Read the Docs Template
Documentation

Read the Docs

Jun 26, 2019
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Get Started</td>
<td></td>
</tr>
<tr>
<td>1.1 ESP32-LyraT V4.3 Getting Started Guide</td>
<td>3</td>
</tr>
<tr>
<td>1.2 ESP32-LyraTD-MSC V2.2 Getting Started Guide</td>
<td>7</td>
</tr>
<tr>
<td>1.3 About ESP-ADF</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Setup ESP-IDF</td>
<td>12</td>
</tr>
<tr>
<td>1.5 Get ESP-ADF</td>
<td>13</td>
</tr>
<tr>
<td>1.6 Setup Path to ESP-ADF</td>
<td>13</td>
</tr>
<tr>
<td>1.7 Start a Project</td>
<td>13</td>
</tr>
<tr>
<td>1.8 Connect and Configure</td>
<td>14</td>
</tr>
<tr>
<td>1.9 Build, Flash and Monitor</td>
<td>14</td>
</tr>
<tr>
<td>1.10 Update ESP-ADF</td>
<td>15</td>
</tr>
<tr>
<td>1.11 Related Documents</td>
<td>16</td>
</tr>
<tr>
<td>2 API Reference</td>
<td></td>
</tr>
<tr>
<td>2.1 Audio Framework</td>
<td>34</td>
</tr>
<tr>
<td>2.2 Audio Streams</td>
<td>64</td>
</tr>
<tr>
<td>2.3 Codecs</td>
<td>73</td>
</tr>
<tr>
<td>2.4 Audio Processing</td>
<td>74</td>
</tr>
<tr>
<td>2.5 Services</td>
<td>76</td>
</tr>
<tr>
<td>2.6 Speech Recognition</td>
<td>77</td>
</tr>
<tr>
<td>2.7 Peripherals</td>
<td>80</td>
</tr>
<tr>
<td>2.8 Abstraction Layer</td>
<td>85</td>
</tr>
<tr>
<td>2.9 Configuration Options</td>
<td>89</td>
</tr>
<tr>
<td>3 Design Guide</td>
<td></td>
</tr>
<tr>
<td>3.1 Project Design</td>
<td>91</td>
</tr>
<tr>
<td>3.2 Design Considerations</td>
<td>93</td>
</tr>
<tr>
<td>3.3 Software Design</td>
<td>96</td>
</tr>
<tr>
<td>3.4 Development Boards</td>
<td>99</td>
</tr>
<tr>
<td>3.5 Audio Samples</td>
<td>109</td>
</tr>
<tr>
<td>4 Resources</td>
<td></td>
</tr>
<tr>
<td>5 Copyrights and Licenses</td>
<td></td>
</tr>
<tr>
<td>5.1 Software Copyrights</td>
<td>115</td>
</tr>
<tr>
<td>6 About</td>
<td></td>
</tr>
</tbody>
</table>

---

1 Get Started

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

2 API Reference

2.1 Audio Framework
2.2 Audio Streams
2.3 Codecs
2.4 Audio Processing
2.5 Services
2.6 Speech Recognition
2.7 Peripherals
2.8 Abstraction Layer
2.9 Configuration Options

3 Design Guide

3.1 Project Design
3.2 Design Considerations
3.3 Software Design
3.4 Development Boards
3.5 Audio Samples

4 Resources

5 Copyrights and Licenses

5.1 Software Copyrights

6 About
This is the documentation for Espressif Audio Development Framework.

<table>
<thead>
<tr>
<th>Get Started</th>
<th>API Reference</th>
<th>Design Guide</th>
</tr>
</thead>
</table>

Contents

1
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.

1.1 ESP32-LyraT V4.3 Getting Started Guide

This guide provides users with functional descriptions, configuration options for ESP32-LyraT V4.3 audio development board, as well as how to get started with the ESP32-LyraT board. Check section Other Versions of LyraT, if you have different version of this board.
The ESP32-LyraT is a hardware platform designed for the dual-core ESP32 audio applications, e.g., Wi-Fi or BT audio speakers, speech-based remote controllers, smart-home appliances with audio functionality(ies), etc.

### 1.1.1 What You Need

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

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

## Overview

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

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

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

### Components

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

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

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

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

**Left Speaker Output**  Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1” pitch.
Fig. 1: ESP32-LyraT Block Diagram

Fig. 2: ESP32-LyraT V4.3 Board Layout Overview
Right Speaker Output  Output socket to connect 4 ohm speaker. The pins have a standard 2.54 mm / 0.1” pitch.

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

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

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

USB Power Port  Provides the power supply for the board.

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

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

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

1.1.2 Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

2. Plug in the Micro-USB cables to the PC and to both USB ports of the ESP32 LyraT.

3. The Standby LED (green) should turn on. Assuming that a battery is not connected, the Charging LED (red) will blink every couple of seconds.

4. Toggle left the Power On Switch.

5. The red Power On LED should turn on.

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

Develop Applications

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

- Setup ESP-IDF in your PC that provides a common framework to develop applications for the ESP32 in C language;
- Get ESP-ADF to have the API specific for the audio applications;
- Setup Path to ESP-ADF to make the framework aware of the audio specific API;
- Start a Project that will provide a sample audio application for the ESP32-LyraT board;
1.1.3 Summary of Key Changes from LyraT V4.2

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

1.1.4 Other Versions of LyraT

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

1.1.5 Related Documents

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

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

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

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

1.2.1 What You Need

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

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

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

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

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

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

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

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

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

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

Fig. 5: ESP32-LyraTD-MSC V2.2 Lower Board (A) Components

USB-UART Port Functions as the communication interface between a PC and the ESP32 WROVER module.
USB Power Port  Provides the power supply for the board.

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

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

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

---

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

    Reset: pressing this button alone resets the system.

---

1.2.2 Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

1. Connect 4-ohm speakers to the Right and Left Speaker Output. Connecting headphones to the Headphone Output is an option.
2. Plug in the Micro-USB cables to the PC and to both USB ports of the ESP32-LyraTD-MSC.

3. The Standby LED (green) should turn on. Assuming that a battery is not connected, the Charging LED (red) will blink every couple of seconds.

4. Toggle right the Power Switch.

5. The red Power On LED should turn on.

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

Develop Applications

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

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

1.2.3 Related Documents

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

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

1.3 About ESP-ADF

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

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

1.4 Setup ESP-IDF

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

You have a choice to compile and upload code to the ESP32 by command line with make or using Eclipse IDE.
Note: We are using ~/esp directory to install the toolchain, ESP-IDF, ESP-ADF and sample applications. You can use a different directory, but need to adjust respective commands.

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

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

1.5 Get ESP-ADF

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

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

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

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

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

1.6 Setup Path to ESP-ADF

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

1.7 Start a Project

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

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

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

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

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

### 1.8 Connect and Configure

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

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

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

Save the configuration.

