Read the Docs Template
Documentation

Read the Docs

Apr 23, 2020
# 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 ESP32-LyraT-Mini V1.2 Getting Started Guide 13  
1.4 About ESP-ADF 18  
1.5 Set up ESP-IDF 18  
1.6 Get ESP-ADF 18  
1.7 Setup Path to ESP-ADF 19  
1.8 Start a Project 19  
1.9 Connect and Configure 19  
1.10 Build, Flash and Monitor 19  
1.11 Update ESP-ADF 21  
1.12 Related Documents 21  

2 API Reference 39  
2.1 Audio Framework 40  
2.2 Audio Streams 84  
2.3 Codecs 93  
2.4 Audio Processing 104  
2.5 Services 114  
2.6 Speech Recognition 118  
2.7 Peripherals 125  
2.8 Abstraction Layer 153  
2.9 Configuration Options 173  

3 Design Guide 175  
3.1 Project Design 175  
3.2 Design Considerations 177  
3.3 Software Design 180  
3.4 Development Boards 183  
3.5 Audio Samples 202  

4 Resources 205  

5 Copyrights and Licenses 207  
5.1 Software Copyrights 207
This is the documentation for Espressif Audio Development Framework (ADF).
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 | Getting Started with ESP32-LyraT-Mini |

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, connected smart-home appliances with one or more audio functionality, etc.
The ESP32-LyraT is a stereo audio board. If you are looking for a mono audio board, intended for lower end applications, check *ESP32-LyraT-Mini V1.2 Getting Started Guide*.

### 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.
1.1. ESP32-LyraT V4.3 Getting Started Guide
**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:

- **Set up 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, MicroSD, 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 Other Boards from LyraT Family

- ESP32-LyraT-Mini V1.2 Getting Started Guide
- ESP32-LyraTD-MSC V2.2 Getting Started Guide

1.1.6 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.

![ESP32-LyraTD-MSC Side View](image)

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.

**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.

**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.

**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:
Fig. 4: ESP32-LyraTD-MSC Block Diagram

Legend
PB = Push Button
PA = Power Amplifier
Fig. 5: ESP32-LyraTD-MSC V2.2 Lower Board (A) Components
Fig. 6: ESP32-LyraTD-MSC V2.2 Upper Board (B) Components
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:

- **Set up 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 Other Boards from LyraT Family

- **ESP32-LyraT V4.3 Getting Started Guide**
- **ESP32-LyraT-Mini V1.2 Getting Started Guide**

1.2.4 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)**

1.3 ESP32-LyraT-Mini V1.2 Getting Started Guide

This guide provides users with functional descriptions, configuration options for ESP32-LyraT-Mini V1.2 audio development board, as well as how to get started with the ESP32-LyraT 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, connected smart-home appliances with one or more audio functionality, etc.
The ESP32-LyraT-Mini is a mono audio board. If you are looking for a stereo audio board, check ESP32-LyraT V4.3 Getting Started Guide.

1.3.1 What You Need

- ESP32-LyraT-Mini V1.2 board
- 4-ohm speaker with Dupont female jumper wires or headphones with a 3.5 mm jack
- Two Micro-USB 2.0 cables, Type A to Micro B
- PC loaded with Windows, Linux or Mac OS

Optional components

- Micro SD-card
- Li-ion Battery

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

Overview

The ESP32-LyraT-Mini V1.2 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 on-board of the ESP32 chip. The specific hardware includes:

- ESP32-WROVER-B module
- Audio codec chip
- ADC chip
- Microphone on board
- Audio output
- 1 x 3-watt speaker output
- MicroSD card slot (1 line)
- Eight keys
- Two system LEDs
- JTAG and UART test points
- Integrated USB-UART Bridge Chip
- Li-ion Battery-Charge Management

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

Components

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT-Mini used in this guide. For detailed technical documentation of this board, please refer to ESP32-LyraT-Mini V1.2 Hardware Reference and ESP32-LyraT-Mini V1.2 schematic (PDF). The list below provides description starting from the picture’s top right corner and going clockwise.
The audio codec chip, ES8311, is a low power mono audio codec. It consists of 1-channel ADC, 1-channel DAC, low noise pre-amplifier, headphone driver, digital sound effects, analog mixing and gain functions. It is interfaced with ESP32-WROVER-B Module over I2S and I2C buses to provide audio processing in hardware independently from the audio application.

**Audio Output** Output socket to connect headphones with a 3.5 mm stereo jack. (Please note that the board outputs a mono signal)

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

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

**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 USB Power Port. The Charging red LED indicates that a battery connected to the Battery Socket is being charged.

**Power On Switch** Power on/off knob: toggling it to the top powers the board on; toggling it to the down powers the board off.

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

**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.

### 1.3.2 Start Application Development

Before powering up the ESP32-LyraT-Mini, please make sure that the board has been received in good condition with no obvious signs of damage.
Fig. 8: ESP32 LyraT-Mini V1.2 Board Layout Overview
Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 4-ohm speaker to the **Speaker Output**. Connecting headphones to the **Audio Output** is an option.
2. Plug in the Micro-USB cables to the PC and to **both USB ports** of the ESP32-LyraT-Mini.
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 top 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:

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

1.3.3 Other Boards from LyraT Family

- *ESP32-LyraT V4.3 Getting Started Guide*
- *ESP32-LyraTD-MSC V2.2 Getting Started Guide*

1.3.4 Related Documents

- *ESP32-LyraT-Mini V1.2 schematic* (PDF)
- *ESP32-LyraT-Mini V1.2 Hardware Reference*
- 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.
1.4 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.

Note: ESP-ADF is developed using v3.3.1 version of ESP-IDF. If you have already set up another version, please switch to the v3.3.1, or you may not be able to compile ESP-ADF applications.

1.5 Set up 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.

1.6 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
```
1.7 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.

1.8 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.

1.9 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.

1.10 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.
1.10.1 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 ---
```

1.10.2 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
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.

## 1.11 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.

## 1.12 Related Documents

### 1.12.1 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 x 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 x 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.

![ESP32-LyraT Block Diagram](image-url)

**Fig. 9: ESP32-LyraT Block Diagram**

**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 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.

**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 busses 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.

1.12. Related Documents
Fig. 10: ESP32-LyraT V4.2 Board Layout
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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
</tr>
<tr>
<td>2</td>
<td>ON</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>ON</td>
</tr>
<tr>
<td>4</td>
<td>ON</td>
</tr>
<tr>
<td>5</td>
<td>ON</td>
</tr>
<tr>
<td>6</td>
<td>ON</td>
</tr>
<tr>
<td>7</td>
<td>ON</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>LED Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO19</td>
<td>Red LED</td>
</tr>
<tr>
<td>GPIO22</td>
<td>Green LED</td>
</tr>
</tbody>
</table>
## Touch Pads

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Touch Pad Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GPIO33</td>
</tr>
<tr>
<td>2</td>
<td>GPIO32</td>
</tr>
<tr>
<td>3</td>
<td>GPIO13</td>
</tr>
<tr>
<td>4</td>
<td>GPIO27</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>MicroSD Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MTDI / GPIO12</td>
</tr>
<tr>
<td>2</td>
<td>MTCK / GPIO13</td>
</tr>
<tr>
<td>3</td>
<td>MTD0 / GPIO15</td>
</tr>
<tr>
<td>4</td>
<td>MTMS / GPIO14</td>
</tr>
<tr>
<td>5</td>
<td>GPIO2</td>
</tr>
<tr>
<td>6</td>
<td>GPIO4</td>
</tr>
<tr>
<td>7</td>
<td>GPIO21</td>
</tr>
</tbody>
</table>

## UART Header / JP2

<table>
<thead>
<tr>
<th>Header Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

## EN and IO0 Headers / JP23 and J24

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Header Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td>EN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Header Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n/a</td>
</tr>
<tr>
<td>2</td>
<td>GPIO0</td>
</tr>
</tbody>
</table>
I2S Header / JP4

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MCLK</td>
<td>GPIO0</td>
</tr>
<tr>
<td>2 SCLK</td>
<td>GPIO5</td>
</tr>
<tr>
<td>1 LRCK</td>
<td>GPIO25</td>
</tr>
<tr>
<td>2 DSDIN</td>
<td>GPIO26</td>
</tr>
<tr>
<td>3 ASDOUT</td>
<td>GPIO35</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>

I2C Header / JP5

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SCL</td>
<td>GPIO23</td>
</tr>
<tr>
<td>2 SDA</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>

JTAG Header / JP7

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MTDO / GPIO15</td>
<td>TDO</td>
</tr>
<tr>
<td>2 MTCK / GPIO13</td>
<td>TCK</td>
</tr>
<tr>
<td>3 MTDI / GPIO12</td>
<td>TDI</td>
</tr>
<tr>
<td>4 MTMS / GPIO14</td>
<td>TMS</td>
</tr>
</tbody>
</table>

Function DIP Switch / JP8

<table>
<thead>
<tr>
<th>Switch OFF</th>
<th>Switch ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GPIO12 not allocated</td>
<td>MicroSD Card 4-wire</td>
</tr>
<tr>
<td>2 Touch Vol. enabled</td>
<td>MicroSD Card 4-wire</td>
</tr>
<tr>
<td>3 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>4 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>5 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>6 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>7 MicroSD Card 4-wire</td>
<td>AUX IN detect</td>
</tr>
<tr>
<td>8 not used</td>
<td>not used</td>
</tr>
</tbody>
</table>

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. 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:

• Set up 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

1.12.2 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.

**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.

**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.
Fig. 12: ESP32 LyraT V4 board layout
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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

In this mode:

1.12. Related Documents 33
• **JTAG** functionality is not available

• Vol- touch button is available for use with the API

**Enable MicroSD Card in 4-wire Mode**

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
</tr>
<tr>
<td>2</td>
<td>ON</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>ON</td>
</tr>
<tr>
<td>4</td>
<td>ON</td>
</tr>
<tr>
<td>5</td>
<td>ON</td>
</tr>
<tr>
<td>6</td>
<td>ON</td>
</tr>
<tr>
<td>7</td>
<td>ON</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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 GPIO00 or GPIO2, have multiple functions. Please refer to tables below or [ESP32 LyraT V4 schematic](#) for specific details.
Red / Green LEDs

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>LED Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO19</td>
<td>Red LED</td>
</tr>
<tr>
<td>GPIO22</td>
<td>Green LED</td>
</tr>
</tbody>
</table>

Touch Pads

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Touch Pad Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO33</td>
<td>Play</td>
</tr>
<tr>
<td>GPIO32</td>
<td>Set</td>
</tr>
<tr>
<td>GPIO13</td>
<td>Vol-</td>
</tr>
<tr>
<td>GPIO27</td>
<td>Vol+</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>MicroSD Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTD1 / GPIO12</td>
<td>DATA2</td>
</tr>
<tr>
<td>MTCK / GPIO13</td>
<td>CD / DATA3</td>
</tr>
<tr>
<td>MTDO / GPIO15</td>
<td>CMD</td>
</tr>
<tr>
<td>MTMS / GPIO14</td>
<td>CLK</td>
</tr>
<tr>
<td>GPIO2</td>
<td>DATA0</td>
</tr>
<tr>
<td>GPIO4</td>
<td>DATA1</td>
</tr>
<tr>
<td>GPIO21</td>
<td>CD</td>
</tr>
</tbody>
</table>

UART Header / JP2

<table>
<thead>
<tr>
<th>Header Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3.3V</td>
</tr>
<tr>
<td>2 TX</td>
</tr>
<tr>
<td>3 RX</td>
</tr>
<tr>
<td>4 GND</td>
</tr>
</tbody>
</table>

I2S Header / JP4

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCLK</td>
<td>GPIO0</td>
</tr>
<tr>
<td>SCLK</td>
<td>GPIO5</td>
</tr>
<tr>
<td>LRCK</td>
<td>GPIO25</td>
</tr>
<tr>
<td>DSDIN</td>
<td>GPIO26</td>
</tr>
<tr>
<td>ASDOUT</td>
<td>GPIO35</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
</tr>
</tbody>
</table>
Read the Docs Template Documentation

I2C Header / JP5

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SCL</td>
<td>GPIO23</td>
</tr>
<tr>
<td>2 SDA</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>

JTAG Header / JP7

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MTDO / GPIO15</td>
<td>TDO</td>
</tr>
<tr>
<td>2 MTCK / GPIO13</td>
<td>TCK</td>
</tr>
<tr>
<td>3 MTDI / GPIO12</td>
<td>TDI</td>
</tr>
<tr>
<td>4 MTMS / GPIO14</td>
<td>TMS</td>
</tr>
</tbody>
</table>

Function DIP Switch / JP8

<table>
<thead>
<tr>
<th>Switch OFF</th>
<th>Switch ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GPIO12 not allocated</td>
<td>MicroSD Card 4-wire</td>
</tr>
<tr>
<td>2 Touch Vol- enabled</td>
<td>MicroSD Card 4-wire</td>
</tr>
<tr>
<td>3 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>4 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>5 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>6 MicroSD Card</td>
<td>JTAG</td>
</tr>
<tr>
<td>7 MicroSD Card 4-wire</td>
<td>AUX IN detect</td>
</tr>
<tr>
<td>8 not used</td>
<td>not used</td>
</tr>
</tbody>
</table>

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:

- **Set up 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 (Decoders and Encoders), Streams or Audio Processing functions.

![Diagram of audio development framework elements]

Fig. 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.

The audio data is typically acquired using an input Stream, processed with Codecs and in some cases with Audio Processing functions, 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 available 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

```c
audio_element_handle_t audio_element_init (audio_element_cfg_t *config)
```

Initialize audio element with config.

Return

- audio_element handle object
- NULL

Parameters

- `config`: The configuration

```c
esp_err_t audio_element_deinit (audio_element_handle_t el)
```

Destroy audio element handle object, stop, clear, delete all.
esp_err_t audio_element_setdata (audio_element_handle_t el, void *data)
Set context data to element handle object. It can be retrieved by calling audio_element_getdata.

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.

Parameters
- el: The audio element handle

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

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.

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 information.

Parameters
- el: The audio element handle
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• info: The information pointer

desc

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• info: The information pointer

desc

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

desc

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

Return URI pointer

Parameters
• el: The audio element handle

desc

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)
Terminate Audio Element. With this function, audio_element will exit the task function. Note: this API only sends request. It does not actually terminate immediately 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 (Time is DEFAULT_MAX_WAIT_TIME) until Element Task has done and exit.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

esp_err_t audio_element_wait_for_stop_ms(audio_element_handle_t el, TickType_t ticks_to_wait)
After the audio_element_stop function is called, the Element task will perform some abort procedures. The maximum amount of time should block waiting for Element task has stopped.

Return
• ESP_OK, Success
• ESP_FAIL, Timeout

Parameters
• el: The audio element handle
  • ticks_to_wait: The maximum amount of time to wait for stop

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

```c
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 events 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.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

```c
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 sent to the `listener`.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

```c
esp_err_t audio_element_set_event_callback(audio_element_handle_t el, event_cb_func cb_func, void *ctx)
```

This function will add a callback to be called from audio element `el`. Any event to caller will cause to call callback function.

Return

- ESP_OK
- ESP_FAIL

Parameters

- el: The audio element handle
- cb_func: The callback function
• **ctx**: Caller context

```c
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 listener.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

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

```c
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

```c
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

```c
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

```c
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 size of 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_report_pos(audio_element_handle_t el)
Element will sendout event with a duplicate information by this function.

Return
• ESP_OK
• ESP_FAIL
• ESP_ERR_NO_MEM

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
Read the Docs Template Documentation

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

dsp_err_t audio_element_reset_input_ringbuf (audio_element_handle_t el)
Reset input buffer.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

dsp_err_t audio_element_finish_state (audio_element_handle_t el)
Set element finish state.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

dsp_err_t audio_element_change_cmd (audio_element_handle_t el, audio_element_msg_cmd_t cmd)
Change element running state with specific command.

Return
• ESP_OK
• ESP_FAIL
• ESP_ERR_INVALID_ARG Element handle is null

Parameters
• el: The audio element handle
• cmd: Specific command from audio_element_msg_cmd_t

dsp_err_t audio_element_reset_output_ringbuf (audio_element_handle_t el)
Reset output buffer.

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 provide Element input data. Depending on setup using ringbuffer or function callback, Element invokes read ringbuffer, or calls read callback function.

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 send out Element output data. Depending on setup using ringbuffer or function callback, Element will invoke write to ringbuffer, or call write callback function.

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

2.1. Audio Framework
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_set_output_ringbuf_size(audio_element_handle_t el, int rb_size)
Set Element output ringbuffer size.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• rb_size: Size of the ringbuffer

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

esp_err_t audio_element_process_init(audio_element_handle_t el)
Provides a way to call element’s open

Return

• ESP_OK
• ESP_FAIL

Parameters

• el: The audio element handle

esp_err_t audio_element_process_deinit(audio_element_handle_t el)
Provides a way to call element’s close

Return

• ESP_OK
• ESP_FAIL

Parameters

• el: The audio element handle

esp_err_t audio_element_seek(audio_element_handle_t el, void *in_data, int in_size, void *out_data, int *out_size)
Call element’s seek

Return

• ESP_OK
• ESP_FAIL
• ESP_ERR_NOT_SUPPORTED

Parameters

• el: The audio element handle
• in_data: A pointer to in data
• in_size: The size of the in_data
• out_data: A pointer to the out data
• out_size: The size of the out_data

Structures

struct audio_element_reserve_data_t
Audio Element user reserved data.
Public Members

```c
int user_data_0
    user data 0
int user_data_1
    user data 1
int user_data_2
    user data 2
int user_data_3
    user data 3
int user_data_4
    user data 4
```

```
struct audio_element_info_t
    Audio Element informations.
```

Public Members

```c
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 bps
    Bit per second
int64_t byte_pos
    The current position (in bytes) being processed for an element
int64_t total_bytes
    The total bytes for an element
int duration
    The duration for an element (optional)
char *uri
    URI (optional)
esp_codec_type_t codec_fmt
    Music format (optional)
```

```
    audio_element_reserve_data_t reserve_data
    This value is reserved for user use (optional)
```

```
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

ctrl_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

const char *tag
Element tag

bool stack_in_ext
Try to allocate stack in external memory

int multi_in_rb_num
The number of multiple input ringbuffer

int multi_out_rb_num
The number of multiple output ringbuffer

Macros

AUDIO_ELEMENT_INFO_DEFAULT()

DEFAULT_ELEMENT_RINGBUF_SIZE

DEFAULT_ELEMENT_BUFFER_LENGTH

DEFAULT_ELEMENT_STACK_SIZE

2.1. Audio Framework
**Type Definitions**

```c
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)
typedef esp_err_t (*event_cb_func) (audio_element_handle_t el, audio_event_iface_msg_t *event, void *ctx)
typedef esp_err_t (*ctrl_func) (audio_element_handle_t self, void *in_data, int in_size, void *out_data, int *out_size)
```

**Enumerations**

```c
eenum 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

eenum 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

eenum 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
AEL_MSG_CMD_REPORT_POSITION = 11

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_STATE_FINISHED = 15
    AEL_STATUS_MOUNTED = 16
    AEL_STATUS_UNMOUNTED = 17

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. 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
Note Because of stop pipeline or pause pipeline depend much on register order. Please register element strictly in the following order: input element first, process middle, output element last.

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 of the linked elements. Used with audio_pipeline_wait_for_stop to keep in sync. The link state of the elements in the pipeline is kept, events are still registered. The stopped audio_pipeline restart by audio_pipeline_resume.

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 portMAX_DELAY 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

```c
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

```c
audio_element_handle_t audio_pipeline_get_el_by_tag(audio_pipeline_handle_t pipeline, const char *tag)
```
Find un-kept element from registered pipeline by tag.

