.6.1 Ring Buffer

Ringbuffer is designed in addition to use as a data buffer, also used to connect Audio Elements. Each Element that requests data from the Ringbuffer will block the task until the data is available. Or block the task when writing data and the Buffer is full. Of course, we can stop this block at any time.

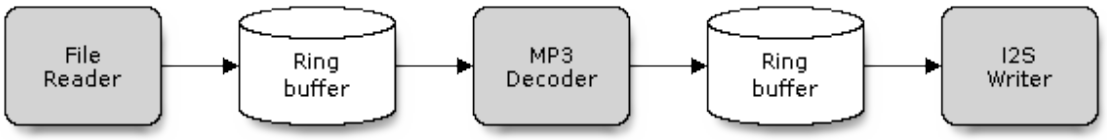


Fig. 2.4: Ring Buffer used in Audio Pipeline

Application Example

In most of ESP-ADF examples connecting of Elements with Ringbuffers is done "behind the scenes" by a function `audio_pipeline_link()`. To see this operation exposed check player/element_sdcard_mp3 example.

API Reference

Header File

- audio_pipeline/include/ringbuf.h

Functions

ringbuf_handle_t **rb_create** (int *size*, int *block_size*)
Create ringbuffer with total size = size * block_size.

Return ringbuf_handle_t

Parameters

- `size`: The size
- `block_size`: The block size

esp_err_t **rb_destroy** (*ringbuf_handle_t rb*)
Cleanup and free all memory created by ringbuf_handle_t.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

esp_err_t **rb_abort** (*ringbuf_handle_t rb*)
Abort waiting until there is space for reading or writing of the ringbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

esp_err_t **rb_reset**(*ringbuf_handle_t rb*)
Reset ringbuffer, clear all values as initial state.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

int **rb_bytes_available**(*ringbuf_handle_t rb*)
Get total bytes available of Ringbuffer.

Return total bytes available

Parameters

- `rb`: The Ringbuffer handle

int **rb_bytes_filled**(*ringbuf_handle_t rb*)
Get the number of bytes that have filled the ringbuffer.

Return The number of bytes that have filled the ringbuffer

Parameters

- `rb`: The Ringbuffer handle

int **rb_get_size**(*ringbuf_handle_t rb*)
Get total size of Ringbuffer (in bytes)

Return total size of Ringbuffer

Parameters

- `rb`: The Ringbuffer handle

int **rb_read**(*ringbuf_handle_t rb*, char **buf*, int *len*, TickType_t *ticks_to_wait*)
Read from Ringbuffer to `buf` with len and wait `tick_to_wait` ticks until enough bytes to read if the ringbuffer bytes available is less than `len`

Return Number of bytes read

Parameters

- `rb`: The Ringbuffer handle
- `buf`: The buffer pointer to read out data
- `len`: The length request
- `ticks_to_wait`: The ticks to wait

int **rb_write**(*ringbuf_handle_t rb*, char **buf*, int *len*, TickType_t *ticks_to_wait*)

Write to Ringbuffer from `buf` with `len` and wait `tick_to_wait` ticks until enough space to write if the ringbuffer space available is less than `len`

Return Number of bytes written

Parameters

- `rb`: The Ringbuffer handle
- `buf`: The buffer
- `len`: The length
- `ticks_to_wait`: The ticks to wait

int **rb_size_get**(*ringbuf_handle_t rb*)

Get total size of ringbuffer.

Return Total size of ringbuffer (in block byte(s))

Parameters

- `rb`: The Ringbuffer handle

esp_err_t **rb_done_write**(*ringbuf_handle_t rb*)

Set status of writing to ringbuffer is done.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

Macros

RB_OK

RB_FAIL

RB_DONE

RB_ABORT

RB_TIMEOUT

Type Definitions

typedef struct ringbuf ***ringbuf_handle_t**

2.6.2 Audio HAL

Abstraction layer for audio board hardware, serves as an interface between the user application and the hardware driver for specific audio board like *ESP32 LyraT*.

The API provides data structures to configure sampling rates of ADC and DAC signal conversion, data bit widths, I2C stream parameters, and selection of signal channels connected to ADC and DAC. It also contains several specific functions to e.g. initialize the audio board, `audio_hal_init()`, control the volume, `audio_hal_get_volume()` and `audio_hal_set_volume()`.

API Reference

Header File

- audio_hal/include/audio_hal.h

Functions

audio_hal_handle_t **audio_hal_init** (*audio_hal_codec_config_t* **audio_hal_conf*, int *index*)
Initialize media codec driver.

Note If selected codec has already been installed, it'll return the audio_hal handle.

Return int, 0success, othersfail

Parameters

- `audio_hal_conf`: Configure structure audio_hal_config_t
- `index`: Indicates which codec will be initialized

esp_err_t **audio_hal_deinit** (*audio_hal_handle_t* *audio_hal*, int *index*)
Uninitialize media codec driver.

Return int, 0success, othersfail

Parameters

- `audio_hal`: reference function pointer for selected audio codec
- `index`: Indicates which codec will be deinitialized

esp_err_t **audio_hal_ctrl_codec** (*audio_hal_handle_t* *audio_hal*, *audio_hal_codec_mode_t* *mode*, *audio_hal_ctrl_t* *audio_hal_ctrl*)
Start/stop codec driver.

Return int, 0success, othersfail

Parameters

- `audio_hal`: reference function pointer for selected audio codec
- `mode`: select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t
- `audio_hal_ctrl`: select start stop state for specific mode

esp_err_t **audio_hal_codec_iface_config**(*audio_hal_handle_t audio_hal*, *audio_hal_codec_mode_t mode*, *audio_hal_codec_i2s_iface_t* *iface)

Set codec I2S interface samples rate & bit width and format either I2S or PCM/DSP.

Return

- 0 Success
- -1 Error

Parameters

- `audio_hal`: reference function pointer for selected audio codec
- `mode`: select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t
- `iface`: I2S sample rate (ex: 16000, 44100), I2S bit width (16, 24, 32),I2s format (I2S, PCM, DSP).

esp_err_t **audio_hal_set_volume**(*audio_hal_handle_t audio_hal*, int *volume*)

Set voice volume.

Note if volume is 0, mute is enabled,range is 0-100.

Return int, 0success, othersfail

Parameters

- `audio_hal`: reference function pointer for selected audio codec
- `volume`: value of volume in percent(%)

esp_err_t **audio_hal_get_volume**(*audio_hal_handle_t audio_hal*, int **volume*)

get voice volume.

Note if volume is 0, mute is enabled, range is 0-100.

Return int, 0success, othersfail

Parameters

- `audio_hal`: reference function pointer for selected audio codec
- `volume`: value of volume in percent returned(%)

Structures

struct audio_hal_codec_i2s_iface_t

I2s interface configuration for audio codec chip.

Public Members

audio_hal_iface_mode_t **mode**

audio codec chip mode

audio_hal_iface_format_t **fmt**

I2S interface format

audio_hal_iface_samples_t **samples**
I2S interface samples per second

audio_hal_iface_bits_t **bits**
i2s interface number of bits per sample

struct audio_hal_codec_config_t
Configure media hal for initialization of audio codec chip.

Public Members

audio_hal_adc_input_t **adc_input**
set adc channel

audio_hal_dac_output_t **dac_output**
set dac channel

audio_hal_codec_mode_t **codec_mode**
select codec mode: adc, dac or both

audio_hal_codec_i2s_iface_t **i2s_iface**
set I2S interface configuration

Macros

AUDIO_HAL_VOL_DEFAULT

AUDIO_HAL_ES8388_DEFAULT

AUDIO_HAL_ES8374_DEFAULT

Type Definitions

typedef struct audio_hal ***audio_hal_handle_t**

Enumerations

enum audio_hal_codec_mode_t
Select media hal codec mode.

Values:

AUDIO_HAL_CODEC_MODE_ENCODE = 1
select adc

AUDIO_HAL_CODEC_MODE_DECODE
select dac

AUDIO_HAL_CODEC_MODE_BOTH
select both adc and dac

AUDIO_HAL_CODEC_MODE_LINE_IN
set adc channel

enum audio_hal_adc_input_t
Select adc channel for input mic signal.

Values:

AUDIO_HAL_ADC_INPUT_LINE1 = 0x00
mic input to adc channel 1

AUDIO_HAL_ADC_INPUT_LINE2
mic input to adc channel 2

AUDIO_HAL_ADC_INPUT_ALL
mic input to both channels of adc

AUDIO_HAL_ADC_INPUT_DIFFERENCE
mic input to adc difference channel

enum audio_hal_dac_output_t
Select channel for dac output.

Values:

AUDIO_HAL_DAC_OUTPUT_LINE1 = 0x00
dac output signal to channel 1

AUDIO_HAL_DAC_OUTPUT_LINE2
dac output signal to channel 2

AUDIO_HAL_DAC_OUTPUT_ALL
dac output signal to both channels

enum audio_hal_ctrl_t
Select operating mode i.e. start or stop for audio codec chip.

Values:

AUDIO_HAL_CTRL_STOP = 0x00
set stop mode

AUDIO_HAL_CTRL_START = 0x01
set start mode

enum audio_hal_iface_mode_t
Select I2S interface operating mode i.e. master or slave for audio codec chip.

Values:

AUDIO_HAL_MODE_SLAVE = 0x00
set slave mode

AUDIO_HAL_MODE_MASTER = 0x01
set master mode

enum audio_hal_iface_samples_t
Select I2S interface samples per second.

Values:

AUDIO_HAL_08K_SAMPLES
set to 8k samples per second

AUDIO_HAL_11K_SAMPLES
set to 11.025k samples per second

AUDIO_HAL_16K_SAMPLES
set to 16k samples in per second

AUDIO_HAL_22K_SAMPLES
set to 22.050k samples per second

AUDIO_HAL_24K_SAMPLES
set to 24k samples in per second

AUDIO_HAL_32K_SAMPLES
set to 32k samples in per second

AUDIO_HAL_44K_SAMPLES
set to 44.1k samples per second

AUDIO_HAL_48K_SAMPLES
set to 48k samples per second

enum audio_hal_iface_bits_t
Select I2S interface number of bits per sample.

Values:

AUDIO_HAL_BIT_LENGTH_16BITS = 1
set 16 bits per sample

AUDIO_HAL_BIT_LENGTH_24BITS
set 24 bits per sample

AUDIO_HAL_BIT_LENGTH_32BITS
set 32 bits per sample

enum audio_hal_iface_format_t
Select I2S interface format for audio codec chip.

Values:

AUDIO_HAL_I2S_NORMAL = 0
set normal I2S format

AUDIO_HAL_I2S_LEFT
set all left format

AUDIO_HAL_I2S_RIGHT
set all right format

AUDIO_HAL_I2S_DSP
set dsp/pcm format

2.6.3 ES8388 Driver

Driver for ES8388 codec chip used in *ESP32 LyraT* audio board.

API Reference

Header File

- audio_hal/driver/es8388/es8388.h

Functions

esp_err_t **es8388_init**(*audio_hal_codec_config_t *cfg*)
Initialize ES8388 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `cfg`: configuration of ES8388

esp_err_t **es8388_deinit**(void)
Deinitialize ES8388 codec chip.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_config_fmt**(*es_module_t mod*, *es_i2s_fmt_t cfg*)
Configure ES8388 I2S format.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mod`: set ADC or DAC or both
- `cfg`: ES8388 I2S format

esp_err_t **es8388_i2s_config_clock**(*es_i2s_clock_t cfg*)
Configure I2s clock in MSATER mode.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `cfg`: set bits clock and WS clock

esp_err_t **es8388_set_bits_per_sample**(*es_module_t mode*, *es_bits_length_t bit_per_sample*)
Configure ES8388 data sample bits.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both
- `bit_per_sample`: bit number of per sample

esp_err_t **es8388_start**(*es_module_t mode*)
Start ES8388 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both

esp_err_t **es8388_stop**(*es_module_t mode*)
Stop ES8388 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both

esp_err_t **es8388_set_voice_volume**(int *volume*)
Set voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `volume`: voice volume (0~100)

esp_err_t **es8388_get_voice_volume**(int **volume*)
Get voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `*volume`: voice volume (0~100)

esp_err_t **es8388_set_voice_mute**(int *enable*)
Configure ES8388 DAC mute or not. Basically you can use this function to mute the output or not.

Return

- ESP_FAIL Parameter error

- ESP_OK Success

Parameters

- `enable`: enable(1) or disable(0)

esp_err_t **es8388_get_voice_mute**(void)
Get ES8388 DAC mute status.

Return

- -1 Parameter error
- 0 voice mute disable
- 1 voice mute enable

esp_err_t **es8388_set_mic_gain**(*es_mic_gain_t gain*)
Set ES8388 mic gain.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `gain`: db of mic gain

esp_err_t **es8388_config_adc_input**(*es_adc_input_t input*)
Set ES8388 adc input mode.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `input`: adc input mode

esp_err_t **es8388_config_dac_output**(*es_dac_output_t output*)
Set ES8388 dac output mode.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `output`: dac output mode

esp_err_t **es8388_write_reg**(uint8_t *reg_add*, uint8_t *data*)
Write ES8388 register.

Return

- ESP_FAIL Parameter error

- ESP_OK Success

Parameters

- `reg_add`: address of register
- `data`: data of register

void **es8388_read_all**()
Print all ES8388 registers.

Return

- void

esp_err_t **es8388_config_i2s**(*audio_hal_codec_mode_t* *mode*, *audio_hal_codec_i2s_iface_t* **iface*)
Configure ES8388 codec mode and I2S interface.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `iface`: I2S config

esp_err_t **es8388_ctrl_state**(*audio_hal_codec_mode_t* *mode*, *audio_hal_ctrl_t* *ctrl_state*)
Control ES8388 codec chip.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `ctrl_state`: start or stop decode or encode progress

void **es8388_pa_power**(bool *enable*)
Set ES8388 PA power.

Return

- void

Parameters

- `enable`: true for enable PA power, false for disable PA power

Structures

struct es_i2s_clock_t
Configure ES8388 clock.

Public Members

es_sclk_div_t **sclk_div**
bits clock divide

es_lclk_div_t **lclk_div**
WS clock divide

Macros

ES8388_ADDR
0x22:CE=1;0x20:CE=0

ES8388_CONTROL1

ES8388_CONTROL2

ES8388_CHIPPOWER

ES8388_ADCPOWER

ES8388_DACPOWER

ES8388_CHIPLOPOW1

ES8388_CHIPLOPOW2

ES8388_ANAVOLMANAG

ES8388_MASTERMODE

ES8388_ADCCONTROL1

ES8388_ADCCONTROL2

ES8388_ADCCONTROL3

ES8388_ADCCONTROL4

ES8388_ADCCONTROL5

ES8388_ADCCONTROL6

ES8388_ADCCONTROL7

ES8388_ADCCONTROL8

ES8388_ADCCONTROL9

ES8388_ADCCONTROL10

ES8388_ADCCONTROL11

ES8388_ADCCONTROL12

ES8388_ADCCONTROL13

ES8388_ADCCONTROL14

ES8388_DACCONTROL1

ES8388_DACCONTROL2

ES8388_DACCONTROL3

ES8388_DACCONTROL4

ES8388_DACCONTROL5

ES8388_DACCONTROL6

ES8388_DACCONTROL7

ES8388_DACCONTROL8

ES8388_DACCONTROL9

ES8388_DACCONTROL10

ES8388_DACCONTROL11

ES8388_DACCONTROL12

ES8388_DACCONTROL13

ES8388_DACCONTROL14

ES8388_DACCONTROL15

ES8388_DACCONTROL16

ES8388_DACCONTROL17

ES8388_DACCONTROL18

ES8388_DACCONTROL19

ES8388_DACCONTROL20

ES8388_DACCONTROL21

ES8388_DACCONTROL22

ES8388_DACCONTROL23

ES8388_DACCONTROL24

ES8388_DACCONTROL25

ES8388_DACCONTROL26

ES8388_DACCONTROL27

ES8388_DACCONTROL28

ES8388_DACCONTROL29

ES8388_DACCONTROL30

Enumerations

enum es_bits_length_t

Values:

BIT_LENGTH_MIN = -1

BIT_LENGTH_16BITS = 0x03

BIT_LENGTH_18BITS = 0x02

BIT_LENGTH_20BITS = 0x01

BIT_LENGTH_24BITS = 0x00

BIT_LENGTH_32BITS = 0x04

BIT_LENGTH_MAX

enum es_sclk_div_t

Values:

MCLK_DIV_MIN = -1

MCLK_DIV_1 = 1

MCLK_DIV_2 = 2

MCLK_DIV_3 = 3

MCLK_DIV_4 = 4

MCLK_DIV_6 = 5

MCLK_DIV_8 = 6

MCLK_DIV_9 = 7

MCLK_DIV_11 = 8

MCLK_DIV_12 = 9

MCLK_DIV_16 = 10

MCLK_DIV_18 = 11

MCLK_DIV_22 = 12

MCLK_DIV_24 = 13

MCLK_DIV_33 = 14

MCLK_DIV_36 = 15

MCLK_DIV_44 = 16

MCLK_DIV_48 = 17

MCLK_DIV_66 = 18

MCLK_DIV_72 = 19

MCLK_DIV_5 = 20

MCLK_DIV_10 = 21

MCLK_DIV_15 = 22

MCLK_DIV_17 = 23

MCLK_DIV_20 = 24

MCLK_DIV_25 = 25

MCLK_DIV_30 = 26

MCLK_DIV_32 = 27

MCLK_DIV_34 = 28

MCLK_DIV_7 = 29

MCLK_DIV_13 = 30

MCLK_DIV_14 = 31

MCLK_DIV_MAX

enum es_lclk_div_t

Values:

LCLK_DIV_MIN = -1

LCLK_DIV_128 = 0

LCLK_DIV_192 = 1

LCLK_DIV_256 = 2

LCLK_DIV_384 = 3

LCLK_DIV_512 = 4

LCLK_DIV_576 = 5

LCLK_DIV_768 = 6

LCLK_DIV_1024 = 7

LCLK_DIV_1152 = 8

LCLK_DIV_1408 = 9

LCLK_DIV_1536 = 10

LCLK_DIV_2112 = 11

LCLK_DIV_2304 = 12

LCLK_DIV_125 = 16

LCLK_DIV_136 = 17

LCLK_DIV_250 = 18

LCLK_DIV_272 = 19

LCLK_DIV_375 = 20

LCLK_DIV_500 = 21

LCLK_DIV_544 = 22

LCLK_DIV_750 = 23

LCLK_DIV_1000 = 24

LCLK_DIV_1088 = 25

LCLK_DIV_1496 = 26

LCLK_DIV_1500 = 27

LCLK_DIV_MAX

enum es_d2se_pga_t

Values:

D2SE_PGA_GAIN_MIN = -1

D2SE_PGA_GAIN_DIS = 0

D2SE_PGA_GAIN_EN = 1

D2SE_PGA_GAIN_MAX =2

enum es_adc_input_t

Values:

ADC_INPUT_MIN = -1

ADC_INPUT_LINPUT1_RINPUT1 = 0x00

ADC_INPUT_MIC1 = 0x05

ADC_INPUT_MIC2 = 0x06

ADC_INPUT_LINPUT2_RINPUT2 = 0x50

ADC_INPUT_DIFFERENCE = 0xf0

ADC_INPUT_MAX

enum es_dac_output_t

Values:

DAC_OUTPUT_MIN = -1

DAC_OUTPUT_LOUT1 = 0x04

DAC_OUTPUT_LOUT2 = 0x08

DAC_OUTPUT_SPK = 0x09

DAC_OUTPUT_ROUT1 = 0x10

DAC_OUTPUT_ROUT2 = 0x20

DAC_OUTPUT_ALL = 0x3c

DAC_OUTPUT_MAX

enum es_mic_gain_t

Values:

MIC_GAIN_MIN = -1

MIC_GAIN_0DB = 0

MIC_GAIN_3DB = 3

MIC_GAIN_6DB = 6

MIC_GAIN_9DB = 9

MIC_GAIN_12DB = 12

MIC_GAIN_15DB = 15

MIC_GAIN_18DB = 18

MIC_GAIN_21DB = 21

MIC_GAIN_24DB = 24

MIC_GAIN_MAX

enum es_module_t

Values:

ES_MODULE_MIN = -1

ES_MODULE_ADC = 0x01

ES_MODULE_DAC = 0x02

ES_MODULE_ADC_DAC = 0x03

ES_MODULE_LINE = 0x04

ES_MODULE_MAX

enum es_mode_t

Values:

ES_MODE_MIN = -1

ES_MODE_SLAVE = 0x00

ES_MODE_MASTER = 0x01

ES_MODE_MAX

enum es_i2s_fmt_t

Values:

ES_I2S_MIN = -1

ES_I2S_NORMAL = 0

ES_I2S_LEFT = 1

ES_I2S_RIGHT = 2

ES_I2S_DSP = 3

ES_I2S_MAX

2.6.4 ES8374 Driver

Driver for ES8374 codec chip.