### 1.9 Build, Flash and Monitor

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

```
  make flash monitor -j5
```

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

#### 1.9.1 Upload

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

```
  ...
  esptool.py v2.1
  Connecting.........______
```

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

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

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

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

```
... 
I (303) PLAY_MP3_FLASH: [1 ] Start audio codec chip 
I (323) PLAY_MP3_FLASH: [2 ] Create audio pipeline, add all elements to pipeline, and subscribe pipeline event 
I (323) PLAY_MP3_FLASH: [2.1] Create mp3 decoder to decode mp3 file and set custom read callback 
I (323) 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

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

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

### 1.10 Update ESP-ADF

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

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

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

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

1.11.1 ESP32-LyraT V4.2 Getting Started Guide

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

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

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

What You Need

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

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.
Fig. 8: ESP32-LyraT V4.2 Board Layout
**Right Microphone**  Onboard microphone connected to IN1 of the Audio Codec Chip.  

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

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

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

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

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

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

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

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

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

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

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

**USB Power Port** Provides the power supply for the board.  

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

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

**Power On Switch** Power on/off knob: twiggling it to the left powers the board on; twiggling 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>1 GPIO19</td>
<td>Red LED</td>
</tr>
<tr>
<td>2 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 GPIO33</td>
<td>Play</td>
</tr>
<tr>
<td>2 GPIO32</td>
<td>Set</td>
</tr>
<tr>
<td>3 GPIO13</td>
<td>Vol-</td>
</tr>
<tr>
<td>4 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>1 MTDI / GPIO12</td>
<td>DATA2</td>
</tr>
<tr>
<td>2 MTCK / GPIO13</td>
<td>CD / DATA3</td>
</tr>
<tr>
<td>3 MTDO / GPIO15</td>
<td>CMD</td>
</tr>
<tr>
<td>4 MTMS / GPIO14</td>
<td>CLK</td>
</tr>
<tr>
<td>5 GPIO2</td>
<td>DATA0</td>
</tr>
<tr>
<td>6 GPIO4</td>
<td>DATA1</td>
</tr>
<tr>
<td>7 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>

EN and IO0 Headers / JP23 and J24

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

<table>
<thead>
<tr>
<th>ESP32 Pin</th>
<th>Header Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 n/a</td>
<td>IO0_Auto</td>
</tr>
<tr>
<td>2 GPIO0</td>
<td>IO0</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:

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

**Related Documents**

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

### 1.11.2 ESP32-LyraT V4 Getting Started Guide

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

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

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

**What You Need**

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

**Overview**

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

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

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

![Fig. 9: ESP32-LyraT block diagram](image)

**Functional Description**

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

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

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

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

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

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

**I2C Header** Provides access to the I2C interface. Both ESP32-WROVER Module and Audio Codec Chip are connected to this interface. See I2C Header / JP5 for pinout details.
**MicroSD Card** The development board supports a MicroSD card in SPI/1-bit/4-bit modes, and can store or play audio files in the MicroSD card. See *MicroSD Card / J5* for pinout details. Note that JTAG cannot be used and should be disconnected by setting **Function DIP Switch** when **MicroSD Card** is in operation, because some of the signals are shared by both devices.

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

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

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

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

![Fig. 10: ESP32 LyraT V4 board layout](image)

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

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

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

**PA Chip** A power amplifier used to amplify stereo audio signal from the **Audio Codec Chip** for driving two 4-ohm speakers.
Boot/Reset Press Keys  
Boot: holding down the Boot button and momentarily pressing the Reset button initiates the firmware upload mode. Then user can upload firmware through the serial port. Reset: pressing this button alone resets the system.

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

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

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

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

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

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

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

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

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

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

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

Hardware Setup Options

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

Enable MicroSD Card in 1-wire Mode

<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 ¹</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-t** 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-t** 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-t** touch button is not available for use with the API
• **AUX Input** detection from the API is not available

### Allocation of ESP32 Pins

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

<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>MTDI / 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>
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:

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

**Related Documents**

- ESP32 LyraT V4 schematic (PDF)
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging
This API provides a way to develop audio applications using *Elements* like *Codecs* (Decoders and Encoders), *Streams* or *Audio Processing* functions.

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_t` 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.
Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

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

Return data pointer

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• tag: The tag name pointer

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

Return Element tag name pointer

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

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

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• uri: The uri pointer

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

Return URI pointer

Parameters
• el: The audio element handle

esp_err_t audio_element_run (audio_element_handle_t el)
Start Audio Element. With this function, audio_element will start as freeRTOS task, and put the task into ‘PAUSED’ state. Note: Element does not actually start when this function returns.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
esp_err_t audio_element_terminate(audio_element_handle_t el)
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.

2.1. Audio Framework 37
Return

• ESP_OK
• ESP_FAIL

Parameters

• el: The audio element handle

\[ \text{esp_err_t audio_element_resume}(\text{audio_element_handle_t el, float wait_for_rb_threshold, TickType_t timeout}) \]

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 \( \text{wait_for_rb_threshold} \). If the timeout period has been exceeded and ringbuffer output has not yet reached \( \text{wait_for_rb_threshold} \) 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

\[ \text{esp_err_t audio_element_msg_set_listener}(\text{audio_element_handle_t el, au-
dio_event_iface_handle_t listener}) \]

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

Return

• ESP_OK
• ESP_FAIL

Parameters

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

\[ \text{esp_err_t audio_element_set_event_callback}(\text{audio_element_handle_t el, event_cb_func cb_func, void *ctx}) \]

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

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

esp_err_t audio_element_set_input_ringbuf(audio_element_handle_t el, ringbuf_handle_t rb)
Set Element input ringbuffer.

Return
• ESP_OK
• ESP_FAIL
Parameters
• el: The audio element handle
• rb: The ringbuffer handle

ringbuf_handle_t audio_element_get_input_ringbuf(audio_element_handle_t el)
Get Element input ringbuffer.

Return ringbuf_handle_t
Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL
Parameters
• el: The audio element handle
• rb: The ringbuffer handle

ringbuf_handle_t audio_element_get_output_ringbuf(audio_element_handle_t el)
Get Element output ringbuffer.

Return ringbuf_handle_t
Parameters
• el: The audio element handle

audio_element_state_t audio_element_get_state(audio_element_handle_t el)
Get current Element state.

Return audio_element_state_t

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

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

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle

esp_err_t audio_element_wait_for_buffer(audio_element_handle_t el, int size_expect, TickType_t timeout)
This function will wait until the 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_set_input_timeout(audio_element_handle_t el, TickType_t timeout)`
Set input read timeout (default is `portMAX_DELAY`).

Return

• ESP_OK
• ESP_FAIL

Parameters

• el: The audio element handle
• timeout: The timeout

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

Return

• ESP_OK
• ESP_FAIL

Parameters

• el: The audio element handle
• timeout: The timeout

2.1. Audio Framework
esp_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

esp_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 output 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

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

```c
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

```c
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

```c
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

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

**Return**

- ESP_OK
- ESP_ERR_INVALID_ARG

**Parameters**

- el: The audio element handle
- buffer: The buffer pointer
- wanted_size: The wanted size
- index: The index of multi input ringbuffer, start from 0, should be less than NUMBER_OF_MULTI_RINGBUF
- ticks_to_wait: Timeout of ringbuffer
esp_err_t audio_element_multi_output (audio_element_handle_t el, char *buffer, int wanted_size, TickType_t ticks_to_wait)

Call this function write data by multi output ringbuffer.

Return
• ESP_OK
• ESP_FAIL

Parameters
• el: The audio element handle
• buffer: The buffer pointer
• wanted_size: The wanted size
• ticks_to_wait: Timeout of ringbuffer

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

Set multi input ringbuffer Element.

Return
• ESP_OK
• ESP_FAIL

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

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

Set multi output ringbuffer Element.

Return
• ESP_OK
• ESP_ERR_INVALID_ARG

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

ringbuf_handle_t audio_element_get_multi_input_ringbuf (audio_element_handle_t el, int index)

Get handle of multi input ringbuffer Element by index.

Return
• NULL Error
• Others ringbuf_handle_t

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

`ringbuf_handle_t audio_element_get_multi_output_ringbuf(audio_element_handle_t el, int index)`

Get handle of multi output ringbuffer Element by index.

Return
• NULL Error
• Others ringbuf_handle_t

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

Structures

`struct audio_element_reserve_data_t`
Audio Element user reserved data.

Public Members

int user_data_0
  user data 0
int user_data_1
  user data 1

`struct audio_element_info_t`
Audio Element informations.