**Return**
- NULL when any errors
- Others on success

**Parameters**
- pipeline: The Audio Pipeline Handle
- tag: A char pointer

```c
audio_element_handle_t audio_pipeline_get_el_once(audio_pipeline_handle_t pipeline, const audio_element_handle_t start_el, const char *tag)
```
Based on beginning element to find un-kept element from registered pipeline by tag.

**Return**
- NULL when any errors
- Others on success

**Parameters**
- pipeline: The Audio Pipeline Handle
- start_el: Specific beginning element
- 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 listener 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 connect_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
- *connect_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.

```c
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.

```c
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.

```c
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_reset_elements(audio_pipeline_handle_t pipeline)

Reset Pipeline linked elements state.

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_reset_kept_state(audio_pipeline_handle_t pipeline, audio_element_handle_t el)

Reset the specific element kept state.

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_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 working (AEL_STATE_RUNNING or AEL_STATE_PAUSED) 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

```
esp_err_t audio_pipeline_relink_more(audio_pipeline_handle_t pipeline, audio_element_handle_t element_1, ...)
```

Adds a NULL-terminated list of elements to audio_pipeline.

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
- **element_1**: The element to add to the audio_pipeline.
- **...**: Additional elements to add to the audio_pipeline.

```c
esp_err_t audio_pipeline_change_state(audio_pipeline_handle_t pipeline, audio_element_state_t new_state)
```

Set the pipeline state.

Return

- **ESP_OK** All linked elements state are same.
- **ESP_FAIL** Error.

Parameters

- **pipeline**: The Audio Pipeline Handle
- **new_state**: The new state will be set

Structures

```c
struct audio_pipeline_cfg
    Audio Pipeline configurations.
```

**Public Members**

```c
int rb_size
    Audio Pipeline ringbuffer size
```

Macros

```c
#define DEFAULT_PIPELINE_RINGBUF_SIZE
#define DEFAULT_AUDIO_PIPELINE_CONFIG
```

Type Definitions

```c
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**

2.1. Audio Framework 67
• 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

```c
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

```c
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

```c
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

```c
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

```c
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.

2.1. Audio Framework
Read the Docs Template Documentation

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

```c
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)
```

```c
enum audio_stream_type_t
    Values:
        AUDIO_STREAM_NONE = 0
        AUDIO_STREAM_READER
        AUDIO_STREAM_WRITER
```

```c
enum audio_codec_type_t
    Values:
        AUDIO_CODEC_TYPE_NONE = 0
        AUDIO_CODEC_TYPE_DECODER
        AUDIO_CODEC_TYPE_ENCODER
```

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

```c
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

`media_source_type_t media_src`
Media source type

Macros

`ESP_ERR_AUDIO_BASE`
Starting number of ESP audio error codes

Type Definitions

```c
typedef void (*esp_audio_event_callback)(esp_audio_state_t *audio, void *ctx)
typedef esp_err_t (*audio_volume_set)(void *hd, int vol)
typedef esp_err_t (*audio_volume_get)(void *hd, int *vol)
```

Enumerations

```c
defined audio_err_t
Values:
```
```c
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_LINK_FAIL = ESP_ERR_AUDIO_BASE + 13
ESP_ERR_AUDIO_UNKNOWN = ESP_ERR_AUDIO_BASE + 14
```
ESP_ERR_AUDIO_OUT_OF_RANGE = ESP_ERR_AUDIO_BASE + 15
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_STOPPED = 3
  AUDIO_STATUS_FINISHED = 4
  AUDIO_STATUS_ERROR = 5

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 when finished
  TERMINATION_TYPE_MAX

enum esp_audio_prefer_t
Values:
  ESP_AUDIO_PREFER_MEM = 0
  ESP_AUDIO_PREFER_SPEED = 1

enum media_source_type_t
Values:
  MEDIA_SRC_TYPE_NULL = 0
  MEDIA_SRC_TYPE_MUSIC_BASE = 0x100
  MEDIA_SRC_TYPE_MUSIC_SD = MEDIA_SRC_TYPE_MUSIC_BASE + 1
  MEDIA_SRC_TYPE_MUSIC_HTTP = MEDIA_SRC_TYPE_MUSIC_BASE + 2
  MEDIA_SRC_TYPE_MUSIC_FLASH = MEDIA_SRC_TYPE_MUSIC_BASE + 3
  MEDIA_SRC_TYPE_MUSIC_A2DP = MEDIA_SRC_TYPE_MUSIC_BASE + 4
  MEDIA_SRC_TYPE_MUSIC_DLNA = MEDIA_SRC_TYPE_MUSIC_BASE + 5
  MEDIA_SRC_TYPE_MUSIC_RAW = MEDIA_SRC_TYPE_MUSIC_BASE + 6
  MEDIA_SRC_TYPE_MUSIC_MAX = 0x1FF
  MEDIA_SRC_TYPE_TONE_BASE = 0x200
  MEDIA_SRC_TYPE_TONE_SD = MEDIA_SRC_TYPE_TONE_BASE + 1
Header File

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

Functions

```c
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

```c
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

```c
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_MEMORY_LACK: allocate memory fail

**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_MEMORY_LACK: allocate memory fail

**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_MEMORY_LACK: allocate memory fail

**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”
- "aadp://44100:2@bt/sink/stream.pcm”

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.
- Enabled CONFIG_FATFS_API_ENCODING_UTF_8, the URI can be support Chinese characters.
- Asynchronous interface
- The maximum of block time can be modify by esp_audio_play_timeout_set, default value is 25 seconds.

Return

- ESP_ERR_AUDIO_NO_ERROR: on succss
- ESP_ERR_AUDIO_TIMEOUT: timeout the play activity
- ESP_ERR_AUDIO_NOT_SUPPORT: Currently status is AUDIO_STATUS_RUNNING
- 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”. If NULL to be set, the uri setup byesp_audio_setup will used.
- **type**: Specific handle type decoder or encoder
- **pos**: Specific starting position by bytes
audio_err_t esp_audio_sync_play(esp_audio_handle_t handle, const char *uri, int pos)
    Play the given uri until music finished or error occurred.

Note
    • All features are same with esp_audio_play
    • Synchronous interface
    • Support decoder mode only
    • No any events post during playing

Return
    • ESP_ERR_AUDIO_NO_ERROR: on success
    • ESP_ERR_AUDIO_TIMEOUT: timeout(8000ms) the play activity
    • ESP_ERR_AUDIO_NOT_SUPPORT: Currently status is AUDIO_STATUS_RUNNING
    • 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”,
    • pos: Specific starting position by bytes

audio_err_t esp_audio_stop(esp_audio_handle_t handle, audio_termination_type_t type)
    A synchronous interface for stop the esp_audio. The maximum of block time is 8000ms.

Note 1. If user queue has been registered by evt_que, AUDIO_STATUS_STOPPED event for success or AUDIO_STATUS_ERROR event for error will be received.

1. TERMINATION_TYPE_DONE only works with input stream which can’t stopped by itself, e.g. raw read/write stream, others streams are no effect.

2. The synchronous interface is used to ensure that working pipeline is stopped.

Return
    • ESP_ERR_AUDIO_NO_ERROR: on success
    • ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
    • ESP_ERR_AUDIO_NOT_READY: The status is not AUDIO_STATUS_RUNNING or AUDIO_STATUS_PAUSED
    • ESP_ERR_AUDIO_TIMEOUT: timeout(8000ms) 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 1. 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.
1. The Paused music must be stoped by `esp_audio_stop` before new playing, otherwise got block on new play.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_NOT_READY: the status is not running
- ESP_ERR_AUDIO_TIMEOUT: timeout(8000ms) 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_RUNNING event for success or AUDIO_STATUS_ERROR event for error will be received.

**Return**
- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_TIMEOUT: timeout(8000ms) 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 success
- 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 success
- 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 success
• 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 success
• ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
• ESP_ERR_AUDIO_NOT_READY: no codec element

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 success
• 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 success
• 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.

audio_err_t esp_audio_media_type_set (esp_audio_handle_t handle, media_source_type_t type)
audio_err_t esp_audio_info_get (esp_audio_handle_t handle, esp_audio_info_t *info)
audio_err_t esp_audio_info_set (esp_audio_handle_t handle, esp_audio_info_t *info)
audio_err_t esp_audio_callback_set (esp_audio_handle_t handle, esp_audio_event_callback cb, void *cb_ctx)
audio_err_t esp_audio_seek (esp_audio_handle_t handle, int seek_time_sec)
  Seek the position in second of currently played music.

Note  This function works only with decoding music.

Return
• ESP_ERR_AUDIO_NO_ERROR: on success
• ESP_ERR_AUDIO_FAIL: codec or allocation fail
• ESP_ERR_AUDIO_TIMEOUT: timeout for sync the element status
• ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
• ESP_ERR_AUDIO_NOT_SUPPORT: codec has finished
• ESP_ERR_AUDIO_OUT_OF_RANGE: the seek_time_ms is out of the range
• ESP_ERR_AUDIO_NOT_READY: the status is neither running nor paused

Parameters
• handle: The esp_audio instance
• seek_time_sec: A pointer to int that indicates esp_audio decoding position.

audio_err_t esp_audio_duration_get (esp_audio_handle_t handle, int *duration)
Get the duration in microseconds of playing music.

Note  This function works only with decoding music.

Return
• ESP_ERR_AUDIO_NO_ERROR: on success
• ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
• ESP_ERR_AUDIO_NOT_READY: no codec element or no in element

Parameters
• handle: The esp_audio instance
• duration: A pointer to int that indicates decoding total time.
audio_err_t esp_audio_play_timeout_set (esp_audio_handle_t handle, int time_ms)
Setting the maximum amount of time to waiting for esp_audio_play only.

Return
  • ESP_ERR_AUDIO_NO_ERROR: on success
  • ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

Parameters
  • handle: The esp_audio instance
  • time_ms: The maximum amount of time

audio_err_t esp_audio_prefer_type_get (esp_audio_handle_t handle, esp_audio_prefer_t *type)
Get the type of esp_audio_prefer_t

Return
  • ESP_ERR_AUDIO_NO_ERROR: on success
  • ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance

Parameters
  • handle: The esp_audio instance
  • type: A pointer to esp_audio_prefer_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 It should be
  make sure same with I2S stream sample_rate

QueueHandle_t evt_que
  For received esp_audio events (optional)

esp_audio_event_callback cb_func
  esp_audio events callback (optional)

void *cb_ctx
  esp_audio callback context (optional)

esp_audio_prefer_t prefer_type
  esp_audio works on sepcific type, default memory is preferred.
  • ESP_AUDIO_PREFER_MEM mode stopped the previous linked elements before the new pipiline
    starting, except out stream element.
• ESP_AUDIO_PREFER_SPEED mode kept the previous linked elements before the new pipeline starting, except out stream element.

```c
void *vol_handle
Volume change instance

audio_volume_set vol_set
Set volume callback

audio_volume_get vol_get
Get volume callback

int task_prio
esp_audio task priority

int task_stack
Size of esp_audio task stack
```

```c
struct esp_audio_setup_t
esp_audio setup parameters by manual
```

## Public Members

```c
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

int set_time
Set starting position of the microseconds time (optional)

char *set_uri
Set URI

char *set_in_stream
Tag of in_stream

char *setCodec
Tag of the codec

char *set_out_stream
Tag of out_stream
```

```c
struct esp_audio_info_t
esp_audio information
```

## Public Members

```c
audio_element_info_t codec_info
Codec information

audio_element_handle_t in_el
Handle of the in stream
```
audio_element_handle_t out_el
Handle of the out stream

audio_element_handle_t codec_el
Handle of the codec

audio_element_handle_t filter_el
Handle of the filter

esp_audio_state_t st
The state of esp_audio

int time_pos
Position of the microseconds time

Macros

DEFAULT_ESP_AUDIO_CONFIG()

Type Definitions

typedef void *esp_audio_handle_t

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)

**esp_err_t i2s_alc_volume_set (audio_element_handle_t i2s_stream, int volume)**

Setup volume of stream by using ALC.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- i2s_stream: The i2s element handle
- volume: The volume of stream will be set.

**esp_err_t i2s_alc_volume_get (audio_element_handle_t i2s_stream, int *volume)**

Get volume of stream.

**Return**

- ESP_OK
- ESP_FAIL
Parameters

- **i2s_stream**: The i2s element handle
- **volume**: The volume of stream

Structures

```c
struct i2s_stream_cfg_t
```

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

**Public Members**

```c
data_stream_type_t type
```
Type of stream

```c
i2s_config_t i2s_config
```
I2S driver configurations

```c
i2s_port_t i2s_port
```
I2S driver hardware port

```c
bool use_alc
```
It is a flag for ALC. If use ALC, the value is true. Or the value is false

```c
int volume
```
The volume of audio input data will be set.

```c
int out_rb_size
```
Size of output ringbuffer

```c
int task_stack
```
Task stack size

```c
int task_core
```
Task running in core (0 or 1)

```c
int task_prio
```
Task priority (based on freeRTOS priority)

```c
bool stack_in_ext
```
Try to allocate stack in external memory

```c
int multi_out_num
```
The number of multiple output

```c
bool uninstall_drv
```
whether uninstall the i2s driver when stream destroyed

Macros

```c
I2S_STREAM_TASK_STACK
```

```c
I2S_STREAM_BUF_SIZE
```

```c
I2S_STREAM_TASK_P prio
```

```c
I2S_STREAM_TASK_CORE
```

```c
I2S_STREAM_RINGBUFFER_SIZE
```
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)

Connect to next track in the playlist.

This function can be used in event_handler of http_stream. User can call this function to connect to next track in playlist when he/she gets HTTP_STREAM_FINISH_TRACK event

Return

- ESP_OK on success
- ESP_FAIL on errors

Parameters

- el: The http_stream element handle

esp_err_t http_stream_restart (audio_element_handle_t el)

esp_err_t http_stream_fetch_again (audio_element_handle_t el)

Try to fetch the tracks again.

If this is live stream we will need to keep fetching URIs.

Return

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if playlist is finished

Parameters

- el: The http_stream element handle
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)

bool stack_in_ext
Try to allocate stack in external memory

http_stream_event_handle_t event_handle
The hook function for HTTP Stream

void *user_data
User data context

bool auto_connect_next_track
connect next track without open/close

bool enable_playlist_parser
Enable playlist parser
int multi_out_num
    The number of multiple output

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 function return the value (-1: ESP_FAIL), HTTP Client will be stopped. If the function return the value > 0, HTTP Stream will ignore the post_field. If the function 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 function return the value (-1: ESP_FAIL), HTTP Client will be stopped. If the function return the value > 0, HTTP Stream will ignore the read function. If the function 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

```c
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

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

Public Members

```c
audio_stream_type_t type
int buf_sz
int out_rb_size
int task_stack
int task_core
int task_prio
```
Stream type
Audio Element Buffer size
Size of output ringbuffer
Task stack size
Task running in core (0 or 1)
Task priority (based on freeRTOS priority)

Macros

```c
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

\[
\text{audio_element_handle_t}\ aac\_decoder\_init\ (aac\_decoder\_cfg\_t \ast config)
\]

Create an Audio Element handle to decode incoming AAC data.

Return  The audio element handle

Parameters

- config: The configuration

Structures

\[
\text{struct aac_decoder_cfg_t}
\]

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)

bool stack_in_ext
    Try to allocate stack in external memory

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/codecs/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)

bool stack_in_ext
  Try to allocate stack in external memory

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)

bool stack_in_ext
    Try to allocate stack in external memory

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)

bool stack_in_ext
    Try to allocate stack in external memory

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)

bool **stack_in_ext**
Try to allocate stack in external memory

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

```c
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)
- `bool stack_in_ext`  
  Try to allocate stack in external memory

**Macros**

```c
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/code/ogg_decoder.h

**Functions**

```c
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)

bool stack_in_ext
Try to allocate stack in external memory

Macros

OGG_DECODER_TASK_STACK_SIZE
OGG_DECODER_TASK_CORE
OGG_DECODER_TASK_P 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 is running

- `int task_prio`
  
  Task priority (based on freeRTOS priority)

- `bool stack_in_ext`
  
  Try to allocate stack in external memory

## 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`
Read the Docs Template Documentation

- 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)
- `bool stack_in_ext`
  Try to allocate stack in external memory

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)

bool stack_in_ext
Try to allocate stack in external memory

Macros

WAV_ENCODER_TASK_STACK
WAV_ENCODER_TASK_CORE
WAV_ENCODER_TASK_PRIO
WAV_ENCODER_RINGBUFFER_SIZE
DEFAULT_WAV_ENCODER_CONFIG()
2.4 Audio Processing

There are couple of options implemented in the ESP-ADF to modify contents of an audio stream:

- Combine contents of two audio streams using Downmix
- Apply ten band Equalizer
- Change audio sampling frequency and convert between single and two channel with Resample Filter
- Modify pitch and speed of the stream using Sonic

Please refer to description of respective APIs below.

2.4.1 Downmix

This API is intended for mixing of two audio files (streams), defined as the base audio file and the newcomer audio file, into one output audio file.

The newcomer audio file will be downmixed into the base audio file with individual gains applied to each file.

Fig. 4: Illustration of Downmixing Process

The number of channel(s) of the output audio file will be the same with that of the base audio file. The number of channel(s) of the newcomer audio file will also be changed to the same with the base audio file, if it is different from that of the base audio file.

The downmix process has 3 states:

- Bypass Downmixing – Only the base audio file will be processed;
- Switch on Downmixing – The base audio file and the target audio file will first enter the transition period, during which the gains of these two files will be changed from the original level to the target level; then enter the stable period, sharing a same target gain;
- Switch off Downmixing – The base audio file and the target audio file will first enter the transition period, during which the gains of these two files will be changed back to their original levels; then enter the stable period, with their original gains, respectively. After that, the downmix process enters the bypass state.