API Reference

Header File

- audio_hal/driver/es8374/es8374.h

Functions

esp_err_t **es8374_init**(*audio_hal_codec_config_t* *cfg)

Initialize ES8374 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `cfg`: configuration of ES8374

esp_err_t **es8374_deinit**(void)

Deinitialize ES8374 codec chip.

Return

- ESP_OK

- ESP_FAIL

esp_err_t **es8374_config_fmt**(*es_module_t mode*, *es_i2s_fmt_t fmt*)
Configure ES8374 I2S format.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both
- `fmt`: ES8374 I2S format

esp_err_t **es8374_i2s_config_clock**(*es_i2s_clock_t cfg*)
Configure I2S clock in MSATER mode.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `cfg`: set bits clock and WS clock

esp_err_t **es8374_set_bits_per_sample**(*es_module_t mode*, *es_bits_length_t bit_per_sample*)
Configure ES8374 data sample bits.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both
- `bit_per_sample`: bit number of per sample

esp_err_t **es8374_start**(*es_module_t mode*)
Start ES8374 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both

esp_err_t **es8374_stop**(*es_module_t mode*)
Stop ES8374 codec chip.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both

esp_err_t **es8374_set_voice_volume**(int *volume*)
Set voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `volume`: voice volume (0~100)

esp_err_t **es8374_get_voice_volume**(int **volume*)
Get voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `*volume`: voice volume (0~100)

esp_err_t **es8374_set_voice_mute**(int *enable*)
Mute or unmute ES8374 DAC. Basically you can use this function to mute or unmute the output.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `enable`: mute(1) or unmute(0)

esp_err_t **es8374_get_voice_mute**(void)
Get ES8374 DAC mute status.

Return

- -1: Parameter error
- 0: Voice is unmuted
- 1: Voice is muted

esp_err_t **es8374_set_mic_gain**(*es_mic_gain_t gain*)
Set ES8374 mic gain.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `gain`: db of mic gain

esp_err_t **es8374_config_adc_input**(*es_adc_input_t input*)
Set ES8374 ADC input mode.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `input`: adc input mode

esp_err_t **es8374_config_dac_output**(*es_dac_output_t output*)
Set ES8374 DAC output mode.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `output`: dac output mode

esp_err_t **es8374_write_reg**(uint8_t *reg_add*, uint8_t *data*)
Write ES8374 register.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `reg_add`: address of register
- `data`: data of register

void **es8374_read_all**()
Print all ES8374 registers.

Return

- void

esp_err_t **es8374_config_i2s**(*audio_hal_codec_mode_t mode*, *audio_hal_codec_i2s_iface_t *iface*)
Configure ES8374 codec mode and I2S interface.

Return

- ESP_FAIL Parameter error

- ESP_OK Success

Parameters

- `mode`: codec mode
- `iface`: I2S config

esp_err_t **es8374_ctrl_state**(*audio_hal_codec_mode_t mode*, *audio_hal_ctrl_t ctrl_state*)
Control ES8374 codec chip.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `ctrl_state`: start or stop decode or encode progress

void **es8374_pa_power**(bool *enable*)
Set ES8374 PA power.

Return

- void

Parameters

- `enable`: true for enable PA power, false for disable PA power

Macros

ES8374_ADDR

2.6.5 ZL38063 Driver

Driver for ZL38063 codec chip used in *ESP32-LyraTD-MSC* audio board.

API Reference

Header File

- audio_hal/driver/zl38063/zl38063.h

Functions

esp_err_t **zl38063_init**(*audio_hal_codec_config_t *cfg*)
Initialize ZL38063 chip.

Return

- ESP_OK

- ESP_FAIL

Parameters

- `cfg`: configuration of ZL38063

esp_err_t **zl38063_deinit**(void)
Deinitialize ZL38063 chip.

Return

- ESP_OK
- ESP_FAIL

esp_err_t **zl38063_ctrl_state**(*audio_hal_codec_mode_t* *mode*, *audio_hal_ctrl_t* *ctrl_state*)
Control ZL38063 chip.

The functions zl38063_ctrl_state and zl38063_config_i2s are not used by this driver. They are kept here to maintain the uniformity and convenience of the interface of the ADF project. These settings for zl38063 are burned in firmware and configuration files. Default i2s configuration: 48000Hz, 16bit, Left-Right channels. Use resampling to be compatible with different file types.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `ctrl_state`: start or stop decode or encode progress

esp_err_t **zl38063_config_i2s**(*audio_hal_codec_mode_t* *mode*, *audio_hal_codec_i2s_iface_t* **iface*)
Configure ZL38063 codec mode and I2S interface.

Return

- ESP_FAIL Parameter error
- ESP_OK Success

Parameters

- `mode`: codec mode
- `iface`: I2S config

esp_err_t **zl38063_set_voice_volume**(int *volume*)
Set voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `volume`: voice volume (0~100)

esp_err_t **zl38063_get_voice_volume**(int *volume)
Get voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `*volume`: voice volume (0~100)

Macros

AUDIO_HAL_ZL38063_DEFAULT

2.7 Configuration Options

Compile-time configuration options specific to ESP-ADF.

2.7.1 ESP HTTP client

ESP_HTTP_CLIENT_ENABLE_HTTPS

Enable https

Found in: Component config > ESP HTTP client

This option will enable https protocol by linking mbedtls library and initializing SSL transport

2.7.2 Audio HAL

AUDIO_BOARD

Audio board

Found in: Audio HAL

Select an audio board to use with the ESP-ADF

Available options:

- ESP_LYRAT_V4_3_BOARD
- ESP_LYRAT_V4_2_BOARD
- ESP_LYRATD_MSC_V2_1_BOARD
- ESP_LYRATD_MSC_V2_2_BOARD

2.7.3 Speech Recognition Configuration

SR_MODEL_SEL

Speech recognition wake net to use

Found in: Speech Recognition Configuration

Select the wake net to be used.

Available options:

- SR_MODEL_WN3_QUANT
- SR_MODEL_WN4_QUANT

NAME_OF_WAKEUP_WORD

Name of wakeup word

Found in: Speech Recognition Configuration

Select wakeup word

Available options:

- WAKEUP_WORD_HI_LEXIN
- WAKEUP_WORD_ALEXA
- WAKEUP_WORD_LIGHT_CONTROL_CN
- WAKEUP_WORD_SPEECH_CMD_CH

2.7.4 HTTP Server

HTTPD_MAX_REQ_HDR_LEN

Max HTTP Request Header Length

Found in: Component config > HTTP Server

This sets the maximum supported size of headers section in HTTP request packet to be processed by the server

HTTPD_MAX_URI_LEN

Max HTTP URI Length

Found in: Component config > HTTP Server

This sets the maximum supported size of HTTP request URI to be processed by the server

CHAPTER 3

Design Guide

The ESP32 is a powerful chip well positioned as a MCU of the audio projects. This section is intended to provide guidance on process of designing an audio project with the ESP32 inside.

3.1 Project Design

When designing a project with ability to process an audio signal or audio data we typically consider a subset of the following components:

Input:

- **Analog signal input** to connect e.g. a microphone
- **Storage media**, e.g. microSD card with audio files to read them
- **WI-Fi** interface to obtain an audio data stream from the internet
- **Bluetooth** interface to obtain an audio data stream from e.g. a BT headset
- **I2S interface** to obtain audio data stream from a codec chip
- **Ethernet** interface to obtain an audio data stream from the internet
- An internal **chip's flash memory** with some audio samples to play
- **User Interface** e.g. buttons or some other means to provide user input

Output:

- **Analog signal output** to connect headphones or and amplifier with speakers
- **Storage media**, e.g. microSD card to write some audio files, e.g. with recording
- **WI-Fi** interface to send out an audio data stream to the internet
- **Bluetooth** interface to stream audio data to e.g. a BT headset
- **I2S interface** to stream some data to a codec chip

- **Ethernet** interface to stream an audio data stream to the internet
- An internal **chip's flash memory** to store some audio recording
- **User Interface** e.g. a display, LEDs or some means of haptic feedback

Main Processing Unit:

A microcontroller or a computer with processing power to read the data from the input, process (e.g. encode / encode) and send to the output.

3.1.1 Project Options

The ESP32 has all the above features or is able to support them (e.g. can drive Ethernet PHY). Considering the ESP32 cost is about $3, and availability of ESP-ADF software development platform, we are able to develop an audio project with minimum additional components at very low price.

Depending on the application, required functionality and performance, we may consider two project groups.

- **Minimum** - having minimum additional components, assuming using on board I2S, or PDM interface as well as DAC, if no high qualify audio on the output is required.
- **Typical** - with an external codec chip and a power amplifier, for high quality output audio and multiple input / output options.

There may be several variation between the above projects, by adding or removing features / components. Below are couple of examples.

3.1.2 Project Minimum

With several peripherals on ESP32, I2S or PDM or DAC interfaces can be used to implement a minimum project. With the digital microphones, we could input voice signals and build a command voice control project minumum that could communicate with a cloud service.

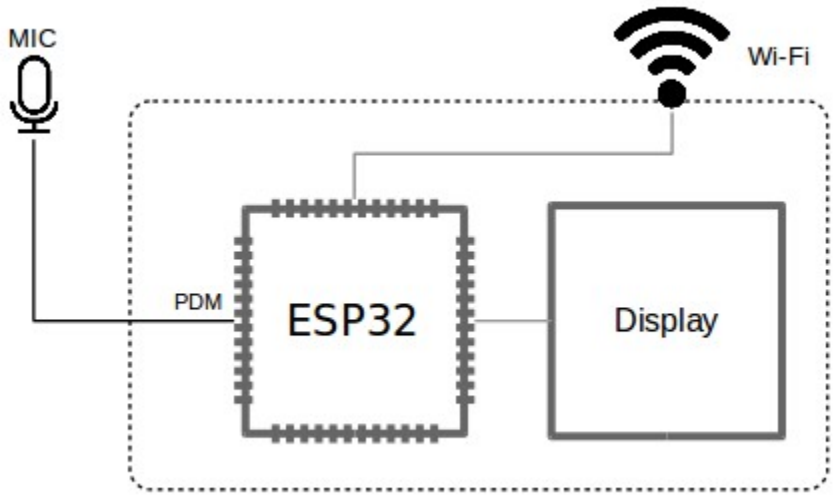


Fig. 3.1: Audio Project Example - Send Voice Commands to Cloud Service

With two on board DACs, if 8-bit width on the output is satisfactory, we may implement another project minimum - a device to play an internet connected radio.

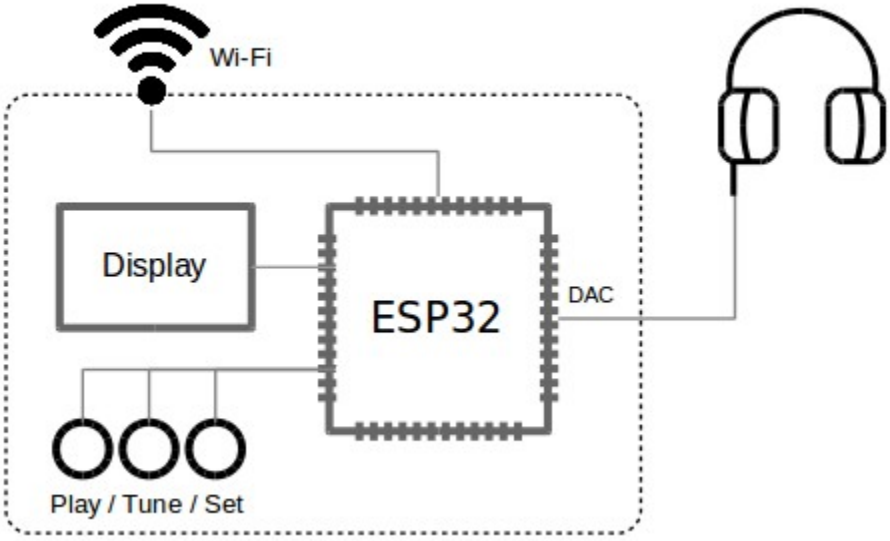


Fig. 3.2: Audio Project Example - Internet Connected Radio Player

3.1.3 Typical Project

When looking for better audio quality and more interfacing options we would use an external I2S codec to do all the analog input and output signal processing. The codec chip, depending on type, may provide additional functionality like audio input signal preamplifier, headphone output amplifier, multiple analog input and outputs, sound effects, etc. The I2S is considered as the industry standard for interfacing with audio codec chips, or in general for a high speed, continuous transfer of the audio data. To optimize performance of audio data processing additional memory may be required. For such cases consider using ESP32-WROVER that provides 4 MB PSRAM on a single module together with the ESP32 chip.

The ESP-ADF is designed primarily to support projects with a codec chip. The *ESP32 LyraT* board is an example of such a project. The software interfacing with the board is done by Audio HAL and a driver. The codec chip used on the ESP32 LyraT is ES8388. Boards with a different codec chip may be supported by providing a different driver.

3.2 Design Considerations

Depending on the audio data format, that may be lossless, lossy or compressed, e.g. WAV, MP3 or FLAC and the quality expressed in sampling rate and bitrate, the project will require different resources: memory, storage space, input / output throughput and the processing power. The resources will also depend on the project type and features discussed in *Project Design*.

This section describes capacity and performance of ESP32 system resources that should be considered when designing an audio project to meet required data format, audio quality and functionality.

3.2.1 Memory

The spare internal Data-RAM is about 290kB with “hello_world” example. For audio system this may be insufficient, and therefore the ESP32 incorporates the ability to use up to 4MB of external SPI RAM (i.e. PSRAM) memory. The

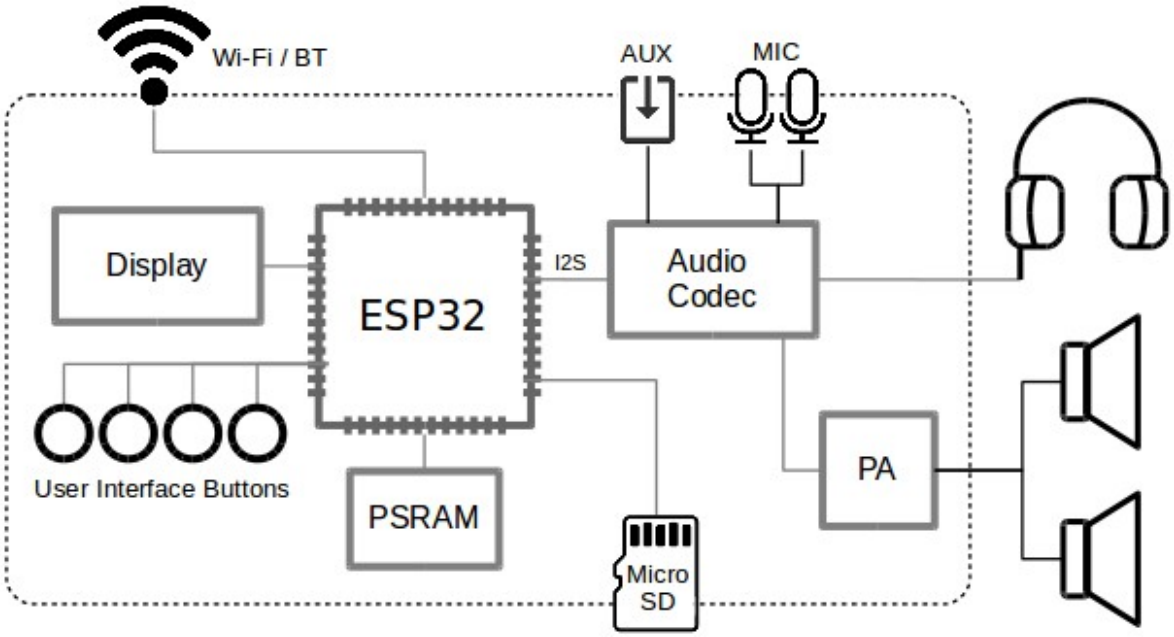


Fig. 3.3: Typical Audio Project Example

external memory is incorporated in the memory map and is, within certain restrictions, usable in the same way internal Data-RAM is.

Refer to External SPI-connected RAM section in IDF documenation for details, especially pay attention to its Restrictions section which is very important.

To be able to use the PSRAM, if installed on your board, it should be enabled in menucofig under *Component config > ESP32-specific > SPI RAM config*. The option *CONFIG_SPIRAM_CACHE_WORKAROUND*, set by default in the same menu, should be kept enabled.

Note: Bluetooth and Wi-Fi can not coexist without PSRAM because it will not leave enough memory for an audio application.