Public Members

int sample_rates
  Sample rates in Hz
int channels
  Number of audio channel, mono is 1, stereo is 2
int bits
  Bit wide (8, 16, 24, 32 bits)
int volume
  Volume in percent
bool mute
  Mute
int64_t *byte_pos
    The current position (in bytes) being processed for an element

int64_t *total_bytes
    The total bytes for an element

char *uri
    URI (optional)

audio_codec_t codec_fmt
    Music format (optional)

audio_element_reserve_data_t *reserve_data
    This value is reserved for user use (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

io_func seek
    Seek callback function

process_func process
    Process callback function

io_func close
    Close callback function

io_func destroy
    Destroy callback function

stream_func read
    Read callback function

stream_func write
    Write callback function

int buffer_len
    Buffer length use for an Element

int task_stack
    Element task stack

int task_prio
    Element task priority (based on freeRTOS priority)

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

int out_rb_size
    Output ringbuffer size

void *data
    User context
char *tag
    Element tag

bool enable_multi_io
    Enable multi input and output ringbuffer

Macros

AUDIO_ELEMENT_INFO_DEFAULT()

DEFAULT_ELEMENT_RINGBUF_SIZE

DEFAULT_ELEMENT_BUFFER_LENGTH

DEFAULT_ELEMENT_STACK_SIZE

DEFAULT_ELEMENT_TASK_PRIO

DEFAULT_ELEMENT_TASK_CORE

DEFAULT_AUDIO_ELEMENT_CONFIG()

Type Definitions

typedef struct audio_element *audio_element_handle_t

typedef esp_err_t (*io_func)(audio_element_handle_t self)

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

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

typedef esp_err_t (*event_cb_func)(audio_element_handle_t el, audio_event_iface_msg_t *event, void *ctx)

Enumerations

enum audio_element_err_t
    Values:
    AEL_IO_OK = ESP_OK
    AEL_IO_FAIL = ESP_FAIL
    AEL_IO_DONE = -2
    AEL_IO_ABORT = -3
    AEL_IO_TIMEOUT = -4
    AEL_PROCESS_FAIL = -5

enum audio_element_state_t
    Audio element state.
    Values:
    AEL_STATE_NONE = 0
    AEL_STATE_INIT
AEL_STATE_RUNNING
AEL_STATE_PAUSED
AEL_STATE_STOPPED
AEL_STATE_FINISHED
AEL_STATE_ERROR

enum audio_element_msg_cmd_t
Audio element action command, process on dispatcher
Values:
AEL_MSG_CMD_NONE = 0
AEL_MSG_CMD_ERROR = 1
AEL_MSG_CMD_FINISH = 2
AEL_MSG_CMD_STOP = 3
AEL_MSG_CMD_PAUSE = 4
AEL_MSG_CMD_RESUME = 5
AEL_MSG_CMD_DESTROY = 6
AEL_MSG_CMD_REPORT_STATUS = 8
AEL_MSG_CMD_REPORT_MUSIC_INFO = 9
AEL_MSG_CMD_REPORT_CODEC_FMT = 10

enum audio_element_status_t
Audio element status report
Values:
AEL_STATUS_NONE = 0
AEL_STATUS_ERROR_OPEN = 1
AEL_STATUS_ERROR_INPUT = 2
AEL_STATUS_ERROR_PROCESS = 3
AEL_STATUS_ERROR_OUTPUT = 4
AEL_STATUS_ERROR_CLOSE = 5
AEL_STATUS_ERROR_TIMEOUT = 6
AEL_STATUS_ERROR_UNKNOWN = 7
AEL_STATUS_INPUT_DONE = 8
AEL_STATUS_INPUT_BUFFERING = 9
AEL_STATUS_OUTPUT_DONE = 10
AEL_STATUS_OUTPUT_BUFFERING = 11
AEL_STATUS_STATE_RUNNING = 12
AEL_STATUS_STATE_PAUSED = 13
AEL_STATUS_STATE_STOPPED = 14
AEL_STATUS_STATE_FINISHED = 15
**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.

![Audio Pipeline Diagram](image)

**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

```c
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

```c
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

```c
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

```c
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.
Read the Docs Template Documentation

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_resume(audio_pipeline_handle_t pipeline)
This function will set all the elements to the RUNNING state and process the audio data as an inherent feature of audio_pipeline.

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_pause(audio_pipeline_handle_t pipeline)
This function will set all the elements to the PAUSED state. Everything remains the same except the data processing is stopped.

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• pipeline: The Audio Pipeline Handle

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

Return

• ESP_OK on success
• ESP_FAIL when any errors

Parameters

• pipeline: The Audio Pipeline Handle

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

Return

• ESP_OK on success
• ESP_FAIL when any errors
Parameters

- **pipeline**: The Audio Pipeline Handle

```c
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
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 conect_rb, audio_element_handle_t next)

Insert the specific audio_element to audio_pipeline, previous element connect to the next element by ring buffer.

Return
• ESP_OK
• ESP_FAIL

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

esp_err_t audio_pipeline_register_more(audio_pipeline_handle_t pipeline, audio_element_handle_t element_1, ...)

Register a NULL-terminated list of elements to audio_pipeline.

Return
• ESP_OK
• ESP_FAIL

Parameters
• pipeline: The audio pipeline handle
• element_1: The element to add to the audio_pipeline.
esp_err_t audio_pipeline_unregister_more(audio_pipeline_handle_t pipeline, audio_element_handle_t element_1, ...)
Unregister a NULL-terminated list of elements to audio_pipeline.

Return
- ESP_OK
- ESP_FAIL

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

esp_err_t audio_pipeline_link_more(audio_pipeline_handle_t pipeline, audio_element_handle_t element_1, ...)
Adds a NULL-terminated list of elements to audio_pipeline.

Return
- ESP_OK
- ESP_FAIL

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

esp_err_t audio_pipeline_listen_more(audio_pipeline_handle_t pipeline, audio_element_handle_t element_1, ...)
Subscribe a NULL-terminated list of element’s events to audio_pipeline.

Return
- ESP_OK
- ESP_FAIL

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

esp_err_t audio_pipeline_check_items_state(audio_pipeline_handle_t pipeline, audio_element_handle_t dest_el, audio_element_status_t status)
Update the destination element state and check the all of linked elements state are same.

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

Parameters
• pipeline: The audio pipeline handle
• dest_el: Destination element
• status: The new status

esp_err_t audio_pipeline_reset_items_state(audio_pipeline_handle_t pipeline)
Reset pipeline element items state to AEL_STATUS_NONE

Return
• ESP_OK on success
• ESP_FAIL when any errors

Parameters
• pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_reset_ringbuffer(audio_pipeline_handle_t pipeline)
Reset pipeline element ringbuffer.

Return
• ESP_OK on success
• ESP_FAIL when any errors

Parameters
• pipeline: The Audio Pipeline Handle

esp_err_t audio_pipeline_breakup_elements(audio_pipeline_handle_t pipeline, audio_element_handle_t kept_ctx_el)
Break up all the linked elements of specific pipeline. The include and before kept_ctx_el 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.

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

struct audio_pipeline_cfg
  Audio Pipeline configurations.

Public Members

int rb_size
  Audio Pipeline ringbuffer size

2.1. Audio Framework
Macros

DEFAULT_PIPELINE_RINGBUF_SIZE
DEFAULT_AUDIO_PIPELINE_CONFIG()

Type Definitions

typedef struct audio_pipeline *audio_pipeline_handle_t

typedef struct audio_pipeline_cfg audio_pipeline_cfg_t
    Audio Pipeline configurations.

2.1.3 Event Interface

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

Application Examples

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

API Reference

Header File

• audio_pipeline/include/audio_event_iface.h

Functions

audio_event_iface_handle_t audio_event_iface_init (audio_event_iface_cfg_t *config)
    Initialize audio event.

    Return
    • ESP_OK
    • ESP_FAIL

    Parameters
    • config: The configurations

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

    Return
    • ESP_OK
    • ESP_FAIL

    Parameters
• **evt**: The event

```c
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 `listener`.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

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

```c
esp_err_t audio_event_iface_remove_listener(audio_event_iface_handle_t listener, audio_event_iface_handle_t evt)
```

Remove audio event `evt` from the listener.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

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

```c
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