Note that, the sample rates of the base audio file and the newcomer audio file must be the same, otherwise an error occurs.

Application Example

Implementation of this API is demonstrated in advanced_examples/downmix_pipeline example.
API Reference

Header File

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

Functions

void **downmix_set_input_rb_timeout** (**audio_element_handle_t** self, **int** ticks_to_wait)
Sets the downmix timeout.

**Parameters**
- self: audio element handle
- ticks_to_wait: input ringbuffer timeout

void **downmix_set_input_rb** (**audio_element_handle_t** self, **ringbuf_handle_t** rb, **int** index)
Sets the downmix input ringbuffer. refer to **ringbuf.h**

**Parameters**
- self: audio element handle
- rb: handle of ringbuffer
- index: The index of multi input ringbuffer.

**esp_err_t** **downmix_set_output_type** (**audio_element_handle_t** self, **esp_downmix_output_type_t** output_type)
Passes number of channels for output stream. Only supported mono and dual.

**Return** ESP_OK ESP_FAIL

**Parameters**
- self: audio element handle
- output_type: down-mixer output type.

**esp_err_t** **downmix_set_work_mode** (**audio_element_handle_t** self, **esp_downmix_work_mode_t** mode)
Sets BYPASS, ON or OFF status of down-mixer.

**Return** ESP_OK ESP_FAIL

**Parameters**
- self: audio element handle
- mode: down-mixer work mode.

**esp_err_t** **downmix_set_out_ctx_info** (**audio_element_handle_t** self, **esp_downmix_out_ctx_type_t** out_ctx)
Passes content of per channel output stream by down-mixer.

**Return** ESP_OK ESP_FAIL

**Parameters**
• self: audio element handle
• out_ctx: content of output stream.

```c
esp_err_t downmix_set_source_stream_info(audio_element_handle_t self, int rate, int ch, int index)
```
Sets the sample rate and the number of channels of input stream to be processed.

**Return** ESP_OK ESP_FAIL

**Parameters**

• self: audio element handle
• rate: sample rate of the input stream
• ch: number of channel(s) of the input stream
• index: The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range.

```c
esp_err_t downmix_set_gain_info(audio_element_handle_t self, float *gain, int index)
```
Sets the audio gain to be processed.

**Return** ESP_OK ESP_FAIL

**Parameters**

• self: audio element handle
• gain: the reset value of gain. The gain is an array of two elements.
• index: The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range.

```c
esp_err_t downmix_set_transit_time_info(audio_element_handle_t self, int transit_time, int index)
```
Sets the audio transit_time to be processed.

**Return** ESP_OK ESP_FAIL

**Parameters**

• self: audio element handle
• transit_time: the reset value of transit_time
• index: The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range

```c
esp_err_t source_info_init(audio_element_handle_t self, esp_downmix_input_info_t *source_num)
```
Initializes information of the source streams for downmixing.

**Return** ESP_OK ESP_FAIL

**Parameters**

• self: audio element handle
• source_num: The information array of source streams

```c
audio_element_handle_t downmix_init(downmix_cfg_t *config)
```
Initializes the Audio Element handle for downmixing.

**Return** The initialized Audio Element handle

**Parameters**
structures

struct downmix_cfg_t
    Downmix configuration.

Public Members

downmix_info_t
    Downmix information

int max_sample
    The number of samples per downmix processing

int out_rb_size
    Size of ring buffer

int task_stack
    Size of task stack

int task_core
    Task running in core

int task_prio
    Task priority (based on the FreeRTOS priority)

bool stack_in_ext
    Try to allocate stack in external memory

Macros

DOWNMIX_TASK_STACK
DOWNMIX_TASK_CORE
DOWNMIX_TASK_PRIO
DOWNMIX_RINGBUFFER_SIZE
DM_BUF_SIZE
DEFAULT_DOWNMIX_CONFIG()

2.4.2 Equalizer

Provided in this API equalizer supports:

- fixed number of ten (10) bands;
- four sample rates: 11025 Hz, 22050 Hz, 44100 Hz and 48000 Hz.

The center frequencies of bands are shown in table below.

<table>
<thead>
<tr>
<th>Band Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>31 Hz</td>
<td>62 Hz</td>
<td>125 Hz</td>
<td>250 Hz</td>
<td>500 Hz</td>
<td>1 kHz</td>
<td>2 kHz</td>
<td>4 kHz</td>
<td>8 kHz</td>
<td>16 kHz</td>
</tr>
</tbody>
</table>
Default gain of each band is -13 dB. To set the gains of all bands use structure `equalizer_cfg`. To set the gain of individual band use function `equalizer_set_gain_info()`.

### Application Example

Implementation of this API is demonstrated in the `audio_processing/equalizer` example.

### API Reference

#### Header File

- `esp-adf-libs/esp_codec/include/codec/equalizer.h`

#### Functions

**esp_err_t equalizer_set_info** *(audio_element_handle_t self, int rate, int ch)*

Set the audio sample rate and the number of channels to be processed by the equalizer.

**Return** ESP_OK ESP_FAIL

**Parameters**

- `self`: Audio element handle
- `rate`: Audio sample rate
- `ch`: Audio channel

**esp_err_t equalizer_set_gain_info** *(audio_element_handle_t self, int index, int value_gain, bool is_channels_gain_equal)*

Set the audio gain to be processed by the equalizer.

**Return** ESP_OK ESP_FAIL

**Parameters**

- `self`: Audio element handle
- `index`: the position of center frequencies of equalizer
- `value_gain`: the value of audio gain which in `index`
- `is_channels_gain_equal`: if Number of audio channel is equal 2, the value of audio gains which two channels are equal by checking `is_channels_gain_equal`. if `is_channels_gain_equal` is true, it means equal, otherwise unequal.

**audio_element_handle_t equalizer_init** *(equalizer_cfg_t *config)*

Create an Audio Element handle that equalizes incoming data.

**Return** The created audio element handle

**Parameters**

- `config`: The configuration
Structures

```c
struct equalizer_cfg
    Equalizer Configuration.
```

Public Members

```c
int samplerate
    Audio sample rate (in Hz)

int channel
    Number of audio channels (Mono=1, Dual=2)

int *set_gain
    Equalizer gain

int out_rb_size
    Size of output ring buffer

int task_stack
    Task stack size

int task_core
    Task running in core...

int task_prio
    Task priority

bool stack_in_ext
    Try to allocate stack in external memory
```

Macros

```c
EQUALIZER_TASK_STACK
EQUALIZER_TASK_CORE
EQUALIZER_TASK_PRIO
EQUALIZER_RINGBUFFER_SIZE
DEFAULT_EQUALIZER_CONFIG()
```

Type Definitions

```c
typedef struct equalizer_cfg equalizer_cfg_t
    Equalizer Configuration.
```

2.4.3 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.
Application Example

Implementation of this API is demonstrated in the following examples:

- audio_processing/pipeline_resample
- audio_processing/pipeline_spiffs_amr_resample
- get-started/play_mp3

API Reference

Header File

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

Functions

```c
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 of channels of stream data

```c
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 esp_resample_mode_t is RESAMPLE_DECODE_MODE, `src_rate` and `src_ch` will be fetched from audio_element_getinfo.
- If the esp_resample_mode_t is RESAMPLE_ENCODE_MODE, `src_rate`, `src_ch`, `dest_rate` and `dest_ch` must be configured.

**Return** The audio element handle

**Parameters**

- `config`: The configuration

Structures

```c
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.

esp_resample_mode_t 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.

esp_resample_type_t type
    The resampling type (Automatic, Upsampling and Downsampling)

int complexity
    Indicates the complexity of the resampling. This parameter is only valid when FIR filter is used. Range: 0~5; 0 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. If user set complexity less than 0, complexity can be set 0. If user set complexity more than 5, complexity can be set 5.

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

bool stack_in_ext
    Try to allocate stack in external memory

Macros

RSP_FILTER_BUFFER_BYTE
2.4.4 Sonic

The Sonic component acts as a multidimensional filter that lets you adjust audio parameters of a WAV stream. This functionality may be useful to e.g. increase playback speed of an audio recording by a user selectable rate.

The following parameters can be adjusted:
- speed
- pitch
- interpolation type

The adjustments of the first two parameters are represented by float values that provide the rate of adjustment. For example, to increase the speed of an audio sample by 2 times, call `sonic_set_pitch_and_speed_info(el, 1.0, 2.0)`. To keep the speed as it is, call `sonic_set_pitch_and_speed_info(el, 1.0, 1.0)`.

For the interpolation type you may select either faster but less accurate linear interpolation, or slower but more accurate FIR interpolation.

Application Example

Implementation of this API is demonstrated in `audio_processing/pipeline_sonic` example.

API Reference

Header File

- esp-adf-libs/esp_codec/include/codecs/audio_sonic.h

Functions

```c
esp_err_t sonic_set_info(audio_element_handle_t self, int rate, int ch)
```
Sets the audio sample rate and the number of channels to be processed by the sonic.

**Return** ESP_OK ESP_FAIL

**Parameters**
- self: Audio element handle
- rate: The sample rate of stream data
- ch: The number channels of stream data

```c
esp_err_t sonic_set_pitch_and_speed_info(audio_element_handle_t self, float pitch, float speed)
```
Sets the audio pitch and speed to be processed by the sonic.
Return ESP_OK ESP_FAIL

Parameters

• self: Audio element handle
• pitch: Scale factor of pitch of audio file. 0 means the original pitch. The range is [0.2 4.0].
• speed: Scale factor of speed of audio file. 0 means the original speed. The range is [0.1 8.0].

`audio_element_handle_t sonic_init (sonic_cfg_t *config)`
Creates an Audio Element handle for sonic.

Return The sonic audio element handle

Parameters

• config: The sonic configuration

Structures

`struct sonic_info_t`
Information on audio file and configuration parameters required by sonic to process the file.

Public Members

int `samplerate`
Audio file sample rate (in Hz)

int `channel`
Number of audio file channels (Mono=1, Dual=2)

int `resample_linear_interpolate`
Flag of using simple linear interpolation. 1 indicates using simple linear interpolation. 0 indicates not using simple linear interpolation.

float `pitch`
Scale factor of pitch of audio file. If the value of ‘pitch’ is 0.3, the pitch of audio file processed by sonic is lower than the original. If the value of ‘pitch’ is 1.3, the pitch of audio file processed by sonic is 30% higher than the original.

float `speed`
Scale factor of speed of audio file. If the value of ‘speed’ is 0.3, the speed of audio file processed by sonic is 70% slower than the original. If the value of ‘speed’ is 1.3, the speed of audio file processed by sonic is 30% faster than the original.

`struct sonic_cfg_t`
Sonic configuration.

Public Members

`sonic_info_t sonic_info`
Information of sonic

int `out_rb_size`
Size of output ring buffer

2.4. Audio Processing
```c
int task_stack
    Task stack size
int task_core
    Task running in core
int task_prio
    Task priority
bool stack_in_ext
    Try to allocate stack in external memory
```

**Macros**

```c
SONIC_SET_VALUE_FOR_INITIALIZATION
SONIC_TASK_STACK
SONIC_TASK_CORE
SONIC_TASK_PRIO
SONIC_RINGBUFFER_SIZE
DEFAULT_SONIC_CONFIG()
```

## 2.5 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 policies to allow device operation by specific user or application.

For details please refer to descriptions listed below.

### 2.5.1 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

- bluetooth_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 Bluetooth Stream and Bluetooth Peripheral created by bluetooth_service_create_stream and bluetooth_service_create_periph

**Return**
- ESP_OK
- ESP_FAIL

2.5. Services
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`
b brief Bluetooth address length

**Type Definitions**

`typedef uint8_t bluetooth_addr_t[BLUETOOTH_ADDR_LEN]`
b brief Bluetooth device address

**Enumerations**

`enum bluetooth_service_mode_t`
b 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

**Header File**

- bluetooth_service/include/bt_keycontrol.h

### 2.6 Speech Recognition

The ESP-ADF comes complete with *speech recognition interface* to recognize voice wakeup commands. Most of currently implemented wakeup commands are in Chinese with one command “Hi Jeson” in English.

Provided in this section functions also include automatic speech detection, also known as *voice activity detection (VAD)*, and *speech recording engine*. 
The Speech Recognition API is designed to easily integrate with existing Audio Framework to retrieve the audio stream from a microphone connected to the audio chip.

### 2.6.1 Speech Recognition Interface

Setting up the speech recognition application to detect a wakeup word may be done using series of Audio Elements linked into a pipeline shown below.

![Fig. 5: Sample Speech Recognition Pipeline](image)

Configuration and use of particular elements is demonstrated in several examples linked to elsewhere in this documentation. What may need clarification is use of the **Filter** and the **RAW stream**. The filter is used to adjust the sample rate of the I2S stream to match the sample rate of the speech recognition model. The RAW stream is the way to feed the audio input to the model.

A code snippet below demonstrates how to initialize the model, determine the number of samples and the sample rate of voice data to feed to the model, and detect the wakeup word.

```c
#include "esp_wn_iface.h"
#include "esp_wn_models.h"
#include "rec_eng_helper.h"

esp_wn_iface_t *wakenet;
model_coeff_getter_t *model_coeff_getter;
model_iface_data_t *model_data;

// Initialize wakeNet model data
get_wakenet_iface(&wakenet);
get_wakenet_coeff(&model_coeff_getter);
model_data = wakenet->create(model_coeff_getter, DET_MODE_90);

// Set parameters of buffer
int audio_chunksize = wakenet->get_samp_chunksize(model_data);
int frequency = wakenet->get_samp_rate(model_data);
int16_t *buffer = malloc(audio_chunksize * sizeof(int16_t));

// Get voice data feed to buffer
...

// Detect
int r = wakenet->detect(model_data, buffer);
if (r > 0) {
   printf("Detection triggered output %d\n", r);
}

// Destroy model
wakenet->destroy(model_data)
```

2.6. Speech Recognition
Application Example

Implementation of the speech recognition API is demonstrated in `speech_recognition/asr` example.

API Reference

For the latest API reference please refer to Espressif Speech recognition repository.

2.6.2 Voice Activity Detection

Voice activity detection (VAD) is a technique used in speech processing to detect the presence (or absence) of human speech. Detection of somebody speaking may be used to activate some processes, e.g. automatically switch on voice recording. It may be also used to deactivate processes, e.g. stop coding and transmission of silence packets to save on computation and network bandwidth.

Provided in this section API implements VAD functionality together with couple of options to configure sensitivity of speech detection, set sample rate or duration of audio samples.

Application Example

Implementation of the voice activity detection API is demonstrated in `speech_recognition/vad` example.

API Reference

For the latest API reference please refer to Espressif Speech recognition repository.

2.6.3 Recorder Engine

The Recorder Engine API is a set of functions to facilitate voice recording. The API is integrated with Voice Activity Detection, providing options to enable and disable VAD to control the incoming audio stream. The Recorder Engine also includes possibility to encode the audio stream using AMR or AMRWB formats.

API Reference

Header File

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

Functions

```c
esp_err_t rec_engine_create(rec_config_t *cfg)
```

Create recorder engine according to parameters.

**Note** Sample rate is 16k, 1 channel, 16bits, by default. Upon completion of this function rec_open callback will be triggered.

**Return**

- 0: Success
• -1: Error

Parameters
  • cfg: See `rec_config_t` structure for additional details

```
int rec_engine_data_read(uint8_t *buffer, int buffer_size, int waiting_time)
```
Read voice data after REC_EVENT_VAD_START.

Return
  • -2: timeout of read
  • -1: parameters invalid or task not running.
  • 0: last voice block.
  • others: voice block index.

Parameters
  • buffer: data pointer
  • buffer_size: Size of buffer, must be equal to REC_ONE_BLOCK_SIZE.
  • waiting_time: Timeout for reading data. Default time of REC_ONE_BLOCK_SIZE is 100ms, larger than 100ms is recommended.

```
esp_err_t rec_engine_detect_suspend(rec_voice_suspend_t flag)
```
Suspend or enable voice detection by vad.

Return
  • 0: Success
  • -1: Error

Parameters
  • flag: REC_VOICE_SUSPEND_ON: Voice detection is suspended
        REC_VOICE_SUSPEND_OFF: Voice detection is not suspended

```
esp_err_t rec_engine_trigger_start(void)
```
Start recording by force.

Return
  • 0: Success
  • -1: Error

```
esp_err_t rec_engine_trigger_stop(void)
```
Stop recording by force.

Return
  • 0: Success
  • -1: Error

```
esp_err_t rec_engine_destroy(void)
```
Destroy the recorder engine.

2.6. Speech Recognition
Note Upon completion of this function rec_close callback will be triggered.

Return
- 0: Success
- -1: Error

```c
esp_err_t rec_engine_vad_enable (bool vad_enable)
```
Disable or enable the VAD (voice activity detection).

Note Enable vad by default. Usage: Call this function before rec_engine_trigger_start to disable voice activity detection. Call this function after rec_engine_trigger_stop to enable voice activity detection. Even if disable voice activity detection, the REC_EVENT_VAD_START and REC_EVENT_VAD_STOP events still notified when rec_engine_trigger_start and rec_engine_trigger_stop called.

Return
- 0: Success
- -1: Error

Parameters
- vad_enable: true is enable vad, false disable vad

```c
esp_err_t rec_engine_enc_enable (bool enc_enable)
```
Enable the recorder encoding, or not.

Note support_encoding must be set, rec_engine_enc_enable can be used. Disable encoding by default.

Return
- 0: Success
- -1: Error

Parameters
- enc_enable: true is enable encoding, false is disable.