Optimization of Internal RAM and Use of PSRAM

Internal RAM is more valuable asset since there are some restrictions on PSRAM. Here are some tips for optimizing internal RAM.

- If PSRAM is in use, set all the static buffer to minimum value in *Component config > Wi-Fi*; if PSRAM is not used then dynamic buffer should be selected to save memory. Refer to Wi-Fi Buffer Usage section in IDF documentation for details.
- If PSRAM and BT are used, then *CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST* and *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY* should be set as "yes" under *Component config > Bluetooth > Bluedroid Enable*, to allocate more of 40kB memory to PSRAM
- If PSRAM and Wi-Fi are used, then *CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST* should be set as "yes" under *Component config > ESP32-specific > SPI RAM config*, to allocate some memory to PSRAM

- Set *CONFIG_WL_SECTOR_SIZE* as 512 in *Component config* > *Wear Levelling*

Note: The smaller the size of sector be, the slower the Write / Read speed will be, and vice versa, but only 512 and 4096 are supported.

- Call `char *buf = heap_caps_malloc(1024 * 10, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT)` instead of `malloc(1024 * 10)` to use PSRAM, and call `char *buf = heap_caps_malloc(512, MALLOC_CAP_INTERNAL | MALLOC_CAP_8BIT)` to use internal RAM.
- Not relying on `malloc()` to automatically allocate PSRAM allows to make a full control of the memory. By avoiding the use of the internal RAM by other `malloc()` calls, you can reserve more memory for high-efficiency usage and task stack since PSRAM cannot be used as task stack memory.
- The task stack will always be allocated at internal RAM. On the other hand you can use of the `xTaskCreateStatic()` function that allows to create tasks with stack on PSRAM (see options in PSRAM and FreeRTOS menuconfig), but pay attention to its help information.

Important: Don't use ROM code in `xTaskCreateStatic` task The ROM code itself is linked in `components/esp32/ld/esp32.rom.ld`. However, you also need to consider other pieces of code that *call* ROM functions, as well as the code that is not recompiled against the *CONFIG_SPIRAM_CACHE_WORKAROUND* patch, like the Wi-Fi and Bluetooth libraries. In general, we advise using this only in threads that do not call any IDF libraries (including libc), doing only calculations and using FreeRTOS primitives to talk to other threads.

Memory Usage by Component Overview

Below is a table that contains ESP-ADF components and their memory usage. Choose the components needed and find out how much internal RAM is left. The table is divided into two parts, when PSRAM is used or not. If PSRAM (external RAM) is in use, then some of the memory will be allocated at PSRAM automatically.

The initial spare internal RAM is 290kB.

Component	Internal RAM Required	
	PSRAM not used	With PSRAM
Wi-Fi [1]	50kB+	50kB+
Bluetooth	140kB (50kB if only BLE needed)	95kB (50kB if only BLE needed)
Flash Card [2]	12kB+	12kB+
I2S [3]	Configurable, 8kB for reference	Configurable, 8kB for reference
RingBuffer [4]	Configurable, 30kB for reference	0kB, all moved into PSRAM

Notes to the table above

1. According to the Wi-Fi menuconfig each Tx and Rx buffer occupies 1.6kB internal RAM. The value of 50kB RAM is assuming use of 5 Rx static buffers and 6 Tx static buffers. If PSRAM is not in use, then the "Type of WiFi Tx Buffer" option should be set as *DYNAMIC* in order to save RAM, in this case, the RAM usage will be far less than 50kB, but programmer should keep at least 50kB available for the Wi-Fi to be able to transmit the data. **[Internal RAM only]**
2. Depending on value of *SD_CARD_OPEN_FILE_NUM_MAX* in audio_hal/board/board.h, that is then used in `sd_card_mount()` function, the RAM needed will increase with a greater number of maximum open files. 12kB is the RAM needed with 5 max files and 512 bytes *CONFIG_WL_SECTOR_SIZE*. **[Internal RAM only]**

3. Depending on configuration settings of the I2S stream, refer to audio_stream/include/i2s_stream.h and audio_stream/i2s_stream.c. **[Internal RAM only]**
4. Depending on configuration setting of the Ringbuffer, refer to *DEFAULT_PIPELINE_RINGBUF_SIZE* in audio_pipeline/include/audio_pipeline.h or user setting, if the buffer is created with e.g. `rb_create()`.

3.2.2 System Settings

The following settings are recommended to achieve a high Wi-Fi performance in an audio project.

Note: Use ESP32 modules and boards from reputable vendors that put attention to product design, component selection and product testing. This is to have confidence of receiving well designed boards with calibrated RF.

- Set these following options in menuconfig.
  - Flash SPI mode as QIO
  - Flash SPI speed as 80MHz
  - CPU frequency as 240MHz
  - Set *Default receive window size* as 5 times greater than *Maximum Segment Size* in *Component config* > *LWIP* > *TCP*
- If external antenna is used, then set *PHY_RF_CAL_PARTIAL* as *PHY_RF_CAL_FULL* in ''esp-idf/components/esp32/phy_init.c''

3.3 Software Design

Espressif audio framework project.

3.3.1 Features

1. All of Streams and Codecs based on audio element.
2. All events based on queue.
3. Audio pipeline supports dynamic combination.
4. Audio pipeline supports multiple elements.
5. Pipeline Support functionality plug-in.
6. Audio common peripherals support work in the one task.
7. Support post-event mechanism in peripherals.
8. Support high level audio play API based on element and audio pipeline.
9. Audio high level interface supports dynamic adding of codec library.
10. Audio high level interface supports dynamic adding of input and output stream.
11. ESP audio supports multiple audio pipelines.

3.3.2 Design Components

Five basic components are - Audio Element, Audio Event, Audio Pipeline, ESP peripherals, ESP audio

Audio Element

Example

```
audio_element_handle_t el;
audio_element_cfg_t cfg = DEFAULT_AUDIO_ELEMENT_CONFIG();
cfg.open = _el_open;
cfg.read = _el_read;
cfg.process = _el_process;
cfg.write = _el_write;
cfg.close = _el_close;
el = audio_element_init(&cfg);
TEST_ASSERT_NOT_NULL(el);
TEST_ASSERT_EQUAL(ESP_OK, audio_element_start(el));
```

Audio Event

Example

```
audio_event_handle_t evt1;
audio_event_cfg_t cfg = AUDIO_EVENT_IFACE_DEFAULT_CFG();
cfg.dispatcher = evt_process;
cfg.queue_size = 10;
cfg.context = &evt1;
cfg.type = AUDIO_EVENT_TYPE_ELEMENT;
evt1 = audio_event_init(&cfg);
TEST_ASSERT_NOT_NULL(evt1);

audio_event_msg_t msg;
int i;
ESP_LOGI(TAG, "[✓] dispatch 10 msg to evt1");
for (i = 0; i < 10; i++) {
    msg.cmd = i;
    TEST_ASSERT_EQUAL(ESP_OK, audio_event_dispatch(evt1, &msg));
}
msg.cmd = 10;
TEST_ASSERT_EQUAL(ESP_FAIL, audio_event_dispatch(evt1, &msg));
ESP_LOGI(TAG, "[✓] listening 10 event have dispatched fron evt1");
while (audio_event_listen(evt1) == ESP_OK);
```

Audio Pipeline

Example

```
audio_element_handle_t first_el, mid_el, last_el;
audio_element_cfg_t el_cfg = DEFAULT_AUDIO_ELEMENT_CONFIG();
```

```
el_cfg.open = _el_open;
el_cfg.read = _el_read;
el_cfg.process = _el_process;
el_cfg.close = _el_close;
first_el = audio_element_init(&el_cfg, "first");
TEST_ASSERT_NOT_NULL(first_el);

el_cfg.read = NULL;
el_cfg.write = NULL;
mid_el = audio_element_init(&el_cfg, "mid");
TEST_ASSERT_NOT_NULL(mid_el);
el_cfg.write = _el_write;
last_el = audio_element_init(&el_cfg, "last");
TEST_ASSERT_NOT_NULL(last_el);

audio_pipeline_cfg_t pipeline_cfg = DEFAULT_AUDIO_PIPELINE_CONFIG();
audio_pipeline_handle_t pipeline = audio_pipeline_init(&pipeline_cfg);
TEST_ASSERT_NOT_NULL(pipeline);
TEST_ASSERT_EQUAL(ESP_OK, audio_pipeline_register(pipeline, first_el, mid_el, last_
↪el));
TEST_ASSERT_EQUAL(ESP_OK, audio_pipeline_link(pipeline, (const char *[]){"first", "mid
↪", "last"}, 3));
```

Audio Peripheral

Example

```
esp_periph_config_t periph_cfg = {
    .event_handle = _periph_event_handle,
    .user_context = NULL,
};
esp_periph_init(&periph_cfg);

// Initialize button peripheral
periph_button_cfg_t btn_cfg = {
    .gpio_mask = GPIO_SEL_36 | GPIO_SEL_39
};
esp_periph_handle_t button_handle = periph_button_init(&btn_cfg);


esp_periph_start(button_handle);

ESP_LOGI(TAG, "wait for button Pressed or touched");

ESP_LOGI(TAG, "running...");
vTaskDelay(5000 / portTICK_RATE_MS);

esp_periph_stop(button_handle);
ESP_LOGI(TAG, "stop button...");
vTaskDelay(5000 / portTICK_RATE_MS);

esp_periph_start(button_handle);
ESP_LOGI(TAG, "start button...");
vTaskDelay(5000 / portTICK_RATE_MS);
```

```
ESP_LOGI(TAG, "destroy...");
esp_periph_destroy();
```

Audio Player

Example

```
esp_audio_cfg_t cfg = {
    .in_stream_buf_size = 4096,                    /*!< Input buffer size */
    .out_stream_buf_size = 4096,                   /*!< Output buffer size */
    .evt_que = NULL,                               /*!< Registered by uesr for␣
↪receiving esp_audio event */
    .resample_rate = 48000,                        /*!< sample rate */
    .hal = NULL,                                   /*!<  */
};
audio_hal_codec_config_t audio_hal_codec_cfg =  AUDIO_HAL_ES8388_DEFAULT();
cfg.hal = audio_hal_init(&audio_hal_codec_cfg, 0);
esp_audio_handle_t player = esp_audio_create(&cfg);
TEST_ASSERT_NOT_EQUAL(player, NULL);
raw_stream_cfg_t raw_cfg = {
    .type = AUDIO_STREAM_READER,
};
audio_element_handle_t raw =  raw_stream_init(&raw_cfg);
wav_decoder_cfg_t wav_cfg = DEFAULT_WAV_DECODER_CONFIG();
audio_element_handle_t wav = wav_decoder_init(&wav_cfg);

 fatfs_stream_cfg_t fatfs_cfg = {
    .type = AUDIO_STREAM_READER,
    .root_path = "/sdcard",
};
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
esp_audio_input_stream_add(player, fatfs_stream_init(&fatfs_cfg));

i2s_cfg.type = AUDIO_STREAM_WRITER;
esp_audio_output_stream_add(player, i2s_stream_init(&i2s_cfg));
wav_decoder_cfg_t wav_cfg = DEFAULT_WAV_DECODER_CONFIG();
esp_audio_codec_lib_add(player, AUDIO_CODEC_TYPE_DECODER, wav);
```

3.4 Development Boards

Hardware details of audio development boards designed by Espressif around ESP32.

3.4.1 ESP32-LyraT V4.3 Hardware Reference

This guide provides functional descriptions, configuration options for ESP32-LyraT V4.3 audio development board. As an introduction to functionality and using the LyraT, please see *ESP32-LyraT V4.3 Getting Started Guide*. Check section *Other Versions of LyraT* if you have different version of the board.

In this Section

- *Overview*
- *Functional Description*
  - *Hardware Setup Options*
    - *Enable MicroSD Card in 1-wire Mode*
    - *Enable MicroSD Card in 4-wire Mode*
    - *Enable JTAG*
    - *Using Automatic Upload*
  - *Allocation of ESP32 Pins*
  - *Pinout of Extension Headers*
    - *UART Header / JP2*
    - *I2S Header / JP4*
    - *I2C Header / JP5*
    - *JTAG Header / JP7*
  - *Notes of Power Distribution*
    - *Power Supply Separation*
    - *Three Dedicated LDOs*
    - *Separate Power Feed for the PAs*
  - *Selecting of the Audio Output*
- *Other Versions of LyraT*
- *Related Documents*

Overview

The ESP32-LyraT development board is a hardware platform designed for the dual-core ESP32 audio applications, e.g., Wi-Fi or BT audio speakers, speech-based remote controllers, smart-home appliances with audio functionality(ies), etc.

The block diagram below presents main components of the ESP32-LyraT.

Functional Description

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT board.

ESP32-WROVER Module The ESP32-WROVER module contains ESP32 chip to provide Wi-Fi / BT connectivity and data processing power as well as integrates 32 Mbit SPI flash and 32 Mbit PSRAM for flexible data storage.

Green LED A general purpose LED controlled by the **ESP32-WROVER Module** to indicate certain operation states of the audio application using dedicated API.

Function DIP Switch Used to configure function of GPIO12 to GPIO15 pins that are shared between devices, primarily between **JTAG Header** and **MicroSD Card**. By default, the **MicroSD Card** is enabled with all switches in *OFF* position. To enable the **JTAG Header** instead, switches in positions 3, 4, 5 and 6 should be put *ON*. If **JTAG** is not used and **MicroSD Card** is operated in the one-line mode, then GPIO12 and GPIO13 may be assigned to other functions. Please refer to ESP32 LyraT V4.3 schematic for more details.

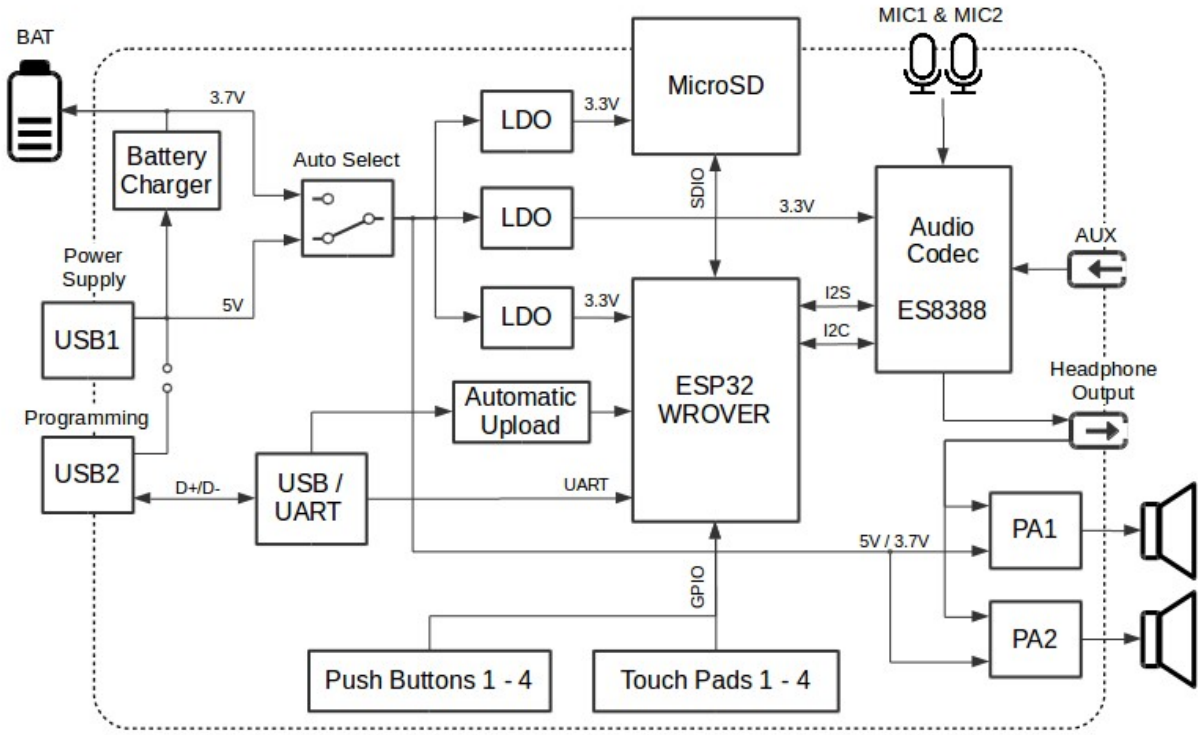


Fig. 3.4: ESP32-LyraT V4.3 Electrical Block Diagram

JTAG Header Provides access to the **JTAG** interface of **ESP32-WROVER Module**. It may be used for debugging, application upload, as well as implementing several other functions, e.g., Application Level Tracing. See *JTAG Header / JP7* for pinout details. Before using **JTAG** signals to the header, **Function DIP Switch** should be enabled. Please note that when **JTAG** is in operation, **MicroSD Card** cannot be used and should be disconnected because some of JTAG signals are shared by both devices.

UART Header Serial port: provides access to the serial TX/RX signals between **ESP32-WROVER Module** and **USB-UART Bridge Chip**.

I2C Header Provides access to the I2C interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2C Header / JP5* for pinout details.

MicroSD Slot The development board supports a MicroSD card in SPI/1-bit/4-bit modes, and can store or play audio files in the MicroSD card. Note that **JTAG** cannot be used and should be disconnected by setting **Function DIP Switch** when **MicroSD Card** is in operation, because some of signals are shared by both devices.

I2S Header Provides access to the I2S interface. Both **ESP32-WROVER Module** and **Audio Codec Chip** are connected to this interface. See *I2S Header / JP4* for pinout details.

Left Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

AUX Input Auxiliary input socket connected to IN2 (left and right channel) of the **Audio Codec Chip**. Use a 3.5 mm stereo jack to connect to this socket.

Headphone Output Output socket to connect headphones with a 3.5 mm stereo jack.

Note: The socket may be used with mobile phone headsets and is compatible with OMPT standard headsets only. It does work with CTIA headsets. Please refer to Phone connector (audio) on Wikipedia.