```c
esp_err_t audio_event_iface_waiting_cmd_msg(audio_event_iface_handle_t evt)
```

Waiting internal queue message.

**Return**

- ESP_OK
- ESP_FAIL

**Parameters**

- **evt**: The event
esp_err_t audio_event_iface_cmd(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

Trigger an event for internal queue with a message.

Return
  • ESP_OK
  • ESP_FAIL

Parameters
  • evt: The event
  • msg: The message

esp_err_t audio_event_iface_cmd_from_isr(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

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

Return
  • ESP_OK
  • ESP_FAIL

Parameters
  • evt: The event
  • msg: The message

esp_err_t audio_event_iface_sendout(audio_event_iface_handle_t evt, audio_event_iface_msg_t *msg)

Trigger and event out with a message.

Return
  • ESP_OK
  • ESP_FAIL

Parameters
  • evt: The event
  • msg: The message

esp_err_t audio_event_iface_discard(audio_event_iface_handle_t evt)

Discard all ongoing event message.

Return
  • ESP_OK
  • ESP_FAIL

Parameters
  • evt: The event

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

Listening and invoke callback function if there are any event are coming.
Return

- ESP_OK
- ESP_FAIL

Parameters

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

QueueHandle_t audio_event_iface_get_queue_handle(audio_event_iface_handle_t evt)

Get External queue handle of Emmitter.

Return External QueueHandle_t

Parameters

- evt: The external queue

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

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

Return

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

Parameters

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

QueueHandle_t audio_event_iface_get_msg_queue_handle(audio_event_iface_handle_t evt)

Get Internal queue handle of Emmitter.

Return Internal QueueHandle_t

Parameters

- evt: The Internal queue

esp_err_t audio_event_iface_set_msg_listener(audio_event_iface_handle_t evt, audio_event_iface_handle_t listener)

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

```c
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
```

```c
struct audio_event_iface_cfg_t
    Event interface configurations

    Public Members

    int internal_queue_size
        It’s optional, Queue size for event internal_queue
    int external_queue_size
        It’s optional, Queue size for event external_queue
    int queue_set_size
        It’s optional, QueueSet size for event queue_set
    on_event_iface_func on_cmd
        Function callback for listener when any event arrived
    void *context
        Context will pass to callback function
    TickType_t wait_time
        Timeout to check for event queue
    int type
        it will pass to audio_event_iface_msg_t source_type (To know where it came from)
```

Macros

- DEFAULT_AUDIO_EVENT_IFACE_SIZE
- AUDIO_EVENT_IFACE_DEFAULT_CFG()
Type Definitions

typedef esp_err_t (*on_event_iface_func)(audio_event_iface_msg_t *, void *)
typedef struct audio_event_iface *audio_event_iface_handle_t

2.1.4 Audio Common

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

API Reference

Header File

- audio_pipeline/include/audio_common.h

Macros

ELEMENT_SUB_TYPE_OFFSET

mem_assert (x)

Enumerations

enum audio_element_type_t
Values:

AUDIO_ELEMENT_TYPE_UNKNOWN = 0x01<<ELEMENT_SUB_TYPE_OFFSET
AUDIO_ELEMENT_TYPE_ELEMENT = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+1)
AUDIO_ELEMENT_TYPE_PLAYER = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+2)
AUDIO_ELEMENT_TYPE_SERVICE = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+3)
AUDIO_ELEMENT_TYPE_PERIPH = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+4)

enum audio_stream_type_t
Values:

AUDIO_STREAM_NONE = 0
AUDIO_STREAM_READER
AUDIO_STREAM_WRITER

enum audio_codec_type_t
Values:

AUDIO_CODEC_TYPE_NONE = 0
AUDIO_CODEC_TYPE_DECODER
AUDIO_CODEC_TYPE_ENCODER

enum audio_codec_t
Values:
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

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
```

12S Stream configurations Default value will be used if any entry is zero.

Public Members

```c
audio_stream_type_t type
```
Type of stream

```c
i2s_config_t i2s_config
```
12S driver configurations

```c
i2s_port_t i2s_port
```
12S 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)

Macros

```c
I2S_STREAM_TASK_STACK
```

```c
I2S_STREAM_BUF_SIZE
```

```c
I2S_STREAM_TASK_PRIO
```

```c
I2S_STREAM_TASK_CORE
```

```c
I2S_STREAM_RINGBUFFER_SIZE
```

```c
I2S_STREAM_CFG_DEFAULT()
```

```c
I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT()
```
2.2.2 HTTP Stream

Header File

- audio_stream/include/http_stream.h

Functions

audio_element_handle_t http_stream_init (http_stream_cfg_t *config)
Create a handle to an Audio Element to stream data from HTTP to another Element or get data from other
elements sent to HTTP, depending on the configuration the stream type, either AUDIO_STREAM_READER or
AUDIO_STREAM_WRITER.

Return The Audio Element handle

Parameters

- config: The configuration

esp_err_t http_stream_next_track (audio_element_handle_t el)
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)

Structures

struct http_stream_event_msg_t
Stream event message.

Public Members

http_stream_event_id_t event_id
Event ID

void *http_client
Reference to HTTP Client using by this HTTP Stream

void *buffer
Reference to Buffer using by the Audio Element

int buffer_len
Length of buffer
void *user_data
    User data context, from http_stream_cfg_t

audio_element_handle_t el
    Audio element context

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

**Public Members**

audio_stream_type_t type
    Type of stream

int out_rb_size
    Size of output ringbuffer

int task_stack
    Task stack size

int task_core
    Task running in core (0 or 1)

int task_prio
    Task priority (based on freeRTOS priority)

http_stream_event_handle_t event_handle
    The hook function for HTTP Stream

void *user_data
    User data context

bool auto_connect_next_track
    connect next track without open/close

bool enable_playlist_parser
    Enable playlist parser

**Macros**

HTTP_STREAM_TASK_STACK

HTTP_STREAM_TASK_CORE

HTTP_STREAM_TASK_PRIO

HTTP_STREAM_RINGBUFFER_SIZE

HTTP_STREAM_CFG_DEFAULT()

**Type Definitions**

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

```c
enum http_stream_event_id_t
{
    HTTP_STREAM_PRE_REQUEST = 0x01,
    HTTP_STREAM_ON_REQUEST,
    HTTP_STREAM_ON_RESPONSE,
    HTTP_STREAM_POST_REQUEST,
    HTTP_STREAM_FINISH_REQUEST,
    HTTP_STREAM_RESOLVE_ALL_TRACKS,
    HTTP_STREAM_FINISH_TRACK,
    HTTP_STREAM_FINISH_PLAYLIST
}
```

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

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

audio_stream_type_t type
Stream type

int buf_sz
Audio Element Buffer size

int out_rb_size
Size of output ringbuffer

int task_stack
Task stack size

int task_core
Task running in core (0 or 1)

int task_prio
Task priority (based on freeRTOS priority)

Macros

FATFS_STREAM_BUF_SIZE
FATFS_STREAM_TASK_STACK
FATFS_STREAM_TASK_CORE
FATFS_STREAM_TASK_PRIO
FATFS_STREAM_RINGBUFFER_SIZE
FATFS_STREAM_CFG_DEFAULT()

2.2.4 Raw Stream

Header File

- audio_stream/include/raw_stream.h

Functions

audio_element_handle_t raw_stream_init (raw_stream_cfg_t *cfg)
Initialize RAW stream.

Return The audio element handle

Parameters

- cfg: The RAW Stream configuration

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

Return Number of bytes actually read.