```c
esp_err_t rec_engine_enc_data_read (uint8_t *buffer, int buffer_size, int waiting_time, int *out_size)
```
Read voice data after REC_EVENT_VAD_START.

Note support_encoding and rec_engine_enc_enable must be set.

Return
- -2: timeout of read
- -1: parameters invalid or not encoding mode.
- 0: success.
- others: voice block index.

Parameters
- buffer: data pointer
- buffer_size: Size of buffer.
- waiting_time: Timeout for reading data.
esp_err_t rec_engine_mute_enable (bool mute_enable)
Enable the recoder mute, or not.

**Note** if enable mute, no data fill the buffer, so the rec_engine_enc_data_read and rec_engine_data_read will be blocked.

**Return**
- 0: Success
- -1: Error

**Parameters**
- mute_enable: true is mute, false is not.

esp_err_t rec_engine_get_wakeup_stat (bool *wakeup_start_t)
Get recorder engine wakeup state.

**Return**
- 0: Success
- -1: Error

**Parameters**
- wakeup_start_t: true is WAKEUP_START, false is not.

**Structures**

```c
struct rec_config_t
```
recorder configuration parameters

**Public Members**

- int one_frame_duration_ms
  Duration of one frame (optional)
- int sensitivity
  For response accuracy rate sensitivity. Default 0: 90%, 1: 95%
- int vad_off_delay_ms
  Vad off delay to stop if no voice is detected
- int wakeup_time_ms
  Time of wakeup
- bool support_encoding
  Support encoding data
- const char *extension
  Encoding format.”amr” or “amrwb” support
- int task_core
  Recorder task running in core (0 or 1)
bool enable_wwe
   Enable Wake Word Engine or not

rec_open open
   Recorder open callback function

rec_fetch fetch
   Recorder fetch data callback function

rec_close close
   Recorder close callback function

rec_callback evt_cb
    Recorder event callback function

void *user_data
   Pointer to user data (optional)

Macros

REC_ONE_BLOCK_SIZE

DEFAULT_REC_ENGINE_CONFIG()

Type Definitions

typedef void (*rec_callback)(rec_event_type_t type, void *user_data)
typedef esp_err_t (*rec_open)(void **handle)
typedef esp_err_t (*rec_fetch)(void *handle, char *data, int data_size)
typedef esp_err_t (*rec_close)(void *handle)

Enumerations

enum rec_event_type_t
   Values:
       REC_EVENT_WAKEUP_START
       REC_EVENT_WAKEUP_END
       REC_EVENT_VAD_START
       REC_EVENT_VAD_STOP

enum rec_voice_suspend_t
   Values:
       REC_VOICE_SUSPEND_OFF
       REC_VOICE_SUSPEND_ON
2.7 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.

2.7.1 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

```c
#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 = DEFAULT_ESP_PERIPH_SET_CONFIG();
    esp_periph_set_handle_t set = esp_periph_set_init(&periph_cfg);
    // Code...
}
```

(continues on next page)
esp_periph_set_register_callback(set, _periph_event_handle, NULL);
// 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(set, button_handle);
esp_periph_start(set, sdcard_handle);
esp_periph_start(set, 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(set, button_handle);
vTaskDelay(10*1000/portTICK_RATE_MS);

// Stop & destroy all peripherals
esp_periph_set_destroy(set);

API Reference

Header File

- esp_peripherals/include/esp_peripherals.h

Functions

```c
esp_periph_set_handle_t esp_periph_set_init(esp_periph_config_t *config)
```
Initialize esp_peripheral sets, create empty peripherals list. Call this function before starting any peripherals (with esp_periph_start). 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 send esp_periph_event_handle_t as *context parameter.

**Return** The peripheral sets instance

**Parameters**
- config: The configurations

esp_err_t esp_periph_set_destroy (esp_periph_set_handle_t periph_set_handle)

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

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- periph_set_handle: The esp_periph_set_handle_t instance

esp_err_t esp_periph_set_stop_all (esp_periph_set_handle_t periph_set_handle)

Stop monitoring all peripherals, the peripheral state is still kept. This function only temporary disables the peripheral.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- periph_set_handle: The esp_periph_set_handle_t instance

esp_periph_handle_t esp_periph_set_get_by_id (esp_periph_set_handle_t periph_set_handle, int peripheral_id)

Get the peripheral handle by Peripheral ID.

**Return** The esp_periph_handle_t

**Parameters**
- periph_set_handle: The esp_periph_set_handle_t instance
- peripheral_id: as esp_periph_id_t, or any ID you use when calling esp_periph_create

audio_event_iface_handle_t esp_periph_set_get_event_iface (esp_periph_set_handle_t periph_set_handle)

Return the event_iface used by this esp_peripherals.

**Return** The audio event iface handle

**Parameters**
- periph_set_handle: The esp_periph_set_handle_t instance
esp_err_t esp_periph_set_register_callback(
    esp_periph_set_handle_t periph_set_handle,
    esp_periph_event_handle_t cb,
    void *user_context)

Register peripheral sets event callback function.

Return
• ESP_OK
• ESP_FAIL

Parameters
• periph_set_handle: The esp_periph_set_handle_t instance
• cb: The event handle callback function
• user_context: The user context pointer

QueueHandle_t esp_periph_set_get_queue(esp_periph_set_handle_t periph_set_handle)
Peripheral is using event_iface to control the event, all events are send out to event_iface queue. This function will be useful in case we want to read events directly from the event_iface queue.

Return The queue handle

Parameters
• periph_set_handle: The esp_periph_set_handle_t instance

esp_err_t esp_periph_set_list_init(esp_periph_set_handle_t periph_set_handle)
Call this function to initialize all the listed peripherals.

Note Work with no task peripheral set only

Return
• ESP_OK
• ESP_FAIL

Parameters
• periph_set_handle: The esp_periph_set_handle_t instance

esp_err_t esp_periph_set_list_run(esp_periph_set_handle_t periph_set_handle, audio_event_iface_msg_t msg)
Call this function to run all the listed peripherals.

Note Work with no task peripheral set only

Return
• ESP_OK
• ESP_FAIL

Parameters
• periph_set_handle: The esp_periph_set_handle_t instance
• msg: The audio_event_iface_msg_t handle message

esp_err_t esp_periph_set_list_destroy(esp_periph_set_handle_t periph_set_handle)
Call this function to destroy all the listed peripherals.
Note Work with no task peripheral set only

Return

• ESP_OK
• ESP_FAIL

Parameters

• periph_set_handle: The esp_periph_set_handle_t instance

`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_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 operations from initialization, execution of commands to destroying the peripheral. These operations are represented by functions passed as call parameters to this function.

Return

• ESP_OK
• ESP_FAIL

Parameters

• periph: The periph
• init: The initialize
• run: The run
• destroy: The destroy

`esp_periph_start` (esp_periph_set_handle_t periph_set_handle, esp_periph_handle_t periph)

Add the peripheral to peripherals list, enable and start monitor task (if task stack size > 0)

Note This peripheral must be first created by calling `esp_periph_create`

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• periph_set_handle: The esp_periph_set_handle_t instance
• periph: The peripheral instance

`esp_periph_stop` (esp_periph_handle_t periph)

Stop monitoring the peripheral, the peripheral state is still kept. This function only temporarily disables the peripheral.
Return

- ESP_OK
- ESP_FAIL

Parameters

- periph: The peripheral

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 peripheral
- 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 peripheral
- cmd: The command
- data: The data
- data_len: The data length

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 peripheral
- event_id: The event identifier
• data: The data
• data_len: The data length

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 peripheral
• 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 peripheral

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 peripheral
• data: The data

void *esp_periph_get_data (esp_periph_handle_t periph)

Get the user data stored in the peripheral.
Return Peripheral data pointer

Parameters

- periph: The peripheral

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

Return The peripheral working state

Parameters

- periph: The handle of peripheral

`esp_periph_id_t esp_periph_get_id(esp_periph_handle_t periph)`
Get Peripheral identifier.

Return The peripheral identifier

Parameters

- periph: The peripheral

`esp_err_t esp_periph_set_id(esp_periph_handle_t periph, esp_periph_id_t periph_id)`
Set Peripheral identifier.

Return

- ESP_OK
- ESP_FAIL

Parameters

- periph: The peripheral
- periph_id: The peripheral identifier

`esp_err_t esp_periph_init(esp_periph_handle_t periph)`
Call this to execute `init` function of peripheral instance.

Return

- ESP_OK
- ESP_FAIL

Parameters

- periph: The peripheral handle

`esp_err_t esp_periph_run(esp_periph_handle_t periph)`
Call this to execute `run` function of peripheral instance.

Return

- ESP_OK
- ESP_FAIL

Parameters
• periph: The peripheral handle

esp_err_t esp_periph_destroy (esp_periph_handle_t periph)
Call this to execute destroy function of peripheral instance.

Return
• ESP_OK
• ESP_FAIL

Parameters
• periph: The peripheral handle

esp_err_t esp_periph_register_on_events (esp_periph_handle_t periph, esp_periph_event_t *evts)
Rigster peripheral on event handle.

Return
• ESP_OK
• ESP_FAIL

Parameters
• periph: The peripheral handle
• evts: The esp_periph_event_t handle

Structures

struct esp_periph_config_t
Common peripherals configurations.

Public Members

int task_stack
>0 Service task stack size; =0 without task created

int task_prio
Service task priority (based on freeRTOS priority)

int task_core
Service task running in core (0 or 1)

struct esp_periph_event
peripheral events

Public Members

void *user_ctx
peripheral context data

esp_periph_event_handle_t cb
peripheral callback function

audio_event_iface_handle_t iface
peripheral event

2.7. Peripherals
Macros

DEFAULT_ESP_PERIPH_STACK_SIZE
DEFAULT_ESP_PERIPH_TASK_Prio
DEFAULT_ESP_PERIPH_TASK_CORE
DEFAULT_ESP_PERIPH_SET_CONFIG()
periph_tick_get

Type Definitions

typedef struct esp_periph_sets *esp_periph_set_handle_t
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)
typedef struct esp_periph_event esp_periph_event_t

Peripheral events

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
PERIPH_ID_GPIO_ISR = AUDIO_ELEMENT_TYPE_PERIPH + 14
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.

2.7.2 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)
2.7.3 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_sdcardUnmount()`. 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

```c
int card_detect_pin
   Card detect gpio number

const char *root
   Base path for vfs
```

Enumerations

```c
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
```

2.7.4 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

```c
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**

```c
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**

```c
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
2.7.5 Console Peripheral

Console Peripheral is used to control the Audio application from the terminal screen. It provides 2 ways to 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

```c
#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_cfg_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,
    };
}
```

(continues on next page)
};
    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

custom_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

int buffer_size
    Size of console input buffer

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_BUFFER_SIZE
CONSOLE_DEFAULT_PROMPT_STRING
```

Type Definitions

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

2.7.6 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

### 2.7.7 Button Peripheral

To control application flow you may use buttons connected and read through the ESP32 GPIOs. This API provides functions to initialize specific 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**

```c
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**

```c
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

2.7.8 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

```c
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)
```

Blink 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

```c
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

```c
struct periph_led_cfg_t
```

The LED peripheral configuration.

Public Members

```c
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)

int **gpio_num**
   Optional, < 0 invalid gpio number

Enumerations

e num **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

2.7.9 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**
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()
ESP32 ADC1 channels and GPIO table ADC1_CHANNEL_0 - GPIO36 ADC1_CHANNEL_1 -
GPIO37 ADC1_CHANNEL_2 - GPIO38 ADC1_CHANNEL_3 - GPIO39 ADC1_CHANNEL_4 - GPIO32
ADC1_CHANNEL_5 - GPIO33 ADC1_CHANNEL_6 - GPIO34 ADC1_CHANNEL_7 - GPIO35

Enumerations

class periph_adc_button_event_id_t
Values:

PERIPH_ADC_BUTTON_IDLE = 0
PERIPH_ADC_BUTTON_PRESSED
PERIPH_ADC_BUTTON_RELEASE
PERIPH_ADC_BUTTON_LONG_PRESSED
PERIPH_ADC_BUTTON_LONG_RELEASE

2.7.10 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)`

Initialize the is31fl3216.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

**Parameters**

- `is31fl3216_config`

`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

`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

```c
esp_err_t periph_is31fl3216_set_duty_step(esp_periph_handle_t periph, uint8_t step)
```  
Set the duty step of flash.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- **periph**: The is31fl3216 handle
- **step**: The step of flash

```c
esp_err_t periph_is31fl3216_set_interval(esp_periph_handle_t periph, uint16_t interval_ms)
```  
Set the interval time.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- **periph**: The is31fl3216 handle
- **interval_ms**: Time of interval

```c
esp_err_t periph_is31fl3216_set_shift_mode(esp_periph_handle_t periph, periph_is31_shift_mode_t mode)
```  
Set the shift mode.

**Return**

- ESP_OK Success
- ESP_FAIL Fail

Parameters

- **periph**: The is31fl3216 handle
- **mode**: Mode of periph_is31_shift_mode_t

```c
esp_err_t periph_is31fl3216_set_light_on_num(esp_periph_handle_t periph, uint16_t light_on_num, uint16_t max_light_num)
```  
Set the light on numbers.

**Return**

- ESP_OK Success
- ESP_FAIL Fail
esp_err_t `periph_is31fl3216_set_act_time` *(esp_periph_handle_t periph, uint16_t act_ms)*
Set the action time.

**Return**
- ESP_OK Success
- ESP_FAIL Fail

**Parameters**
- `periph`: The is31fl3216 handle
- `act_ms`: Action time, unit is millisecond, 0 is infinite

**Structures**

```c
struct periph_is31fl3216_cfg_t
    The configuration of is31fl3216.
```

**Public Members**

```c
    uint32_t duties[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**

```c
    IS31FL3216_CH_NUM

    BLUE_LED_MAX_NUM
```

**Enumerations**

```c
enum periph_is31fl3216_state_t
    Values:
        IS31FL3216_STATE_UNKNOWN
        IS31FL3216_STATE_OFF
        IS31FL3216_STATE_ON
        IS31FL3216_STATE_FLASH
        IS31FL3216_STATE_BY_AUDIO
```
**IS31FL3216_STATE_SHIFT**

enum **periph_is31_shift_mode_t**

Values:

- **PERIPH_IS31_SHIFT_MODE_UNKNOWN**
- **PERIPH_IS31_SHIFT_MODE_ACC**
  accumulation mode
- **PERIPH_IS31_SHIFT_MODE_SINGLE**

<table>
<thead>
<tr>
<th>Name of Peripheral</th>
<th>ESP32-LyraT</th>
<th>ESP32-LyraTD-MSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wi-Fi</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SD Card</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Spiffs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Console</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Touch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Button</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LED</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ADC Button</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LED Controller</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### 2.8 Abstraction Layer

#### 2.8.1 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. 6: 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 block_size, int n_blocks)`
Create ringbuffer with total size = block_size * n_blocks.

Return `ringbuf_handle_t`

Parameters

• `block_size`: Size of each block
• `n_blocks`: Number of blocks

`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

```c
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

```c
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

```c
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 `ticks_to_wait` ticks until enough bytes to read if the ringbuffer bytes available is less than `len`. If `buf` argument provided is NULL, then ringbuffer do pseudo reads by simply advancing pointers.

**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

```c
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 `ticks_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

```c
esp_err_t rb_done_write (ringbuf_handle_t rb)
```
Set status of writing to ringbuffer is done.

**Return**
Read the Docs Template Documentation

• ESP_OK
• ESP_FAIL

Parameters
• rb: The Ringbuffer handle

esp_err_t rb_unblock_reader(ringbuf_handle_t rb)
  Unblock from rb_read.

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

2.8.2 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, audio_hal_func_t *audio_hal_func)
Initialize media codec driver.

Note If selected codec has already been installed, it’ll return the audio_hal handle.

Return int, 0 success, others fail

Parameters

• audio_hal_conf: Configure structure audio_hal_config_t
• audio_hal_func: Structure containing functions used to operate audio the codec chip

esp_err_t audio_hal_deinit (audio_hal_handle_t audio_hal)
Uninitialize media codec driver.

Return int, 0 success, others fail

Parameters

• audio_hal: reference function pointer for selected audio codec

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, 0 success, others fail

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_mute (audio_hal_handle_t audio_hal, bool mute)
Set voice mute. Enables or disables DAC mute of a codec.

2.8. Abstraction Layer
Note audio_hal_get_volume will still give a non-zero number in mute state. It will be set to that number when speaker is unmuted.

Return int, 0 success, others fail

Parameters
  • audio_hal: reference function pointer for selected audio codec
  • mute: true/false. If true speaker will be muted and if false speaker will be unmuted.

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, 0 success, others fail

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, 0 success, others fail

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

struct audio_hal
Configuration of functions and variables used to operate audio codec chip.

Public Members

esp_err_t (*audio_codec_initialize)(audio_hal_codec_config_t *codec_cfg)
initialize codec

esp_err_t (*audio_codec_deinitialize)(void)
deinitialize codec

esp_err_t (*audio_codec_ctrl)(audio_hal_codec_mode_t mode, audio_hal_ctrl_t ctrl_state)
control codec mode and state

esp_err_t (*audio_codec_config_iface)(audio_hal_codec_mode_t mode, audio_hal_codec_i2s_iface_t *iface)
configure i2s interface

esp_err_t (*audio_codec_set_mute)(bool mute)
set codec mute

esp_err_t (*audio_codec_set_volume)(int volume)
set codec volume

esp_err_t (*audio_codec_get_volume)(int *volume)
get codec volume

xSemaphoreHandle audio_hal_lock
semaphore of codec

void *handle
handle of audio codec

Macros

AUDIO_HAL_VOL_DEFAULT

Type Definitions

typedef struct audio_hal *audio_hal_handle_t

typedef struct audio_hal audio_hal_func_t
Configuration of functions and variables used to operate audio codec chip.
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:
\texttt{AUDIO\_HAL\_MODE\_SLAVE} = 0x00
set slave mode

\texttt{AUDIO\_HAL\_MODE\_MASTER} = 0x01
set master mode

\texttt{enum audio\_hal\_iface\_samples\_t}
Select I2S interface samples per second.
\texttt{Values:}

\texttt{AUDIO\_HAL\_08K\_SAMPLES}
set to 8k samples per second

\texttt{AUDIO\_HAL\_11K\_SAMPLES}
set to 11.025k samples per second

\texttt{AUDIO\_HAL\_16K\_SAMPLES}
set to 16k samples in per second

\texttt{AUDIO\_HAL\_22K\_SAMPLES}
set to 22.050k samples per second

\texttt{AUDIO\_HAL\_24K\_SAMPLES}
set to 24k samples in per second

\texttt{AUDIO\_HAL\_32K\_SAMPLES}
set to 32k samples in per second

\texttt{AUDIO\_HAL\_44K\_SAMPLES}
set to 44.1k samples per second

\texttt{AUDIO\_HAL\_48K\_SAMPLES}
set to 48k samples per second

\texttt{enum audio\_hal\_iface\_bits\_t}
Select I2S interface number of bits per sample.
\texttt{Values:}

\texttt{AUDIO\_HAL\_BIT\_LENGTH\_16BITS} = 1
set 16 bits per sample

\texttt{AUDIO\_HAL\_BIT\_LENGTH\_24BITS}
set 24 bits per sample

\texttt{AUDIO\_HAL\_BIT\_LENGTH\_32BITS}
set 32 bits per sample

\texttt{enum audio\_hal\_iface\_format\_t}
Select I2S interface format for audio codec chip.
\texttt{Values:}

\texttt{AUDIO\_HAL\_I2S\_NORMAL} = 0
set normal I2S format

\texttt{AUDIO\_HAL\_I2S\_LEFT}
set all left format

\texttt{AUDIO\_HAL\_I2S\_RIGHT}
set all right format

\texttt{AUDIO\_HAL\_I2S\_DSP}
set dsp/pcm format
2.8.3 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

```c
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

```c
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

```c
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

```c
esp_err_t es8388_set_voice_volume(int volume)
```
Set voice volume.

**Return**
- ESP_OK
- ESP_FAIL

**Parameters**
- `volume`: voice volume (0~100)

```c
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(bool enable)`  
Configure ES8388 DAC mute or not. Basically you can use this function to mute the output or unmute.