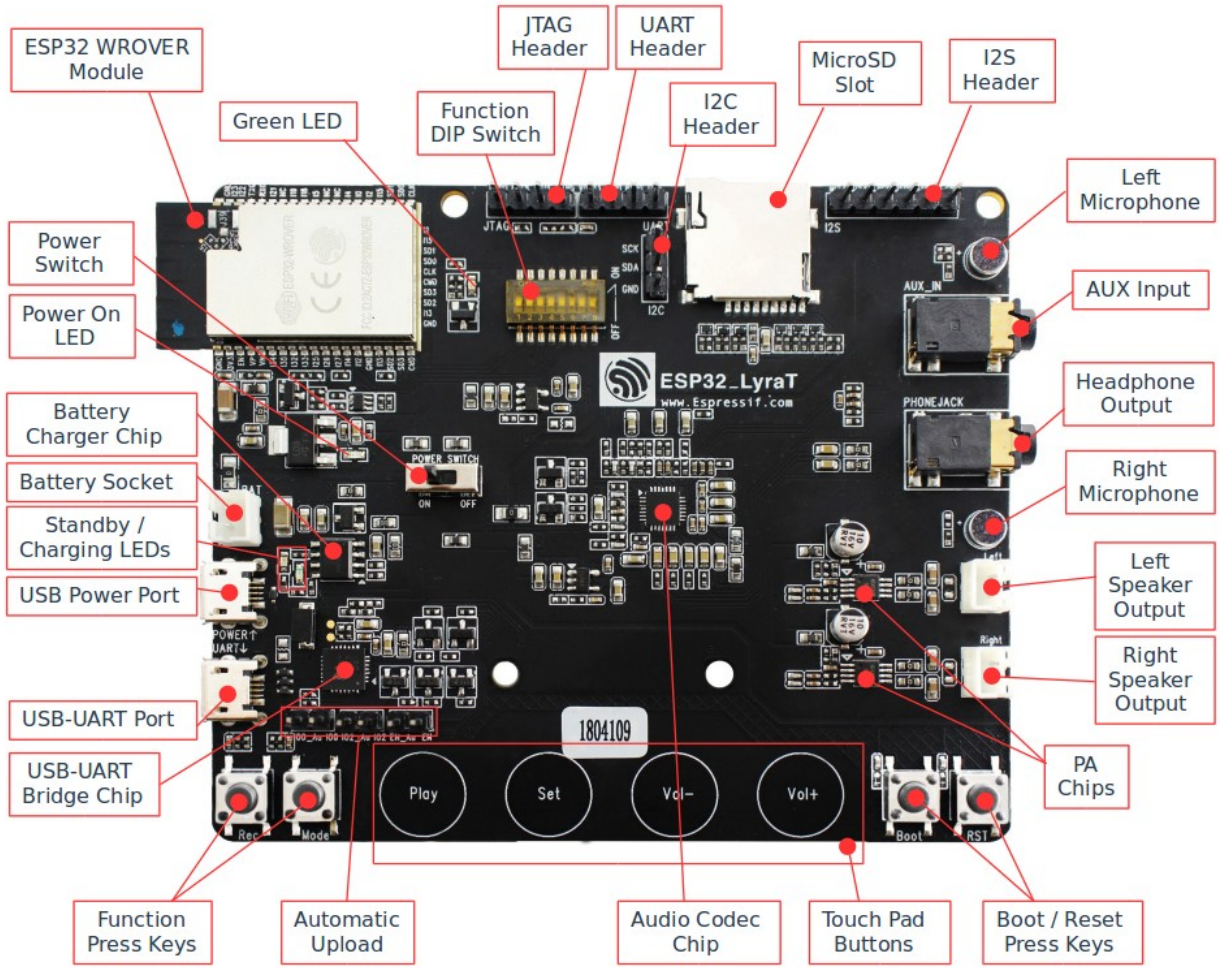


Fig. 3.5: ESP32-LyraT V4.3 Board Layout

Right Microphone Onboard microphone connected to IN1 of the **Audio Codec Chip**.

Left Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

Right Speaker Output Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1" pitch.

PA Chip A power amplifier used to amplify stereo audio signal from the **Audio Codec Chip** for driving two 4-ohm speakers.

Boot/Reset Press Keys Boot button: holding down the **Boot** button and momentarily pressing the **Reset** button to initiate the firmware download mode. Then you can download firmware through the serial port. Reset button: pressing this button alone resets the system.

Touch Pad Buttons Four touch pads labeled *Play*, *Sel*, *Vol+* and *Vol-*. They are routed to **ESP32-WROVER Module** and intended for development and testing of a UI for audio applications using dedicated API.

Audio Codec Chip The Audio Codec Chip, ES8388, is a low power stereo audio codec with a headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER Module** over I2S and I2S buses to provide audio processing in hardware independently from the audio application.

Automatic Upload Install three jumpers on this header to enable automatic loading of application to the ESP32. Install all jumpers together on all three headers. Remove all jumpers after upload is complete.

Function Press Keys Two key labeled *Rec* and *Mode*. They are routed to **ESP32-WROVER Module** and intended for developing and testing a UI for audio applications using dedicated API.

USB-UART Bridge Chip A single chip USB-UART bridge provides up to 1 Mbps transfers rate.

USB-UART Port Functions as the communication interface between a PC and the ESP32 module.

USB Power Port Provides the power supply for the board.

Standby / Charging LEDs The **Standby** green LED indicates that power has been applied to the **Micro USB Port**. The **Charging** red LED indicates that a battery connected to the **Battery Socket** is being charged.

Battery Socket Two pins socket to connect a single cell Li-ion battery.

Note: Please verify if polarity on the battery plug matches polarity of the socket as marked on the board's soldermask besides the socket.

Battery Charger Chip Constant current & constant voltage linear charger for single cell lithium-ion batteries AP5056. Used for charging of a battery connected to the **Battery Socket** over the **Micro USB Port**.

Power On LED Red LED indicating that **Power On Switch** is turned on.

Note: The **Power On Switch** does not affect / disconnect the Li-ion battery charging.

Power Switch Power on/off knob: toggling it to the left powers the board on; toggling it to the right powers the board off.

Hardware Setup Options

There are a couple of options to change the hardware configuration of the ESP32-LyraT board. The options are selectable with the **Function DIP Switch**.

Enable MicroSD Card in 1-wire Mode

DIP SW	Position
1	OFF
2	OFF
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF [1]
8	n/a

1. **AUX Input** detection may be enabled by toggling the DIP SW 7 *ON*. Note that the **AUX Input** signal pin should not be be plugged in when the system powers up. Otherwise the ESP32 may not be able to boot correctly.

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is available for use with the API

Enable MicroSD Card in 4-wire Mode

DIP SW	Position
1	ON
2	ON
3	OFF
4	OFF
5	OFF
6	OFF
7	OFF
8	n/a

In this mode:

- **JTAG** functionality is not available
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Enable JTAG

DIP SW	Position
1	OFF
2	OFF
3	ON
4	ON
5	ON
6	ON
7	ON
8	n/a

In this mode:

- **MicroSD Card** functionality is not available, remove the card from the slot
- *Vol-* touch button is not available for use with the API
- **AUX Input** detection from the API is not available

Using Automatic Upload

Entering of the ESP32 into upload mode may be done in two ways:

- Manually by pressing both **Boot** and **RST** keys and then releasing first **RST** and then **Boot** key.
- Automatically by software performing the upload. The software is using **DTR** and **RTS** signals of the serial interface to control states of **EN**, **IO0** and **IO2** pins of the ESP32. This functionality is enabled by installing jumpers in three headers **JP23**, **JP24** and **JP25**. For details see ESP32 LyraT V4.3 schematic. Remove all jumpers after upload is complete.

Allocation of ESP32 Pins

Several pins ESP32 module are allocated to the on board hardware. Some of them, like GPIO0 or GPIO2, have multiple functions. Please refer to the table below or ESP32 LyraT V4.3 schematic for specific details.

GPIO Pin	Type	Function Definition
SENSOR_VP	I	Audio **Rec** (PB)
SENSOR_VN	I	Audio **Mode** (PB)
IO32	I/O	Audio **Set** (TP)
IO33	I/O	Audio **Play** (TP)
IO27	I/O	Audio **Vol+** (TP)
IO13	I/O	JTAG **MTCK**, MicroSD **D3**, Audio **Vol-** (TP)
IO14	I/O	JTAG **MTMS**, MicroSD **CLK**
IO12	I/O	JTAG **MTDI**, MicroSD **D2**, Aux signal **detect**
IO15	I/O	JTAG **MTDO**, MicroSD **CMD**
IO2	I/O	Automatic Upload, MicroSD **D0**
IO4	I/O	MicroSD **D1**
IO34	I	MicroSD insert **detect**
IO0	I/O	Automatic Upload, I2S **MCLK**
IO5	I/O	I2S **SCLK**
IO25	I/O	I2S **LRCK**
IO26	I/O	I2S **DSDIN**
IO35	I	I2S **ASDOUT**
IO19	I/O	Headphone jack insert **detect**
IO22	I/O	Green LED indicator
IO21	I/O	PA Enable output
IO18	I/O	I2C **SDA**
IO23	I/O	I2C **SCL**

- (TP) - touch pad
- (PB) - push button

Pinout of Extension Headers

There are several pin headers available to connect external components, check the state of particular signal bus or debug operation of ESP32. Note that some signals are shared, see section *Allocation of ESP32 Pins* for details.

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

I2S Header / JP4

	I2C Header Pin	ESP32 Pin
1	MCLK	GPI0
2	SCLK	GPIO5
1	LRCK	GPIO25
2	DSDIN	GPIO26
3	ASDOUT	GPIO35
3	GND	GND

I2C Header / JP5

	I2C Header Pin	ESP32 Pin
1	SCL	GPIO23
2	SDA	GPIO18
3	GND	GND

JTAG Header / JP7

	ESP32 Pin	JTAG Signal
1	MTDO / GPIO15	TDO
2	MTCK / GPIO13	TCK
3	MTDI / GPIO12	TDI
4	MTMS / GPIO14	TMS

Notes of Power Distribution

The board features quite extensive power distribution system. It provides independent power supplies to all critical components. This should reduce noise in the audio signal from digital components and improve overall performance of the components.

Power Supply Separation

The main power supply is 5V and provided by a USB. The secondary power supply is 3.7V and provided by an optional battery. The USB power itself is fed with a dedicated cable, separate from a USB cable used for an application upload. To further reduce noise from the USB, the battery may be used instead of the USB.

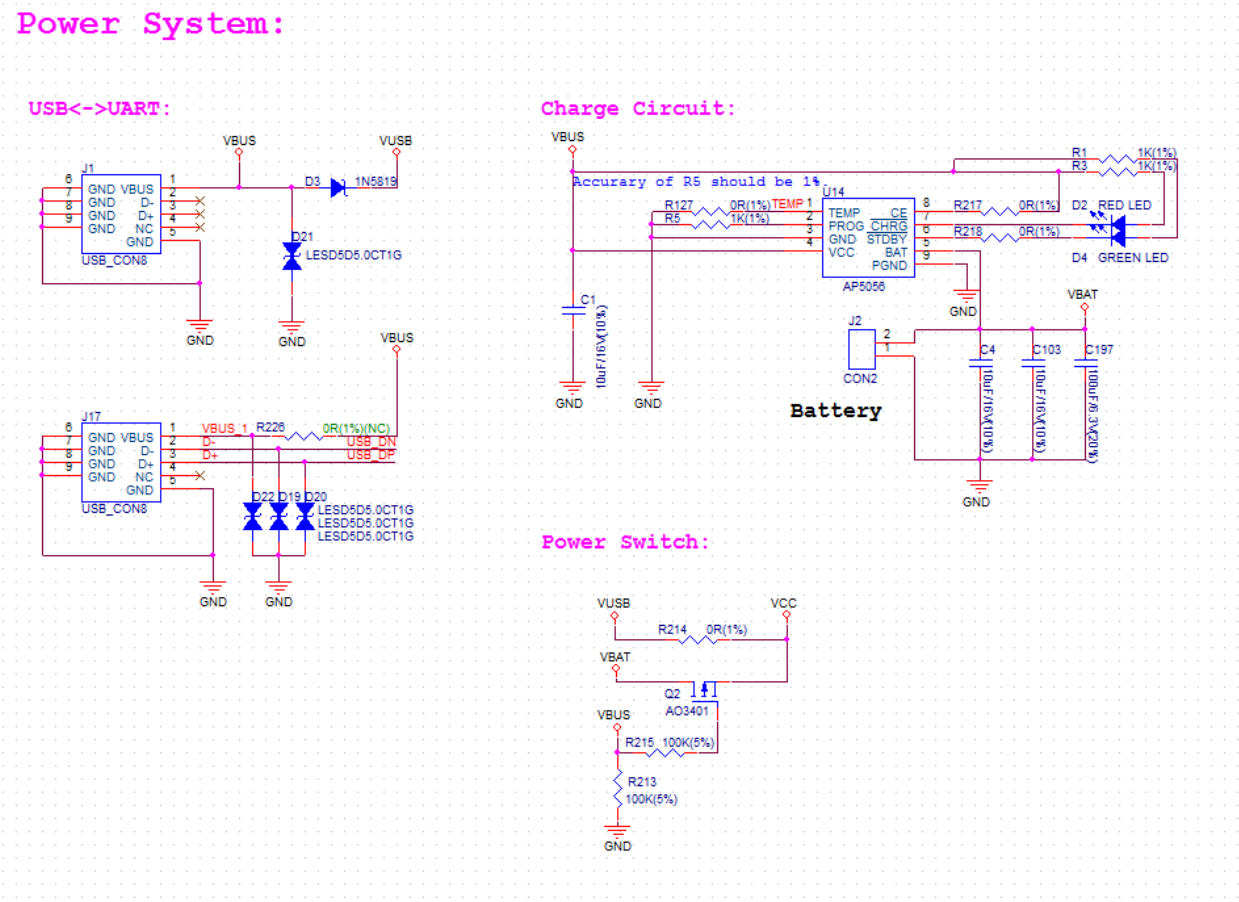


Fig. 3.6: ESP32 LyraT V4.3 - Power Supply Separation

Three Dedicated LDOs

ESP32 Module

To provide enough current the ESP32, the development board adopts LD1117S33CTR LDO capable to supply the maximum output current of 800mA.

MicroSD Card and **Audio Codec**

Two separate LDOs are provided for the MicorSD Card and the Audio Codec. Both circuits have similar design that includes an inductor and double decoupling capacitors on both the input and output of the LDO.

Separate Power Feed for the PAs

The audio amplifier unit features two NS4150 that require a large power supply for driving external speakers with the maximum output power of 3W. The power is supplied directly to both PAs from the battery or the USB. The

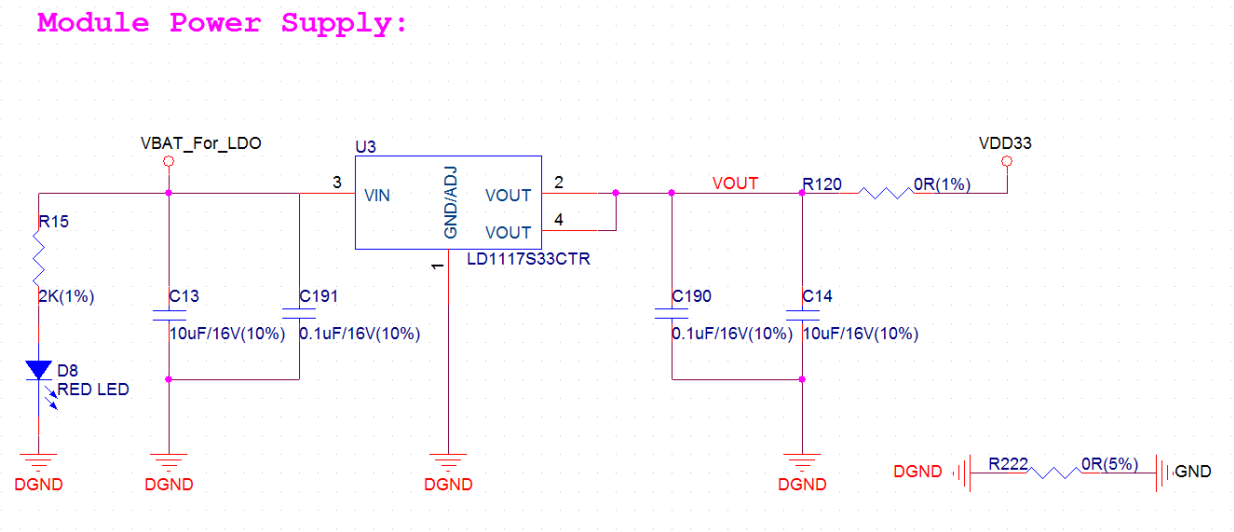


Fig. 3.7: ESP32 LyraT V4.3 - Dedicated LDO for the ESP32 Module

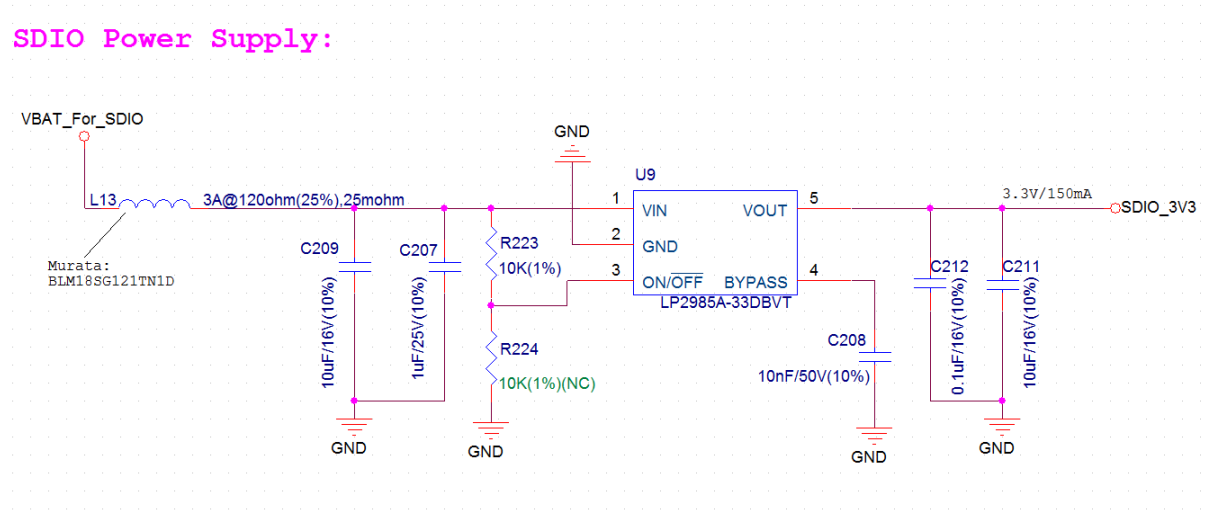


Fig. 3.8: ESP32 LyraT V4.3 - Dedicated LDO for the MicroSD Card

development board adds a set of LC circuits at the front of the PA power supply, where L uses 1.5A magnetic beads and C uses 10uF aluminum electrolytic capacitors, to effectively filter out power crosstalk.

