Read the Docs Template Documentation

Release

Read the Docs

Jul 01, 2018

Contents

This is the documentation for Espressif Audio Development Framework.

Get Started	API Reference	Design Guide

CHAPTER 1

Get Started

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

To make the start with ESP-ADF quicker, Espressif designed *ESP32 LyraT*, a development board intended to build an audio application with the ESP32.

Fig. 1.1: ESP32 LyraT audio development board

Click the link below to get started with this board.

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.

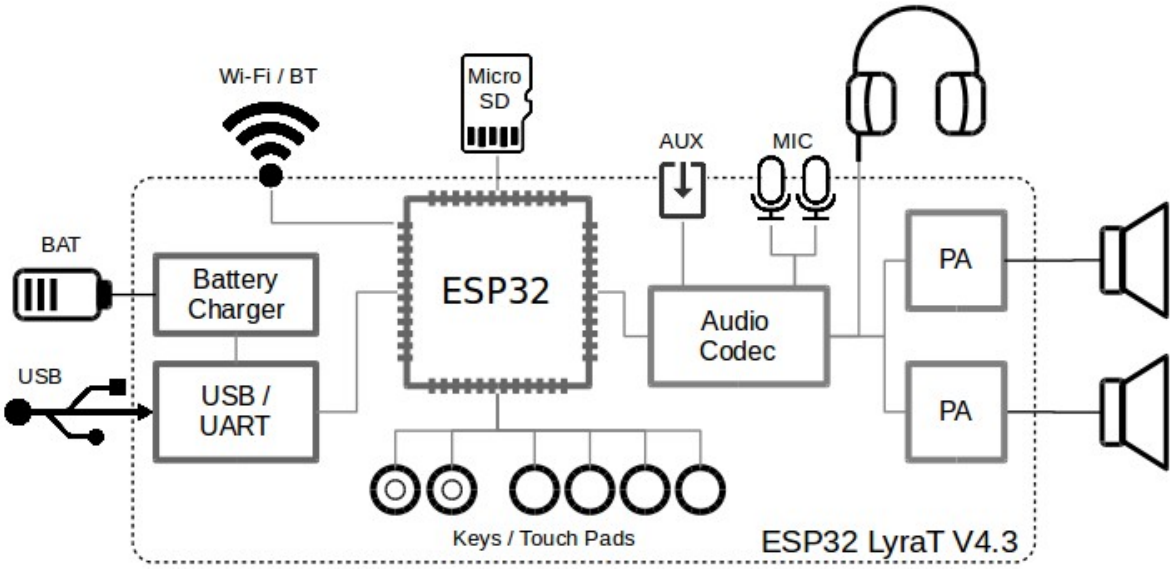

Fig. 1.2: ESP32-LyraT Block Diagram

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.

Boot/Reset Press Keys Boot button: holding down the **Boot** button and momentarily pressing the **Reset** button to initiate the firmware upload mode. Then you can upload firmware through the serial port. Reset button: 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.

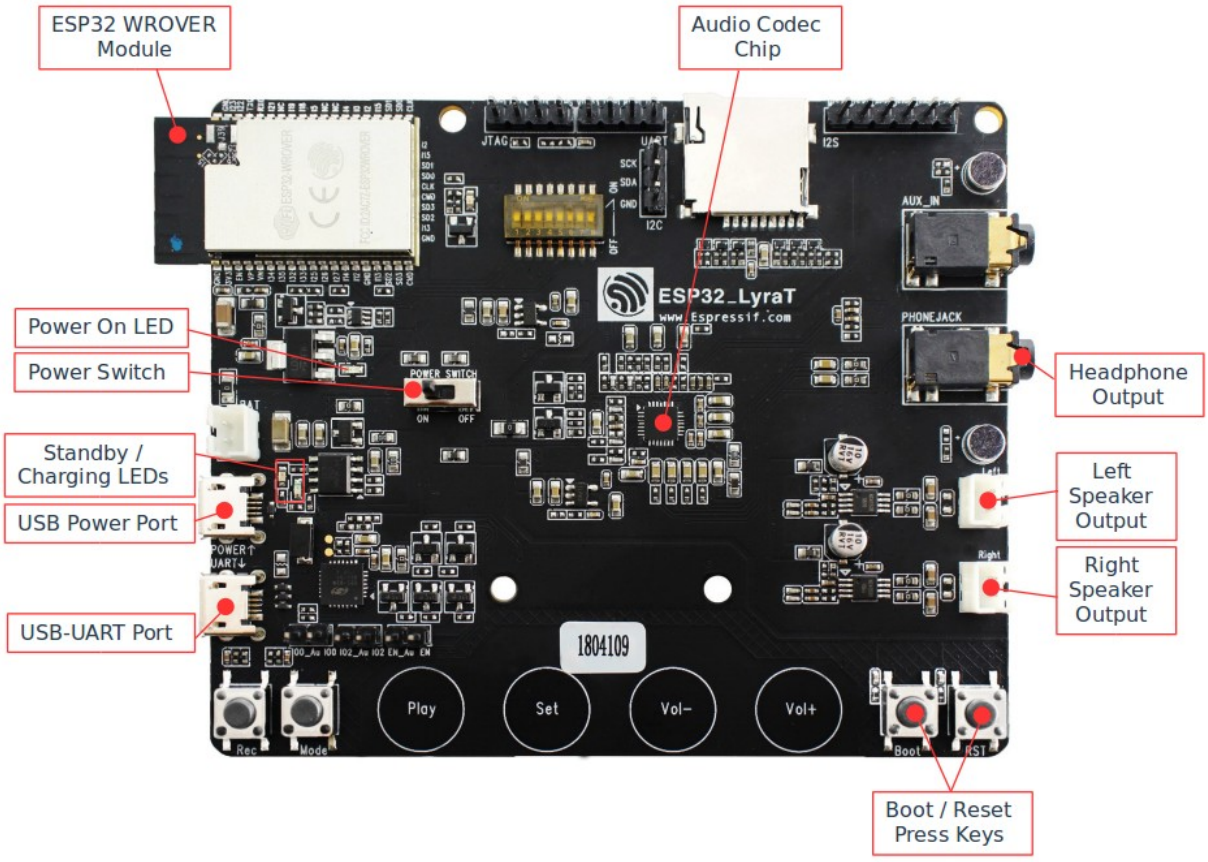

Fig. 1.3: ESP32-LyraT V4.3 Board Layout Overview

1.1.2 Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

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

Develop Applications

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

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

1.1.3 Summary of Key Changes from LyraT V4.2

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

1.1.4 Other Versions of LyraT

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

1.1.5 Related Documents

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

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

1.2 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.

1.3 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.

1.4 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
```

1.5 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.

1.6 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.

1.7 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 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 and upload speed:

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

Save the configuration.

1.8 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 binaries to your ESP32 board and start the monitor.

```
...

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

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

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

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

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

1.9 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.