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
  • ESP_FAIL Parameter error
  • ESP_OK Success

`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

```c
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

```c
void es8388_read_all()
```
Print all ES8388 registers.

**Return**
- void

```c
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

```c
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

```c
void es8388_pa_power(bool enable)
```
Set ES8388 PA power.

**Return**
- void
Parameters

- enable: true for enable PA power, false for disable PA power

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
2.8.4 ES8374 Driver

Driver for ES8374 codec chip.

API Reference

Header File

- audio_hal/driver/es8374/es8374.h

Functions

esp_err_t es8374_codec_init (audio_hal_codec_config_t *cfg)

Initialize ES8374 codec chip.

Return
• ESP_OK
• ESP_FAIL

Parameters
• \texttt{cfg}: configuration of ES8374

\texttt{esp_err_t es8374_codec_deinit (void)}
Deinitialize ES8374 codec chip.

Return
• ESP_OK
• ESP_FAIL

\texttt{esp_err_t es8374_config_fmt (es\_module\_t \texttt{mode}, es\_i2s\_fmt\_t \texttt{fmt})}
Configure ES8374 I2S format.

Return
• ESP_OK
• ESP_FAIL

Parameters
• \texttt{mode}: set ADC or DAC or both
• \texttt{fmt}: ES8374 I2S format

\texttt{esp_err_t es8374_i2s_config_clock (es\_i2s\_clock\_t \texttt{cfg})}
Configure I2S clock in MSATER mode.

Return
• ESP_OK
• ESP_FAIL

Parameters
• \texttt{cfg}: set bits clock and WS clock

\texttt{esp_err_t es8374_set_bits_per_sample (es\_module\_t \texttt{mode}, es\_bits\_length\_t \texttt{bit\_per\_sample})}
Configure ES8374 data sample bits.

Return
• ESP_OK
• ESP_FAIL

Parameters
• \texttt{mode}: set ADC or DAC or both
• \texttt{bit\_per\_sample}: bit number of per sample

\texttt{esp_err_t es8374_start (es\_module\_t \texttt{mode})}
Start ES8374 codec chip.
Return

- ESP_OK
- ESP_FAIL

Parameters

- `mode`: set ADC or DAC or both

```c
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

```c
esp_err_t es8374_codec_set_voice_volume(int volume)
```
Set voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `volume`: voice volume (0~100)

```c
esp_err_t es8374_codec_get_voice_volume(int *volume)
```
Get voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `volume`: voice volume (0~100)

```c
esp_err_t es8374_set_voice_mute(bool 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)

```c
esp_err_t es8374_get_voice_mute(void)
```
Get ES8374 DAC mute status.

2.8. Abstraction Layer
```c
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.
```
es8374_codec_config_i2s(mode, *iface)
Configure ES8374 codec mode and I2S interface.

Return
• ESP_FAIL Parameter error
• ESP_OK Success

Parameters
• mode: codec mode
• iface: I2S config

es8374_codec_ctrl_state(mode, 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(enable)
Set ES8374 PA power.

Return
• void

Parameters
• enable: true for enable PA power, false for disable PA power

Macros

ES8374_ADDR

2.8.5 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_codec_init(audio_hal_codec_config_t *cfg)
Initialize ZL38063 chip.

    Return
    • ESP_OK
    • ESP_FAIL

    Parameters
    • cfg: configuration of ZL38063

esp_err_t zl38063_codec_deinit(void)
Deinitialize ZL38063 chip.

    Return
    • ESP_OK
    • ESP_FAIL

esp_err_t zl38063_codec_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_codec_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

```c
esp_err_t zl38063_codec_set_voice_mute(bool mute)
```

mute or unmute the codec

Return

- ESP_OK
- ESP_FAIL

Parameters

- mute: true, false

```c
esp_err_t zl38063_codec_set_voice_volume(int volume)
```

Set voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- volume: voice volume (0~100)

```c
esp_err_t zl38063_codec_get_voice_volume(int *volume)
```

Get voice volume.

Return

- ESP_OK
- ESP_FAIL

Parameters

- *volume: voice volume (0~100)

## 2.9 Configuration Options

Compile-time configuration options specific to ESP-ADF.

### 2.9.1 Recorder Engine Configuration

**RECORD_ENGINE_MODE**

Choose recorder engine functionality

*Found in: Recorder Engine Configuration*

Recorder engine have VAD, WWE and AMR encoding functionality. AMR encoding enabled, the binary size increase 144kB. WWE enabled, the binary size increase 103kB.
Available options:

- REC_ENG_ENABLE_VAD_ONLY
- REC_ENG_ENABLE_VAD_WWE
- REC_ENG_ENABLE_VAD_WWE_AMR

### 2.9.2 Audio HAL

**AUDIO_BOARD**

Audio board

*Found in: Audio HAL*

Select an audio board to use with the ESP-ADF

Available options:

- AUDIO_BOARD_CUSTOM
- ESP_LYRAT_V4_3_BOARD
- ESP_LYRAT_V4_2_BOARD
- ESP_LYRATD_MSC_V2_1_BOARD
- ESP_LYRATD_MSC_V2_2_BOARD
- ESP_LYRAT_MINI_V1_1_BOARD
- ESP32_KORVO_DU1906_BOARD
- ESP32_S2_KALUGA_V1_BOARD

**ESP32_KORVO_DU1906_DAC**

ESP32 KORVO DU1906 Board DAC chip

*Found in: Audio HAL*

Select DAC chip to use on ESP32_KORVO_DU1906 board

Available options:

- ESP32_KORVO_DU1906_DAC_TAS5805M
- ESP32_KORVO_DU1906_DAC_ES7148

**ESP32_KORVO_DU1906_ADC**

ESP32 KORVO DU1906 Board ADC chip

*Found in: Audio HAL*

Select ADC chip to use on ESP32_KORVO_DU1906 board

Available options:

- ESP32_KORVO_DU1906_ADC_ES7243
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 quality 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 minimum that could communicate with a cloud service.

![Fig. 1: Audio Project Example - Send Voice Commands to Cloud Service](image)

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.
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
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 documentation 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 menuconfig 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
  ```c
  char *buf = heap_caps_malloc(1024 * 10, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT)
  ```
  instead of `malloc(1024 * 10)` to use PSRAM, and call

  ```c
  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/id/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.

<table>
<thead>
<tr>
<th>Component</th>
<th>Internal RAM Required</th>
<th>With PSRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PSRAM not used</td>
<td>With PSRAM</td>
</tr>
<tr>
<td>Wi-Fi</td>
<td>50kB+</td>
<td>50kB+</td>
</tr>
<tr>
<td>Bluetooth</td>
<td>140kB (50kB if only BLE needed)</td>
<td>95kB (50kB if only BLE needed)</td>
</tr>
<tr>
<td>Flash Card</td>
<td>12kB+</td>
<td>12kB+</td>
</tr>
<tr>
<td>I2S</td>
<td>Configurable, 8kB for reference</td>
<td>Configurable, 8kB for reference</td>
</tr>
<tr>
<td>RingBuffer</td>
<td>Configurable, 30kB for reference</td>
<td>0kB, all moved into PSRAM</td>
</tr>
</tbody>
</table>

**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**

```c
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**

```c
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 from evt1");
while (audio_event_listen(evt1) == ESP_OK);
```

#### Audio Pipeline

**Example**

```c
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

```c
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_periph_stop(button_handle);
ESP_LOGI(TAG, "running...");
vTaskDelay(5000 / portTICK_RATE_MS);

esp_periph_start(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);
```

(continues on previous page)
ESP_LOGI(TAG, "destroy...");
esp_periph_destroy();

Audio Player

Example

```c
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 user for receiving esp_audio event */
    .resample_rate = 48000,       /*!< sample rate */
    .hal = NULL,                  /*!< */
};
```

Audio Player Example

```c
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);
```

```c
raw_stream_cfg_t raw_cfg = {
    .type = AUDIO_STREAM_READER,
};
```

Audio Player Example

```c
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);
```

```c
fatfs_stream_cfg_t fatfs_cfg = {
    .type = AUDIO_STREAM_READER,
    .root_path = "/sdcard",
};
```