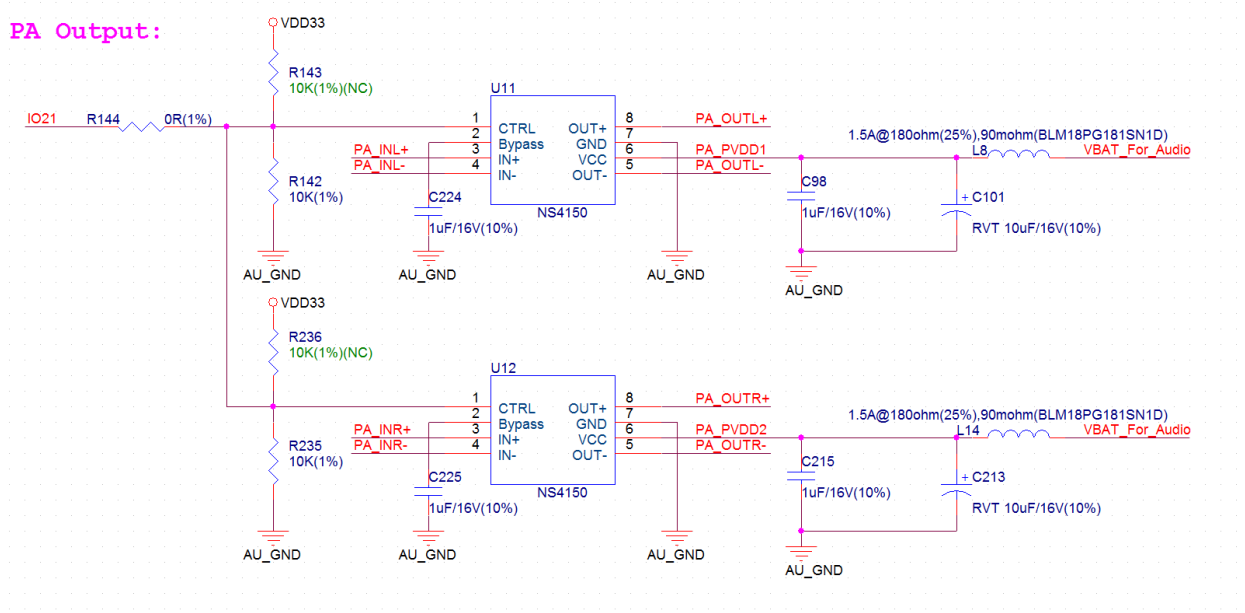


Fig. 3.9: ESP32 LyraT V4.3 - Power Supply for the PAs

Selecting of the Audio Output

The development board uses two mono Class D amplifier ICs, model number NS4150 with maximum output power of 3W and operating voltage from 3.0V to 5.25V.

The audio input source is the digital-to-analog converter (DAC) output of the ES8388. Audio output supports two external speakers.

An optional audio output is a pair of headphones feed from the same DACs as the amplifier ICs.

To switch between using headphones and speakers, the board provides a digital input signal to detect when a headphone jack is inserted and a digital output signal to enable or disable the amplifier ICs. In other words selection between speakers and headphones is under software control instead of using mechanical contacts that would disconnect speakers once a headphone jack is inserted.

Other Versions of LyraT

- *ESP32-LyraT V4.2 Getting Started Guide*
- *ESP32-LyraT V4 Getting Started Guide*

Related Documents

- ESP32 LyraT V4.3 schematic (PDF)
- *ESP32-LyraT V4.3 Getting Started Guide*
- ESP32 Datasheet (PDF)

- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging

3.5 Audio Samples

Music files in this section are intended for testing of audio applications. The files are organized into different *Formats* and *Sample Rates*.

3.5.1 Formats

The tables below provides an audio file converted from 'wav' format into several other audio formats.

Long Samples

The audio track duration in this section is 3 minutes and 7 seconds.

Two Channel Audio

No	Format	Audio File	Size [kB]
1	aac	ff-16b-2c-44100hz.aac	2,995
2	ac3	ff-16b-2c-44100hz.ac3	2,994
3	aiff	ff-16b-2c-44100hz.aiff	33,002
4	flac	ff-16b-2c-44100hz.flac	22,406
5	m4a	ff-16b-2c-44100hz.m4a	3,028
6	mp3	ff-16b-2c-44100hz.mp3	2,994
7	mp4	ff-16b-2c-44100hz.mp4	3,079
8	ogg	ff-16b-2c-44100hz.ogg	2,612
9	opus	ff-16b-2c-44100hz.opus	2,598
10	ts	ff-16b-2c-44100hz.ts	5,510
11	wav	ff-16b-2c-44100hz.wav	49,504
12	wma	ff-16b-2c-44100hz.wma	3,227

Playlist containing all above files: ff-16b-2c-playlist.m3u

Single Channel Audio

No	Format	Audio File	Size [kB]
1	aac	ff-16b-1c-44100hz.aac	1,650
2	ac3	ff-16b-1c-44100hz.ac3	2,193
3	aiff	ff-16b-1c-44100hz.aiff	16,115
4	amr	ff-16b-1c-8000hz.amr	299
5	flac	ff-16b-1c-44100hz.flac	10,655
6	m4a	ff-16b-1c-44100hz.m4a	1,628
7	mp3	ff-16b-1c-44100hz.mp3	1,463
8	ogg	ff-16b-1c-44100hz.ogg	1,558
9	opus	ff-16b-1c-44100hz.opus	1,641
10	wav	ff-16b-1c-44100hz.wav	16,115
11	wma	ff-16b-1c-44100hz.wma	3,151

Playlist containing all above files: ff-16b-1c-playlist.m3u

Short Samples

If you need shorter audio files for testing, this section provides 16 seconds audio tracks.

Two Channel Audio

No	Format	Audio File	Size [kB]
1	aac	gs-16b-2c-44100hz.aac	241
2	ac3	gs-16b-2c-44100hz.ac3	380
3	aiff	gs-16b-2c-44100hz.aiff	2,792
4	flac	gs-16b-2c-44100hz.flac	1,336
5	m4a	gs-16b-2c-44100hz.m4a	1,367
6	mp3	gs-16b-2c-44100hz.mp3	254
7	mp4	gs-16b-2c-44100hz.mp4	259
8	ogg	gs-16b-2c-44100hz.ogg	229
9	opus	gs-16b-2c-44100hz.opus	219
10	ts	gs-16b-2c-44100hz.ts	286
11	wav	gs-16b-2c-44100hz.wav	2,792
12	wma	gs-16b-2c-44100hz.wma	276

Playlist containing all above files: gs-16b-2c-playlist.m3u

Single Channel Audio

No	Format	Audio File	Size [kB]
1	amr	gs-16b-1c-8000hz.amr	25
2	aac	gs-16b-1c-44100hz.aac	137
3	ac3	gs-16b-1c-44100hz.ac3	190
4	aiff	gs-16b-1c-44100hz.aiff	1,397
5	flac	gs-16b-1c-44100hz.flac	645
6	m4a	gs-16b-1c-44100hz.m4a	650
7	mp3	gs-16b-1c-44100hz.mp3	127
8	ogg	gs-16b-1c-44100hz.ogg	144
9	opus	gs-16b-1c-44100hz.opus	132
10	wav	gs-16b-1c-44100hz.wav	1,497
11	wma	gs-16b-1c-44100hz.wma	276

Playlist containing all above files: gs-16b-1c-playlist.m3u

3.5.2 Sample Rates

The files in this section have been prepared by converting a single audio file into different sampling rates defined in MPEG Layer III specification. Both mono and stereo versions of files are provided. The bit depth of files is 16 bits.

	Sample Rate	MPEG III	Channels	Bit Rate	Size
Audio File	[Hz]	ver		[kbit/s]	[kB]
ff-16b-1c-8000hz.mp3	8000	2.5	mono	8	183
ff-16b-1c-11025hz.mp3	11025	2.5	mono	16	366
ff-16b-1c-12000hz.mp3	12000	2.5	mono	16	366
ff-16b-1c-16000hz.mp3	16000	2	mono	24	548
ff-16b-1c-22050hz.mp3	22050	2	mono	32	731
ff-16b-1c-24000hz.mp3	24000	2	mono	32	731
ff-16b-1c-32000hz.mp3	32000	1	mono	48	1,097
ff-16b-1c-44100hz.mp3	44100	1	mono	64	1,462
ff-16b-2c-8000hz.mp3	8000	2.5	joint stereo	24	549
ff-16b-2c-11025hz.mp3	11025	2.5	joint stereo	32	731
ff-16b-2c-12000hz.mp3	12000	2.5	joint stereo	32	731
ff-16b-2c-16000hz.mp3	16000	2	joint stereo	48	1,097
ff-16b-2c-22050hz.mp3	22050	2	joint stereo	64	1,462
ff-16b-2c-24000hz.mp3	24000	2	joint stereo	64	1,462
ff-16b-2c-32000hz.mp3	32000	1	joint stereo	96	2,194
ff-16b-2c-44100hz.mp3	44100	1	joint stereo	128	2,924

Playlist containing all above files: ff-16b-mp3-playlist.m3u

Original music files: “Furious Freak” and “Galway”, Kevin MacLeod (incompetech.com), Licensed under Creative Commons: By Attribution 3.0, http://creativecommons.org/licenses/by/3.0/

CHAPTER 4

Resources

- The esp32.com forum is a place to ask questions and find community resources. The forum has a section dedicated to ESP-ADF.
- This ESP Audio Development Framework inherits from ESP IoT Development Framework and you can learn about it in ESP-IDF Programming Guide.
- Check the Issues section on GitHub if you find a bug or have a feature request. Please check existing Issues before opening a new one.
- If you're interested in contributing to ESP Audio Development Framework, please check the Contributions Guide.
- Several books have been written about ESP32 and they are listed on Espressif web site.
- For additional ESP32 product related information, please refer to documentation section of Espressif site.
- Where to buy audio development boards produced by Espressif:
  - ESP32-LyraT - Espressif Official Sample Provider,
  - ESP32-LyraTD MSC - Espressif Official Sample Provider,

CHAPTER 5

Copyrights and Licenses

5.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2018 Espressif Systems. This source code is licensed under the ESPRESSIF MIT License as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses:

- esp-stagefright is Copyright (c) 2005-2008, The Android Open Source Project, and is licensed under the Apache License Version 2.0.

Please refer to the COPYRIGHT in ESP-IDF Programming Guide

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

CHAPTER 6

About

This is documentation of ESP-ADF, the framework to develop audio applications for ESP32 chip by Espressif.

The **ESP32** is 2.4 GHz Wi-Fi and Bluetooth combo, 32 bit dual core chip running up to 240 MHz, designed for mobile, wearable electronics, and Internet-of-Things (IoT) applications. It has several peripherals on board including I2S interfaces to easy integrate with dedicated audio chips. These hardware features together with the ESP-ADF software provide a powerful platform to implement audio applications including native wireless networking and powerful user interface.

The **ESP-ADF** provides a range of API components including **Audio Streams**, **Codecs** and **Services** organized in **Audio Pipeline**, all integrated with audio hardware through **Media HAL** and with **Peripherals** onboard of **ESP32**.

The ESP-ADF also provides integration with **Baidu DauerOS** cloud services. A range of components is coming to provide integration with DeepBrain, Amazon, Google, Alibaba and Turing cloud services.

The **ESP-ADF** builds on well established, FreeRTOS based, Espressif IOT Development Framework ESP-IDF.

- genindex

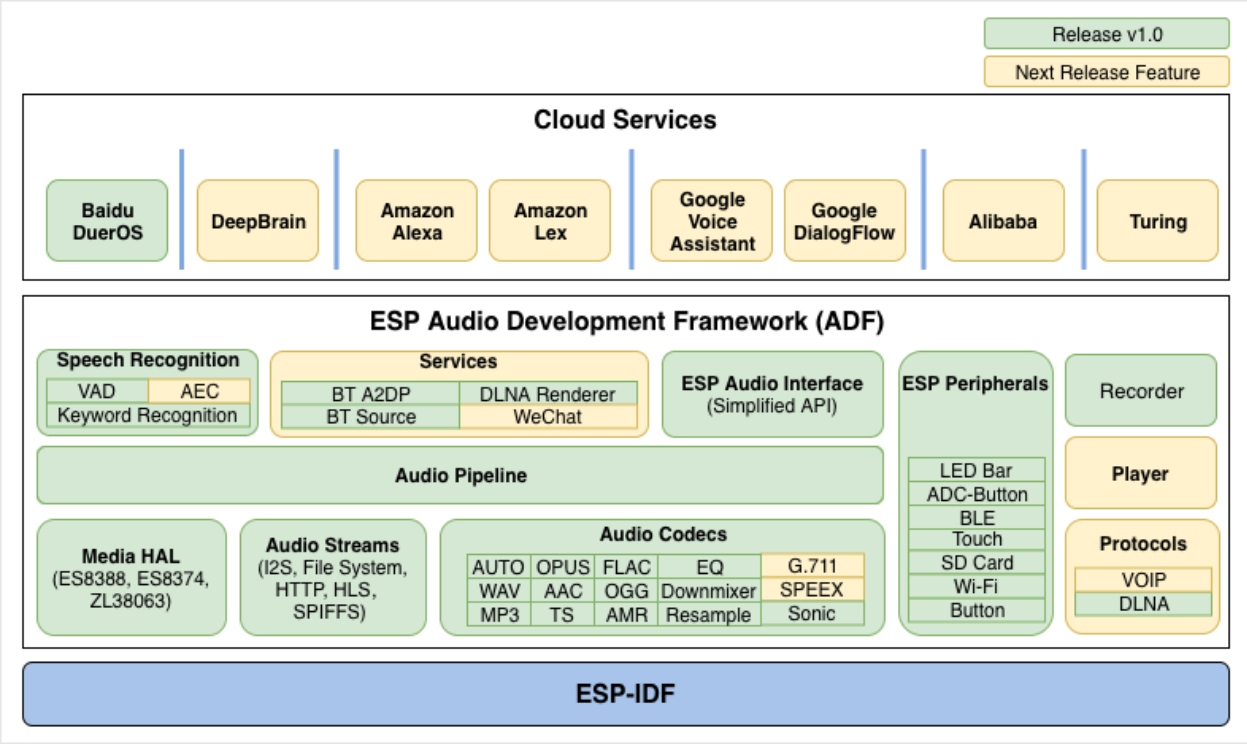


Fig. 6.1: Espressif Audio Development Framework

Index

A

aac_decoder_cfg_t (C++ class), 79
aac_decoder_cfg_t::out_rb_size (C++ member), 79
aac_decoder_cfg_t::task_core (C++ member), 79
aac_decoder_cfg_t::task_prio (C++ member), 79
aac_decoder_cfg_t::task_stack (C++ member), 79
aac_decoder_init (C++ function), 78
AAC_DECODER_RINGBUFFER_SIZE (C macro), 79
AAC_DECODER_TASK_CORE (C macro), 79
AAC_DECODER_TASK_PRIO (C macro), 79
AAC_DECODER_TASK_STACK_SIZE (C macro), 79
ADC_DEFAULT_ARR (C macro), 117
ADC_INPUT_DIFFERENCE (C++ enumerator), 135
ADC_INPUT_LINPUT1_RINPUT1 (C++ enumerator), 135
ADC_INPUT_LINPUT2_RINPUT2 (C++ enumerator), 135
ADC_INPUT_MAX (C++ enumerator), 135
ADC_INPUT_MIC1 (C++ enumerator), 135
ADC_INPUT_MIC2 (C++ enumerator), 135
ADC_INPUT_MIN (C++ enumerator), 135
AEL_IO_ABORT (C++ enumerator), 47
AEL_IO_DONE (C++ enumerator), 47
AEL_IO_FAIL (C++ enumerator), 47
AEL_IO_OK (C++ enumerator), 47
AEL_IO_TIMEOUT (C++ enumerator), 47
AEL_MSG_CMD_DESTROY (C++ enumerator), 48
AEL_MSG_CMD_ERROR (C++ enumerator), 48
AEL_MSG_CMD_FINISH (C++ enumerator), 48
AEL_MSG_CMD_NONE (C++ enumerator), 48
AEL_MSG_CMD_PAUSE (C++ enumerator), 48
AEL_MSG_CMD_REPORT_CODEC_FMT (C++ enumerator), 48
AEL_MSG_CMD_REPORT_MUSIC_INFO (C++ enumerator), 48
AEL_MSG_CMD_REPORT_STATUS (C++ enumerator), 48
AEL_MSG_CMD_RESUME (C++ enumerator), 48
AEL_MSG_CMD_STOP (C++ enumerator), 48
AEL_PROCESS_FAIL (C++ enumerator), 47
AEL_STATE_ERROR (C++ enumerator), 48
AEL_STATE_FINISHED (C++ enumerator), 48
AEL_STATE_INIT (C++ enumerator), 48
AEL_STATE_NONE (C++ enumerator), 48
AEL_STATE_PAUSED (C++ enumerator), 48
AEL_STATE_RUNNING (C++ enumerator), 48
AEL_STATE_STOPPED (C++ enumerator), 48
AEL_STATUS_ERROR_CLOSE (C++ enumerator), 48
AEL_STATUS_ERROR_INPUT (C++ enumerator), 48
AEL_STATUS_ERROR_OPEN (C++ enumerator), 48
AEL_STATUS_ERROR_OUTPUT (C++ enumerator), 48
AEL_STATUS_ERROR_PROCESS (C++ enumerator), 48
AEL_STATUS_ERROR_TIMEOUT (C++ enumerator), 48
AEL_STATUS_ERROR_UNKNOWN (C++ enumerator), 48
AEL_STATUS_INPUT_BUFFERING (C++ enumerator), 48
AEL_STATUS_INPUT_DONE (C++ enumerator), 48
AEL_STATUS_MOUNTED (C++ enumerator), 49
AEL_STATUS_NONE (C++ enumerator), 48
AEL_STATUS_OUTPUT_BUFFERING (C++ enumerator), 49
AEL_STATUS_OUTPUT_DONE (C++ enumerator), 48
AEL_STATUS_STATE_PAUSED (C++ enumerator), 49
AEL_STATUS_STATE_RUNNING (C++ enumerator), 49
AEL_STATUS_STATE_STOPPED (C++ enumerator), 49
AEL_STATUS_UNMOUNTED (C++ enumerator), 49
amr_decoder_cfg_t (C++ class), 80
amr_decoder_cfg_t::out_rb_size (C++ member), 80
amr_decoder_cfg_t::task_core (C++ member), 80
amr_decoder_cfg_t::task_prio (C++ member), 80
amr_decoder_cfg_t::task_stack (C++ member), 80
amr_decoder_init (C++ function), 80
AMR_DECODER_RINGBUFFER_SIZE (C macro), 80