1.10 Related Documents

1.10.1 ESP32-LyraT V4.2 Getting Started Guide

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

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

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

What You Need

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

Overview

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

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

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

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

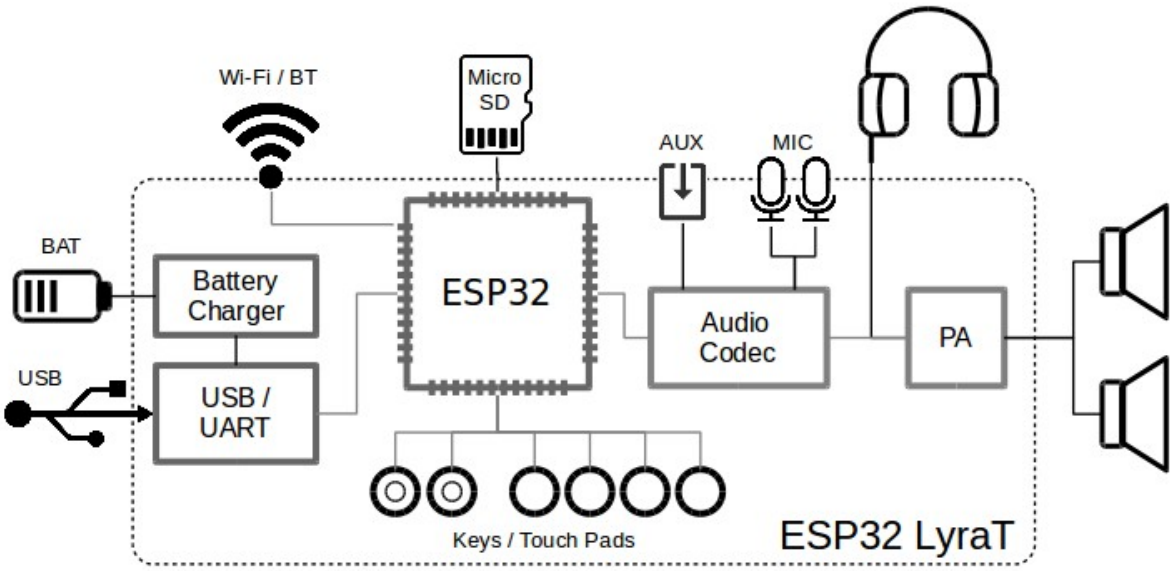

Fig. 1.4: ESP32-LyraT Block Diagram

Functional Description

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

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

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

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

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

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

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

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

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

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

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

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

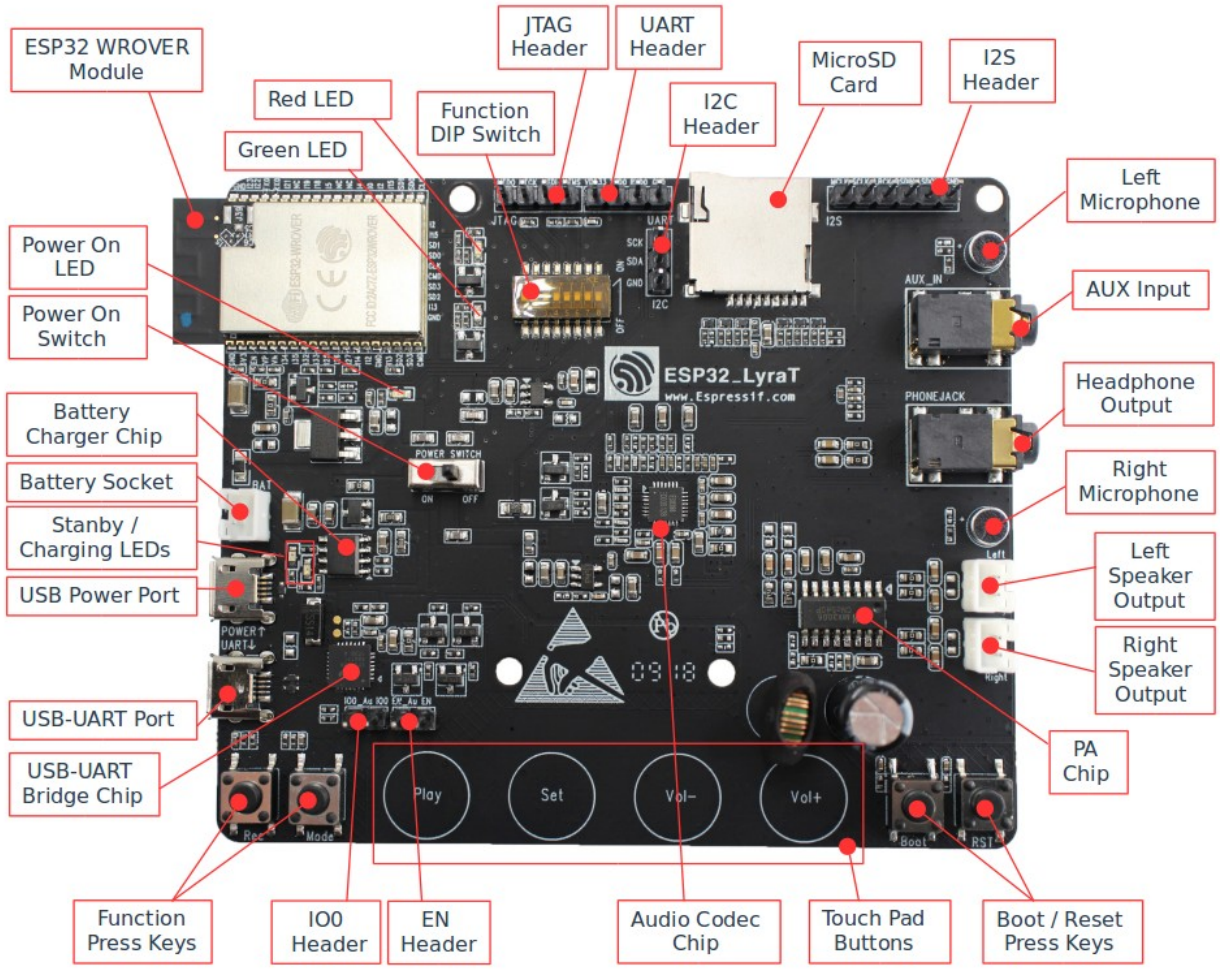

Fig. 1.5: ESP32-LyraT V4.2 Board Layout

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 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.

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

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

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

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

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

USB Power Port Provides the power supply for the board.

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

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

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

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

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

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

Hardware Setup Options

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

Enable MicroSD Card in 1-wire Mode

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

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

In this mode:

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

Enable MicroSD Card in 4-wire Mode

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

In this mode:

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

Enable JTAG

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

In this mode:

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

Allocation of ESP32 Pins

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

Red / Green LEDs

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

Touch Pads

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

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

MicroSD Card / J5

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

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

EN and IO0 Headers / JP23 and J24

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

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

I2S Header / JP4

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

I2C Header / JP5

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

JTAG Header / JP7

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

Function DIP Switch / JP8

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

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

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

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

Develop Applications

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

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

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

Related Documents

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

1.10.2 ESP32-LyraT V4 Getting Started Guide

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

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

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

What You Need

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

Overview

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

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

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

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

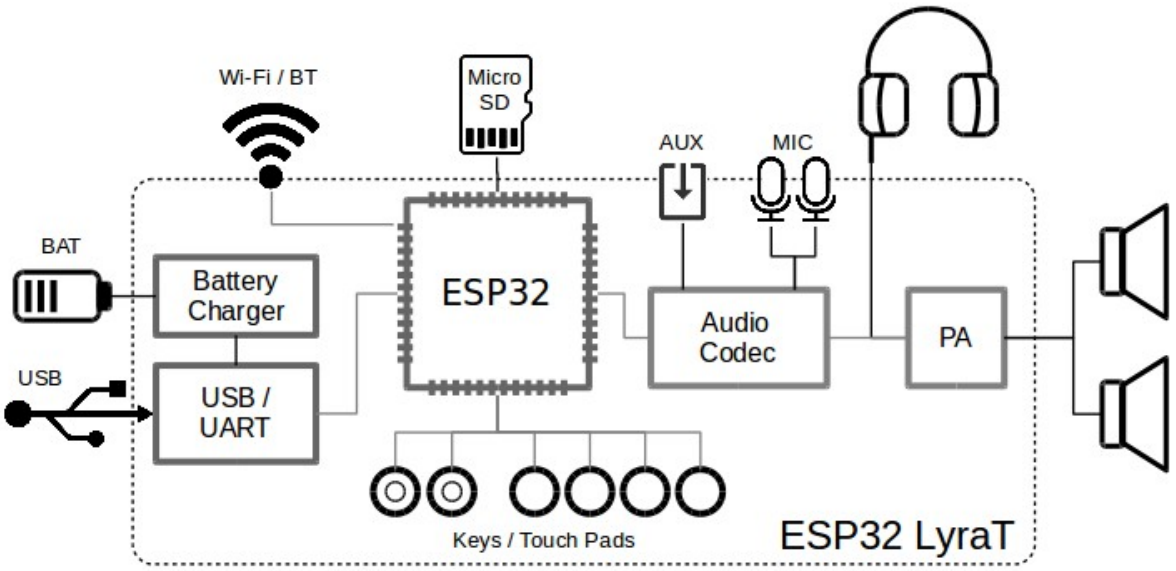

Fig. 1.6: ESP32-LyraT block diagram

Functional Description

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

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

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

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

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

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

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

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

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

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

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

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

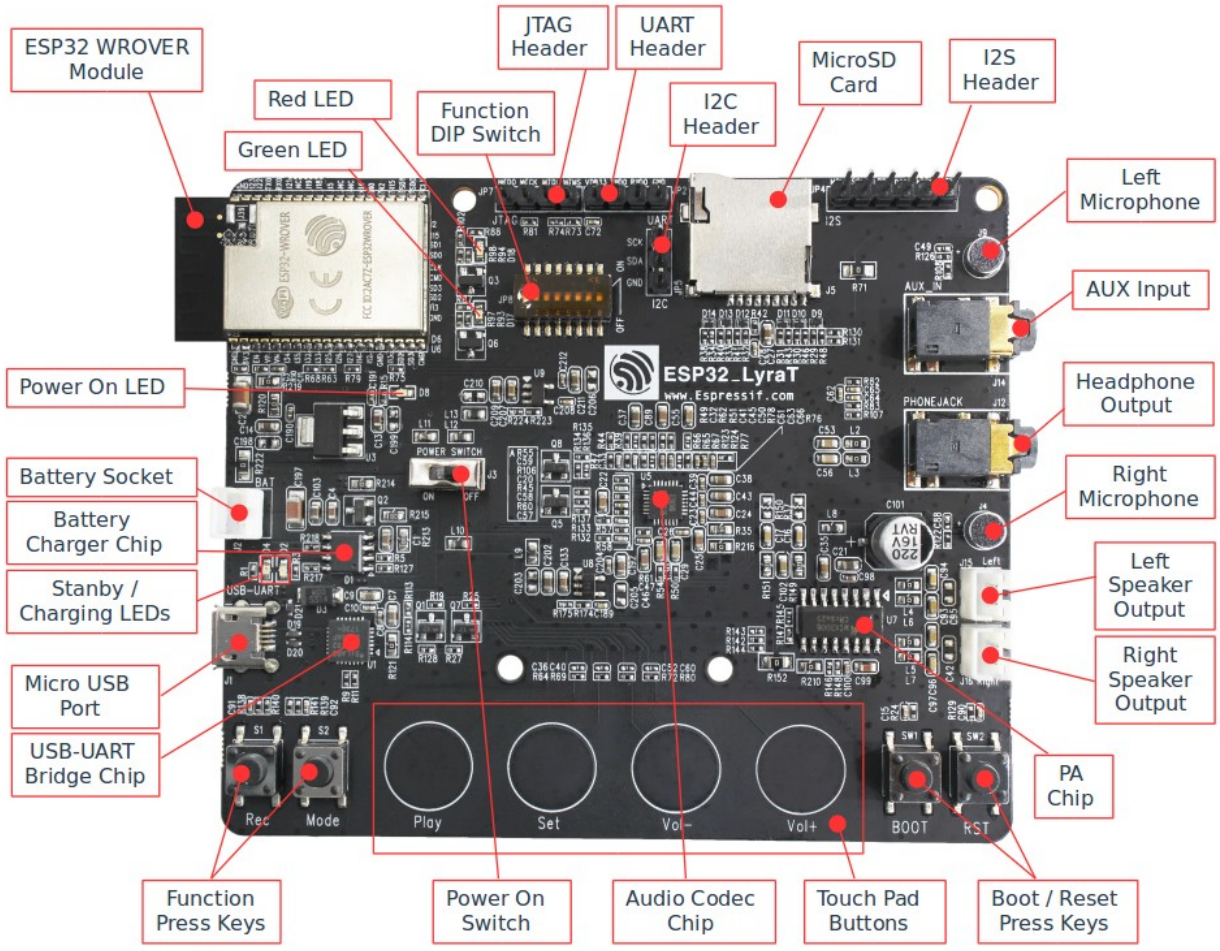

Fig. 1.7: ESP32 LyraT V4 board layout

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

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

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

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

Boot/Reset Press Keys Boot button: holding down the **Boot** button and momentarily pressing the **Reset** button initiates the firmware download mode. Then user 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 headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER Module** over I2S and I2S buses to provide audio processing in hardware independently from the audio application.

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

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

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

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

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

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

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

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

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

Hardware Setup Options

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

Enable MicroSD Card in 1-wire Mode

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

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

In this mode:

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

Enable MicroSD Card in 4-wire Mode

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

In this mode:

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

Enable JTAG

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

In this mode:

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

Allocation of ESP32 Pins

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

Red / Green LEDs

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

Touch Pads

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

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

MicroSD Card / J5

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

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

I2S Header / JP4

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

I2C Header / JP5

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

JTAG Header / JP7

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

Function DIP Switch / JP8

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

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

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

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

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

Develop Applications

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

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

Related Documents

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

CHAPTER 2

API Reference

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

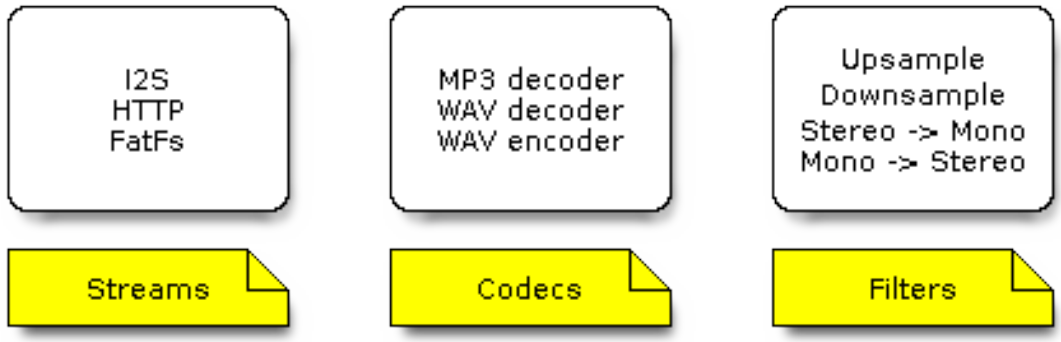

Fig. 2.1: **Elements** of the Audio Development Framework

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

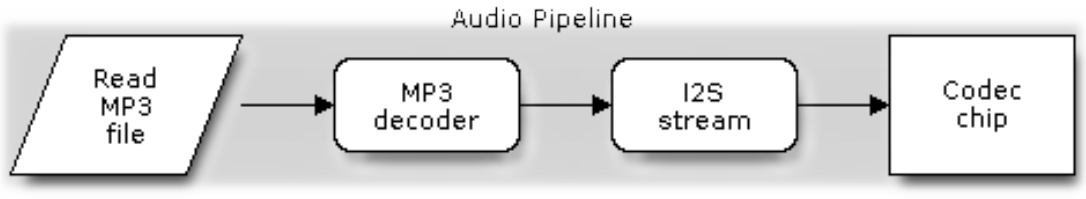

Fig. 2.2: Sample Organization of Elements in Audio Pipeline

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

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

2.1 Audio Framework

2.1.1 Audio Element

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

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

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

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

API Reference

Header File

- audio_pipeline/include/audio_element.h

Functions

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

Return

- audio_elemenent handle object
- NULL

Parameters

- `config`: The configuration

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return data pointer

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return Element tag name pointer

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return URI pointer

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

- ESP_FAIL

Return

- ESP_OK

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Set Element input ringbuffer.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Get Element input ringbuffer.

Return ringbuf_handle_t

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return ringbuf_handle_t

Parameters

- `el`: The audio element handle

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

Return audio_element_state_t

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Set input read timeout (default is `portMAX_DELAY`)

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

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

Parameters

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

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

Call this function to sendout Element output data.

Return

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

Parameters

- `el`: Depend on setup using ringbuffer or function callback, Element will invokes write t ringbuffer, or call write callback funtion
- `buffer`: The buffer pointer
- `write_size`: The write size

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return QueueHandle_t

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `el`: The audio element handle

Structures

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)

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)

void ***data**
User context

char ***tag**
Element tag

Macros

AUDIO_ELEMENT_INFO_DEFAULT

DEFAULT_ELEMENT_BUFFER_LENGTH

DEFAULT_ELEMENT_STACK_SIZE

DEFAULT_ELEMENT_TASK_PRIO

DEFAULT_ELEMENT_TASK_CORE

DEFAULT_AUDIO_ELEMENT_CONFIG

Type Definitions

typedef struct audio_element ***audio_element_handle_t**

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

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

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

Enumerations

enum audio_element_err_t

Values:

AEL_IO_OK = ESP_OK

AEL_IO_FAIL = ESP_FAIL

AEL_IO_DONE = -2

AEL_IO_ABORT = -3

AEL_IO_TIMEOUT = -4

AEL_PROCESS_FAIL = -5

enum audio_element_state_t

Audio element state.

Values:

AEL_STATE_NONE = 0

AEL_STATE_INIT

AEL_STATE_RUNNING

AEL_STATE_PAUSED

AEL_STATE_STOPPED

AEL_STATE_FINISHED

AEL_STATE_ERROR

enum audio_element_msg_cmd_t

Audio element action command, process on dispatcher

Values:

AEL_MSG_CMD_NONE = 0

AEL_MSG_CMD_ERROR = 1

AEL_MSG_CMD_FINISH = 2

AEL_MSG_CMD_STOP = 3

AEL_MSG_CMD_PAUSE = 4

AEL_MSG_CMD_RESUME = 5

AEL_MSG_CMD_DESTROY = 6

AEL_MSG_CMD_REPORT_STATUS = 8

AEL_MSG_CMD_REPORT_MUSIC_INFO = 9

AEL_MSG_CMD_REPORT_CODEC_FMT = 10

enum audio_element_status_t

Audio element status report

Values:

AEL_STATUS_NONE = 0

AEL_STATUS_ERROR_OPEN = 1

AEL_STATUS_ERROR_INPUT = 2

AEL_STATUS_ERROR_PROCESS = 3

AEL_STATUS_ERROR_OUTPUT = 4

AEL_STATUS_ERROR_CLOSE = 5

AEL_STATUS_ERROR_TIMEOUT = 6

AEL_STATUS_ERROR_UNKNOWN = 7

AEL_STATUS_INPUT_DONE = 8

AEL_STATUS_INPUT_BUFFERING = 9

AEL_STATUS_OUTPUT_DONE = 10

AEL_STATUS_OUTPUT_BUFFERING = 11

AEL_STATUS_STATE_RUNNING = 12

AEL_STATUS_STATE_PAUSED = 13

AEL_STATUS_STATE_STOPPED = 14

AEL_STATUS_MOUNTED = 15

AEL_STATUS_UNMOUNTED = 16

2.1.2 Audio Pipeline

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

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

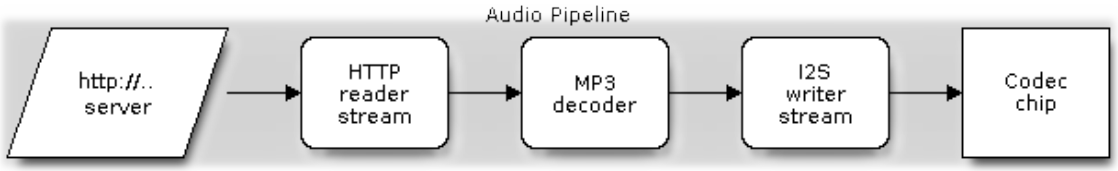

Fig. 2.3: Sample Organization of Elements in Audio Pipeline

API Reference

Header File

- audio_pipeline/include/audio_pipeline.h

Functions

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

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

Return

- audio_pipeline_handle_t on success
- NULL when any errors

Parameters

- `config`: The configuration - audio_pipeline_cfg_t

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

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

Return ESP_OK

Parameters

- `pipeline`: The Audio Pipeline Handle

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

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

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

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

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

Start Audio Pipeline, with this function, audio_pipeline will start all task elements, which have been registered using the `audio_pipeline_register` function. In addition this also puts all the task elements in the 'PAUSED' state.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

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

Start Audio Pipeline, with this function, audio_pipeline will start all task elements, which have been registered using the `audio_pipeline_register` function. In addition this also puts all the task elements in the 'PAUSED' state.

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_run` function.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

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

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

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

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_remove_listener**(*audio_pipeline_handle_t pipeline*)

Remove event listener from this audio_pipeline.

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

esp_err_t **audio_pipeline_set_listener**(*audio_pipeline_handle_t pipeline*, *audio_event_iface_handle_t evt*)

Set event listner for this audio_pipeline, any event from this pipeline can be listen to by `evt`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

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

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

Return The Event Handle

Parameters

- `pipeline`: The pipeline

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

- `...`: Additional elements to add to the audio_pipeline.

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

Adds a NULL-terminated list of elements to audio_pipeline.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

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

Return

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

Parameters

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

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

Reset pipeline element items state to `AEL_STATUS_NONE`

Return

- ESP_OK on success
- ESP_FAIL when any errors

Parameters

- `pipeline`: The Audio Pipeline Handle

Structures

struct audio_pipeline_cfg
Audio Pipeline configurations.

Public Members

int **rb_size**
Audio Pipeline ringbuffer size

Macros

DEFAULT_PIPELINE_RINGBUF_SIZE

DEFAULT_AUDIO_PIPELINE_CONFIG

Type Definitions

typedef struct audio_pipeline ***audio_pipeline_handle_t**

typedef struct *audio_pipeline_cfg* **audio_pipeline_cfg_t**
Audio Pipeline configurations.

2.1.3 Event Interface

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

Application Examples

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

API Reference

Header File

- audio_pipeline/include/audio_event_iface.h

Functions

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `config`: The configurations

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t **audio_event_iface_set_cmd_waiting_timeout** (*audio_event_iface_handle_t evt*, TickType_t *wait_time*)
Set current queue wait time for the event.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t **audio_event_iface_waiting_cmd_msg**(*audio_event_iface_handle_t evt*)
Waiting internal queue message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

esp_err_t **audio_event_iface_cmd**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)
Trigger an event for internal queue with a message.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t **audio_event_iface_cmd_from_isr**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)
It's same with `audio_event_iface_cmd`, but can send a message from ISR.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

esp_err_t **audio_event_iface_sendout**(*audio_event_iface_handle_t evt*, *audio_event_iface_msg_t *msg*)
Trigger and event out with a message.

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Discard all ongoing event message.

Return

- ESP_OK
- ESP_FAIL

Parameters

- `evt`: The event

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

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

Get External queue handle of Emmitter.

Return External QueueHandle_t

Parameters

- `evt`: The external queue

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

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

Return

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

Parameters

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

QueueHandle_t **audio_event_iface_get_msg_queue_handle**(*audio_event_iface_handle_t evt*)
Get Internal queue handle of Emmitter.

Return Internal QueueHandle_t

Parameters

- `evt`: The Internal queue

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

Structures

struct audio_event_iface_msg_t
Event message

Public Members

int **cmd**
Command id

void ***data**
Data pointer

int **data_len**
Data length

void ***source**
Source event

int **source_type**
Source type (To know where it came from)

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

struct audio_event_iface_cfg_t
Event interface configurations

Public Members

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

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

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

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

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

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

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

Macros

DEFAULT_AUDIO_EVENT_IFACE_SIZE

AUDIO_EVENT_IFACE_DEFAULT_CFG

Type Definitions

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

typedef struct audio_event_iface ***audio_event_iface_handle_t**

2.1.4 Audio Common

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

API Reference

Header File

- audio_pipeline/include/audio_common.h

Macros

ELEMENT_SUB_TYPE_OFFSET

mem_assert(x)

Enumerations

enum audio_element_type_t
: *Values:*

  AUDIO_ELEMENT_TYPE_UNKNOW = 0x01<<ELEMENT_SUB_TYPE_OFFSET

AUDIO_ELEMENT_TYPE_ELEMENT = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+1)

AUDIO_ELEMENT_TYPE_PLAYER = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+2)

AUDIO_ELEMENT_TYPE_SERVICE = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+3)

AUDIO_ELEMENT_TYPE_PERIPH = 0x01<<(ELEMENT_SUB_TYPE_OFFSET+4)

enum audio_stream_type_t

Values:

AUDIO_STREAM_NONE = 0

AUDIO_STREAM_READER

AUDIO_STREAM_WRITER

enum audio_codec_type_t

Values:

AUDIO_CODEC_TYPE_NONE = 0

AUDIO_CODEC_TYPE_DECODER

AUDIO_CODEC_TYPE_ENCODER

enum audio_codec_t

Values:

AUDIO_CODEC_NONE = 0

AUDIO_CODEC_RAW

AUDIO_CODEC_WAV

AUDIO_CODEC_MP3

AUDIO_CODEC_AAC

AUDIO_CODEC_OPUS

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:

Stream Name	Stream Type
I2S	Reader / Writer
HTTP	Reader / Writer
FatFs	Reader / Writer

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, depend on the configuration of stream type is AUDIO_STREAM_READER or AUDIO_STREAM_WRITER.

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

Return The Audio Element handle

Parameters

- `config`: The configuration

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

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

Structures

struct i2s_stream_cfg_t
I2S Stream configurations Default value will be used if any entry is zero.

Public Members

audio_stream_type_t **type**
Type of stream

i2s_config_t **i2s_config**
I2S driver configurations

i2s_pin_config_t **i2s_pin_config**
I2S driver hardware pin configurations

i2s_port_t **i2s_port**
I2S driver hardware port

Macros

I2S_STREAM_CFG_DEFAULT

I2S_STREAM_INTERNAL_DAC_CFG_DEFAULT

2.2.2 HTTP Stream

Header File

- audio_stream/include/http_stream.h

Functions

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

Return The Audio Element handle

Parameters

- `config`: The configuration

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**
Lenght of buffer

void ***user_data**
User data context, from *http_stream_cfg_t*

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

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

void ***user_data**
User data context

Macros

HTTP_STREAM_CFG_DEFAULT

Type Definitions

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

Enumerations

enum http_stream_event_id_t
HTTP Stream hook type.

Values:

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

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

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

HTTP_STREAM_POST_REQUEST
The event handler will be called after HTTP Client send header and body to the serve, 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

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

Return The Audio Element handle

Parameters

- `config`: The configuration

Structures

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

Public Members

audio_stream_type_t **type**
Stream type

int **buf_sz**
Audio Element Buffer size

2.3 Codecs

2.3.1 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

2.3.2 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

2.3.3 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.