Audio Player Example

```c
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
esp_audio_input_stream_add(player, fatfs_stream_init(&fatfs_cfg));
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-Mini V1.2 Hardware Reference

This guide provides functional descriptions, configuration options for ESP32-LyraT-Mini V1.2 audio development board. As an introduction to functionality and using the LyraT, please see ESP32-LyraT-Mini V1.2 Getting Started Guide.
Overview

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, connected smart-home appliances with one or more audio functionality, etc. The block diagram below presents main components of the ESP32-LyraT-Mini.

Functional Description

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT-Mini board. The list provides description starting from the picture’s top right corner and going clockwise.

**MicroSD Card** The development board supports a MicroSD card in SPI/1-bit modes, and can store or play audio files in the MicroSD card. See MicroSD Card for pinout details.

**Microphone** On-board microphone connected to AINRP/AINRP of the Audio ADC Chip.

**System LEDs** Two general purpose LEDs (green and red) controlled by ESP32-WROVER-B Module to indicate certain operation states of the audio application using dedicated API.

**Audio Codec Chip** The audio codec chip, ES8311, is a low power mono audio codec. It consists of 1-channel ADC, 1-channel DAC, low noise pre-amplifier, headphone driver, digital sound effects, analog mixing and gain functions. It is interfaced with ESP32-WROVER-B Module over I2S and I2C buses to provide audio processing in hardware independently from the audio application.

**Audio Output** Output socket to connect headphones with a 3.5 mm stereo jack. One of the socket’s terminals is wired to ESP32 to provide jack insertion detection.

**Audio ADC Chip** The audio codec chip, ES7243, is a low power multi-bit delta-sigma audio ADC and DAC. In this board this chip is used as the microphone interface.

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

**Speaker Output** Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1” pitch.
Audio Function Press Keys  Six press keys labeled Rec, Mode, Play, Set, Vol- and Vol+. They are routed to ESP32-WROVER-B Module and intended for development and testing of a UI for audio applications using dedicated API.

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.

Automatic Upload  A simple two transistor circuit to put ESP32 into firmware upload mode depending on the status of UART DTR and RTS signals. The signals are controlled by an external application to upload the firmware over the USB-UART interface.

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

USB-UART Bridge Chip  A single chip USB-UART bridge CP2102N provides up to 3 Mbps transfers rates.

Standby / Charging LEDs  The Standby green LED indicates that power has been applied to the USB Power 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 and constant voltage linear charger for single cell lithium-ion batteries AP5056. Used for charging of a battery connected to the Battery Socket over the USB Power Port.

Power On Switch  Power on/off knob: toggling it to the top powers the board on; toggling it to the down powers the
Fig. 5: ESP32 LyraT-Mini V1.2 Board Layout
board off.

**Note:** The **Power On Switch** does not affect / disconnect the Li-ion battery charging. More information, you can refer to ESP32-LyraT-Mini V1.2 schematic (PDF).

**Power Supervisor**  Provides EN signal to enable ESP32 once power supply voltage stabilizes.

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

**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 64 Mbit SPI flash and 64 Mbit PSRAM for flexible data storage.

**UART Test Point**  Serial port: provides access to the serial TX/RX signals between **ESP32-WROVER-B Module** and **USB-UART Bridge Chip**. See **UART Test Point** for pinout details.

**JTAG Test Point**  Provides access to the **JTAG** interface of **ESP32-WROVER-B Module**. It may be used for debugging, application upload, as well as implementing several other functions, e.g., Application Level Tracing. See **JTAG Test Point** for pinout details.

### Allocation of ESP32 Pins to Test Points

This section describes allocation of test points available on the ESP32-LyraT-Mini board. The test points are bare through hole solder pads and have standard 2.54 mm / 0.1 inch pitch. User may need to populate them with pin headers or sockets for easy connection of external hardware.

#### JTAG Test Point

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTDO / GPIO15</td>
<td>TDO</td>
</tr>
<tr>
<td>MTCK / GPIO15</td>
<td>TCK</td>
</tr>
<tr>
<td>MTDI / GPIO15</td>
<td>TDI</td>
</tr>
<tr>
<td>MTMS / GPIO15</td>
<td>TMS</td>
</tr>
</tbody>
</table>

#### UART Test Point

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Pin Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RXD0</td>
<td>RX</td>
</tr>
<tr>
<td>TXD0</td>
<td>TX</td>
</tr>
<tr>
<td>GND</td>
<td>GND</td>
</tr>
<tr>
<td>n/a</td>
<td>3.3 V</td>
</tr>
</tbody>
</table>

**MicroSD Card**

Implemented on this board MicoSD card interface operates in SPI/1-bit mode. The board is able to support SPI/4-bit mode after populating couple of additional components on locations reserved on the PCB. See ESP32-LyraT-Mini V1.2 schematic (PDF) for additional information. Not populated components are marked *(NC)* on the schematic.
### ESP32 Pin allocation for MicroSD Signal

<table>
<thead>
<tr>
<th>Pin</th>
<th>ESP32 Pin</th>
<th>MicroSD Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MTDI / GPIO12</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>MTCK / GPIO13</td>
<td>–</td>
</tr>
<tr>
<td>3</td>
<td>MTDO / GPIO15</td>
<td>CMD</td>
</tr>
<tr>
<td>4</td>
<td>MTMS / GPIO14</td>
<td>CLK</td>
</tr>
<tr>
<td>5</td>
<td>GPIO2</td>
<td>DATA0</td>
</tr>
<tr>
<td>6</td>
<td>GPIO4</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>GPIO34</td>
<td>CD</td>
</tr>
</tbody>
</table>

### GPIO Allocation Summary

The table below provides allocation of GPIOs exposed on terminals of **ESP32-WROVER-B Module** to control specific components or functions of the board.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Pin Name</th>
<th>ES8311</th>
<th>ES7243</th>
<th>Keys</th>
<th>MicroSD</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>EN</td>
<td>–</td>
<td>–</td>
<td>EN_KEY</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>S_VP</td>
<td>–</td>
<td>–</td>
<td>I2S_DATA</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>S_VN</td>
<td>–</td>
<td>–</td>
<td>REC, MODE, PLAY, SET, VOL-, VOL+</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>IO34</td>
<td>–</td>
<td>–</td>
<td>CD</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>IO35</td>
<td>I2S0_ASDOUT</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>IO32</td>
<td>I2S1_SCLK</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>IO33</td>
<td>I2S1_LRCK</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>IO25</td>
<td>I2S0_LRCK</td>
<td>I2S0_DSDIN</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>IO26</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>IO27</td>
<td>–</td>
<td>–</td>
<td>Blue_LED</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>13</td>
<td>IO14</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>14</td>
<td>IO12</td>
<td>–</td>
<td>–</td>
<td>NC (DATA2)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>15</td>
<td>IO13</td>
<td>–</td>
<td>–</td>
<td>NC (DATA3)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>17</td>
<td>SD2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>18</td>
<td>SD3</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>19</td>
<td>CMD</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>20</td>
<td>CLK</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>21</td>
<td>SD0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>22</td>
<td>SD1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>23</td>
<td>IO15</td>
<td>–</td>
<td>–</td>
<td>CMD</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>24</td>
<td>IO2</td>
<td>IO2_KEY</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>25</td>
<td>IO0</td>
<td>I2S0_MCLK</td>
<td>I2S1_MCLK</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>26</td>
<td>IO4</td>
<td>I00_KEY</td>
<td>–</td>
<td>DATA0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>27</td>
<td>NC (IO16)</td>
<td>–</td>
<td>–</td>
<td>NC (DATA1)</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>28</td>
<td>NC (IO17)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>29</td>
<td>IO5</td>
<td>I2S0_SCLK</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>30</td>
<td>IO18</td>
<td>I2C_SDA</td>
<td>I2C_SDA</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>31</td>
<td>IO19</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>32</td>
<td>IO21</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>33</td>
<td>RXD0</td>
<td>–</td>
<td>–</td>
<td>RXD0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>34</td>
<td>TXD0</td>
<td>–</td>
<td>–</td>
<td>TXD0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>35</td>
<td>IO22</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>36</td>
<td>IO23</td>
<td>I2C_SCK</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

1. **Pin** - ESP32-WROVER-B module pin number, GND and power supply pins are not listed
2. **PJ_DET** - phone jack insertion detect signal
3. **PA_CTRL** - NS4150 power amplifier chip control signal
4. **RXD0, TXD0** - serial communication signals connected to TXD and RXD pins of CP2102N USB-UART bridge
5. **NC** - not connected

**Notes on Power Distribution**

The ESP32-LyraT-Mini board provides some basic features to isolate noise from digital components by providing separate power distribution for audio and digital subsystems.

**Power Supply over USB and from Battery**

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.

---

![Power System diagram](image)

*Fig. 6: ESP32-LyraT-Mini V1.2 - Dedicated USB Power Supply Socket*
Independent Audio and Digital Power Supply

The board features independent power supplies to the audio components and ESP32 module. This should reduce noise in the audio signal from digital components and improve overall performance of the components.

Selecting of the Audio Output

The board provides a mono audio output signal on pins OUTN and OUTP of the ES8311 codec chip. The signal is routed to two outputs:

- power amplifier (PA) to drive an external speaker,
- phone jack socket to drive external headphones.

The board design assumes that selection between one of these outputs is implemented in software, as opposed to using traditional mechanical contacts in a phone jack socket, that would disconnect the speaker once a headphone jack is inserted.

Two digital IO signals are provided to implement selection between the speaker and the headphones:

- **PJ_DET** - digital input signal to detect when a headphone jack is inserted,
- **PA_CTRL** - digital output signal to enable or disable the amplifier IC.

The application running on ESP32 may then enable of disable the PA with **PA_CTRL** basing on status of **PJ_DET**. Please see **GPIO Allocation Summary** for specific GPIO numbers allocated to these signals.
3.4. Development Boards

Fig. 8: ESP32-LyraT-Mini V1.2 - Digital Power Supply

Fig. 9: ESP32-LyraT-Mini V1.2 - Audio Power Supply
3.4.2 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.

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 functional-
The block diagram below presents main components of the ESP32-LyraT.

![ESP32-LyraT V4.3 Electrical Block Diagram](image)

**Fig. 10: ESP32-LyraT V4.3 Electrical Block Diagram**

### 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.

- **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.

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

---

Fig. 11: ESP32-LyraT V4.3 Board Layout
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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
</tr>
<tr>
<td>2</td>
<td>ON</td>
</tr>
<tr>
<td>3</td>
<td>OFF</td>
</tr>
<tr>
<td>4</td>
<td>OFF</td>
</tr>
<tr>
<td>5</td>
<td>OFF</td>
</tr>
<tr>
<td>6</td>
<td>OFF</td>
</tr>
<tr>
<td>7</td>
<td>OFF</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>DIP SW</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OFF</td>
</tr>
<tr>
<td>2</td>
<td>OFF</td>
</tr>
<tr>
<td>3</td>
<td>ON</td>
</tr>
<tr>
<td>4</td>
<td>ON</td>
</tr>
<tr>
<td>5</td>
<td>ON</td>
</tr>
<tr>
<td>6</td>
<td>ON</td>
</tr>
<tr>
<td>7</td>
<td>ON</td>
</tr>
<tr>
<td>8</td>
<td>n/a</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>GPIO Pin</th>
<th>Type</th>
<th>Function Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_VP</td>
<td>I</td>
<td>Audio Rec (PB)</td>
</tr>
<tr>
<td>SENSOR_VN</td>
<td>I</td>
<td>Audio Mode (PB)</td>
</tr>
<tr>
<td>IO32</td>
<td>I/O</td>
<td>Audio Set (TP)</td>
</tr>
<tr>
<td>IO33</td>
<td>I/O</td>
<td>Audio Play (TP)</td>
</tr>
<tr>
<td>IO27</td>
<td>I/O</td>
<td>Audio Vol+ (TP)</td>
</tr>
<tr>
<td>IO13</td>
<td>I/O</td>
<td>JTAG MTCK, MicroSD D3, Audio Vol- (TP)</td>
</tr>
<tr>
<td>IO14</td>
<td>I/O</td>
<td>JTAG MTMS, MicroSD CLK</td>
</tr>
<tr>
<td>IO12</td>
<td>I/O</td>
<td>JTAG MTDI, MicroSD D2, Aux signal detect</td>
</tr>
<tr>
<td>IO15</td>
<td>I/O</td>
<td>JTAG MTDO, MicroSD CMD</td>
</tr>
<tr>
<td>IO2</td>
<td>I/O</td>
<td>Automatic Upload, MicroSD D0</td>
</tr>
<tr>
<td>IO4</td>
<td>I/O</td>
<td>MicroSD D1</td>
</tr>
<tr>
<td>IO34</td>
<td>I</td>
<td>MicroSD insert detect</td>
</tr>
<tr>
<td>IO0</td>
<td>I/O</td>
<td>Automatic Upload, I2S MCLK</td>
</tr>
<tr>
<td>IO5</td>
<td>I/O</td>
<td>I2S SCLK</td>
</tr>
<tr>
<td>IO25</td>
<td>I/O</td>
<td>I2S LRCK</td>
</tr>
<tr>
<td>IO26</td>
<td>I/O</td>
<td>I2S DSDIN</td>
</tr>
<tr>
<td>IO35</td>
<td>I</td>
<td>I2S ASDOUT</td>
</tr>
<tr>
<td>IO19</td>
<td>I/O</td>
<td>Headphone jack insert detect</td>
</tr>
<tr>
<td>IO22</td>
<td>I/O</td>
<td>Green LED indicator</td>
</tr>
<tr>
<td>IO21</td>
<td>I/O</td>
<td>PA Enable output</td>
</tr>
<tr>
<td>IO18</td>
<td>I/O</td>
<td>I2C SDA</td>
</tr>
<tr>
<td>IO23</td>
<td>I/O</td>
<td>I2C SCL</td>
</tr>
</tbody>
</table>

- (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**

<table>
<thead>
<tr>
<th>Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3.3V</td>
<td></td>
</tr>
<tr>
<td>2 TX</td>
<td></td>
</tr>
<tr>
<td>3 RX</td>
<td></td>
</tr>
<tr>
<td>4 GND</td>
<td></td>
</tr>
</tbody>
</table>

**I2S Header / JP4**

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MCLK</td>
<td>GPIO0</td>
</tr>
<tr>
<td>2 SCLK</td>
<td>GPIO5</td>
</tr>
<tr>
<td>1 LRCK</td>
<td>GPIO25</td>
</tr>
<tr>
<td>2 DSDIN</td>
<td>GPIO26</td>
</tr>
<tr>
<td>3 ASDOUT</td>
<td>GPIO35</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>

**I2C Header / JP5**

<table>
<thead>
<tr>
<th>I2C Header Pin</th>
<th>ESP32 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SCL</td>
<td>GPIO23</td>
</tr>
<tr>
<td>2 SDA</td>
<td>GPIO18</td>
</tr>
<tr>
<td>3 GND</td>
<td>GND</td>
</tr>
</tbody>
</table>

**JTAG Header / JP7**

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>JTAG Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MTDO / GPIO15</td>
<td>TDO</td>
</tr>
<tr>
<td>2 MTCK / GPIO13</td>
<td>TCK</td>
</tr>
<tr>
<td>3 MTDI / GPIO12</td>
<td>TDI</td>
</tr>
<tr>
<td>4 MTMS / GPIO14</td>
<td>TMS</td>
</tr>
</tbody>
</table>

**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.

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.
Module Power Supply:

![Module Power Supply Diagram]

Fig. 13: ESP32 LyraT V4.3 - Dedicated LDO for the ESP32 Module

SDIO Power Supply:

![SDIO Power Supply Diagram]

Fig. 14: 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.

![PA Output Diagram](image)

Fig. 15: 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)

3.4. Development Boards
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 provide 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**

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aac</td>
<td>ff-16b-2c-44100hz.aac</td>
<td>2,995</td>
</tr>
<tr>
<td>2</td>
<td>ac3</td>
<td>ff-16b-2c-44100hz.ac3</td>
<td>2,994</td>
</tr>
<tr>
<td>3</td>
<td>aiff</td>
<td>ff-16b-2c-44100hz.aiff</td>
<td>33,002</td>
</tr>
<tr>
<td>4</td>
<td>flac</td>
<td>ff-16b-2c-44100hz.flac</td>
<td>22,406</td>
</tr>
<tr>
<td>5</td>
<td>m4a</td>
<td>ff-16b-2c-44100hz.m4a</td>
<td>3,028</td>
</tr>
<tr>
<td>6</td>
<td>mp3</td>
<td>ff-16b-2c-44100hz.mp3</td>
<td>2,994</td>
</tr>
<tr>
<td>7</td>
<td>mp4</td>
<td>ff-16b-2c-44100hz.mp4</td>
<td>3,079</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>ff-16b-2c-44100hz.ogg</td>
<td>2,612</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>ff-16b-2c-44100hz.opus</td>
<td>2,598</td>
</tr>
<tr>
<td>10</td>
<td>ts</td>
<td>ff-16b-2c-44100hz.ts</td>
<td>5,510</td>
</tr>
<tr>
<td>11</td>
<td>wav</td>
<td>ff-16b-2c-44100hz.wav</td>
<td>49,504</td>
</tr>
<tr>
<td>12</td>
<td>wma</td>
<td>ff-16b-2c-44100hz.wma</td>
<td>3,227</td>
</tr>
</tbody>
</table>

Playlist containing all above files: ff-16b-2c-playlist.m3u
Single Channel Audio

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aac</td>
<td>ff-16b-1c-44100hz.aac</td>
<td>1,650</td>
</tr>
<tr>
<td>2</td>
<td>ac3</td>
<td>ff-16b-1c-44100hz.ac3</td>
<td>2,193</td>
</tr>
<tr>
<td>3</td>
<td>aiff</td>
<td>ff-16b-1c-44100hz.aiff</td>
<td>16,115</td>
</tr>
<tr>
<td>4</td>
<td>amr</td>
<td>ff-16b-1c-8000hz.amr</td>
<td>299</td>
</tr>
<tr>
<td>5</td>
<td>flac</td>
<td>ff-16b-1c-44100hz.flac</td>
<td>10,655</td>
</tr>
<tr>
<td>6</td>
<td>m4a</td>
<td>ff-16b-1c-44100hz.m4a</td>
<td>1,628</td>
</tr>
<tr>
<td>7</td>
<td>mp3</td>
<td>ff-16b-1c-44100hz.mp3</td>
<td>1,463</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>ff-16b-1c-44100hz.ogg</td>