AMR_DECODER_TASK_CORE (C macro), 80
AMR_DECODER_TASK_PRIO (C macro), 80
AMR_DECODER_TASK_STACK_SIZE (C macro), 80
amrnb_encoder_cfg_t (C++ class), 81
amrnb_encoder_cfg_t::out_rb_size (C++ member), 81
amrnb_encoder_cfg_t::task_core (C++ member), 81
amrnb_encoder_cfg_t::task_prio (C++ member), 81
amrnb_encoder_cfg_t::task_stack (C++ member), 81
amrnb_encoder_init (C++ function), 80
AMRNB_ENCODER_RINGBUFFER_SIZE (C macro), 81
AMRNB_ENCODER_TASK_CORE (C macro), 81
AMRNB_ENCODER_TASK_PRIO (C macro), 81
AMRNB_ENCODER_TASK_STACK (C macro), 81
amrwb_encoder_cfg_t (C++ class), 82
amrwb_encoder_cfg_t::out_rb_size (C++ member), 82
amrwb_encoder_cfg_t::task_core (C++ member), 82
amrwb_encoder_cfg_t::task_prio (C++ member), 82
amrwb_encoder_cfg_t::task_stack (C++ member), 82
amrwb_encoder_init (C++ function), 81
AMRWB_ENCODER_RINGBUFFER_SIZE (C macro), 82
AMRWB_ENCODER_TASK_CORE (C macro), 82
AMRWB_ENCODER_TASK_PRIO (C macro), 82
AMRWB_ENCODER_TASK_STACK (C macro), 82
AUDIO_CODEC_AAC (C++ enumerator), 62
AUDIO_CODEC_MP3 (C++ enumerator), 62
AUDIO_CODEC_NONE (C++ enumerator), 62
AUDIO_CODEC_OPUS (C++ enumerator), 62
AUDIO_CODEC_RAW (C++ enumerator), 62
audio_codec_t (C++ type), 62
AUDIO_CODEC_TYPE_DECODER (C++ enumerator), 62
AUDIO_CODEC_TYPE_ENCODER (C++ enumerator), 62
AUDIO_CODEC_TYPE_NONE (C++ enumerator), 62
audio_codec_type_t (C++ type), 62
AUDIO_CODEC_WAV (C++ enumerator), 62
audio_element_abort_input_ringbuf (C++ function), 39
audio_element_abort_output_ringbuf (C++ function), 39
audio_element_cfg_t (C++ class), 46
audio_element_cfg_t::buffer_len (C++ member), 46
audio_element_cfg_t::close (C++ member), 46
audio_element_cfg_t::data (C++ member), 47
audio_element_cfg_t::destroy (C++ member), 46
audio_element_cfg_t::enable_multi_io (C++ member), 47
audio_element_cfg_t::open (C++ member), 46
audio_element_cfg_t::out_rb_size (C++ member), 47
audio_element_cfg_t::process (C++ member), 46
audio_element_cfg_t::read (C++ member), 46
audio_element_cfg_t::seek (C++ member), 46
audio_element_cfg_t::tag (C++ member), 47
audio_element_cfg_t::task_core (C++ member), 47
audio_element_cfg_t::task_prio (C++ member), 46
audio_element_cfg_t::task_stack (C++ member), 46
audio_element_cfg_t::write (C++ member), 46
audio_element_deinit (C++ function), 34
audio_element_err_t (C++ type), 47
audio_element_get_event_queue (C++ function), 43
audio_element_get_input_ringbuf (C++ function), 38
audio_element_get_multi_input_ringbuf (C++ function), 45
audio_element_get_multi_output_ringbuf (C++ function), 45
audio_element_get_output_ringbuf (C++ function), 39
audio_element_get_output_ringbuf_size (C++ function), 43
audio_element_get_state (C++ function), 39
audio_element_get_tag (C++ function), 35
audio_element_get_uri (C++ function), 36
audio_element_getdata (C++ function), 35
audio_element_getinfo (C++ function), 36
audio_element_handle_t (C++ type), 47
AUDIO_ELEMENT_INFO_DEFAULT (C macro), 47
audio_element_info_t (C++ class), 45
audio_element_info_t::bits (C++ member), 46
audio_element_info_t::byte_pos (C++ member), 46
audio_element_info_t::channels (C++ member), 45
audio_element_info_t::codec_fmt (C++ member), 46
audio_element_info_t::mute (C++ member), 46
audio_element_info_t::reserve_data (C++ member), 46
audio_element_info_t::sample_rates (C++ member), 45
audio_element_info_t::total_bytes (C++ member), 46
audio_element_info_t::uri (C++ member), 46
audio_element_info_t::volume (C++ member), 46
audio_element_init (C++ function), 34
audio_element_input (C++ function), 41
audio_element_msg_cmd_t (C++ type), 48
audio_element_msg_remove_listener (C++ function), 38
audio_element_msg_set_listener (C++ function), 38
audio_element_multi_input (C++ function), 43
audio_element_multi_output (C++ function), 44
audio_element_output (C++ function), 42
audio_element_pause (C++ function), 37
audio_element_report_codec_fmt (C++ function), 40
audio_element_report_info (C++ function), 40
audio_element_report_status (C++ function), 40
audio_element_reserve_data_t (C++ class), 45
audio_element_reserve_data_t::user_data_0 (C++ member), 45
audio_element_reserve_data_t::user_data_1 (C++ member), 45
audio_element_reset_input_ringbuf (C++ function), 41
audio_element_reset_output_ringbuf (C++ function), 41
audio_element_reset_state (C++ function), 43
audio_element_resume (C++ function), 37
audio_element_run (C++ function), 36

audio_element_set_input_ringbuf (C++ function), 38
audio_element_set_input_timeout (C++ function), 40
audio_element_set_multi_input_ringbuf (C++ function), 44
audio_element_set_multi_output_ringbuf (C++ function), 44
audio_element_set_output_ringbuf (C++ function), 38
audio_element_set_output_timeout (C++ function), 41
audio_element_set_read_cb (C++ function), 42
audio_element_set_ringbuf_done (C++ function), 43
audio_element_set_tag (C++ function), 35
audio_element_set_uri (C++ function), 36
audio_element_set_write_cb (C++ function), 42
audio_element_setdata (C++ function), 35
audio_element_setinfo (C++ function), 35
audio_element_state_t (C++ type), 48
audio_element_status_t (C++ type), 48
audio_element_stop (C++ function), 37
audio_element_terminate (C++ function), 36
AUDIO_ELEMENT_TYPE_ELEMENT (C++ enumerator), 62
AUDIO_ELEMENT_TYPE_PERIPH (C++ enumerator), 62
AUDIO_ELEMENT_TYPE_PLAYER (C++ enumerator), 62
AUDIO_ELEMENT_TYPE_SERVICE (C++ enumerator), 62
audio_element_type_t (C++ type), 62
AUDIO_ELEMENT_TYPE_UNKNOW (C++ enumerator), 62
audio_element_wait_for_buffer (C++ function), 39
audio_element_wait_for_stop (C++ function), 37
audio_err_t (C++ type), 63
audio_event_iface_cfg_t (C++ class), 61
audio_event_iface_cfg_t::context (C++ member), 61
audio_event_iface_cfg_t::external_queue_size (C++ member), 61
audio_event_iface_cfg_t::internal_queue_size (C++ member), 61
audio_event_iface_cfg_t::on_cmd (C++ member), 61
audio_event_iface_cfg_t::queue_set_size (C++ member), 61
audio_event_iface_cfg_t::type (C++ member), 61
audio_event_iface_cfg_t::wait_time (C++ member), 61
audio_event_iface_cmd (C++ function), 58
audio_event_iface_cmd_from_isr (C++ function), 58
AUDIO_EVENT_IFACE_DEFAULT_CFG (C macro), 61
audio_event_iface_destroy (C++ function), 57
audio_event_iface_discard (C++ function), 59
audio_event_iface_get_msg_queue_handle (C++ function), 60
audio_event_iface_get_queue_handle (C++ function), 59
audio_event_iface_handle_t (C++ type), 61
audio_event_iface_init (C++ function), 57
audio_event_iface_listen (C++ function), 59
audio_event_iface_msg_t (C++ class), 60
audio_event_iface_msg_t::cmd (C++ member), 60
audio_event_iface_msg_t::data (C++ member), 60
audio_event_iface_msg_t::data_len (C++ member), 60
audio_event_iface_msg_t::need_free_data (C++ member), 60
audio_event_iface_msg_t::source (C++ member), 60
audio_event_iface_msg_t::source_type (C++ member), 60
audio_event_iface_read (C++ function), 59
audio_event_iface_remove_listener (C++ function), 57
audio_event_iface_sendout (C++ function), 59
audio_event_iface_set_cmd_waiting_timeout (C++ function), 58
audio_event_iface_set_listener (C++ function), 57
audio_event_iface_set_msg_listener (C++ function), 60
audio_event_iface_waiting_cmd_msg (C++ function), 58
AUDIO_HAL_08K_SAMPLES (C++ enumerator), 126
AUDIO_HAL_11K_SAMPLES (C++ enumerator), 126
AUDIO_HAL_16K_SAMPLES (C++ enumerator), 126
AUDIO_HAL_22K_SAMPLES (C++ enumerator), 126
AUDIO_HAL_24K_SAMPLES (C++ enumerator), 126
AUDIO_HAL_32K_SAMPLES (C++ enumerator), 127
AUDIO_HAL_44K_SAMPLES (C++ enumerator), 127
AUDIO_HAL_48K_SAMPLES (C++ enumerator), 127
AUDIO_HAL_ADC_INPUT_ALL (C++ enumerator), 126
AUDIO_HAL_ADC_INPUT_DIFFERENCE (C++ enumerator), 126
AUDIO_HAL_ADC_INPUT_LINE1 (C++ enumerator), 125
AUDIO_HAL_ADC_INPUT_LINE2 (C++ enumerator), 126
audio_hal_adc_input_t (C++ type), 125
AUDIO_HAL_BIT_LENGTH_16BITS (C++ enumerator), 127
AUDIO_HAL_BIT_LENGTH_24BITS (C++ enumerator), 127
AUDIO_HAL_BIT_LENGTH_32BITS (C++ enumerator), 127
audio_hal_codec_config_t (C++ class), 125
audio_hal_codec_config_t::adc_input (C++ member), 125
audio_hal_codec_config_t::codec_mode (C++ member), 125
audio_hal_codec_config_t::dac_output (C++ member), 125
audio_hal_codec_config_t::i2s_iface (C++ member), 125
audio_hal_codec_i2s_iface_t (C++ class), 124
audio_hal_codec_i2s_iface_t::bits (C++ member), 124
audio_hal_codec_i2s_iface_t::fmt (C++ member), 124
audio_hal_codec_i2s_iface_t::mode (C++ member), 124

audio_hal_codec_i2s_iface_t::samples (C++ member), 124
audio_hal_codec_iface_config (C++ function), 123
AUDIO_HAL_CODEC_MODE_BOTH (C++ enumerator), 125
AUDIO_HAL_CODEC_MODE_DECODE (C++ enumerator), 125
AUDIO_HAL_CODEC_MODE_ENCODE (C++ enumerator), 125
AUDIO_HAL_CODEC_MODE_LINE_IN (C++ enumerator), 125
audio_hal_codec_mode_t (C++ type), 125
audio_hal_ctrl_codec (C++ function), 123
AUDIO_HAL_CTRL_START (C++ enumerator), 126
AUDIO_HAL_CTRL_STOP (C++ enumerator), 126
audio_hal_ctrl_t (C++ type), 126
AUDIO_HAL_DAC_OUTPUT_ALL (C++ enumerator), 126
AUDIO_HAL_DAC_OUTPUT_LINE1 (C++ enumerator), 126
AUDIO_HAL_DAC_OUTPUT_LINE2 (C++ enumerator), 126
audio_hal_dac_output_t (C++ type), 126
audio_hal_deinit (C++ function), 123
AUDIO_HAL_ES8374_DEFAULT (C macro), 125
AUDIO_HAL_ES8388_DEFAULT (C macro), 125
audio_hal_get_volume (C++ function), 124
audio_hal_handle_t (C++ type), 125
AUDIO_HAL_I2S_DSP (C++ enumerator), 127
AUDIO_HAL_I2S_LEFT (C++ enumerator), 127
AUDIO_HAL_I2S_NORMAL (C++ enumerator), 127
AUDIO_HAL_I2S_RIGHT (C++ enumerator), 127
audio_hal_iface_bits_t (C++ type), 127
audio_hal_iface_format_t (C++ type), 127
audio_hal_iface_mode_t (C++ type), 126
audio_hal_iface_samples_t (C++ type), 126
audio_hal_init (C++ function), 123
AUDIO_HAL_MODE_MASTER (C++ enumerator), 126
AUDIO_HAL_MODE_SLAVE (C++ enumerator), 126
audio_hal_set_volume (C++ function), 124
AUDIO_HAL_VOL_DEFAULT (C macro), 125
AUDIO_HAL_ZL38063_DEFAULT (C macro), 143
audio_pipeline_breakup_elements (C++ function), 55
audio_pipeline_cfg (C++ class), 56
audio_pipeline_cfg::rb_size (C++ member), 56
audio_pipeline_cfg_t (C++ type), 56
audio_pipeline_check_items_state (C++ function), 55
audio_pipeline_deinit (C++ function), 50
audio_pipeline_get_el_by_tag (C++ function), 52
audio_pipeline_get_event_iface (C++ function), 53
audio_pipeline_handle_t (C++ type), 56
audio_pipeline_init (C++ function), 49
audio_pipeline_link (C++ function), 52
audio_pipeline_link_insert (C++ function), 53
audio_pipeline_link_more (C++ function), 54
audio_pipeline_listen_more (C++ function), 54
audio_pipeline_pause (C++ function), 51
audio_pipeline_register (C++ function), 50
audio_pipeline_register_more (C++ function), 53
audio_pipeline_relink (C++ function), 55
audio_pipeline_remove_listener (C++ function), 52
audio_pipeline_reset_items_state (C++ function), 55
audio_pipeline_reset_ringbuffer (C++ function), 55
audio_pipeline_resume (C++ function), 51
audio_pipeline_run (C++ function), 50
audio_pipeline_set_listener (C++ function), 53
audio_pipeline_stop (C++ function), 51
audio_pipeline_terminate (C++ function), 50
audio_pipeline_unlink (C++ function), 52
audio_pipeline_unregister (C++ function), 50
audio_pipeline_unregister_more (C++ function), 54
audio_pipeline_wait_for_stop (C++ function), 51
AUDIO_PLAYLIST (C++ enumerator), 62
AUDIO_STATUS_ERROR (C++ enumerator), 64
AUDIO_STATUS_PAUSED (C++ enumerator), 64
AUDIO_STATUS_RUNNING (C++ enumerator), 64
AUDIO_STATUS_STOPED (C++ enumerator), 64
AUDIO_STATUS_UNKNOWN (C++ enumerator), 64
AUDIO_STREAM_NONE (C++ enumerator), 62
AUDIO_STREAM_READER (C++ enumerator), 62
audio_stream_type_t (C++ type), 62
AUDIO_STREAM_WRITER (C++ enumerator), 62
audio_termination_type_t (C++ type), 64

B

BIT_LENGTH_16BITS (C++ enumerator), 133
BIT_LENGTH_18BITS (C++ enumerator), 133
BIT_LENGTH_20BITS (C++ enumerator), 133
BIT_LENGTH_24BITS (C++ enumerator), 133
BIT_LENGTH_32BITS (C++ enumerator), 133
BIT_LENGTH_MAX (C++ enumerator), 133
BIT_LENGTH_MIN (C++ enumerator), 133
BLUETOOTH_A2DP_SINK (C++ enumerator), 94
BLUETOOTH_A2DP_SOURCE (C++ enumerator), 94
BLUETOOTH_ADDR_LEN (C macro), 94
bluetooth_addr_t (C++ type), 94
bluetooth_service_cfg_t (C++ class), 94
bluetooth_service_cfg_t::device_name (C++ member), 94
bluetooth_service_cfg_t::mode (C++ member), 94
bluetooth_service_cfg_t::remote_name (C++ member), 94
bluetooth_service_create_periph (C++ function), 91
bluetooth_service_create_stream (C++ function), 91
bluetooth_service_destroy (C++ function), 94
bluetooth_service_mode_t (C++ type), 94
bluetooth_service_start (C++ function), 91

C

console_cmd_callback_t (C++ type), 111
CONSOLE_DEFAULT_PROMPT_STRING (C macro), 111
CONSOLE_DEFAULT_TASK_PRIO (C macro), 111
CONSOLE_DEFAULT_TASK_STACK (C macro), 111

D

D2SE_PGA_GAIN_DIS (C++ enumerator), 135
D2SE_PGA_GAIN_EN (C++ enumerator), 135
D2SE_PGA_GAIN_MAX (C++ enumerator), 135
D2SE_PGA_GAIN_MIN (C++ enumerator), 135
DAC_OUTPUT_ALL (C++ enumerator), 136
DAC_OUTPUT_LOUT1 (C++ enumerator), 136
DAC_OUTPUT_LOUT2 (C++ enumerator), 136
DAC_OUTPUT_MAX (C++ enumerator), 136
DAC_OUTPUT_MIN (C++ enumerator), 136
DAC_OUTPUT_ROUT1 (C++ enumerator), 136
DAC_OUTPUT_ROUT2 (C++ enumerator), 136
DAC_OUTPUT_SPK (C++ enumerator), 136
decoder_opus_init (C++ function), 86
DEFAULT_AAC_DECODER_CONFIG (C macro), 79
DEFAULT_AMR_DECODER_CONFIG (C macro), 80
DEFAULT_AMRNB_ENCODER_CONFIG (C macro), 81
DEFAULT_AMRWB_ENCODER_CONFIG (C macro), 82
DEFAULT_AUDIO_ELEMENT_CONFIG (C macro), 47
DEFAULT_AUDIO_EVENT_IFACE_SIZE (C macro), 61
DEFAULT_AUDIO_PIPELINE_CONFIG (C macro), 56
DEFAULT_ELEMENT_BUFFER_LENGTH (C macro), 47
DEFAULT_ELEMENT_RINGBUF_SIZE (C macro), 47
DEFAULT_ELEMENT_STACK_SIZE (C macro), 47
DEFAULT_ELEMENT_TASK_CORE (C macro), 47
DEFAULT_ELEMENT_TASK_PRIO (C macro), 47
DEFAULT_FLAC_DECODER_CONFIG (C macro), 83
DEFAULT_MP3_DECODER_CONFIG (C macro), 84
DEFAULT_OGG_DECODER_CONFIG (C macro), 85
DEFAULT_OPUS_DECODER_CONFIG (C macro), 86
DEFAULT_PIPELINE_RINGBUF_SIZE (C macro), 56
DEFAULT_RESAMPLE_FILTER_CONFIG (C macro), 90
DEFAULT_WAV_DECODER_CONFIG (C macro), 87
DEFAULT_WAV_ENCODER_CONFIG (C macro), 88