Parameters
• **pipeline**: The audio pipeline handle
• **buffer**: The buffer
• **buf_size**: Maximum number of bytes to be read.

```c
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**

```c
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**

```c
audio_stream_type_t type
```

Type of stream

```c
int out_rb_size
```

Size of output ringbuffer

**Macros**

```c
RAW_STREAM_RINGBUFFER_SIZE
RAW_STREAM_CFG_DEFAULT()
```

### 2.2.5 Spiffs Stream

**Header File**

```c
• 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

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

API Reference - AMRNB Encoder

API Reference - AMRWB Encoder

2.3.3 FLAC Decoder

Decode an audio data stream provided in FLAC format.

API Reference

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

2.3.5 OGG Decoder

Decode an audio data stream provided in OGG format.
API Reference

2.3.6 OPUS Decoder

Decode an audio data stream provided in OPUS format.

API Reference

2.3.7 WAV Decoder and Encoder

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

Application Examples

Implementation of this API is demonstrated in the following examples:

- player/pipeline_sdcard_wav
- recorder/pipeline_wav_sdcard

API Reference - Decoder

API Reference - Encoder

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.

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 newcome 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

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

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

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

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 polices 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

2.6 Speech Recognition

The ESP-ADF comes complete with *wakeup word libraries* and *speech recognition interface* to recognize voice wakeup commands. Most of currently implemented wakeup commands are in Chinese with one command “Alexa” 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 easy integrate with existing *Audio Framework* to retrieve the audio stream from a microphone connected to the audio chip.

2.6.1 Wakeup Word Libraries

Espressif speech recognition libraries contain several wakeup words split into models. Two models are provided:

- SR_MODEL_WN3_QUANT used for a single wakeup word,
- SR_MODEL_WN4_QUANT used for multi wakeup words.

Model selection is done in menuconfig by setting CONFIG_SR_MODEL_SEL.

**Single Wakeup Word Model**

This model is defined as SR_MODEL_WN3_QUANT in configuration and contains two libraries, one with wake word in Chinese and the other one in English.

<table>
<thead>
<tr>
<th>Library</th>
<th>Language</th>
<th>Wakeup Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>libnn_model_hilexin_wn3.a</td>
<td>Chinese</td>
<td>(Hài, lè xīn)</td>
</tr>
<tr>
<td>libnn_model_alexa_wn3.a</td>
<td>English</td>
<td>Alexa</td>
</tr>
</tbody>
</table>

To select desired wakeup word set CONFIG_NAME_OF_WAKEUP_WORD.
Multiple Wakeup Word Model

This model is defined as SR_MODEL_WN4_QUANT in configuration and contains two libraries with wakeup words in Chinese.

**Library** libnn_model_light_control_ch_wn4.a (Chinese)

<table>
<thead>
<tr>
<th>No.</th>
<th>Wakeup Words</th>
<th>Pronunciation</th>
<th>English Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dākāi diàndēng</td>
<td></td>
<td>Turn on the light</td>
</tr>
<tr>
<td>2</td>
<td>Guānbì diàndēng</td>
<td></td>
<td>Turn off the light</td>
</tr>
</tbody>
</table>

**Library** libnn_model_speech_cmd_ch_wn4 (Chinese)

<table>
<thead>
<tr>
<th>No.</th>
<th>Wakeup Words</th>
<th>Pronunciation</th>
<th>English Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hái, lè xīn</td>
<td></td>
<td>Hi, Espressif</td>
</tr>
<tr>
<td>2</td>
<td>Dākāi diàndēng</td>
<td></td>
<td>Turn on the light</td>
</tr>
<tr>
<td>3</td>
<td>Guānbì diàndēng</td>
<td></td>
<td>Turn off the light</td>
</tr>
<tr>
<td>4</td>
<td>Yīnlìāng jiā dà</td>
<td></td>
<td>Increase volume</td>
</tr>
<tr>
<td>5</td>
<td>Yīnlìāng jiān xiāo</td>
<td></td>
<td>Volume down</td>
</tr>
<tr>
<td>6</td>
<td>Bōfāng</td>
<td></td>
<td>Play</td>
</tr>
<tr>
<td>7</td>
<td>Zànting</td>
<td></td>
<td>Pause</td>
</tr>
<tr>
<td>8</td>
<td>Jingyīn</td>
<td></td>
<td>Mute</td>
</tr>
<tr>
<td>9</td>
<td>Bōfāng běndī gēqū</td>
<td></td>
<td>Play local music</td>
</tr>
</tbody>
</table>

To select desired set of multi wakeup words set CONFIG_NAME_OF_WAKEUP_WORD.

**API Reference**

Declarations of all available speech recognition models is contained in a header file esp-adf-libs/esp_sr/include/esp_sr_models.h.

**2.6.2 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.

![Sample Speech Recognition Pipeline](image)

Fig. 5: Sample Speech Recognition Pipeline

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_sr_iface.h"
#include "esp_sr_models.h"

static const sr_model_iface_t *model = &sr_model_wakenet3_quantized;

// Initialize wakeNet model data
static model_iface_data_t *model_data = model->create(DET_MODE_90);

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

// Get voice data feed to buffer
...

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

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

**Application Example**

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

**API Reference**

2.6.3 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**

2.6.4 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.
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);

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**

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

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_sdcard_unmount(). The data reading / writing is implemented in a separate API described in FatFs Stream.

Application Examples

Implementation of this API is demonstrated in couple of examples:

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

API Reference

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

2.7.5 Console Peripheral

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

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

Code example
```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_config_t periph_cfg = {
        .event_handle = _periph_event_handle,
        .user_context = NULL,
    };
    esp_periph_init(&periph_cfg);

    const periph_console_cmd_t cmd[] = {
        { .cmd = "play", .id = 1, .help = "Play audio" },
        { .cmd = "stop", .id = 2, .help = "Stop audio" },
        { .cmd = "test", .help = "test console", .func = console_test_cb },
    };

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

2.7. Peripherals
API Reference

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

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

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

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 single 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

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

<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.

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

```c
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

```c
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

```c
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

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

86 Chapter 2. API Reference
Return
  • ESP_OK
  • ESP_FAIL

Parameters
  • rb: The Ringbuffer handle

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

  Return total bytes available

Parameters
  • rb: The Ringbuffer handle

int rb_bytes_filled(ringbuf_handle_t rb)
  Get the number of bytes that have filled the ringbuffer.

  Return The number of bytes that have filled the ringbuffer

Parameters
  • rb: The Ringbuffer handle

int rb_get_size(ringbuf_handle_t rb)
  Get total size of Ringbuffer (in bytes)

  Return total size of Ringbuffer

Parameters
  • rb: The Ringbuffer handle

int rb_read(ringbuf_handle_t rb, char *buf, int len, TickType_t ticks_to_wait)
  Read from Ringbuffer to buf with len and wait tick_to_wait ticks until enough bytes to read if the ringbuffer bytes available is less than len. 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

int rb_write(ringbuf_handle_t rb, char *buf, int len, TickType_t ticks_to_wait)
  Write to Ringbuffer from buf with len and wait tick_to_wait ticks until enough space to write if the ringbuffer space available is less than len

  Return Number of bytes written

Parameters
Read the Docs Template Documentation

- rb: The Ringbuffer handle
- buf: The buffer
- len: The length
- ticks_to_wait: The ticks to wait

```c
int rb_size_get(ringbuf_handle_t rb)
Get total size of ringbuffer.

Return Total size of ringbuffer (in block byte(s))

Parameters
- rb: The Ringbuffer handle
```

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

Return
- ESP_OK
- ESP_FAIL

Parameters
- rb: The Ringbuffer handle
```

```c
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

```c
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

2.8.3 ES8388 Driver

Driver for ES8388 codec chip used in *ESP32 LyraT* audio board.

API Reference

2.8.4 ES8374 Driver

Driver for ES8374 codec chip.

API Reference

2.8.5 ZL38063 Driver

Driver for ZL38063 codec chip used in *ESP32-LyraTD-MSC* audio board.

API Reference

2.9 Configuration Options

Compile-time configuration options specific to ESP-ADF.