<td>1,558</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>ff-16b-1c-44100hz.opus</td>
<td>1,641</td>
</tr>
<tr>
<td>10</td>
<td>wav</td>
<td>ff-16b-1c-44100hz.wav</td>
<td>16,115</td>
</tr>
<tr>
<td>11</td>
<td>wma</td>
<td>ff-16b-1c-44100hz.wma</td>
<td>3,151</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aac</td>
<td>gs-16b-2c-44100hz.aac</td>
<td>241</td>
</tr>
<tr>
<td>2</td>
<td>ac3</td>
<td>gs-16b-2c-44100hz.ac3</td>
<td>380</td>
</tr>
<tr>
<td>3</td>
<td>aiff</td>
<td>gs-16b-2c-44100hz.aiff</td>
<td>2,792</td>
</tr>
<tr>
<td>4</td>
<td>flac</td>
<td>gs-16b-2c-44100hz.flac</td>
<td>1,336</td>
</tr>
<tr>
<td>5</td>
<td>m4a</td>
<td>gs-16b-2c-44100hz.m4a</td>
<td>1,367</td>
</tr>
<tr>
<td>6</td>
<td>mp3</td>
<td>gs-16b-2c-44100hz.mp3</td>
<td>254</td>
</tr>
<tr>
<td>7</td>
<td>mp4</td>
<td>gs-16b-2c-44100hz.mp4</td>
<td>259</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>gs-16b-2c-44100hz.ogg</td>
<td>229</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>gs-16b-2c-44100hz.opus</td>
<td>219</td>
</tr>
<tr>
<td>10</td>
<td>ts</td>
<td>gs-16b-2c-44100hz.ts</td>
<td>286</td>
</tr>
<tr>
<td>11</td>
<td>wav</td>
<td>gs-16b-2c-44100hz.wav</td>
<td>2,792</td>
</tr>
<tr>
<td>12</td>
<td>wma</td>
<td>gs-16b-2c-44100hz.wma</td>
<td>276</td>
</tr>
</tbody>
</table>

Playlist containing all above files: gs-16b-2c-playlist.m3u
### Single Channel Audio

<table>
<thead>
<tr>
<th>No</th>
<th>Format</th>
<th>Audio File</th>
<th>Size [kB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>amr</td>
<td>gs-16b-1c-8000hz.amr</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>aac</td>
<td>gs-16b-1c-44100hz.aac</td>
<td>137</td>
</tr>
<tr>
<td>3</td>
<td>ac3</td>
<td>gs-16b-1c-44100hz.ac3</td>
<td>190</td>
</tr>
<tr>
<td>4</td>
<td>aiff</td>
<td>gs-16b-1c-44100hz.aiff</td>
<td>1,397</td>
</tr>
<tr>
<td>5</td>
<td>flac</td>
<td>gs-16b-1c-44100hz.flac</td>
<td>645</td>
</tr>
<tr>
<td>6</td>
<td>m4a</td>
<td>gs-16b-1c-44100hz.m4a</td>
<td>650</td>
</tr>
<tr>
<td>7</td>
<td>mp3</td>
<td>gs-16b-1c-44100hz.mp3</td>
<td>127</td>
</tr>
<tr>
<td>8</td>
<td>ogg</td>
<td>gs-16b-1c-44100hz.ogg</td>
<td>144</td>
</tr>
<tr>
<td>9</td>
<td>opus</td>
<td>gs-16b-1c-44100hz.opus</td>
<td>132</td>
</tr>
<tr>
<td>10</td>
<td>wav</td>
<td>gs-16b-1c-44100hz.wav</td>
<td>1,497</td>
</tr>
<tr>
<td>11</td>
<td>wma</td>
<td>gs-16b-1c-44100hz.wma</td>
<td>276</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Audio File</th>
<th>Sample Rate</th>
<th>MPEG III</th>
<th>Channels</th>
<th>Bit Rate</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff-16b-1c-8000hz.mp3</td>
<td>8000</td>
<td>2.5</td>
<td>mono</td>
<td>8</td>
<td>183</td>
</tr>
<tr>
<td>ff-16b-1c-11025hz.mp3</td>
<td>11025</td>
<td>2.5</td>
<td>mono</td>
<td>16</td>
<td>366</td>
</tr>
<tr>
<td>ff-16b-1c-12000hz.mp3</td>
<td>12000</td>
<td>2.5</td>
<td>mono</td>
<td>16</td>
<td>366</td>
</tr>
<tr>
<td>ff-16b-1c-16000hz.mp3</td>
<td>16000</td>
<td>2</td>
<td>mono</td>
<td>24</td>
<td>548</td>
</tr>
<tr>
<td>ff-16b-1c-22050hz.mp3</td>
<td>22050</td>
<td>2</td>
<td>mono</td>
<td>32</td>
<td>731</td>
</tr>
<tr>
<td>ff-16b-1c-24000hz.mp3</td>
<td>24000</td>
<td>2</td>
<td>mono</td>
<td>32</td>
<td>731</td>
</tr>
<tr>
<td>ff-16b-1c-32000hz.mp3</td>
<td>32000</td>
<td>1</td>
<td>mono</td>
<td>48</td>
<td>1,097</td>
</tr>
<tr>
<td>ff-16b-1c-44100hz.mp3</td>
<td>44100</td>
<td>1</td>
<td>mono</td>
<td>64</td>
<td>1,462</td>
</tr>
<tr>
<td>ff-16b-2c-8000hz.mp3</td>
<td>8000</td>
<td>2.5</td>
<td>joint stereo</td>
<td>24</td>
<td>549</td>
</tr>
<tr>
<td>ff-16b-2c-11025hz.mp3</td>
<td>11025</td>
<td>2.5</td>
<td>joint stereo</td>
<td>32</td>
<td>731</td>
</tr>
<tr>
<td>ff-16b-2c-12000hz.mp3</td>
<td>12000</td>
<td>2.5</td>
<td>joint stereo</td>
<td>32</td>
<td>731</td>
</tr>
<tr>
<td>ff-16b-2c-16000hz.mp3</td>
<td>16000</td>
<td>2</td>
<td>joint stereo</td>
<td>48</td>
<td>1,097</td>
</tr>
<tr>
<td>ff-16b-2c-22050hz.mp3</td>
<td>22050</td>
<td>2</td>
<td>joint stereo</td>
<td>64</td>
<td>1,462</td>
</tr>
<tr>
<td>ff-16b-2c-24000hz.mp3</td>
<td>24000</td>
<td>2</td>
<td>joint stereo</td>
<td>64</td>
<td>1,462</td>
</tr>
<tr>
<td>ff-16b-2c-32000hz.mp3</td>
<td>32000</td>
<td>1</td>
<td>joint stereo</td>
<td>96</td>
<td>2,194</td>
</tr>
<tr>
<td>ff-16b-2c-44100hz.mp3</td>
<td>44100</td>
<td>1</td>
<td>joint stereo</td>
<td>128</td>
<td>2,924</td>
</tr>
</tbody>
</table>

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,
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.
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. 1: Espressif Audio Development Framework
A
aac_decoder_cfg_t (C++ class), 93
aac_decoder_cfg_t::out_rb_size (C++ member), 93
aac_decoder_cfg_t::stack_in_ext (C++ member), 94
aac_decoder_cfg_t::task_core (C++ member), 93
aac_decoder_cfg_t::task_prio (C++ member), 93
aac_decoder_cfg_t::task_stack (C++ member), 93
aac_decoder_init (C++ function), 93
AAC_DECODER_RINGBUFFER_SIZE (C macro), 94
AAC_DECODER_TASK_CORE (C macro), 94
AAC_DECODER_TASK_PRIO (C macro), 94
AAC_DECODER_TASK_STACK_SIZE (C macro), 94
ADC_DEFAULT_ARR (C macro), 149
AEL_IO_ABORT (C++ enumerator), 56
AEL_IO_DONE (C++ enumerator), 56
AEL_IO_FAIL (C++ enumerator), 56
AEL_IO_OK (C++ enumerator), 56
AEL_IO_TIMEOUT (C++ enumerator), 56
AEL_MSG_CMD_DESTROY (C++ enumerator), 57
AEL_MSG_CMD_ERROR (C++ enumerator), 57
AEL_MSG_CMD_FINISH (C++ enumerator), 57
AEL_MSG_CMD_NONE (C++ enumerator), 56
AEL_MSG_CMD_PAUSE (C++ enumerator), 57
AEL_MSG_CMD_REPORT_CODEC_FMT (C++ enumerator), 57
AEL_MSG_CMD_REPORT_MUSIC_INFO (C++ enumerator), 57
AEL_MSG_CMD_REPORT_POSITION (C++ enumerator), 57
AEL_MSG_CMD_REPORT_STATUS (C++ enumerator), 57
AEL_MSG_CMD_RESUME (C++ enumerator), 57
AEL_MSG_CMD_STOP (C++ enumerator), 57
AEL_STATE_ERROR (C++ enumerator), 56
AEL_STATE_FINISHED (C++ enumerator), 56
AEL_STATE_INIT (C++ enumerator), 56
AEL_STATE_NONE (C++ enumerator), 56
AEL_STATE_PAUSED (C++ enumerator), 56
AEL_STATE_RUNNING (C++ enumerator), 56
AEL_STATE_STOPPED (C++ enumerator), 56
AEL_STATUS_ERROR_CLOSE (C++ enumerator), 57
AEL_STATUS_ERROR_INPUT (C++ enumerator), 57
AEL_STATUS_ERROR_OPEN (C++ enumerator), 57
AEL_STATUS_ERROR_OUTPUT (C++ enumerator), 57
AEL_STATUS_ERROR_PROCESS (C++ enumerator), 57
AEL_STATUS_ERROR_TIMEOUT (C++ enumerator), 57
AEL_STATUS_ERROR_UNKNOWN (C++ enumerator), 57
AEL_STATUS_INPUT_BUFFERING (C++ enumerator), 57
AEL_STATUS_INPUT_DONE (C++ enumerator), 57
AEL_STATUS_MOUNTED (C++ enumerator), 57
AEL_STATUS_NONE (C++ enumerator), 57
AEL_STATUS_OUTPUT_BUFFERING (C++ enumerator), 57
AEL_STATUS_OUTPUT_DONE (C++ enumerator), 57
AEL_STATUS_STATE_FINISHED (C++ enumerator), 57
AEL_STATUS_STATE_PAUSED (C++ enumerator), 57
AEL_STATUS_STATE_RUNNING (C++ enumerator), 57
AEL_STATUS_STATE_STOPPED (C++ enumerator), 57
AEL_STATUS_UNMOUNTED (C++ enumerator), 57
amr_decoder_cfg_t (C++ class), 94
amr_decoder_cfg_t::out_rb_size (C++ member), 95
amr_decoder_cfg_t::stack_in_ext (C++ member), 95
amr_decoder_cfg_t::task_core (C++ member), 95
amr_decoder_cfg_t::task_prio (C++ member), 95
amr_decoder_cfg_t::task_stack (C++ member), 95
amr_decoder_init (C++ function), 94
AMR_DECODER_RINGBUFFER_SIZE (C macro), 95
AMR_DECODER_TASK_CORE (C macro), 95
AMR_DECODER_TASK_PRIO (C macro), 95
AMR_DECODER_TASK_STACK_SIZE (C macro), 95
amrnb_encoder_cfg_t (C++ class), 95
amrnb_encoder_cfg_t::out_rb_size (C++ member), 96
amrnb_encoder_cfg_t::stack_in_ext (C++ member), 96
amrnb_encoder_cfg_t::task_core (C++ member), 96
amrnb_encoder_cfg_t::task_prio (C++ member), 96
amrnb_encoder_cfg_t::task_stack (C++ member), 96
amrnb_encoder_init (C++ function), 95
AMRNB_ENCODER_RINGBUFFER_SIZE (C macro), 96
AMRNB_ENCODER_TASK_CORE (C macro), 96
AMRNB_ENCODER_TASK_PRIO (C macro), 96
AMRNB_ENCODER_TASK_STACK_SIZE (C macro), 96
amrwb_encoder_cfg_t (C++ class), 96
amrwb_encoder_cfg_t::out_rb_size (C++ member), 97
amrwb_encoder_cfg_t::stack_in_ext (C++ member), 97
amrwb_encoder_cfg_t::task_core (C++ member), 97
amrwb_encoder_cfg_t::task_prio (C++ member), 97
amrwb_encoder_cfg_t::task_stack (C++ member), 97
amrwb_encoder_init (C++ function), 96
AMRWB_ENCODER_RINGBUFFER_SIZE (C macro), 97
AMRWB_ENCODER_TASK_CORE (C macro), 97
AMRWB_ENCODER_TASK_PRIO (C macro), 97
AMRWB_ENCODER_TASK_STACK_SIZE (C macro), 97
AUDIO_CODEC_TYPE_DECODER (C++ enumerator), 72
AUDIO_CODEC_TYPE_ENCODER (C++ enumerator), 72
AUDIO_CODEC_TYPE_NONE (C++ enumerator), 72
audio_element_abort_input_ringbuf (C++ function), 46
audio_element_abort_output_ringbuf (C++ function), 46
audio_elementcfg_t (C++ class), 54
audio_elementcfg_t::buffer_len (C++ member), 55
audio_elementcfg_t::close (C++ member), 55
audio_elementcfg_t::data (C++ member), 55
audio_elementcfg_t::destroy (C++ member), 55
audio_elementcfg_t::multi_in_rb_num (C++ member), 55
audio_elementcfg_t::multi_out_rb_num (C++ member), 55
audio_elementcfg_t::open (C++ member), 55
audio_elementcfg_t::out_rb_size (C++ member), 55
audio_elementcfg_t::process (C++ member), 55
audio_elementcfg_t::read (C++ member), 55
audio_elementcfg_t::seek (C++ member), 55
audio_elementcfg_t::stack_in_ext (C++ member), 55
audio_elementcfg_t::tag (C++ member), 55
audio_elementcfg_t::task_core (C++ member), 55
audio_elementcfg_t::task_prio (C++ member), 55
audio_elementcfg_t::task_stack (C++ member), 55
audio_elementcfg_t::write (C++ member), 55
audio_element_change_cmd (C++ function), 48
audio_element_deinit (C++ function), 40
audio_element_err_t (C++ type), 56
audio_element_finish_state (C++ function), 48
audio_element_get_event_queue (C++ function), 50
audio_element_get_input_ringbuf (C++ function), 45
audio_element_get_multi_input_ringbuf (C++ function), 52
audio_element_get_multi_output_ringbuf (C++ function), 52
audio_element_get_output_ringbuf (C++ function), 45
audio_element_get_output_ringbuf_size (C++ function), 50
audio_element_get_state (C++ function), 46
audio_element_get_tag (C++ function), 41
audio_element_get_uri (C++ function), 42
audio_element_getdata (C++ function), 41
audio_element_getinfo (C++ function), 42
audio_element_handle_t (C++ type), 56
read the docs template documentation

audio_element_info_t (C++ class), 54
audio_element_info_t::bits (C++ member), 54
audio_element_info_t::bps (C++ member), 54
audio_element_info_t::byte_pos (C++ member), 54
audio_element_info_t::channels (C++ member), 54
audio_element_info_t::codec_fmt (C++ member), 54
audio_element_info_t::duration (C++ member), 54
audio_element_info_t::reserve_data (C++ member), 54
audio_element_info_t::sample_rates (C++ member), 54
audio_element_info_t::total_bytes (C++ member), 54
audio_element_info_t::uri (C++ member), 54
audio_element_init (C++ function), 40
audio_element_input (C++ function), 49
audio_element_msg_cmd_t (C++ type), 56
audio_element_msg_remove_listener (C++ function), 45
audio_element_msg_set_listener (C++ function), 44
audio_element_multi_input (C++ function), 51
audio_element_multi_output (C++ function), 51
audio_element_output (C++ function), 49
audio_element_pause (C++ function), 43
audio_element_process_deinit (C++ function), 53
audio_element_process_init (C++ function), 53
audio_element_report_codec_fmt (C++ function), 47
audio_element_report_info (C++ function), 47
audio_element_report_pos (C++ function), 47
audio_element_report_status (C++ function), 46
audio_element_reserve_data_t (C++ class), 54
audio_element_reserve_data_t::user_data_0 (C++ member), 54
audio_element_reserve_data_t::user_data_1 (C++ member), 54
audio_element_reserve_data_t::user_data_2 (C++ member), 54
audio_element_reserve_data_t::user_data_3 (C++ member), 54
audio_element_reserve_data_t::user_data_4 (C++ member), 54
audio_element_reset_input_ringbuf (C++ function), 48
audio_element_reset_output_ringbuf (C++ function), 48
audio_element_reset_state (C++ function), 50
audio_element_resume (C++ function), 44
audio_element_run (C++ function), 42
audio_element_seek (C++ function), 53
audio_element_set_event_callback (C++ function), 44
audio_element_set_input_ringbuf (C++ function), 45
audio_element_set_input_timeout (C++ function), 47
audio_element_set_multi_input_ringbuf (C++ function), 51
audio_element_set_multi_output_ringbuf (C++ function), 52
audio_element_set_output_ringbuf (C++ function), 45
audio_element_set_output_ringbuf_size (C++ function), 51
audio_element_set_output_timeout (C++ function), 48
audio_element_set_read_cb (C++ function), 49
audio_element_set_ringbuf_done (C++ function), 50
audio_element_set_tag (C++ function), 41
audio_element_set_uri (C++ function), 42
audio_element_set_write_cb (C++ function), 49
audio_element_setdata (C++ function), 41
audio_element_setinfo (C++ function), 41
audio_element_state_t (C++ type), 56
audio_element_status_t (C++ type), 57
audio_element_stop (C++ function), 43
audio_element_terminate (C++ function), 42
AUDIO_ELEMENT_TYPE_ELEMENT (C++ enumerator), 72
AUDIO_ELEMENT_TYPE_PERIPH (C++ enumerator), 72
AUDIO_ELEMENT_TYPE_PLAYER (C++ enumerator), 72
AUDIO_ELEMENT_TYPE_SERVICE (C++ enumerator), 72
audio_element_type_t (C++ type), 72
audio_element_wait_for_buffer (C++ function), 46
audio_element_wait_for_stop (C++ function), 43
audio_element_wait_for_stop_ms (C++ function), 43
audio_err_t (C++ type), 73
audio_event_iface_cfg_t (C++ class), 71
audio_event_iface_cfg_t::context (C++ member), 71
audio_event_iface_cfg_t::external_queue (C++ member), 71
audio_event_iface_cfg_t::internal_queue (C++ member), 71
audio_event_iface_cfg_t::on_cmd (C++ member), 71
audio_event_iface_cfg_t::queue_set_size (C++ member), 71
audio_event_iface_cfg_t::type (C++ member), 71
audio_event_iface_cfg_t::wait_time (C++ member), 71
audio_event_iface_cmd (C++ function), 68
audio_event_iface_cmd_from_isr (C++ function), 68
AUDIO_EVENT_IFACE_DEFAULT_CFG (C macro), 71
audio_event_iface_destroy (C++ function), 67
audio_event_iface_discard (C++ function), 69
audio_event_iface_get_msg_queue_handle (C++ function), 69
audio_event_iface_get_queue_handle (C++ function), 69
audio_event_iface_handle_t (C++ type), 71
audio_event_iface_init (C++ function), 67
audio_event_iface_list (C++ function), 69
audio_event_iface_msg_t (C++ class), 70
audio_event_iface_msg_t::cmd (C++ member), 70
audio_event_iface_msg_t::data (C++ member), 70
audio_event_iface_msg_t::data_len (C++ member), 70
audio_event_iface_msg_t::need_free_data (C++ member), 70
audio_event_iface_msg_t::source (C++ member), 70
audio_event_iface_msg_t::source_type (C++ member), 70
audio_event_iface_read (C++ function), 69
audio_event_iface_remove_listener (C++ function), 67
audio_event_iface_sendout (C++ function), 69
audio_event_iface_set_cmd_waiting_timeout (C++ function), 68
audio_event_iface_set_listener (C++ function), 67
audio_event_iface_set_msg_listener (C++ function), 70
audio_event_iface_waiting_cmd_msg (C++ function), 68
audio_hal (C++ class), 159
audio_hal::audio_codec_config_iface (C++ member), 159
audio_hal::audio_codec_ctrl (C++ member), 159
audio_hal::audio_codec_deinitialize (C++ member), 159
audio_hal::audio_codec_get_volume (C++ member), 159
audio_hal::audio_codec_initialize (C++ member), 159
audio_hal::audio_codec_set_mute (C++ member), 159
audio_hal::audio_codec_set_volume (C++ member), 159
audio_hal::audio_hal_lock (C++ member), 159
audio_hal::handle (C++ member), 159
AUDIO_HAL_08K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_11K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_16K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_22K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_24K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_32K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_48K_SAMPLES (C++ enumerator), 161
AUDIO_HAL_ADC_INPUT_ALL (C++ enumerator), 160
AUDIO_HAL_ADC_INPUT_DIFFERENCE (C++ enumerator), 160
AUDIO_HAL_ADC_INPUT_LINE1 (C++ enumerator), 160
AUDIO_HAL_ADC_INPUT_LINE2 (C++ enumerator), 160
AUDIO_HAL_BIT_LENGTH_16BITS (C++ enumerator), 161
AUDIO_HAL_BIT_LENGTH_24BITS (C++ enumerator), 161
AUDIO_HAL_BIT_LENGTH_32BITS (C++ enumerator), 161
AUDIO_HAL_CODEC_CONFIG (C++ class), 158
audio_hal::audio_codec_config_iface (C++ member), 159
audio_hal::audio_codec_config_iface::dac_output (C++ member), 159
audio_hal::audio_codec_config_iface::codec_mode (C++ member), 159
audio_hal::audio_codec_config_iface::i2s_iface (C++ member), 159
audio_hal::audio_codec_i2s_iface (C++ class), 158
audio_hal::audio_codec_i2s_iface::bits (C++
Read the Docs Template Documentation

AUDIO_STREAM_NONE (C++ enumerator), 72
AUDIO_STREAM_READER (C++ enumerator), 72
audio_stream_type_t (C++ type), 72
AUDIO_STREAM_WRITER (C++ enumerator), 72
audio_termination_type_t (C++ type), 74
audio_volume_get (C++ type), 73
audio_volume_set (C++ type), 73

BLUE_LED_MAX_NUM (C macro), 152
BLUETOOTH_A2DP_SINK (C++ enumerator), 118
BLUETOOTH_A2DP_SOURCE (C++ enumerator), 118
BLUETOOTH_ADDR_LEN (C macro), 118
bluetooth_service_cfg_t (C++ class), 118
bluetooth_service_cfg_t::device_name (C++ member), 118
bluetooth_service_cfg_t::mode (C++ member), 118
bluetooth_service_cfg_t::remote_name (C++ member), 118
bluetooth_service_create_periph (C++ function), 115
bluetooth_service_create_stream (C++ function), 115
bluetooth_service_destroy (C++ function), 117
bluetooth_service_mode_t (C++ type), 118
bluetooth_service_start (C++ function), 115

console_cmd_callback_t (C++ type), 143
CONSOLE_DEFAULT_BUFFER_SIZE (C macro), 143
CONSOLE_DEFAULT_PROMPT_STRING (C macro), 143
CONSOLE_DEFAULT_TASK_PRIO (C macro), 143
ctrl_func (C++ type), 56

decoder_opus_init (C++ function), 101
default_element_stack_size (C macro), 55
default_element_task_core (C macro), 56
default_element_task_prio (C macro), 55
default_equalizer_config (C macro), 109
default_esp_audio_config (C macro), 84
default_esp_periph_set_config (C macro), 134
default_esp_periph_stack_size (C macro), 134
default_esp_periph_task_core (C macro), 134
default_esp_periph_task_prio (C macro), 134
default_flac_decoder_config (C macro), 98
default_mp3_decoder_config (C macro), 99