E

ELEMENT_SUB_TYPE_OFFSET (C macro), 62
ES8374_ADDR (C macro), 141
es8374_config_adc_input (C++ function), 139
es8374_config_dac_output (C++ function), 140
es8374_config_fmt (C++ function), 137
es8374_config_i2s (C++ function), 140
es8374_ctrl_state (C++ function), 140
es8374_deinit (C++ function), 137
es8374_get_voice_mute (C++ function), 139
es8374_get_voice_volume (C++ function), 139
es8374_i2s_config_clock (C++ function), 138
es8374_init (C++ function), 137
es8374_pa_power (C++ function), 141
es8374_read_all (C++ function), 140
es8374_set_bits_per_sample (C++ function), 138
es8374_set_mic_gain (C++ function), 139
es8374_set_voice_mute (C++ function), 139
es8374_set_voice_volume (C++ function), 138
es8374_start (C++ function), 138
es8374_stop (C++ function), 138
es8374_write_reg (C++ function), 140
ES8388_ADCCONTROL1 (C macro), 132
ES8388_ADCCONTROL10 (C macro), 132
ES8388_ADCCONTROL11 (C macro), 132
ES8388_ADCCONTROL12 (C macro), 132
ES8388_ADCCONTROL13 (C macro), 132
ES8388_ADCCONTROL14 (C macro), 132
ES8388_ADCCONTROL2 (C macro), 132
ES8388_ADCCONTROL3 (C macro), 132
ES8388_ADCCONTROL4 (C macro), 132
ES8388_ADCCONTROL5 (C macro), 132
ES8388_ADCCONTROL6 (C macro), 132
ES8388_ADCCONTROL7 (C macro), 132
ES8388_ADCCONTROL8 (C macro), 132
ES8388_ADCCONTROL9 (C macro), 132
ES8388_ADCPOWER (C macro), 132
ES8388_ADDR (C macro), 132
ES8388_ANAVOLMANAG (C macro), 132
ES8388_CHIPLOPOW1 (C macro), 132
ES8388_CHIPLOPOW2 (C macro), 132
ES8388_CHIPPOWER (C macro), 132
es8388_config_adc_input (C++ function), 130
es8388_config_dac_output (C++ function), 130
es8388_config_fmt (C++ function), 128
es8388_config_i2s (C++ function), 131
ES8388_CONTROL1 (C macro), 132
ES8388_CONTROL2 (C macro), 132
es8388_ctrl_state (C++ function), 131
ES8388_DACCONTROL1 (C macro), 132
ES8388_DACCONTROL10 (C macro), 132
ES8388_DACCONTROL11 (C macro), 132
ES8388_DACCONTROL12 (C macro), 133
ES8388_DACCONTROL13 (C macro), 133
ES8388_DACCONTROL14 (C macro), 133
ES8388_DACCONTROL15 (C macro), 133
ES8388_DACCONTROL16 (C macro), 133
ES8388_DACCONTROL17 (C macro), 133
ES8388_DACCONTROL18 (C macro), 133

ES8388_DACCONTROL19 (C macro), 133
ES8388_DACCONTROL2 (C macro), 132
ES8388_DACCONTROL20 (C macro), 133
ES8388_DACCONTROL21 (C macro), 133
ES8388_DACCONTROL22 (C macro), 133
ES8388_DACCONTROL23 (C macro), 133
ES8388_DACCONTROL24 (C macro), 133
ES8388_DACCONTROL25 (C macro), 133
ES8388_DACCONTROL26 (C macro), 133
ES8388_DACCONTROL27 (C macro), 133
ES8388_DACCONTROL28 (C macro), 133
ES8388_DACCONTROL29 (C macro), 133
ES8388_DACCONTROL3 (C macro), 132
ES8388_DACCONTROL30 (C macro), 133
ES8388_DACCONTROL4 (C macro), 132
ES8388_DACCONTROL5 (C macro), 132
ES8388_DACCONTROL6 (C macro), 132
ES8388_DACCONTROL7 (C macro), 132
ES8388_DACCONTROL8 (C macro), 132
ES8388_DACCONTROL9 (C macro), 132
ES8388_DACPOWER (C macro), 132
es8388_deinit (C++ function), 128
es8388_get_voice_mute (C++ function), 129
es8388_get_voice_volume (C++ function), 129
es8388_i2s_config_clock (C++ function), 128
es8388_init (C++ function), 127
ES8388_MASTERMODE (C macro), 132
es8388_pa_power (C++ function), 131
es8388_read_all (C++ function), 130
es8388_set_bits_per_sample (C++ function), 128
es8388_set_mic_gain (C++ function), 130
es8388_set_voice_mute (C++ function), 129
es8388_set_voice_volume (C++ function), 129
es8388_start (C++ function), 128
es8388_stop (C++ function), 129
es8388_write_reg (C++ function), 130
es_adc_input_t (C++ type), 135
es_bits_length_t (C++ type), 133
es_d2se_pga_t (C++ type), 135
es_dac_output_t (C++ type), 136
es_i2s_clock_t (C++ class), 131
es_i2s_clock_t::lclk_div (C++ member), 131
es_i2s_clock_t::sclk_div (C++ member), 131
ES_I2S_DSP (C++ enumerator), 137
es_i2s_fmt_t (C++ type), 137
ES_I2S_LEFT (C++ enumerator), 137
ES_I2S_MAX (C++ enumerator), 137
ES_I2S_MIN (C++ enumerator), 137
ES_I2S_NORMAL (C++ enumerator), 137
ES_I2S_RIGHT (C++ enumerator), 137
es_lclk_div_t (C++ type), 134
es_mic_gain_t (C++ type), 136
ES_MODE_MASTER (C++ enumerator), 136
ES_MODE_MAX (C++ enumerator), 136
ES_MODE_MIN (C++ enumerator), 136
ES_MODE_SLAVE (C++ enumerator), 136
es_mode_t (C++ type), 136
ES_MODULE_ADC (C++ enumerator), 136
ES_MODULE_ADC_DAC (C++ enumerator), 136
ES_MODULE_DAC (C++ enumerator), 136
ES_MODULE_LINE (C++ enumerator), 136
ES_MODULE_MAX (C++ enumerator), 136
ES_MODULE_MIN (C++ enumerator), 136
es_module_t (C++ type), 136
es_sclk_div_t (C++ type), 133
esp_audio_cfg_t (C++ class), 69
esp_audio_cfg_t (C++ type), 70
esp_audio_cfg_t::evt_que (C++ member), 70
esp_audio_cfg_t::hal (C++ member), 70
esp_audio_cfg_t::in_stream_buf_size (C++ member), 69
esp_audio_cfg_t::out_stream_buf_size (C++ member), 69
esp_audio_cfg_t::resample_rate (C++ member), 70
esp_audio_cfg_t::task_prio (C++ member), 70
esp_audio_codec_lib_add (C++ function), 65
esp_audio_codec_lib_query (C++ function), 66
esp_audio_create (C++ function), 64
esp_audio_destroy (C++ function), 65
esp_audio_handle_t (C++ type), 70
esp_audio_input_stream_add (C++ function), 65
esp_audio_output_stream_add (C++ function), 65
esp_audio_pause (C++ function), 67
esp_audio_play (C++ function), 66
esp_audio_pos_get (C++ function), 68
esp_audio_resume (C++ function), 67
esp_audio_setup (C++ function), 69
esp_audio_setup_t (C++ class), 70
esp_audio_setup_t (C++ type), 70
esp_audio_setup_t::set_channel (C++ member), 70
esp_audio_setup_t::set_codec (C++ member), 70
esp_audio_setup_t::set_in_stream (C++ member), 70
esp_audio_setup_t::set_out_stream (C++ member), 70
esp_audio_setup_t::set_path (C++ member), 70
esp_audio_setup_t::set_pos (C++ member), 70
esp_audio_setup_t::set_sample_rate (C++ member), 70
esp_audio_setup_t::set_type (C++ member), 70
esp_audio_setup_t::set_uri (C++ member), 70
esp_audio_state_get (C++ function), 68
esp_audio_state_t (C++ class), 63
esp_audio_state_t::err_msg (C++ member), 63
esp_audio_state_t::status (C++ member), 63
esp_audio_status_t (C++ type), 63, 64
esp_audio_stop (C++ function), 67
esp_audio_time_get (C++ function), 69
esp_audio_vol_get (C++ function), 68
esp_audio_vol_set (C++ function), 68
ESP_ERR_AUDIO_ALREADY_EXISTS (C++ enumerator), 64

ESP_ERR_AUDIO_BASE (C macro), 63
ESP_ERR_AUDIO_CLOSE (C++ enumerator), 64
ESP_ERR_AUDIO_FAIL (C++ enumerator), 63
ESP_ERR_AUDIO_HAL_FAIL (C++ enumerator), 63
ESP_ERR_AUDIO_INPUT (C++ enumerator), 64
ESP_ERR_AUDIO_INVALID_PARAMETER (C++ enumerator), 63
ESP_ERR_AUDIO_INVALID_PATH (C++ enumerator), 63
ESP_ERR_AUDIO_INVALID_URI (C++ enumerator), 63
ESP_ERR_AUDIO_MEMORY_LACK (C++ enumerator), 63
ESP_ERR_AUDIO_NO_CODEC (C++ enumerator), 63
ESP_ERR_AUDIO_NO_ERROR (C++ enumerator), 63
ESP_ERR_AUDIO_NO_INPUT_STREAM (C++ enumerator), 63
ESP_ERR_AUDIO_NO_OUTPUT_STREAM (C++ enumerator), 63
ESP_ERR_AUDIO_NOT_READY (C++ enumerator), 63
ESP_ERR_AUDIO_NOT_SUPPORT (C++ enumerator), 64
ESP_ERR_AUDIO_OPEN (C++ enumerator), 64
ESP_ERR_AUDIO_OUTPUT (C++ enumerator), 64
ESP_ERR_AUDIO_PROCESS (C++ enumerator), 64
ESP_ERR_AUDIO_TIMEOUT (C++ enumerator), 64
ESP_ERR_AUDIO_UNKNOWN (C++ enumerator), 64
esp_periph_config_t (C++ class), 102
esp_periph_config_t::event_handle (C++ member), 102
esp_periph_config_t::user_context (C++ member), 102
esp_periph_create (C++ function), 101
esp_periph_destroy (C++ function), 98
esp_periph_event_handle_t (C++ type), 102
esp_periph_func (C++ type), 102
esp_periph_get_by_id (C++ function), 98
esp_periph_get_data (C++ function), 99
esp_periph_get_event_iface (C++ function), 101
esp_periph_get_id (C++ function), 100
esp_periph_get_queue (C++ function), 100
esp_periph_get_state (C++ function), 98
esp_periph_handle_t (C++ type), 102
esp_periph_id_t (C++ type), 102
esp_periph_init (C++ function), 97
esp_periph_run_func (C++ type), 102
esp_periph_send_cmd (C++ function), 99
esp_periph_send_cmd_from_isr (C++ function), 99
esp_periph_send_event (C++ function), 98
esp_periph_set_callback (C++ function), 101
esp_periph_set_data (C++ function), 99
esp_periph_set_function (C++ function), 101
esp_periph_set_id (C++ function), 100
esp_periph_start (C++ function), 97
esp_periph_start_timer (C++ function), 100
esp_periph_state_t (C++ type), 103
esp_periph_stop (C++ function), 97
esp_periph_stop_all (C++ function), 98
esp_periph_stop_timer (C++ function), 100
esp_periph_tick_get (C++ function), 101
esp_touch_pad_sel_t (C++ type), 112

F

FATFS_STREAM_BUF_SIZE (C macro), 76
FATFS_STREAM_CFG_DEFAULT (C macro), 76
fatfs_stream_cfg_t (C++ class), 75
fatfs_stream_cfg_t::buf_sz (C++ member), 75
fatfs_stream_cfg_t::out_rb_size (C++ member), 75
fatfs_stream_cfg_t::task_core (C++ member), 75
fatfs_stream_cfg_t::task_prio (C++ member), 76
fatfs_stream_cfg_t::task_stack (C++ member), 75
fatfs_stream_cfg_t::type (C++ member), 75
fatfs_stream_init (C++ function), 75
FATFS_STREAM_RINGBUFFER_SIZE (C macro), 76
FATFS_STREAM_TASK_CORE (C macro), 76
FATFS_STREAM_TASK_PRIO (C macro), 76
FATFS_STREAM_TASK_STACK (C macro), 76
flac_decoder_cfg_t (C++ class), 83
flac_decoder_cfg_t::out_rb_size (C++ member), 83
flac_decoder_cfg_t::task_core (C++ member), 83
flac_decoder_cfg_t::task_prio (C++ member), 83
flac_decoder_cfg_t::task_stack (C++ member), 83
flac_decoder_init (C++ function), 82
FLAC_DECODER_RINGBUFFER_SIZE (C macro), 83
FLAC_DECODER_TASK_CORE (C macro), 83
FLAC_DECODER_TASK_PRIO (C macro), 83
FLAC_DECODER_TASK_STACK_SIZE (C macro), 83

H

HTTP_STREAM_CFG_DEFAULT (C macro), 74
http_stream_cfg_t (C++ class), 73
http_stream_cfg_t::enable_playlist_parser (C++ member), 74
http_stream_cfg_t::event_handle (C++ member), 74
http_stream_cfg_t::out_rb_size (C++ member), 74
http_stream_cfg_t::task_core (C++ member), 74
http_stream_cfg_t::task_prio (C++ member), 74
http_stream_cfg_t::task_stack (C++ member), 74
http_stream_cfg_t::type (C++ member), 74
http_stream_cfg_t::user_data (C++ member), 74
http_stream_event_handle_t (C++ type), 74
http_stream_event_id_t (C++ type), 74
http_stream_event_msg_t (C++ class), 73
http_stream_event_msg_t::buffer (C++ member), 73
http_stream_event_msg_t::buffer_len (C++ member), 73
http_stream_event_msg_t::el (C++ member), 73
http_stream_event_msg_t::event_id (C++ member), 73
http_stream_event_msg_t::http_client (C++ member), 73
http_stream_event_msg_t::user_data (C++ member), 73

HTTP_STREAM_FINISH_PLAYLIST (C++ enumerator), 75
HTTP_STREAM_FINISH_REQUEST (C++ enumerator), 75
HTTP_STREAM_FINISH_TRACK (C++ enumerator), 75
http_stream_init (C++ function), 73
http_stream_next_track (C++ function), 73
HTTP_STREAM_ON_REQUEST (C++ enumerator), 74
HTTP_STREAM_ON_RESPONSE (C++ enumerator), 74
HTTP_STREAM_POST_REQUEST (C++ enumerator), 75
HTTP_STREAM_PRE_REQUEST (C++ enumerator), 74
HTTP_STREAM_RESOLVE_ALL_TRACKS (C++ enumerator), 75
http_stream_restart (C++ function), 73
HTTP_STREAM_RINGBUFFER_SIZE (C macro), 74
HTTP_STREAM_TASK_CORE (C macro), 74
HTTP_STREAM_TASK_PRIO (C macro), 74
HTTP_STREAM_TASK_STACK (C macro), 74

I

I2S_STREAM_BUF_SIZE (C macro), 72
I2S_STREAM_CFG_DEFAULT (C macro), 72
i2s_stream_cfg_t (C++ class), 72
i2s_stream_cfg_t::i2s_config (C++ member), 72
i2s_stream_cfg_t::i2s_pin_config (C++ member), 72
i2s_stream_cfg_t::i2s_port (C++ member), 72
i2s_stream_cfg_t::out_rb_size (C++ member), 72
i2s_stream_cfg_t::task_core (C++ member), 72
i2s_stream_cfg_t::task_prio (C++ member), 72
i2s_stream_cfg_t::task_stack (C++ member), 72
i2s_stream_cfg_t::type (C++ member), 72
i2s_stream_init (C++ function), 71
I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT (C macro), 72
I2S_STREAM_RINGBUFFER_SIZE (C macro), 72
i2s_stream_set_clk (C++ function), 71
I2S_STREAM_TASK_CORE (C macro), 72
I2S_STREAM_TASK_PRIO (C macro), 72
I2S_STREAM_TASK_STACK (C macro), 72
io_func (C++ type), 47
IS31FL3216_CH_NUM (C macro), 119
IS31FL3216_STATE_BY_AUDIO (C++ enumerator), 119
IS31FL3216_STATE_FLASH (C++ enumerator), 119
IS31FL3216_STATE_OFF (C++ enumerator), 119
IS31FL3216_STATE_ON (C++ enumerator), 119

L

LCLK_DIV_1000 (C++ enumerator), 135
LCLK_DIV_1024 (C++ enumerator), 135
LCLK_DIV_1088 (C++ enumerator), 135
LCLK_DIV_1152 (C++ enumerator), 135
LCLK_DIV_125 (C++ enumerator), 135
LCLK_DIV_128 (C++ enumerator), 134
LCLK_DIV_136 (C++ enumerator), 135
LCLK_DIV_1408 (C++ enumerator), 135
LCLK_DIV_1496 (C++ enumerator), 135
LCLK_DIV_1500 (C++ enumerator), 135
LCLK_DIV_1536 (C++ enumerator), 135
LCLK_DIV_192 (C++ enumerator), 134
LCLK_DIV_2112 (C++ enumerator), 135
LCLK_DIV_2304 (C++ enumerator), 135
LCLK_DIV_250 (C++ enumerator), 135
LCLK_DIV_256 (C++ enumerator), 134
LCLK_DIV_272 (C++ enumerator), 135
LCLK_DIV_375 (C++ enumerator), 135
LCLK_DIV_384 (C++ enumerator), 134
LCLK_DIV_500 (C++ enumerator), 135
LCLK_DIV_512 (C++ enumerator), 135
LCLK_DIV_544 (C++ enumerator), 135
LCLK_DIV_576 (C++ enumerator), 135
LCLK_DIV_750 (C++ enumerator), 135
LCLK_DIV_768 (C++ enumerator), 135
LCLK_DIV_MAX (C++ enumerator), 135
LCLK_DIV_MIN (C++ enumerator), 134

M

MCLK_DIV_1 (C++ enumerator), 133
MCLK_DIV_10 (C++ enumerator), 134
MCLK_DIV_11 (C++ enumerator), 134
MCLK_DIV_12 (C++ enumerator), 134
MCLK_DIV_13 (C++ enumerator), 134
MCLK_DIV_14 (C++ enumerator), 134
MCLK_DIV_15 (C++ enumerator), 134
MCLK_DIV_16 (C++ enumerator), 134
MCLK_DIV_17 (C++ enumerator), 134
MCLK_DIV_18 (C++ enumerator), 134
MCLK_DIV_2 (C++ enumerator), 133
MCLK_DIV_20 (C++ enumerator), 134
MCLK_DIV_22 (C++ enumerator), 134
MCLK_DIV_24 (C++ enumerator), 134
MCLK_DIV_25 (C++ enumerator), 134
MCLK_DIV_3 (C++ enumerator), 133
MCLK_DIV_30 (C++ enumerator), 134
MCLK_DIV_32 (C++ enumerator), 134
MCLK_DIV_33 (C++ enumerator), 134
MCLK_DIV_34 (C++ enumerator), 134
MCLK_DIV_36 (C++ enumerator), 134
MCLK_DIV_4 (C++ enumerator), 134
MCLK_DIV_44 (C++ enumerator), 134
MCLK_DIV_48 (C++ enumerator), 134
MCLK_DIV_5 (C++ enumerator), 134
MCLK_DIV_6 (C++ enumerator), 134
MCLK_DIV_66 (C++ enumerator), 134