2.9.1 Audio HAL

**AUDIO_BOARD**

Audio board

*Found in: Audio HAL*

Select an audio board to use with the ESP-ADF

**Available options:**

- ESP_LYRAT_V4_3_BOARD
- ESP_LYRAT_V4_2_BOARD
- ESP_LYRATD_MSC_V2_1_BOARD
- ESP_LYRATD_MSC_V2_2_BOARD
• ESP_LYRAT_MINI_V1_1_BOARD
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.

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.
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.

3.2.1 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 `char *buf = heap_caps_malloc(1024 * 10, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT)` instead of `malloc(1024 * 10)` to use PSRAM, and call `char *buf = heap_caps_malloc(512, MALLOC_CAP_INTERNAL | MALLOC_CAP_8BIT)` to use internal RAM.

• Not relying on `malloc()` to automatically allocate PSRAM allows to make a full control of the memory. By avoiding the use of the internal RAM by other `malloc()` calls, you can reserve more memory for high-efficiency usage and task stack since PSRAM cannot be used as task stack memory.

• The task stack will always be allocated at internal RAM. On the other hand you can use of the `xTaskCreateStatic()` function that allows to create tasks with stack on PSRAM (see options in PSRAM and FreeRTOS menuconfig), but pay attention to its help information.

**Important:** Don’t use ROM code in `xTaskCreateStatic` task. The ROM code itself is linked in `components/esp32/id/esp32.rom.id`. 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></td>
</tr>
<tr>
<td>Wi-Fi ¹</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 ²</td>
<td>12kB+</td>
<td>12kB+</td>
</tr>
<tr>
<td>I2S ³</td>
<td>Configurable, 8kB for reference</td>
<td>Configurable, 8kB for reference</td>
</tr>
<tr>
<td>RingBuffer ⁴</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]

3.2. Design Considerations
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 fron 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

esp_periph_config_t periph_cfg = {
  .event_handle = _periph_event_handle,
  .user_context = NULL,
};
esp_periph_init(&periph_cfg);

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

esp_periph_start(button_handle);
ESP_LOGI(TAG, "wait for button Pressed or touched");

ESP_LOGI(TAG, "running...");
vTaskDelay(5000 / portTICK_RATE_MS);

esp_periph_stop(button_handle);
ESP_LOGI(TAG, "stop button...");
vTaskDelay(5000 / portTICK_RATE_MS);

esp_periph_start(button_handle);
ESP_LOGI(TAG, "start button...");
vTaskDelay(5000 / portTICK_RATE_MS);
ESP_LOGI(TAG, "destroy...");
esp_periph_destroy();

Audio Player

Example

esp_audio_cfg_t cfg = {
    .in_stream_buf_size = 4096, /*!< Input buffer size */
    .out_stream_buf_size = 4096, /*!< Output buffer size */
    .evt_que = NULL, /*!< Registered by user for receiving esp_audio event */
    .resample_rate = 48000, /*!< sample rate */
    .hal = NULL, /*!< */
};
 audio_hal_codec_config_t audio_hal_codec_cfg = AUDIO_HAL_ES8388_DEFAULT();
cfg.hal = audio_hal_init(&audio_hal_codec_cfg, 0);
 esp_audio_handle_t player = esp_audio_create(&cfg);
TEST_ASSERT_NOT_EQUAL(player, NULL);
 raw_stream_cfg_t raw_cfg = {
    .type = AUDIO_STREAM_READER,
};
 audio_element_handle_t raw = raw_stream_init(&raw_cfg);
 wav_decoder_cfg_t wav_cfg = DEFAULT_WAV_DECODER_CONFIG();
 audio_element_handle_t wav = wav_decoder_init(&wav_cfg);

 fatfs_stream_cfg_t fatfs_cfg = {
    .type = AUDIO_STREAM_READER,
    .root_path = "/sdcard",
};
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
 esp_audio_input_stream_add(player, fatfs_stream_init(&fatfs_cfg));
i2s_cfg.type = AUDIO_STREAM_WRITER;
 esp_audio_output_stream_add(player, i2s_stream_init(&i2s_cfg));
 wav_decoder_cfg_t wav_cfg = DEFAULT_WAV_DECODER_CONFIG();
 esp_audio_codec_lib_add(player, AUDIO_CODEC_TYPE_DECODER, wav);

3.4 Development Boards

Hardware details of audio development boards designed by Espressif around ESP32.

3.4.1 ESP32-LyraT V4.3 Hardware Reference

This guide provides functional descriptions, configuration options for ESP32-LyraT V4.3 audio development board. As an introduction to functionality and using the LyraT, please see ESP32-LyraT V4.3 Getting Started Guide. Check section Other Versions of LyraT if you have different version of the board.
Overview

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

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

Functional Description

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

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

**Green LED**  A general purpose LED controlled by the ESP32-WROVER Module to indicate certain operation states of the audio application using dedicated API.

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

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.

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 ¹</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:
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>
</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>1 MCLK</td>
<td>GPIO10</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
Power System:

USB<>UART:

Charge Circuit:

Accuracy of R5 should be 1%

Power Switch:

Module Power Supply:

Fig. 6: ESP32 LyraT V4.3 - Power Supply Separation

Fig. 7: ESP32 LyraT V4.3 - Dedicated LDO for the ESP32 Module
Two separate LDOs are provided for the MicroSD 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.

SDIO Power Supply:

![SDIO Power Supply Diagram](image)

*Fig. 8: ESP32 LyraT V4.3 - Dedicated LDO for the MicroSD Card*

Separate Power Feed for the PAs

The audio amplifier unit features two NS4150 that require a large power supply for driving external speakers with the maximum output power of 3W. The power is supplied directly to both PAs from the battery or the USB. The 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:

![PA Output Diagram](image)

*Fig. 9: ESP32 LyraT V4.3 - Power Supply for the PAs*
**Selecting of the Audio Output**

The development board uses two mono Class D amplifier ICs, model number NS4150 with maximum output power of 3W and operating voltage from 3.0V to 5.25V.

The audio input source is the digital-to-analog converter (DAC) output of the ES8388. Audio output supports two external speakers.

An optional audio output is a pair of headphones feed from the same DACs as the amplifier ICs.

To switch between using headphones and speakers, the board provides a digital input signal to detect when a headphone jack is inserted and a digital output signal to enable or disable the amplifier ICs. In other words selection between speakers and headphones is under software control instead of using mechanical contacts that would disconnect speakers once a headphone jack is inserted.

**Other Versions of LyraT**

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

**Related Documents**

- ESP32 LyraT V4.3 schematic (PDF)
- ESP32-LyraT V4.3 Getting Started Guide
- ESP32 Datasheet (PDF)
- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging

**3.5 Audio Samples**

Music files in this section are intended for testing of audio applications. The files are organized into different Formats and Sample Rates.

**3.5.1 Formats**

The tables below provides an audio file converted from ‘wav’ format into several other audio formats.

**Long Samples**

The audio track duration in this section is 3 minutes and 7 seconds.
## Two Channel Audio

<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/
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.
CHAPTER 6

About

This is documentation of ESP-ADF, the framework to develop audio applications for ESP32 chip by Espressif.

The ESP32 is 2.4 GHz Wi-Fi and Bluetooth combo, 32 bit dual core chip running up to 240 MHz, designed for mobile, wearable electronics, and Internet-of-Things (IoT) applications. It has several peripherals on board including I2S interfaces to easy integrate with dedicated audio chips. These hardware features together with the ESP-ADF software provide a powerful platform to implement audio applications including native wireless networking and powerful user interface.

The ESP-ADF provides a range of API components including Audio Streams, Codecs and Services organized in Audio Pipeline, all integrated with audio hardware through Media HAL and with Peripherals onboard of ESP32.

The ESP-ADF also provides integration with Baidu DauerOS cloud services. A range of components is coming to provide integration with DeepBrain, Amazon, Google, Alibaba and Turing cloud services.

The ESP-ADF builds on well established, FreeRTOS based, Espressif IOT Development Framework ESP-IDF.