API Reference

2.4 Peripherals

2.4.1 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

2.4.2 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

2.4.3 Button Peripheral

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

API Reference

2.4.4 Touch Peripheral

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

API Reference

2.4.5 Console Peripheral

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

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

Code example

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

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

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

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

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

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

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

}
```

API Reference

2.4.6 ESP Peripherals

There are several ESP32 peripherals to use in the ESP-ADF, from buttons, touchpads, SD Card to Wi-Fi. This library simplifies the management of peripherals, by pooling and monitoring in a single task, adding basic functions to send and receive events. And it also provides APIs to easily integrate new peripherals.

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

Examples

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

static const char *TAG = "ESP_PERIPH_TEST";

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

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

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

    // Setup BUTTON peripheral
    periph_button_cfg_t btn_cfg = {
        .gpio_mask = GPIO_SEL_36 | GPIO_SEL_39
    };
```

```
    esp_periph_handle_t button_handle = periph_button_init(&btn_cfg);

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

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

    //Stop button peripheral
    esp_periph_stop(button_handle);
    vTaskDelay(10*1000/portTICK_RATE_MS);

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

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

API Reference

2.5 Abstraction Layer

2.5.1 Ring Buffer

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

Fig. 2.4: Ring Buffer used in Audio Pipeline

Application Example

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

API Reference

Header File

- audio_pipeline/include/ringbuf.h

Functions

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

Return ringbuf_handle_t

Parameters

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

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

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

Return total bytes available

Parameters

- `rb`: The Ringbuffer handle

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

Return The number of bytes that have filled the ringbuffer

Parameters

- `rb`: The Ringbuffer handle

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

Return total size of Ringbuffer

Parameters

- `rb`: The Ringbuffer handle

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

Return Number of bytes read

Parameters

- `rb`: The Ringbuffer handle
- `buf`: The buffer pointer to read out data
- `len`: The length request
- `ticks_to_wait`: The ticks to wait

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

Return Number of bytes written

Parameters

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

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

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

Parameters

- `rb`: The Ringbuffer handle

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

Return

- ESP_OK
- ESP_FAIL

Parameters

- `rb`: The Ringbuffer handle

Macros

RB_OK

RB_FAIL

RB_DONE

RB_ABORT

RB_TIMEOUT

Type Definitions

typedef struct ringbuf ***ringbuf_handle_t**

2.5.2 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

2.5.3 ES8388 Driver

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

API Reference

2.6 Configuration Options

Compile-time configuration options specific to ESP-ADF.

2.6.1 ESP HTTP client

ESP_HTTP_CLIENT_ENABLE_HTTPS

Enable https

Found in: Component config > ESP HTTP client

This option will enable https protocol by linking mbedtls library and initializing SSL transport

2.6.2 Audio HAL

AUDIO_BOARD

Audio board

Found in: Audio HAL

Select an audio board to use with the ESP-ADF

Available options:

- ESP_LYRAT_V4_3_BOARD
- ESP_LYRAT_V4_2_BOARD

CHAPTER 3

Design Guide

The ESP32 is a powerful chip well positioned as a MCU of the audio projects. This section is intended to provide guidance on process of designing an audio project with the ESP32 inside.

3.1 Project Design

When designing a project with ability to process an audio signal or audio data we typically consider a subset of the following components:

Input:

- **Analog signal input** to connect e.g. a microphone
- **Storage media**, e.g. microSD card with audio files to read them
- **WI-Fi** interface to obtain an audio data stream from the internet
- **Bluetooth** interface to obtain an audio data stream from e.g. a BT headset
- **I2S interface** to obtain audio data stream from a codec chip
- **Ethernet** interface to obtain an audio data stream from the internet
- An internal **chip's flash memory** with some audio samples to play
- **User Interface** e.g. buttons or some other means to provide user input

Output:

- **Analog signal output** to connect headphones or and amplifier with speakers
- **Storage media**, e.g. microSD card to write some audio files, e.g. with recording
- **WI-Fi** interface to send out an audio data stream to the internet
- **Bluetooth** interface to stream audio data to e.g. a BT headset
- **I2S interface** to stream some data to a codec chip

- **Ethernet** interface to stream an audio data stream to the internet
- An internal **chip's flash memory** to store some audio recording
- **User Interface** e.g. a display, LEDs or some means of haptic feedback

Main Processing Unit:

A microcontroller or a computer with processing power to read the data from the input, process (e.g. encode / encode) and send to the output.

3.1.1 Project Options

The ESP32 has all the above features or is able to support them (e.g. can drive Ethernet PHY). Considering the ESP32 cost is about $3, and availability of ESP-ADF software development platform, we are able to develop an audio project with minimum additional components at very low price.

Depending on the application, required functionality and performance, we may consider two project groups.

- **Minimum** - having minimum additional components, assuming using on board I2S, or PDM interface as well as DAC, if no high qualify audio on the output is required.
- **Typical** - with an external codec chip and a power amplifier, for high quality output audio and multiple input / output options.

There may be several variation between the above projects, by adding or removing features / components. Below are couple of examples.

3.1.2 Project Minimum

With several peripherals on ESP32, I2S or PDM or DAC interfaces can be used to implement a minimum project. With the digital microphones, we could input voice signals and build a command voice control project minumum that could communicate with a cloud service.

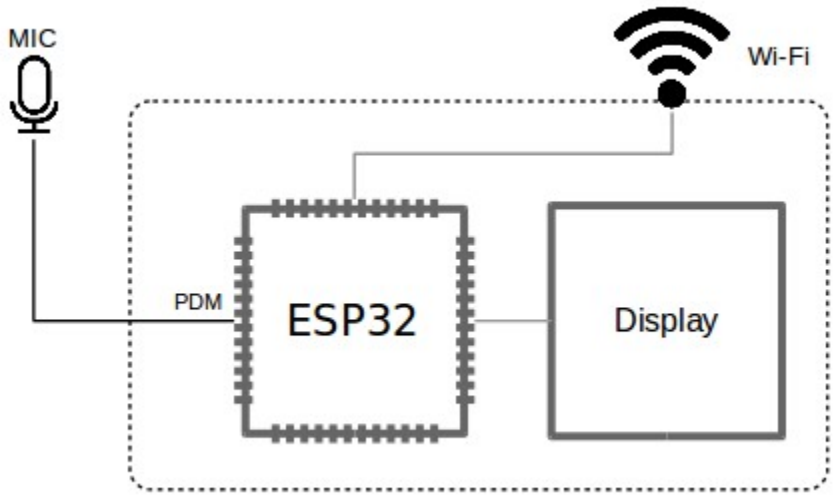

Fig. 3.1: Audio Project Example - Send Voice Commands to Cloud Service

With two on board DACs, if 8-bit width on the output is satisfactory, we may implement another project minimum - a device to play an internet connected radio.

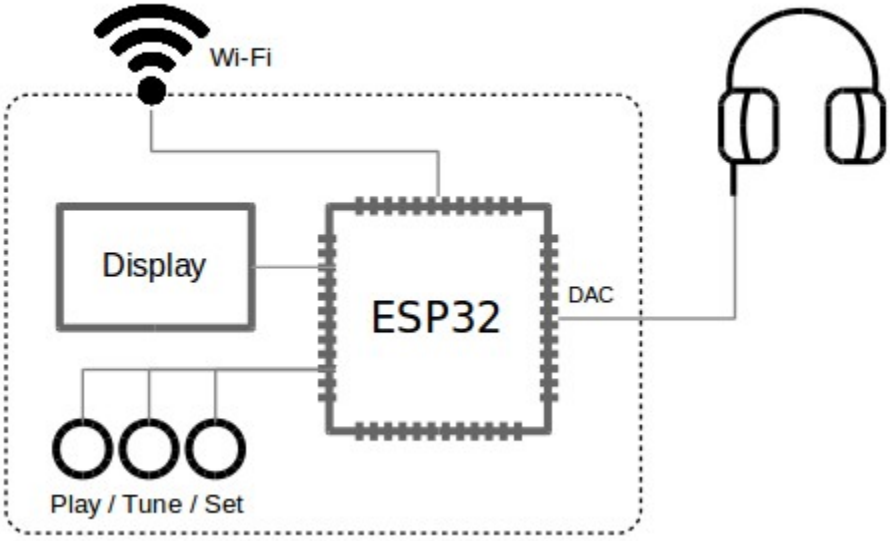

Fig. 3.2: Audio Project Example - Internet Connected Radio Player

3.1.3 Typical Project

When looking for better audio quality and more interfacing options we would use an external I2S codec to do all the analog input and output signal processing. The codec chip, depending on type, may provide additional functionality like audio input signal preamplifier, headphone output amplifier, multiple analog input and outputs, sound effects, etc. The I2S is considered as the industry standard for interfacing with audio codec chips, or in general for a high speed, continuous transfer of the audio data. To optimize performance of audio data processing additional memory may be required. For such cases consider using ESP32-WROVER that provides 4 MB PSRAM on a single module together with the ESP32 chip.