MCLK_DIV_7 (C++ enumerator), 134
MCLK_DIV_72 (C++ enumerator), 134
MCLK_DIV_8 (C++ enumerator), 134
MCLK_DIV_9 (C++ enumerator), 134
MCLK_DIV_MAX (C++ enumerator), 134
MCLK_DIV_MIN (C++ enumerator), 133
mem_assert (C macro), 62
MIC_GAIN_0DB (C++ enumerator), 136
MIC_GAIN_12DB (C++ enumerator), 136
MIC_GAIN_15DB (C++ enumerator), 136
MIC_GAIN_18DB (C++ enumerator), 136
MIC_GAIN_21DB (C++ enumerator), 136
MIC_GAIN_24DB (C++ enumerator), 136
MIC_GAIN_3DB (C++ enumerator), 136
MIC_GAIN_6DB (C++ enumerator), 136
MIC_GAIN_9DB (C++ enumerator), 136
MIC_GAIN_MAX (C++ enumerator), 136
MIC_GAIN_MIN (C++ enumerator), 136
mp3_decoder_cfg_t (C++ class), 84
mp3_decoder_cfg_t::out_rb_size (C++ member), 84
mp3_decoder_cfg_t::task_core (C++ member), 84
mp3_decoder_cfg_t::task_prio (C++ member), 84
mp3_decoder_cfg_t::task_stack (C++ member), 84
mp3_decoder_init (C++ function), 84
MP3_DECODER_RINGBUFFER_SIZE (C macro), 84
MP3_DECODER_TASK_CORE (C macro), 84
MP3_DECODER_TASK_PRIO (C macro), 84
MP3_DECODER_TASK_STACK_SIZE (C macro), 84

O

ogg_decoder_cfg_t (C++ class), 85
ogg_decoder_cfg_t::out_rb_size (C++ member), 85
ogg_decoder_cfg_t::task_core (C++ member), 85
ogg_decoder_cfg_t::task_prio (C++ member), 85
ogg_decoder_cfg_t::task_stack (C++ member), 85
ogg_decoder_init (C++ function), 85
OGG_DECODER_RINGBUFFER_SIZE (C macro), 85
OGG_DECODER_TASK_CORE (C macro), 85
OGG_DECODER_TASK_PRIO (C macro), 85
OGG_DECODER_TASK_STACK_SIZE (C macro), 85
on_event_iface_func (C++ type), 61
opus_decoder_cfg_t (C++ class), 86
opus_decoder_cfg_t::out_rb_size (C++ member), 86
opus_decoder_cfg_t::task_core (C++ member), 86
opus_decoder_cfg_t::task_prio (C++ member), 86
opus_decoder_cfg_t::task_stack (C++ member), 86
OPUS_DECODER_RINGBUFFER_SIZE (C macro), 86
OPUS_DECODER_TASK_CORE (C macro), 86
OPUS_DECODER_TASK_PRIO (C macro), 86
OPUS_DECODER_TASK_STACK_SIZE (C macro), 86

P

periph_adc_button_cfg_t (C++ class), 116
periph_adc_button_cfg_t::arr (C++ member), 117
periph_adc_button_cfg_t::arr_size (C++ member), 117
PERIPH_ADC_BUTTON_CLICK (C++ enumerator), 117
periph_adc_button_event_id_t (C++ type), 117
PERIPH_ADC_BUTTON_IDLE (C++ enumerator), 117
periph_adc_button_init (C++ function), 116
PERIPH_ADC_BUTTON_PRESS (C++ enumerator), 117
PERIPH_ADC_BUTTON_RELEASE (C++ enumerator), 117
PERIPH_BLUETOOTH_AUDIO_STARTED (C++ enumerator), 95
PERIPH_BLUETOOTH_AUDIO_STOPPED (C++ enumerator), 95
PERIPH_BLUETOOTH_AUDIO_SUSPENDED (C++ enumerator), 95
periph_bluetooth_cancel_discover (C++ function), 93
periph_bluetooth_connect (C++ function), 93
PERIPH_BLUETOOTH_CONNECTED (C++ enumerator), 95
PERIPH_BLUETOOTH_DISCONNECTED (C++ enumerator), 95
periph_bluetooth_discover (C++ function), 93
periph_bluetooth_event_id_t (C++ type), 94
periph_bluetooth_fast_forward (C++ function), 93
periph_bluetooth_next (C++ function), 92
periph_bluetooth_pause (C++ function), 92
periph_bluetooth_play (C++ function), 91
periph_bluetooth_prev (C++ function), 92
periph_bluetooth_rewind (C++ function), 92
periph_bluetooth_stop (C++ function), 92
PERIPH_BLUETOOTH_UNKNOWN (C++ enumerator), 95
periph_button_cfg_t (C++ class), 113
periph_button_cfg_t::gpio_mask (C++ member), 113
periph_button_cfg_t::long_press_time_ms (C++ member), 113
periph_button_event_id_t (C++ type), 114
periph_button_init (C++ function), 113
PERIPH_BUTTON_LONG_PRESSED (C++ enumerator), 114
PERIPH_BUTTON_LONG_RELEASE (C++ enumerator), 114
PERIPH_BUTTON_PRESSED (C++ enumerator), 114
PERIPH_BUTTON_RELEASE (C++ enumerator), 114
PERIPH_BUTTON_UNCHANGE (C++ enumerator), 114
periph_console_cfg_t (C++ class), 110
periph_console_cfg_t::command_num (C++ member), 111
periph_console_cfg_t::commands (C++ member), 111
periph_console_cfg_t::prompt_string (C++ member), 111
periph_console_cfg_t::task_prio (C++ member), 111
periph_console_cfg_t::task_stack (C++ member), 111

periph_console_cmd_t (C++ class), 110
periph_console_cmd_t::cmd (C++ member), 110
periph_console_cmd_t::func (C++ member), 110
periph_console_cmd_t::help (C++ member), 110
periph_console_cmd_t::id (C++ member), 110
periph_console_init (C++ function), 110
PERIPH_ID_ADC (C++ enumerator), 102
PERIPH_ID_ADC_BTN (C++ enumerator), 102
PERIPH_ID_AUXIN (C++ enumerator), 102
PERIPH_ID_BLUETOOTH (C++ enumerator), 102
PERIPH_ID_BUTTON (C++ enumerator), 102
PERIPH_ID_CONSOLE (C++ enumerator), 102
PERIPH_ID_FLASH (C++ enumerator), 102
PERIPH_ID_IS31FL3216 (C++ enumerator), 103
PERIPH_ID_LED (C++ enumerator), 102
PERIPH_ID_SDCARD (C++ enumerator), 102
PERIPH_ID_SPIFFS (C++ enumerator), 102
PERIPH_ID_TOUCH (C++ enumerator), 102
PERIPH_ID_WIFI (C++ enumerator), 102
periph_is31fl3216_cfg_t (C++ class), 118
periph_is31fl3216_cfg_t::duty (C++ member), 119
periph_is31fl3216_cfg_t::is31fl3216_pattern (C++ member), 119
periph_is31fl3216_cfg_t::state (C++ member), 119
periph_is31fl3216_init (C++ function), 117
periph_is31fl3216_set_blink_pattern (C++ function), 118
periph_is31fl3216_set_duty (C++ function), 118
periph_is31fl3216_set_state (C++ function), 118
periph_is31fl3216_state_t (C++ type), 119
periph_led_blink (C++ function), 114
PERIPH_LED_BLINK_FINISH (C++ enumerator), 116
periph_led_cfg_t (C++ class), 115
periph_led_cfg_t::led_duty_resolution (C++ member), 115
periph_led_cfg_t::led_freq_hz (C++ member), 115
periph_led_cfg_t::led_speed_mode (C++ member), 115
periph_led_cfg_t::led_timer_num (C++ member), 115
periph_led_event_id_t (C++ type), 116
periph_led_init (C++ function), 114
periph_led_stop (C++ function), 115
PERIPH_LED_UNCHANGE (C++ enumerator), 116
periph_sdcard_cfg_t (C++ class), 106
periph_sdcard_cfg_t::card_detect_pin (C++ member), 107
periph_sdcard_cfg_t::root (C++ member), 107
periph_sdcard_event_id_t (C++ type), 107
periph_sdcard_init (C++ function), 106
periph_sdcard_is_mounted (C++ function), 106
periph_spiffs_cfg_t (C++ class), 108
periph_spiffs_cfg_t::format_if_mount_failed (C++ member), 108
periph_spiffs_cfg_t::max_files (C++ member), 108
periph_spiffs_cfg_t::partition_label (C++ member), 108
periph_spiffs_cfg_t::root (C++ member), 108
periph_spiffs_event_id_t (C++ type), 108
periph_spiffs_init (C++ function), 108
periph_spiffs_is_mounted (C++ function), 108
PERIPH_STATE_ERROR (C++ enumerator), 103
PERIPH_STATE_INIT (C++ enumerator), 103
PERIPH_STATE_NULL (C++ enumerator), 103
PERIPH_STATE_PAUSE (C++ enumerator), 103
PERIPH_STATE_RUNNING (C++ enumerator), 103
PERIPH_STATE_STATUS_MAX (C++ enumerator), 103
PERIPH_STATE_STOPPING (C++ enumerator), 103
periph_tick_get (C macro), 102
periph_touch_cfg_t (C++ class), 112
periph_touch_cfg_t::long_tap_time_ms (C++ member), 112
periph_touch_cfg_t::tap_threshold_percent (C++ member), 112
periph_touch_cfg_t::touch_mask (C++ member), 112
periph_touch_event_id_t (C++ type), 112
periph_touch_init (C++ function), 111
PERIPH_TOUCH_LONG_RELEASE (C++ enumerator), 113
PERIPH_TOUCH_LONG_TAP (C++ enumerator), 113
PERIPH_TOUCH_RELEASE (C++ enumerator), 112
PERIPH_TOUCH_TAP (C++ enumerator), 112
PERIPH_TOUCH_UNCHANGE (C++ enumerator), 112
periph_wifi_cfg_t (C++ class), 104
periph_wifi_cfg_t::disable_auto_reconnect (C++ member), 105
periph_wifi_cfg_t::password (C++ member), 105
periph_wifi_cfg_t::reconnect_timeout_ms (C++ member), 105
periph_wifi_cfg_t::ssid (C++ member), 105
PERIPH_WIFI_CONFIG_DONE (C++ enumerator), 105
PERIPH_WIFI_CONFIG_ERROR (C++ enumerator), 105
periph_wifi_config_mode_t (C++ type), 105
periph_wifi_config_start (C++ function), 104
periph_wifi_config_wait_done (C++ function), 104
PERIPH_WIFI_CONNECTED (C++ enumerator), 105
PERIPH_WIFI_CONNECTING (C++ enumerator), 105
PERIPH_WIFI_DISCONNECTED (C++ enumerator), 105
PERIPH_WIFI_ERROR (C++ enumerator), 105
periph_wifi_init (C++ function), 103
periph_wifi_is_connected (C++ function), 104
PERIPH_WIFI_SETTING (C++ enumerator), 105
periph_wifi_state_t (C++ type), 105
PERIPH_WIFI_UNCHANGE (C++ enumerator), 105
periph_wifi_wait_for_connected (C++ function), 103
process_func (C++ type), 47

R

RAW_STREAM_CFG_DEFAULT (C macro), 77

raw_stream_cfg_t (C++ class), 77
raw_stream_cfg_t::out_rb_size (C++ member), 77
raw_stream_cfg_t::type (C++ member), 77
raw_stream_init (C++ function), 76
raw_stream_read (C++ function), 76
RAW_STREAM_RINGBUFFER_SIZE (C macro), 77
raw_stream_write (C++ function), 76
RB_ABORT (C macro), 122
rb_abort (C++ function), 120
rb_bytes_available (C++ function), 121
rb_bytes_filled (C++ function), 121
rb_create (C++ function), 120
rb_destroy (C++ function), 120
RB_DONE (C macro), 122
rb_done_write (C++ function), 122
RB_FAIL (C macro), 122
rb_get_size (C++ function), 121
RB_OK (C macro), 122
rb_read (C++ function), 121
rb_reset (C++ function), 121
rb_size_get (C++ function), 122
RB_TIMEOUT (C macro), 122
rb_write (C++ function), 121
ringbuf_handle_t (C++ type), 122
RSP_FILTER_BUFFER_BYTE (C macro), 90
rsp_filter_cfg_t (C++ class), 89
rsp_filter_cfg_t::complexity (C++ member), 90
rsp_filter_cfg_t::dest_ch (C++ member), 89
rsp_filter_cfg_t::dest_rate (C++ member), 89
rsp_filter_cfg_t::down_ch_idx (C++ member), 90
rsp_filter_cfg_t::max_indata_bytes (C++ member), 89
rsp_filter_cfg_t::mode (C++ member), 89
rsp_filter_cfg_t::out_len_bytes (C++ member), 90
rsp_filter_cfg_t::out_rb_size (C++ member), 90
rsp_filter_cfg_t::sample_bits (C++ member), 89
rsp_filter_cfg_t::src_ch (C++ member), 89
rsp_filter_cfg_t::src_rate (C++ member), 89
rsp_filter_cfg_t::task_core (C++ member), 90
rsp_filter_cfg_t::task_prio (C++ member), 90
rsp_filter_cfg_t::task_stack (C++ member), 90
rsp_filter_cfg_t::type (C++ member), 90
rsp_filter_init (C++ function), 89
RSP_FILTER_RINGBUFFER_SIZE (C macro), 90
rsp_filter_set_src_info (C++ function), 89
RSP_FILTER_TASK_CORE (C macro), 90
RSP_FILTER_TASK_PRIO (C macro), 90
RSP_FILTER_TASK_STACK (C macro), 90

S

SDCARD_STATUS_CARD_DETECT_CHANGE (C++ enumerator), 107
SDCARD_STATUS_MOUNT_ERROR (C++ enumerator), 107
SDCARD_STATUS_MOUNTED (C++ enumerator), 107
SDCARD_STATUS_UNKNOWN (C++ enumerator), 107
SDCARD_STATUS_UNMOUNT_ERROR (C++ enumerator), 107
SDCARD_STATUS_UNMOUNTED (C++ enumerator), 107
SPIFFS_STATUS_MOUNT_ERROR (C++ enumerator), 108
SPIFFS_STATUS_MOUNTED (C++ enumerator), 108
SPIFFS_STATUS_UNKNOWN (C++ enumerator), 108
SPIFFS_STATUS_UNMOUNT_ERROR (C++ enumerator), 109
SPIFFS_STATUS_UNMOUNTED (C++ enumerator), 108
SPIFFS_STREAM_BUF_SIZE (C macro), 78
SPIFFS_STREAM_CFG_DEFAULT (C macro), 78
spiffs_stream_cfg_t (C++ class), 77
spiffs_stream_cfg_t::buf_sz (C++ member), 78
spiffs_stream_cfg_t::out_rb_size (C++ member), 78
spiffs_stream_cfg_t::task_core (C++ member), 78
spiffs_stream_cfg_t::task_prio (C++ member), 78
spiffs_stream_cfg_t::task_stack (C++ member), 78
spiffs_stream_cfg_t::type (C++ member), 78
spiffs_stream_init (C++ function), 77
SPIFFS_STREAM_RINGBUFFER_SIZE (C macro), 78
SPIFFS_STREAM_TASK_CORE (C macro), 78
SPIFFS_STREAM_TASK_PRIO (C macro), 78
SPIFFS_STREAM_TASK_STACK (C macro), 78
stream_func (C++ type), 47

T

TERMINATION_TYPE_DONE (C++ enumerator), 64
TERMINATION_TYPE_MAX (C++ enumerator), 64
TERMINATION_TYPE_NOW (C++ enumerator), 64
timer_callback (C++ type), 102
TOUCH_PAD_SEL0 (C++ enumerator), 112
TOUCH_PAD_SEL1 (C++ enumerator), 112
TOUCH_PAD_SEL2 (C++ enumerator), 112
TOUCH_PAD_SEL3 (C++ enumerator), 112
TOUCH_PAD_SEL4 (C++ enumerator), 112
TOUCH_PAD_SEL5 (C++ enumerator), 112
TOUCH_PAD_SEL6 (C++ enumerator), 112
TOUCH_PAD_SEL7 (C++ enumerator), 112
TOUCH_PAD_SEL8 (C++ enumerator), 112
TOUCH_PAD_SEL9 (C++ enumerator), 112

W

wav_decoder_cfg_t (C++ class), 87
wav_decoder_cfg_t::out_rb_size (C++ member), 87
wav_decoder_cfg_t::task_core (C++ member), 87
wav_decoder_cfg_t::task_prio (C++ member), 87
wav_decoder_cfg_t::task_stack (C++ member), 87
wav_decoder_init (C++ function), 87
WAV_DECODER_RINGBUFFER_SIZE (C macro), 87

WAV_DECODER_TASK_CORE (C macro), 87
WAV_DECODER_TASK_PRIO (C macro), 87
WAV_DECODER_TASK_STACK (C macro), 87
wav_encoder_cfg_t (C++ class), 88
wav_encoder_cfg_t::out_rb_size (C++ member), 88
wav_encoder_cfg_t::task_core (C++ member), 88
wav_encoder_cfg_t::task_prio (C++ member), 88
wav_encoder_cfg_t::task_stack (C++ member), 88
wav_encoder_init (C++ function), 88
WAV_ENCODER_RINGBUFFER_SIZE (C macro), 88
WAV_ENCODER_TASK_CORE (C macro), 88
WAV_ENCODER_TASK_PRIO (C macro), 88
WAV_ENCODER_TASK_STACK (C macro), 88
WIFI_CONFIG_AIRKISS (C++ enumerator), 105
WIFI_CONFIG_BLUEFI (C++ enumerator), 105
WIFI_CONFIG_ESPTOUCH (C++ enumerator), 105
WIFI_CONFIG_ESPTOUCH_AIRKISS (C++ enumerator), 105
WIFI_CONFIG_WEB (C++ enumerator), 105
WIFI_CONFIG_WPS (C++ enumerator), 105

Z

zl38063_config_i2s (C++ function), 142
zl38063_ctrl_state (C++ function), 142
zl38063_deinit (C++ function), 141
zl38063_get_voice_volume (C++ function), 142
zl38063_init (C++ function), 141
zl38063_set_voice_volume (C++ function), 142