• genindex
Fig. 1: Espressif Audio Development Framework
Index

A

AEL_IO_ABORT (C++ enumerator), 48
AEL_IO_DONE (C++ enumerator), 48
AEL_IO_FAIL (C++ enumerator), 48
AEL_IO_OK (C++ enumerator), 48
AEL_IO_TIMEOUT (C++ enumerator), 48
AEL_MSG_CMD_DESTROY (C++ enumerator), 49
AEL_MSG_CMD_ERROR (C++ enumerator), 49
AEL_MSG_CMD_FINISH (C++ enumerator), 49
AEL_MSG_CMD_NONE (C++ enumerator), 49
AEL_MSG_CMD_PAUSE (C++ enumerator), 49
AEL_MSG_CMD_REPORT_CODEC_FMT (C++ enumerator), 49
AEL_MSG_CMD_REPORT_MUSIC_INFO (C++ enumerator), 49
AEL_MSG_CMD_RESUME (C++ enumerator), 49
AEL_MSG_CMD_STOP (C++ enumerator), 49
AEL_PROCESS_FAIL (C++ enumerator), 48
AEL_STATE_ERROR (C++ enumerator), 49
AEL_STATE_FINISHED (C++ enumerator), 49
AEL_STATE_INIT (C++ enumerator), 48
AEL_STATE_NONE (C++ enumerator), 48
AEL_STATE_PAUSED (C++ enumerator), 49
AEL_STATE_RUNNING (C++ enumerator), 49
AEL_STATE_STOPPED (C++ enumerator), 49
AEL_STATUS_CLOSE (C++ enumerator), 49
AEL_STATUS_ERROR_INPUT (C++ enumerator), 49
AEL_STATUS_ERROR_OUTPUT (C++ enumerator), 49
AEL_STATUS_ERROR_PROCESS (C++ enumerator), 49
AEL_STATUS_ERROR_TIMEOUT (C++ enumerator), 49
AEL_STATUS_ERROR_UNKNOWN (C++ enumerator), 49
AEL_STATUS_INPUT_BUFFERING (C++ enumerator), 49
AEL_STATUS_INPUT_DONE (C++ enumerator), 49
AEL_STATUS_MOUNTED (C++ enumerator), 49
AEL_STATUS_NONE (C++ enumerator), 49
AEL_STATUS_OUTPUT_BUFFERING (C++ enumerator), 49
AEL_STATUS_OUTPUT_DONE (C++ enumerator), 49
AEL_STATUS_STATE_FINISHED (C++ enumerator), 49
AEL_STATUS_STATE_PAUSED (C++ enumerator), 49
AEL_STATUS_STATE_RUNNING (C++ enumerator), 49
AEL_STATUS_STATE_STOPPED (C++ enumerator), 49
AEL_STATUS_UNMOUNTED (C++ enumerator), 50
AUDIO_CODEC_AAC (C++ enumerator), 64
AUDIO_CODEC_AMR (C++ enumerator), 64
AUDIO_CODEC_M4A (C++ enumerator), 64
AUDIO_CODEC_MP3 (C++ enumerator), 64
AUDIO_CODEC_NONE (C++ enumerator), 63
AUDIO_CODEC_OPUS (C++ enumerator), 64
AUDIO_CODEC_RAW (C++ enumerator), 64
audio_codec_t (C++ type), 63
audio_codec_t::buffer_len (C++ member), 63
audio_codec_t::close (C++ member), 63
audio_codec_type_t (C++ type), 63
audio_codec_type_t::buffer_len (C++ member), 63
audio_codec_type_t::close (C++ member), 63
audio_element_abort_input_ringbuf (C++ function), 40
audio_element_abort_output_ringbuf (C++ function), 40
audio_element_abort_input_ringbuf (C++ function), 40
audio_element_cfg_t (C++ class), 47
audio_element_cfg_t::buffer_len (C++ member), 47
audio_element_cfg_t::close (C++ member), 47
audio_element_abort_input_ringbuf (C++ function), 40
audio_element_abort_output_ringbuf (C++ function), 40
audio_element_abort_input_ringbuf (C++ function), 40
audio_element_cfg_t (C++ class), 47
audio_element_cfg_t::buffer_len (C++ member), 47
audio_element_cfg_t::close (C++ member), 47
audio_element_cfg_t (C++ class), 47
audio_element_cfg_t::buffer_len (C++ member), 47
audio_element_cfg_t::close (C++ member), 47
audio_element_cfg_t::data (C++ member), 47
audio_element_cfg_t::destroy (C++ member), 47
audio_element_cfg_t::enable_multi_io (C++ member), 48
audio_element_cfg_t::open (C++ member), 47
audio_element_cfg_t::out_rb_size (C++ member), 47
audio_element_cfg_t::process (C++ member), 47
audio_element_cfg_t::read (C++ member), 47
audio_element_cfg_t::seek (C++ member), 47
audio_element_cfg_t::tag (C++ member), 47
audio_element_cfg_t::task_core (C++ member), 47
audio_element_cfg_t::task_prio (C++ member), 47
audio_element_cfg_t::task_stack (C++ member), 47
audio_element_cfg_t::write (C++ member), 47
audio_element_deinit (C++ function), 34
audio_element_err_t (C++ type), 48
audio_element_get_event_queue (C++ function), 43
audio_element_get_input_ringbuf (C++ function), 39
audio_element_get_multi_input_ringbuf (C++ function), 45
audio_element_get_output_ringbuf (C++ function), 39
audio_element_get_output_ringbuf_size (C++ function), 44
audio_element_get_output_ringbuf (C++ function), 46
audio_element_get_multi_output_ringbuf (C++ function), 46
audio_element_get_output_ringbuf (C++ function), 39
audio_element_get_state (C++ function), 40
audio_element_get_tag (C++ function), 35
audio_element_get_uri (C++ function), 36
audio_element_getdata (C++ function), 35
audio_element_getinfo (C++ function), 36
audio_element_histogram_t (C++ type), 48
AUDIO_ELEMENT_INFO_DEFAULT (C macro), 48
audio_element_info_t (C++ class), 46
audio_element_info_t::bits (C++ member), 46
audio_element_info_t::byte_pos (C++ member), 46
audio_element_info_t::channels (C++ member), 46
audio_element_info_t::codec_fmt (C++ member), 47
audio_element_info_t::mute (C++ member), 46
audio_element_info_t::reserve_data (C++ member), 47
audio_element_info_t::sample_rates (C++ member), 46
audio_element_info_t::total_bytes (C++ member), 47
audio_element_info_t::uri (C++ member), 47
audio_element_info_t::volume (C++ member), 46
audio_element_init (C++ function), 34
audio_element_input (C++ function), 42
audio_element_message_queue_t (C++ type), 49
audio_element_msg_remove_listener (C++ function), 39
audio_element_msg_set_listener (C++ function), 38
audio_element_multi_input (C++ function), 44
audio_element_multi_output (C++ function), 45
audio_element_output (C++ function), 42
audio_element_pause (C++ function), 37
audio_element_report_codec_fmt (C++ function), 41
audio_element_report_info (C++ function), 41
audio_element_report_status (C++ function), 40
audio_element_reserve_data (C++ class), 46
audio_element_reserve_data_t::user_data_0 (C++ member), 46
audio_element_reserve_data (C++ class), 46
audio_element_reserve_data_t::user_data_1 (C++ member), 46
audio_element_reset_input_ringbuf (C++ function), 42
audio_element_reset_output_ringbuf (C++ function), 42
audio_element_resume (C++ function), 38
audio_element_run (C++ function), 36
audio_element_set_event_callback (C++ function), 38
audio_element_set_input_timeout (C++ function), 41
audio_element_set_output_ringbuf (C++ function), 39
audio_element_set_output_ringbuf_size (C++ function), 44
audio_element_set_output_timeout (C++ function), 41
audio_element_set_multi_input_ringbuf (C++ function), 45
audio_element_set_multi_output_ringbuf (C++ function), 45
audio_element_set_output_ringbuf (C++ function), 39
audio_element_set_output_ringbuf_size (C++ function), 44
audio_element_set_output_timeout (C++ function), 41
audio_element_set_read_cb (C++ function), 42
audio_element_set_ringbuf_done (C++ function), 43
audio_element_set_tag (C++ function), 35
audio_element_set_uri (C++ function), 36
audio_element_set_write_cb (C++ function), 43
audio_element_setdata (C++ function), 35
audio_element_setinfo (C++ function), 35
audio_element_state_t (C++ type), 48
audio_element_stop (C++ function), 49
audio_element_terminate (C++ function), 36
AUDIO_ELEMENT_TYPE_ELEMENT (C++ enumerator), 63
AUDIO_ELEMENT_TYPE_PERIPH (C++ enumerator), 63
AUDIO_ELEMENT_TYPE_PLAYER (C++ enumerator), 63
AUDIO_ELEMENT_TYPE_SERVICE (C++ enumerator), 63
audio_element_type_t (C++ type), 63
AUDIO_ELEMENT_TYPE_UNKNOW (C++ enumerator), 63
audio_element_wait_for_buffer (C++ function), 40
audio_element_wait_for_stop (C++ function), 37
audio_element_wait_for_stop_ms (C++ function), 37
audio_event_iface_cfg_t (C++ class), 62
audio_event_iface_cfg_t::context (C++ member), 62
audio_event_iface_cfg_t::external_queue (C++ member), 62
audio_event_iface_cfg_t::internal_queue (C++ member), 62
audio_event_iface_cfg_t::on_cmd (C++ member), 62
audio_event_iface_cfg_t::queue_set_size (C++ member), 62
audio_event_iface_cfg_t::type (C++ member), 62
audio_event_iface_cfg_t::wait_time (C++ member), 62
audio_event_iface_cmd (C++ function), 59
audio_event_iface_cmd_from_isr (C++ function), 60
AUDIO_EVENT_IFACE_DEFAULT_CFG (C macro), 62
audio_event_iface_destroy (C++ function), 58
audio_event_iface_discard (C++ function), 60
audio_event_iface_get_msg_queue_handle (C++ function), 61
audio_event_iface_get_queue_handle (C++ function), 61
audio_event_iface_handle_t (C++ type), 63
audio_event_iface_init (C++ function), 58
audio_event_iface_listen (C++ function), 60
audio_event_iface_msg_t (C++ class), 62
audio_event_iface_msg_t::cmd (C++ member), 62
audio_event_iface_msg_t::data (C++ member), 62
audio_event_iface_msg_t::data_len (C++ member), 62
audio_event_iface_msg_t::need_free_data (C++ member), 62
audio_event_iface_msg_t::source (C++ member), 62
audio_event_iface_msg_t::source_type (C++ member), 62
audio_event_iface_remove_listener (C++ function), 61
audio_event_iface_sendout (C++ function), 60
audio_event_iface_set_cmd_waiting_timeout (C++ function), 59
audio_event_iface_set_listener (C++ function), 59
audio_event_iface_set_msg_listener (C++ function), 61
audio_event_iface_waiting_cmd_msg (C++ function), 59
audio_pipeline_breakup_elements (C++ function), 56
audio_pipeline_cfg (C++ class), 57
audio_pipeline_cfg::rb_size (C++ member), 57
audio_pipeline_cfg_t (C++ type), 58
audio_pipeline_change_state (C++ function), 57
audio_pipeline_check_items_state (C++ function), 55
audio_pipeline_deinit (C++ function), 50
audio_pipeline_get_el_by_tag (C++ function), 53
audio_pipeline_get_event_iface (C++ function), 54
audio_pipeline_handle_t (C++ type), 58
audio_pipeline_init (C++ function), 50
audio_pipeline_link (C++ function), 53
audio_pipeline_link_insert (C++ function), 54
audio_pipeline_link_more (C++ function), 55
audio_pipeline_listen_more (C++ function), 55
audio_event_iface_get_queue_handle (C++ function), 61
audio_pipeline_pause (C++ function), 52
audio_pipeline_register (C++ function), 51
audio_pipeline_register_more (C++ function), 54
audio_pipeline_relink (C++ function), 56
audio_pipeline_relink_more (C++ function), 57
audio_pipeline_remove_listener (C++ function), 53
audio_pipeline_reset_items_state (C++ function), 56
audio_pipeline_reset_ringbuffer (C++ function), 56
audio_pipeline_resume (C++ function), 52
audio_pipeline_run (C++ function), 51
audio_pipeline_set_listener (C++ function), 54
audio_pipeline_stop (C++ function), 52
audio_pipeline_terminate (C++ function), 51
audio_pipeline_unlink (C++ function), 53
audio_pipeline_unregister (C++ function), 51
audio_pipeline_unregister_more (C++ function), 55
audio_pipeline_wait_for_stop (C++ function), 52