The ESP-ADF is designed primarily to support projects with a codec chip. The *ESP32 LyraT* board is an example of such a project. The software interfacing with the board is done by Audio HAL and a driver. The codec chip used on the ESP32 LyraT is ES8388. Boards with a different codec chip may be supported by providing a different driver.

3.2 Design Considerations

Depending on the audio data format, that may be lossless, lossy or compressed, e.g. WAV, MP3 or FLAC and the quality expressed in sampling rate and bitrate, the project will require different resources: memory, storage space, input / output throughput and the processing power. The resources will also depend on the project type and features discussed in *Project Design*.

This section describes capacity and performance of ESP32 system resources that should be considered when designing an audio project to meet required data format, audio quality and functionality.

3.2.1 Memory

The spare internal Data-RAM is about 290kB with “hello_world” example. For audio system this may be insufficient, and therefore the ESP32 incorporates the ability to use up to 4MB of external SPI RAM (i.e. PSRAM) memory. The

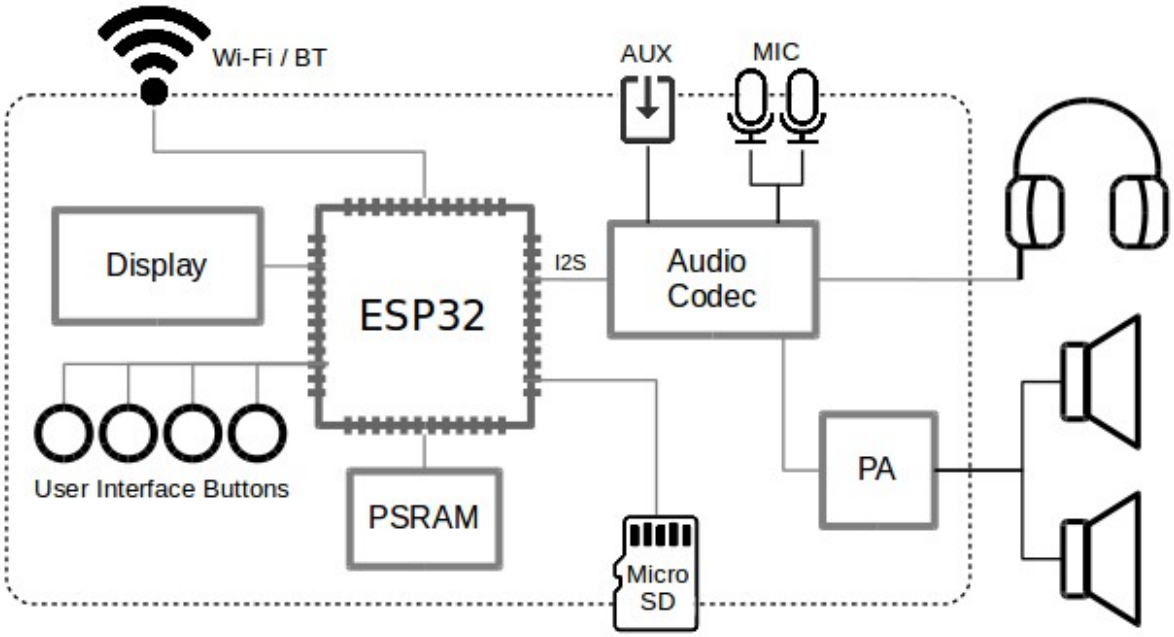

Fig. 3.3: Typical Audio Project Example

external memory is incorporated in the memory map and is, within certain restrictions, usable in the same way internal Data-RAM is.

Refer to External SPI-connected RAM section in IDF documenation for details, especially pay attention to its Restrictions section which is very important.

To be able to use the PSRAM, if installed on your board, it should be enabled in menucofig under *Component config > ESP32-specific > SPI RAM config*. The option *CONFIG_SPIRAM_CACHE_WORKAROUND*, set by default in the same menu, should be kept enabled.

Note: Bluetooth and Wi-Fi can not coexist without PSRAM because it will not leave enough memory for an audio application.

Optimization of Internal RAM and Use of PSRAM

Internal RAM is more valuable asset since there are some restrictions on PSRAM. Here are some tips for optimizing internal RAM.

- If PSRAM is in use, set all the static buffer to minimum value in *Component config > Wi-Fi*; if PSRAM is not used then dynamic buffer should be selected to save memory. Refer to Wi-Fi Buffer Usage section in IDF documentation for details.
- If PSRAM and BT are used, then *CONFIG_BT_ALLOCATION_FROM_SPIRAM_FIRST* and *CONFIG_BT_BLE_DYNAMIC_ENV_MEMORY* should be set as "yes" under *Component config > Bluetooth > Bluedroid Enable*, to allocate more of 40kB memory to PSRAM
- If PSRAM and Wi-Fi are used, then *CONFIG_WIFI_LWIP_ALLOCATION_FROM_SPIRAM_FIRST* should be set as "yes" under *Component config > ESP32-specific > SPI RAM config*, to allocate some memory to PSRAM

- Set *CONFIG_WL_SECTOR_SIZE* as 512 in *Component config* > *Wear Levelling*

Note: The smaller the size of sector be, the slower the Write / Read speed will be, and vice versa, but only 512 and 4096 are supported.

- Call `char *buf = heap_caps_malloc(1024 * 10, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT)` instead of `malloc(1024 * 10)` to use PSRAM, and call `char *buf = heap_caps_malloc(512, MALLOC_CAP_INTERNAL | MALLOC_CAP_8BIT)` to use internal RAM.
- Not relying on `malloc()` to automatically allocate PSRAM allows to make a full control of the memory. By avoiding the use of the internal RAM by other `malloc()` calls, you can reserve more memory for high-efficiency usage and task stack since PSRAM cannot be used as task stack memory.
- The task stack will always be allocated at internal RAM. On the other hand you can use of the `xTaskCreateStatic()` function that allows to create tasks with stack on PSRAM (see options in PSRAM and FreeRTOS menuconfig), but pay attention to its help information.

Important: Don't use ROM code in `xTaskCreateStatic` task The ROM code itself is linked in `components/esp32/ld/esp32.rom.ld`. However, you also need to consider other pieces of code that *call* ROM functions, as well as the code that is not recompiled against the *CONFIG_SPIRAM_CACHE_WORKAROUND* patch, like the Wi-Fi and Bluetooth libraries. In general, we advise using this only in threads that do not call any IDF libraries (including libc), doing only calculations and using FreeRTOS primitives to talk to other threads.

Memory Usage by Component Overview

Below is a table that contains ESP-ADF components and their memory usage. Choose the components needed and find out how much internal RAM is left. The table is divided into two parts, when PSRAM is used or not. If PSRAM (external RAM) is in use, then some of the memory will be allocated at PSRAM automatically.

The initial spare internal RAM is 290kB.

Component	Internal RAM Required	
	PSRAM not used	With PSRAM
Wi-Fi [1]	50kB+	50kB+
Bluetooth	140kB (50kB if only BLE needed)	95kB (50kB if only BLE needed)
Flash Card [2]	12kB+	12kB+
I2S [3]	Configurable, 8kB for reference	Configurable, 8kB for reference
RingBuffer [4]	Configurable, 30kB for reference	0kB, all moved into PSRAM

Notes to the table above

1. According to the Wi-Fi menuconfig each Tx and Rx buffer occupies 1.6kB internal RAM. The value of 50kB RAM is assuming use of 5 Rx static buffers and 6 Tx static buffers. If PSRAM is not in use, then the "Type of WiFi Tx Buffer" option should be set as *DYNAMIC* in order to save RAM, in this case, the RAM usage will be far less than 50kB, but programmer should keep at least 50kB available for the Wi-Fi to be able to transmit the data. **[Internal RAM only]**
2. Depending on value of *SD_CARD_OPEN_FILE_NUM_MAX* in audio_hal/board/board.h, that is then used in `sd_card_mount()` function, the RAM needed will increase with a greater number of maximum open files. 12kB is the RAM needed with 5 max files and 512 bytes *CONFIG_WL_SECTOR_SIZE*. **[Internal RAM only]**

3. Depending on configuration settings of the I2S stream, refer to audio_stream/include/i2s_stream.h and audio_stream/i2s_stream.c. **[Internal RAM only]**
4. Depending on configuration setting of the Ringbuffer, refer to *DEFAULT_PIPELINE_RINGBUF_SIZE* in audio_pipeline/include/audio_pipeline.h or user setting, if the buffer is created with e.g. `rb_create()`.

3.2.2 System Settings

The following settings are recommended to achieve a high Wi-Fi performance in an audio project.

Note: Use ESP32 modules and boards from reputable vendors that put attention to product design, component selection and product testing. This is to have confidence of receiving well designed boards with calibrated RF.

- Set these following options in menuconfig.
  - Flash SPI mode as QIO
  - Flash SPI speed as 80MHz
  - CPU frequency as 240MHz
  - Set *Default receive window size* as 5 times greater than *Maximum Segment Size* in *Component config* > *LWIP* > *TCP*
- If external antenna is used, then set *PHY_RF_CAL_PARTIAL* as *PHY_RF_CAL_FULL* in ''esp-idf/components/esp32/phy_init.c''

3.3 Software Design

Espressif audio framework project.

3.3.1 Features

1. All of Streams and Codecs based on audio element.
2. All events based on queue.
3. Audio pipeline supports dynamic combination.
4. Audio pipeline supports multiple elements.
5. Pipeline Support functionality plug-in.
6. Audio common peripherals support work in the one task.
7. Support post-event mechanism in peripherals.
8. Support high level audio play API based on element and audio pipeline.
9. Audio high level interface supports dynamic adding of codec library.
10. Audio high level interface supports dynamic adding of input and output stream.
11. ESP audio supports multiple audio pipelines.

3.3.2 Design Components

Five basic components are - Audio Element, Audio Event, Audio Pipeline, ESP peripherals, ESP audio

Audio Element

Example