default_ogg_decoder_config (C macro), 100
default_opus_decoder_config (C macro), 101
default_pipeline_ringbuf_size (C macro), 66
default_rec_engine_config (C macro), 124
default_resample_filter_config (C macro), 112
default_sonic_config (C macro), 114
default_nav_decoder_config (C macro), 102
default_nav_encoder_config (C macro), 103
dm_buf_size (C macro), 107
downmix_cfg_t (C++ class), 107
downmix_cfg_t::downmix_info (C++ member), 107
downmix_cfg_t::max_sample (C++ member), 107
downmix_cfg_t::out_rb_size (C++ member), 107
downmix_cfg_t::stack_in_ext (C++ member), 107
downmix_cfg_t::task_core (C++ member), 107
downmix_cfg_t::task_prio (C++ member), 107
downmix_cfg_t::task_stack (C++ member), 107
downmix_init (C++ function), 106
downmix_ringbuffer_size (C macro), 107
downmix_set_gain_info (C++ function), 106
downmix_set_input_rb (C++ function), 105
downmix_set_input_rb_timeout (C++ function), 105
downmix_set_out_ctx_info (C++ function), 105
downmix_set_output_type (C++ function), 105
downmix_set_source_stream_info (C++ function), 106
downmix_set_transit_time_info (C++ function), 106
downmix_set_work_mode (C++ function), 105
downmix_task_core (C macro), 107
downmix_task_prio (C macro), 107
downmix_task_stack (C macro), 107
ELEMENT_SUB_TYPE_OFFSET (C macro), 72
equalizer_cfg (C++ class), 109
equalizer_cfg::channel (C++ member), 109
equalizer_cfg::out_rb_size (C++ member), 109
equalizer_cfg::samplerate (C++ member), 109
equalizer_cfg::set_gain (C++ member), 109
equalizer_cfg::stack_in_ext (C++ member), 109
equalizer_cfg::task_core (C++ member), 109
equalizer_cfg::task_prio (C++ member), 109
equalizer_cfg::task_stack (C++ member), 109
equalizer_cfg_t (C++ type), 109
equalizer_init (C++ function), 108
EQUALIZER_RINGBUFFER_SIZE (C macro), 109
equalizer_set_gain_info (C++ function), 108
equalizer_set_info (C++ function), 108
EQUALIZER_TASK_CORE (C macro), 109
EQUALIZER_TASK_PRIO (C macro), 109
EQUALIZER_TASK_STACK (C macro), 109
ES8374_ADDR (C macro), 171
es8374_codec_config_i2s (C++ function), 171
es8374_codec_ctrl_state (C++ function), 171
es8374_codec_deinit (C++ function), 168
es8374_codec_get_voice_volume (C++ function), 169
es8374_codec_init (C++ function), 167
es8374_codec_set_voice_volume (C++ function), 169
es8374_config_adc_input (C++ function), 170
es8374_config_dac_output (C++ function), 170
es8374_config_fmt (C++ function), 168
es8374_get_voice_mute (C++ function), 169
es8374_i2s_config_clock (C++ function), 168
es8374_pa_power (C++ function), 171
es8374_read_all (C++ function), 168
es8374_set_bits_per_sample (C++ function), 168
es8374_set_mic_gain (C++ function), 170
es8374_set_voice_mute (C++ function), 169
es8374_stop (C++ function), 169
es8374_write_reg (C++ function), 170
ES8388_ADCCONTROL1 (C macro), 166
ES8388_ADCCONTROL10 (C macro), 166
ES8388_ADCCONTROL11 (C macro), 166
ES8388_ADCCONTROL12 (C macro), 166
ES8388_ADCCONTROL13 (C macro), 166
ES8388_ADCCONTROL14 (C macro), 166
ES8388_ADCCONTROL15 (C macro), 166
ES8388_ADCCONTROL16 (C macro), 166
ES8388_ADCCONTROL17 (C macro), 166
ES8388_ADCCONTROL18 (C macro), 166
ES8388_ADCCONTROL19 (C macro), 166
ES8388_ADCCONTROL20 (C macro), 166
ES8388_ADCCONTROL21 (C macro), 166
ES8388_ADCCONTROL22 (C macro), 166
ES8388_ADCCONTROL23 (C macro), 166
ES8388_ADCCONTROL24 (C macro), 166
ES8388_ADCCONTROL25 (C macro), 167
ES8388_ADCCONTROL26 (C macro), 167
ES8388_ADCCONTROL27 (C macro), 167
ES8388_ADCCONTROL28 (C macro), 167
ES8388_ADCCONTROL29 (C macro), 167
ES8388_ADCCONTROL3 (C macro), 166
ES8388_ADCCONTROL4 (C macro), 166
ES8388_ADCCONTROL5 (C macro), 166
ES8388_ADCCONTROL6 (C macro), 166
ES8388_ADCCONTROL7 (C macro), 166
ES8388_ADCCONTROL8 (C macro), 166
ES8388_ADCCONTROL9 (C macro), 166
ES8388_ADCPOWER (C macro), 166
ES8388_ADDR (C macro), 166
ES8388_ANAVOLMANAG (C macro), 166
ES8388_CHIPLOPOWER (C macro), 166
ES8388_CHIPPOWER (C macro), 166
es8388_config_adc_input (C++ function), 164
es8388_config_dac_output (C++ function), 164
es8388_config_fmt (C++ function), 162
es8388_config_i2s (C++ function), 165
ES8388_CONTROL1 (C macro), 166
ES8388_CONTROL2 (C macro), 166
es8388_crtl_state (C++ function), 165
ES8388_DACCONTROL1 (C macro), 166
ES8388_DACCONTROL10 (C macro), 167
ES8388_DACCONTROL11 (C macro), 167
ES8388_DACCONTROL12 (C macro), 167
ES8388_DACCONTROL13 (C macro), 167
ES8388_DACCONTROL14 (C macro), 167
ES8388_DACCONTROL15 (C macro), 167
ES8388_DACCONTROL16 (C macro), 167
ES8388_DACCONTROL17 (C macro), 167
ES8388_DACCONTROL18 (C macro), 167
ES8388_DACCONTROL19 (C macro), 167
ES8388_DACCONTROL2 (C macro), 166
ES8388_DACCONTROL20 (C macro), 167
ES8388_DACCONTROL21 (C macro), 167
ES8388_DACCONTROL22 (C macro), 167
ES8388_DACCONTROL23 (C macro), 167
ES8388_DACCONTROL24 (C macro), 167
ES8388_DACCONTROL25 (C macro), 167
ES8388_DACCONTROL26 (C macro), 167
ES8388_DACCONTROL27 (C macro), 167
ES8388_DACCONTROL28 (C macro), 167
ES8388_DACCONTROL29 (C macro), 167
ES8388_DACCONTROL3 (C macro), 166
ES8388_DACCONTROL4 (C macro), 166
ES8388_DACCONTROL5 (C macro), 166
ES8388_DACCONTROL6 (C macro), 166
ES8388_DACCONTROL7 (C macro), 166
ES8388_DACCONTROL8 (C macro), 166
ES8388_DACCONTROL9 (C macro), 166
ES8388_DACPOWER (C macro), 166
es8388_deinit (C++ function), 162
es8388_get_voice_mute (C++ function), 164
es8388_get_voice_volume (C++ function), 163
es8388_i2s_config_clock (C++ function), 162
ESP_ERR_AUDIO_HAL_FAIL (C++ enumerator), 73
ESP_ERR_AUDIO_INPUT (C++ enumerator), 74
ESP_ERR_AUDIO_INVALID_PARAMETER (C++ enumerator), 73
ESP_ERR_AUDIO_INVALID_PATH (C++ enumerator), 73
ESP_ERR_AUDIO_INVALID_URI (C++ enumerator), 73
ESP_ERR_AUDIO_LINK_FAIL (C++ enumerator), 73
ESP_ERR_AUDIO_MEMORY_LACK (C++ enumerator), 73
ESP_ERR_AUDIO_NO_CODEC (C++ enumerator), 73
ESP_ERR_AUDIO_NO_ERROR (C++ enumerator), 73
ESP_ERR_AUDIO_OUTPUT (C++ enumerator), 73
esp_periph_config_t::task_stack (C++ member), 133
esp_periph_config_t::task_prio (C++ member), 133
esp_periph_config_t::task_stack (C++ member), 133
esp_periph_create (C++ function), 129
esp_periph_destroy (C++ function), 133
esp_periph_event (C++ class), 133
esp_periph_event::cb (C++ member), 133
esp_periph_event::iface (C++ member), 133
esp_periph_event::user_ctx (C++ member), 133
esp_periph_event_handle_t (C++ type), 134
esp_periph_event_t (C++ type), 134
esp_periph_func (C++ type), 134
esp_periph_get_data (C++ function), 131
esp_periph_get_id (C++ function), 132
esp_periph_get_state (C++ function), 132
esp_periph_handle_t (C++ type), 134
esp_periph_id_t (C++ type), 134
esp_periph_init (C++ function), 132
esp_periph_register_on_events (C++ function), 133
esp_periph_run (C++ function), 132
esp_periph_run_func (C++ type), 134
esp_periph_send_cmd (C++ function), 130
esp_periph_send_cmd_from_isr (C++ function), 130
esp_periph_send_event (C++ function), 130
esp_periph_set_data (C++ function), 131
esp_periph_set_destroy (C++ function), 127
esp_periph_set_function (C++ function), 129
esp_periph_set_get_by_id (C++ function), 127
esp_periph_set_get_event_iface (C++ function), 127
esp_periph_set_get_queue (C++ function), 128
esp_periph_set_handle_t (C++ type), 134
esp_periph_set_id (C++ function), 132
esp_periph_set_init (C++ function), 126
esp_periph_set_list_destroy (C++ function), 128
esp_periph_set_list_init (C++ function), 128
esp_periph_set_list_run (C++ function), 128
esp_periph_set_register_callback (C++ function), 127
esp_periph_set_stop_all (C++ function), 127
esp_periph_start (C++ function), 129
esp_periph_start_timer (C++ function), 131
esp_periph_state_t (C++ type), 135
esp_periph_stop (C++ function), 129
esp_periph_stop_timer (C++ function), 131
touch_pad_sel_t (C++ type), 144
event_cb_func (C++ type), 56
F
FATFS_STREAM_BUF_SIZE (C macro), 90
FATFS_STREAM_CFG_DEFAULT (C macro), 90
fatfs_streamCfg_t (C++ class), 90
fatfs_streamCfg_t::buf_sz (C++ member), 90
fatfs_streamCfg_t::out_rb_size (C++ member), 90
fatfs_streamCfg_t::task_core (C++ member), 90
fatfs_streamCfg_t::task_stack (C++ member), 90
fatfs_streamCfg_t::task_pr (C++ member), 90
fatfs_streamCfg_t::task_stack (C++ member), 90
fatfs_streamCfg_t::type (C++ member), 90
fatfs_stream_init (C++ function), 90
FATFS_STREAM_RINGBUFFER_SIZE (C macro), 90
FATFS_STREAM_TASK_CORE (C macro), 90
FATFS_STREAM_TASK_PRIO (C macro), 90
FATFS_STREAM_TASK_STACK (C macro), 90
flac_decoderCfg_t (C++ class), 97
flac_decoderCfg_t::out_rb_size (C++ member), 98
i2s_stream_set_clk (C++ function), 85
I2S_STREAM_TASK_CORE (C macro), 86
I2S_STREAM_TASK_PTO (C macro), 86
I2S_STREAM_TASK_STACK (C macro), 86
io_func (C++ type), 56
IS31FL3216_CH_NUM (C macro), 152
IS31FL3216_STATE_BY_AUDIO (C++ enumerator), 152
IS31FL3216_STATE_FLASH (C++ enumerator), 152
IS31FL3216_STATE_OFF (C++ enumerator), 152
IS31FL3216_STATE_ON (C++ enumerator), 152
IS31FL3216_STATE_SHIFT (C++ enumerator), 152
IS31FL3216_STATE_UNKNOWN (C++ enumerator), 152

M
media_source_type_t (C++ type), 74
MEDIA_SRC_TYPE_MUSIC_A2DP (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_BASE (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_DLNA (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_FLASH (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_HTTP (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_MAX (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_RAW (C++ enumerator), 74
MEDIA_SRC_TYPE_MUSIC_SD (C++ enumerator), 74
MEDIA_SRC_TYPE_NULL (C++ enumerator), 74
MEDIA_SRC_TYPE_reserve_BASE (C++ enumerator), 75
MEDIA_SRC_TYPE_reserve_MAX (C++ enumerator), 75
MEDIA_SRC_TYPE_TONE_BASE (C++ enumerator), 74
MEDIA_SRC_TYPE_TONE_FLASH (C++ enumerator), 74
MEDIA_SRC_TYPE_TONE_HTTP (C++ enumerator), 74
MEDIA_SRC_TYPE_TONE_MAX (C++ enumerator), 75
MEDIA_SRC_TYPE_TONE_SD (C++ enumerator), 74
mem_assert (C macro), 72
mp3_decoder_cfg_t (C++ class), 99
mp3_decoder_cfg_t::task_core (C++ member), 99
mp3_decoder_cfg_t::task_prio (C++ member), 99
mp3_decoder_cfg_t::task_stack (C++ member), 99
mp3_decoder_init (C++ function), 98
MP3_DECODER_RINGBUFFER_SIZE (C macro), 99
MP3_DECODER_TASK_CORE (C macro), 99
MP3_DECODER_TASK_Prio (C macro), 99
MP3_DECODER_TASK_STACK_SIZE (C macro), 99

O
ogg_decoder_cfg_t (C++ class), 100
ogg_decoder_cfg_t::out_rb_size (C++ member), 100
ogg_decoder_cfg_t::stack_in_ext (C++ member), 100
ogg_decoder_cfg_t::task_core (C++ member), 100
ogg_decoder_cfg_t::task_prio (C++ member), 100
ogg_decoder_cfg_t::task_stack (C++ member), 100
ogg_decoder_init (C++ function), 99
OGG_DECODER_RINGBUFFER_SIZE (C macro), 100
OGG_DECODER_TASK_CORE (C macro), 100
OGG_DECODER_TASK_Prio (C macro), 100
OGG_DECODER_TASK_STACK_SIZE (C macro), 100
on_event_iface_func (C++ type), 71
opus_decoder_cfg_t (C++ class), 101
opus_decoder_cfg_t::out_rb_size (C++ member), 101
opus_decoder_cfg_t::stack_in_ext (C++ member), 101
opus_decoder_cfg_t::task_core (C++ member), 101
opus_decoder_cfg_t::task_prio (C++ member), 101
opus_decoder_cfg_t::task_stack (C++ member), 101

P
periph_adc_button_cfg_t (C++ class), 149
periph_adc_button_cfg_t::arr (C++ member), 149
periph_adc_button_cfg_t::arr_size (C++ member), 149
periph_adc_button_event_id_t (C++ type), 149
PERIPH_ADC_BUTTON_IDLE (C++ enumerator), 149
periph_adc_button_init (C++ function), 148
PERIPH_ADC_BUTTON_LONG_PRESSED (C++ enumerator), 149
PERIPH_ADC_BUTTON_LONG_RELEASE (C++ enumerator), 149
PERIPH_ADC_BUTTON_PRESSED (C++ enumerator), 149
PERIPH_ADC_BUTTON_RELEASE (C++ enumerator), 149
periph_bluetooth_cancel_discover (C++ function), 117
periph_bluetooth_connect (C++ function), 117
periph_bluetooth_discover (C++ function), 117
periph_bluetooth_fast_forward (C++ function), 116
periph_bluetooth_next (C++ function), 116
periph_bluetooth_pause (C++ function), 115
periph_bluetooth_play (C++ function), 115
periph_bluetooth_prev (C++ function), 116
periph_bluetooth_rewind (C++ function), 116
periph_bluetooth_stop (C++ function), 115
periph_button_cfg_t (C++ class), 145
periph_button_cfg_t::gpio_mask (C++ member), 146
periph_button_cfg_t::long_press_time_ms (C++ member), 146
periph_button_event_id_t (C++ type), 146
periph_button_init (C++ function), 145
PERIPH_BUTTON_LONG_PRESSED (C++ enumerator), 146
PERIPH_BUTTON_LONG_RELEASE (C++ enumerator), 146
PERIPH_BUTTON_PRESSED (C++ enumerator), 146
PERIPH_BUTTON_RELEASE (C++ enumerator), 146
periph_console_cfg_t (C++ class), 142
periph_console_cfg_t::buffer_size (C++ member), 143
periph_console_cfg_t::command_num (C++ member), 143
periph_console_cfg_t::commands (C++ member), 143
periph_console_cfg_t::prompt_string (C++ member), 143
periph_console_cfg_t::task_prio (C++ member), 143
periph_console_cfg_t::task_stack (C++ member), 143
periph_console_cmd_t (C++ class), 142
periph_console_cmd_t::cmd (C++ member), 142
periph_console_cmd_t::func (C++ member), 142
periph_console_cmd_t::help (C++ member), 142
periph_console_cmd_t::id (C++ member), 142
periph_console_init (C++ function), 142
PERIPH_ID_ADC (C++ enumerator), 142
PERIPH_ID_ADC_BTN (C++ enumerator), 142
PERIPH_ID_AUXIN (C++ enumerator), 142
PERIPH_ID_AW2013 (C++ enumerator), 135
PERIPH_ID_BLUETOOTH (C++ enumerator), 134
PERIPH_ID_BUTTON (C++ enumerator), 134
PERIPH_ID_CONSOLE (C++ enumerator), 134
PERIPH_ID_FLASH (C++ enumerator), 134
PERIPH_ID_GPIO_ISR (C++ enumerator), 134
PERIPH_ID_IS31FL3216 (C++ enumerator), 134
PERIPH_ID_LED (C++ enumerator), 134
PERIPH_ID_SDCARD (C++ enumerator), 134
PERIPH_ID_SPIFFS (C++ enumerator), 134
PERIPH_ID_TOUCH (C++ enumerator), 134
PERIPH_ID_WIFI (C++ enumerator), 134
PERIPH_ID_WS2812 (C++ enumerator), 134
PERIPH_IS31_SHIFT_MODE_ACC (C++ enumerator), 153
PERIPH_IS31_SHIFT_MODE_SINGLE (C++ enumerator), 153
periph_is31_shift_mode_t (C++ type), 153
PERIPH_IS31_SHIFT_MODE_UNKNOWN (C++ enumerator), 153
periph_is31f13216_cfg_t (C++ class), 152
periph_is31f13216_cfg_t::duty (C++ member), 152
periph_is31f13216_cfg_t::is31f13216_pattern (C++ member), 152
periph_is31f13216_cfg_t::state (C++ member), 152
periph_is31f13216_cfg_t::state (C++ member), 152
periph_is31f13216_init (C++ function), 150
periph_is31f13216_set_act_time (C++ function), 152
periph_is31f13216_set_blink_pattern (C++ function), 150
periph_is31f13216_set_duty (C++ function), 150
periph_is31f13216_set_duty (C++ function), 150
periph_is31f13216_set_shift_mode (C++ function), 151
periph_is31f13216_set_shift_mode (C++ function), 151
periph_is31f13216_set_state (C++ function), 150
periph_is31f13216_state_t (C++ type), 152
periph_led_blink (C++ function), 147
periph_is31f13216_set_shift_mode (C++ function), 151
periph_is31f13216_set_shift_mode (C++ function), 151
periph_is31f13216_set_state (C++ function), 150
periph_is31f13216_state_t (C++ type), 152
periph_led_blink (C++ function), 147
rb_done_write (C++ function), 155
RB_FAIL (C macro), 156
rb_get_size (C++ function), 155
RB_OK (C macro), 156
rb_read (C++ function), 155
rb_reset (C++ function), 154
RB_TIMEOUT (C macro), 156
rb_unblock_reader (C++ function), 156
rb_write (C++ function), 155
rec_callback (C++ type), 124
rec_close (C++ type), 124
rec_config_t (C++ class), 123
rec_config_t::close (C++ member), 124
rec_config_t::enable_wwe (C++ member), 123
rec_config_t::evt_cb (C++ member), 124
rec_config_t::extension (C++ member), 123
rec_config_t::fetch (C++ member), 124
rec_config_t::one_frame_duration_ms (C++ member), 123
rec_config_t::open (C++ member), 124
rec_config_t::sensitivity (C++ member), 123
rec_config_t::support_encoding (C++ member), 123
rec_config_t::task_core (C++ member), 123
rec_config_t::user_data (C++ member), 124
rec_config_t::vad_off_delay_ms (C++ member), 123
rec_config_t::wakeup_time_ms (C++ member), 123
rec_engine_create (C++ function), 120
rec_engine_data_read (C++ function), 121
rec_engine_destroy (C++ function), 121
rec_engine_detect_suspend (C++ function), 121
rec_engine_enc_data_read (C++ function), 122
rec_engine_enc_enable (C++ function), 122
rec_engine_get_wakeup_stat (C++ function), 123
rec_engine_mute_enable (C++ function), 123
rec_engine_trigger_start (C++ function), 121
rec_engine_trigger_stop (C++ function), 121
rec_engine_vad_enable (C++ function), 122
rec_event_type_t (C++ type), 124
REC_EVENT_VAD_START (C++ enumerator), 124
REC_EVENT_VAD_STOP (C++ enumerator), 124
REC_EVENT_WAKEUP_END (C++ enumerator), 124
REC_EVENT_WAKEUP_START (C++ enumerator), 124
rec_fetch (C++ type), 124
REC_ONE_BLOCK_SIZE (C macro), 124
rec_open (C++ type), 124
REC_VOICE_SUSPEND_OFF (C++ enumerator), 124
REC_VOICE_SUSPEND_ON (C++ enumerator), 124
rec_voice_suspend_t (C++ type), 124
ringbuf_handle_t (C++ type), 156
RSP_FILTER_BUFFER_BYTE (C macro), 111
rsp_filter_cfg_t (C++ class), 110
rsp_filter_cfg_t::complexity (C++ member), 111
rsp_filter_cfg_t::dest_ch (C++ member), 111
rsp_filter_cfg_t::dest_rate (C++ member), 111
rsp_filter_cfg_t::down_ch_idx (C++ member), 111
rsp_filter_cfg_t::max_indata_bytes (C++ member), 111
rsp_filter_cfg_t::mode (C++ member), 111
rsp_filter_cfg_t::out_len_bytes (C++ member), 111
rsp_filter_cfg_t::out_rb_size (C++ member), 111
rsp_filter_cfg_t::sample_bits (C++ member), 111
rsp_filter_cfg_t::src_ch (C++ member), 111
rsp_filter_cfg_t::src_rate (C++ member), 111
rsp_filter_cfg_t::task_stack (C++ member), 111
rsp_filter_cfg_t::task_prio (C++ member), 111
S
SDCARD_STATUS_CARD_DETECT_CHANGE (C++ enumerator), 139
SDCARD_STATUS_MOUNT_ERROR (C++ enumerator), 139
SDCARD_STATUS_MOUNTED (C++ enumerator), 139
SDCARD_STATUS_UNKNOWN (C++ enumerator), 139
SDCARD_STATUS_UNMOUNT_ERROR (C++ enumerator), 139
SDCARD_STATUS_UNMOUNTED (C++ enumerator), 139
sonic_cfg_t (C++ class), 113
sonic_cfg_t::out_rb_size (C++ member), 113
sonic_cfg_t::sonic_info (C++ member), 113
WIFI_CONFIG_WEB (C++ enumerator), 137
WIFI_CONFIG_WPS (C++ enumerator), 137

Z
zl38063_codec_config_i2s (C++ function), 172
zl38063_codec_ctrl_state (C++ function), 172
zl38063_codec_deinit (C++ function), 172
zl38063_codec_get_voice_volume (C++ function), 173
zl38063_codec_init (C++ function), 172
zl38063_codec_set_voice_mute (C++ function), 173
zl38063_codec_set_voice_volume (C++ function), 173