AUDIO_PLAYLIST (C++ enumerator), 64
AUDIO_STREAM_NONE (C++ enumerator), 63
AUDIO_STREAM_READER (C++ enumerator), 63
AUDIO_STREAM_TYPE_T (C++ type), 63
AUDIO_STREAM_WRITER (C++ enumerator), 63

DEFAULT_AUDIO_ELEMENT_CONFIG (C macro), 48
DEFAULT_AUDIO_EVENT_IFACE_SIZE (C macro), 62
DEFAULT_AUDIO_PIPELINE_CONFIG (C macro), 58
DEFAULT_ELEMENT_BUFFER_LENGTH (C macro), 48
DEFAULT_ELEMENT_RINGBUF_SIZE (C macro), 48
DEFAULT_ELEMENT_STACK_SIZE (C macro), 48
DEFAULT_ELEMENT_TASK_CORE (C macro), 48
DEFAULT_ELEMENT_TASK_PRIO (C macro), 48
DEFAULT_PIPELINE_RINGBUF_SIZE (C macro), 58

ELEMENT_SUB_TYPE_OFFSET (C macro), 63
event_cb_func (C++ type), 48

F
FATFS_STREAM_BUF_SIZE (C macro), 70
FATFS_STREAM_CFG_DEFAULT (C macro), 70
fatfs_stream_cfg_t::buf_sz (C++ member), 70
fatfs_stream_cfg_t::out_rb_size (C++ member), 70
fatfs_stream_cfg_t::task_core (C++ member), 70
fatfs_stream_cfg_t::task_prio (C++ member), 70
fatfs_stream_cfg_t::task_stack (C++ member), 70
fatfs_stream_cfg_t::type (C++ member), 70
fatfs_stream_init (C++ function), 69
FATFS_STREAM_RINGBUFFER_SIZE (C macro), 70
FATFS_STREAM_TASK_CORE (C macro), 70
FATFS_STREAM_TASK_PRIO (C macro), 70
FATFS_STREAM_TASK_STACK (C macro), 70

H
HTTP_STREAM_CFG_DEFAULT (C macro), 68
http_stream_cfg_t::auto_connect_next_track (C++ member), 68
http_stream_cfg_t::enable_playlist_parser (C++ member), 68
http_stream_cfg_t::event_handle (C++ member), 68
http_stream_cfg_t::out_rb_size (C++ member), 68
http_stream_cfg_t::task_core (C++ member), 68
http_stream_cfg_t::task_prio (C++ member), 68
http_stream_event_msg_t::buffer (C++ member), 67
http_stream_event_msg_t::buffer_len (C++ member), 67
http_stream_event_msg_t::el (C++ member), 68
http_stream_event_msg_t::event_id (C++ member), 67
http_stream_event_msg_t::http_client (C++ member), 67
http_stream_event_msg_t::user_data (C++ member), 68

Index
SPIFFS_STREAM_TASK_CORE (C macro), 72
SPIFFS_STREAM_TASK_Prio (C macro), 72
SPIFFS_STREAM_TASK_STACK (C macro), 72
stream_func (C++ type), 48