```
audio_element_handle_t el;
audio_element_cfg_t cfg = DEFAULT_AUDIO_ELEMENT_CONFIG();
cfg.open = _el_open;
cfg.read = _el_read;
cfg.process = _el_process;
cfg.write = _el_write;
cfg.close = _el_close;
el = audio_element_init(&cfg);
TEST_ASSERT_NOT_NULL(el);
TEST_ASSERT_EQUAL(ESP_OK, audio_element_start(el));
```

Audio Event

Example

```
audio_event_handle_t evt1;
audio_event_cfg_t cfg = AUDIO_EVENT_IFACE_DEFAULT_CFG();
cfg.dispatcher = evt_process;
cfg.queue_size = 10;
cfg.context = &evt1;
cfg.type = AUDIO_EVENT_TYPE_ELEMENT;
evt1 = audio_event_init(&cfg);
TEST_ASSERT_NOT_NULL(evt1);

audio_event_msg_t msg;
int i;
ESP_LOGI(TAG, "[✓] dispatch 10 msg to evt1");
for (i = 0; i < 10; i++) {
    msg.cmd = i;
    TEST_ASSERT_EQUAL(ESP_OK, audio_event_dispatch(evt1, &msg));
}
msg.cmd = 10;
TEST_ASSERT_EQUAL(ESP_FAIL, audio_event_dispatch(evt1, &msg));
ESP_LOGI(TAG, "[✓] listening 10 event have dispatched fron evt1");
while (audio_event_listen(evt1) == ESP_OK);
```

Audio Pipeline

Example

```
audio_element_handle_t first_el, mid_el, last_el;
audio_element_cfg_t el_cfg = DEFAULT_AUDIO_ELEMENT_CONFIG();
```

```
el_cfg.open = _el_open;
el_cfg.read = _el_read;
el_cfg.process = _el_process;
el_cfg.close = _el_close;
first_el = audio_element_init(&el_cfg, "first");
TEST_ASSERT_NOT_NULL(first_el);

el_cfg.read = NULL;
el_cfg.write = NULL;
mid_el = audio_element_init(&el_cfg, "mid");
TEST_ASSERT_NOT_NULL(mid_el);
el_cfg.write = _el_write;
last_el = audio_element_init(&el_cfg, "last");
TEST_ASSERT_NOT_NULL(last_el);

audio_pipeline_cfg_t pipeline_cfg = DEFAULT_AUDIO_PIPELINE_CONFIG();
audio_pipeline_handle_t pipeline = audio_pipeline_init(&pipeline_cfg);
TEST_ASSERT_NOT_NULL(pipeline);
TEST_ASSERT_EQUAL(ESP_OK, audio_pipeline_register(pipeline, first_el, mid_el, last_
↪el));
TEST_ASSERT_EQUAL(ESP_OK, audio_pipeline_link(pipeline, (const char *[]){"first", "mid
↪", "last"}, 3));
```

Audio Peripheral

Example

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

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


esp_periph_start(button_handle);

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

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

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

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

```
ESP_LOGI(TAG, "destroy...");
esp_periph_destroy();
```

Audio Player

Example

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

 fatfs_stream_cfg_t fatfs_cfg = {
    .type = AUDIO_STREAM_READER,
    .root_path = "/sdcard",
};
i2s_stream_cfg_t i2s_cfg = I2S_STREAM_CFG_DEFAULT();
esp_audio_input_stream_add(player, fatfs_stream_init(&fatfs_cfg));

i2s_cfg.type = AUDIO_STREAM_WRITER;
esp_audio_output_stream_add(player, i2s_stream_init(&i2s_cfg));
wav_decoder_cfg_t wav_cfg = DEFAULT_WAV_DECODER_CONFIG();
esp_audio_codec_lib_add(player, AUDIO_CODEC_TYPE_DECODER, wav);
```

3.4 Development Boards

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

3.4.1 ESP32-LyraT V4.3 Hardware Reference

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

In this Section

Overview

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

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

Functional Description

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

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

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

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

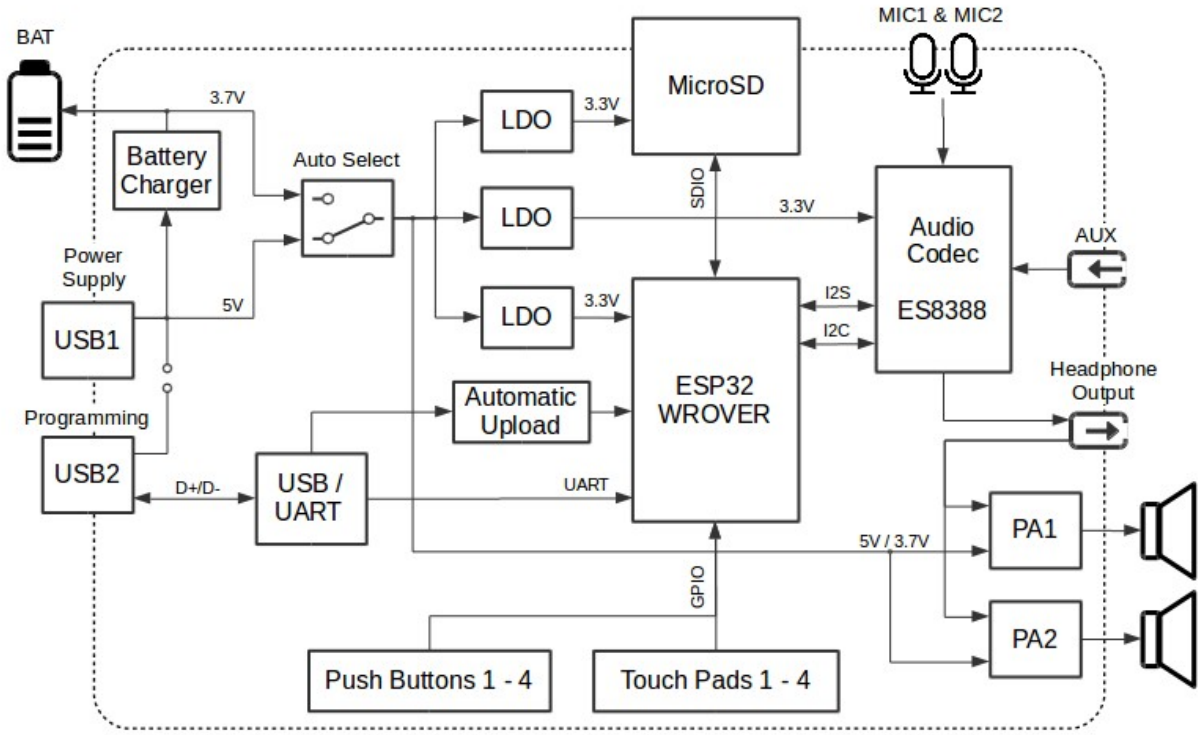

Fig. 3.4: ESP32-LyraT V4.3 Electrical Block Diagram

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

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

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

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

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

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

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

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

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

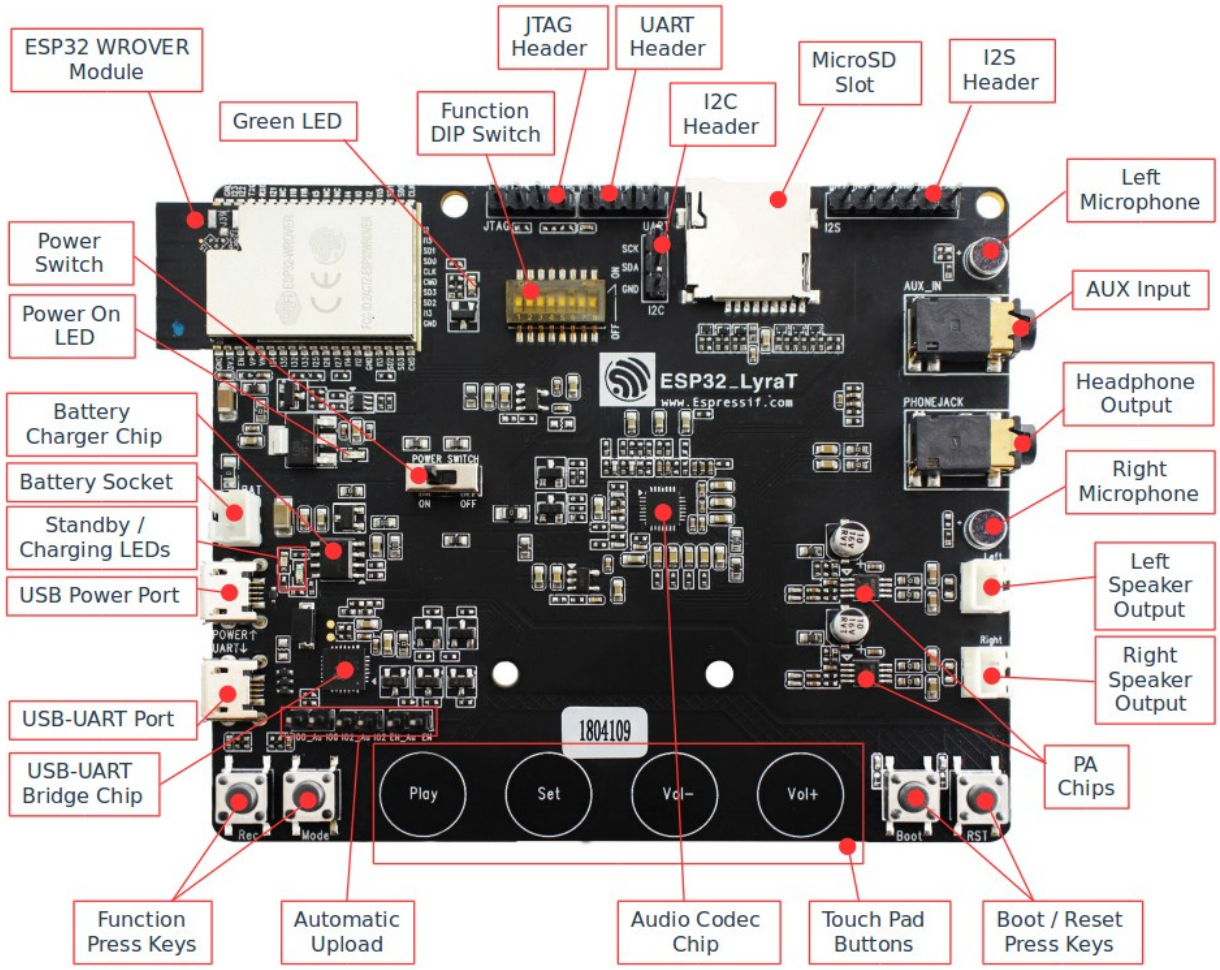

Fig. 3.5: ESP32-LyraT V4.3 Board Layout

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

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

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

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

Boot/Reset Press Keys Boot button: holding down the **Boot** button and momentarily pressing the **Reset** button to initiate the firmware download mode. Then you can download firmware through the serial port. Reset button: pressing this button alone resets the system.

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

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

Automatic Upload Install three jumpers on this header to enable automatic loading of application to the ESP32. Install all jumpers together on all three headers. Remove all jumpers after upload is complete.

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

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

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

USB Power Port Provides the power supply for the board.

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

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

Note: Please verify if polarity on the battery plug matches polarity of the socket as marked on the board's soldermask besides the socket.

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

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

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

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

Hardware Setup Options

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

Enable MicroSD Card in 1-wire Mode

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

1. **AUX Input** detection may be enabled by toggling the DIP SW 7 *ON*. Note that the **AUX Input** signal pin should not be be plugged in when the system powers up. Otherwise the ESP32 may not be able to boot correctly.

In this mode:

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

Enable MicroSD Card in 4-wire Mode

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

In this mode:

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

Enable JTAG

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

In this mode:

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

Using Automatic Upload

Entering of the ESP32 into upload mode may be done in two ways:

- Manually by pressing both **Boot** and **RST** keys and then releasing first **RST** and then **Boot** key.
- Automatically by software performing the upload. The software is using **DTR** and **RTS** signals of the serial interface to control states of **EN**, **IO0** and **IO2** pins of the ESP32. This functionality is enabled by installing jumpers in three headers **JP23**, **JP24** and **JP25**. For details see ESP32 LyraT V4.3 schematic. Remove all jumpers after upload is complete.

Allocation of ESP32 Pins

Several pins ESP32 module are allocated to the on board hardware. Some of them, like GPIO0 or GPIO2, have multiple functions. Please refer to the table below or ESP32 LyraT V4.3 schematic for specific details.

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

- (TP) - touch pad
- (PB) - push button

Pinout of Extension Headers

There are several pin headers available to connect external components, check the state of particular signal bus or debug operation of ESP32. Note that some signals are shared, see section *Allocation of ESP32 Pins* for details.

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

I2S Header / JP4

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

I2C Header / JP5

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

JTAG Header / JP7

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

Notes of Power Distribution

The board features quite extensive power distribution system. It provides independent power supplies to all critical components. This should reduce noise in the audio signal from digital components and improve overall performance of the components.

Power Supply Separation

The main power supply is 5V and provided by a USB. The secondary power supply is 3.7V and provided by an optional battery. The USB power itself is fed with a dedicated cable, separate from a USB cable used for an application upload. To further reduce noise from the USB, the battery may be used instead of the USB.

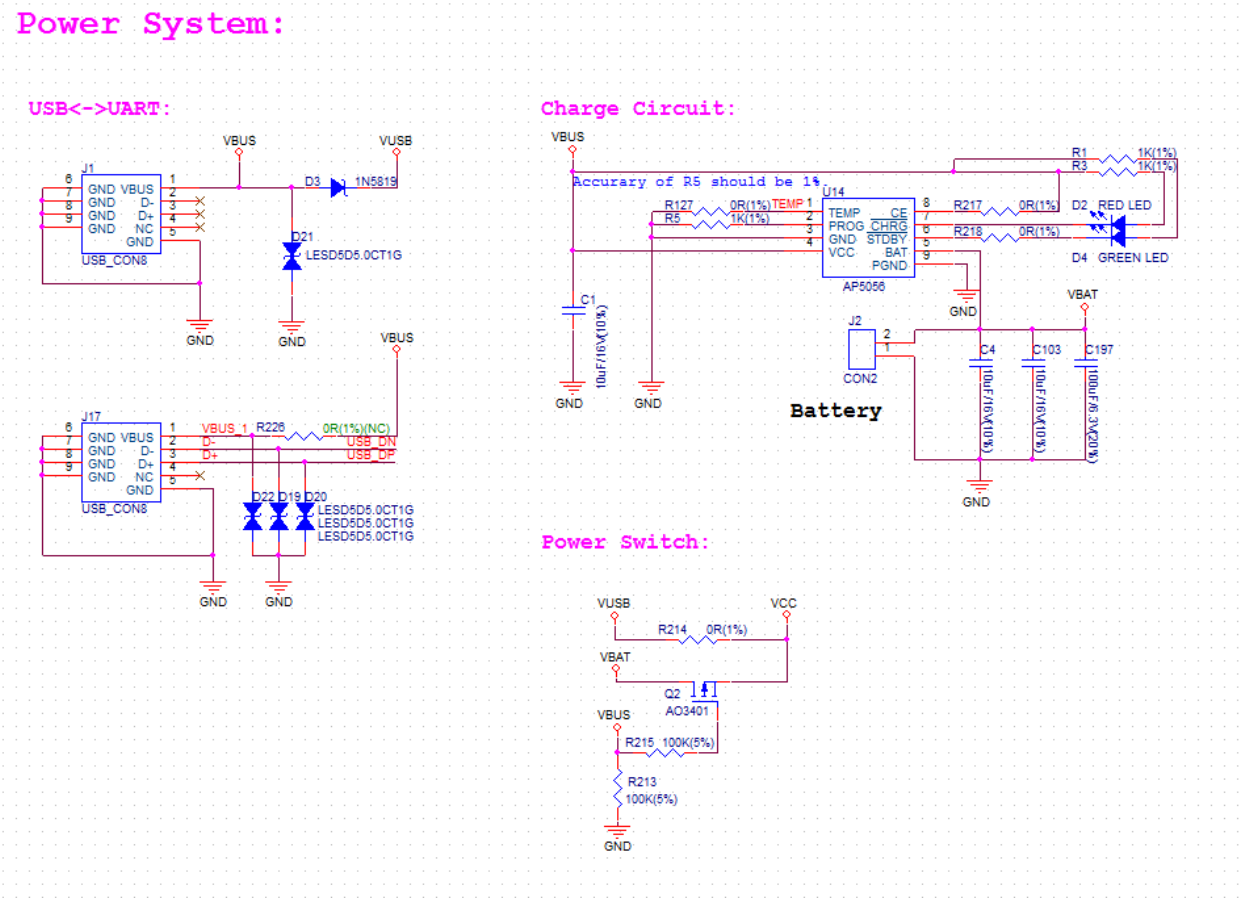

Fig. 3.6: ESP32 LyraT V4.3 - Power Supply Separation

Three Dedicated LDOs

ESP32 Module

To provide enough current the ESP32, the development board adopts LD1117S33CTR LDO capable to supply the maximum output current of 800mA.

MicroSD Card and **Audio Codec**

Two separate LDOs are provided for the MicorSD Card and the Audio Codec. Both circuits have similar design that includes an inductor and double decoupling capacitors on both the input and output of the LDO.

Separate Power Feed for the PAs

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

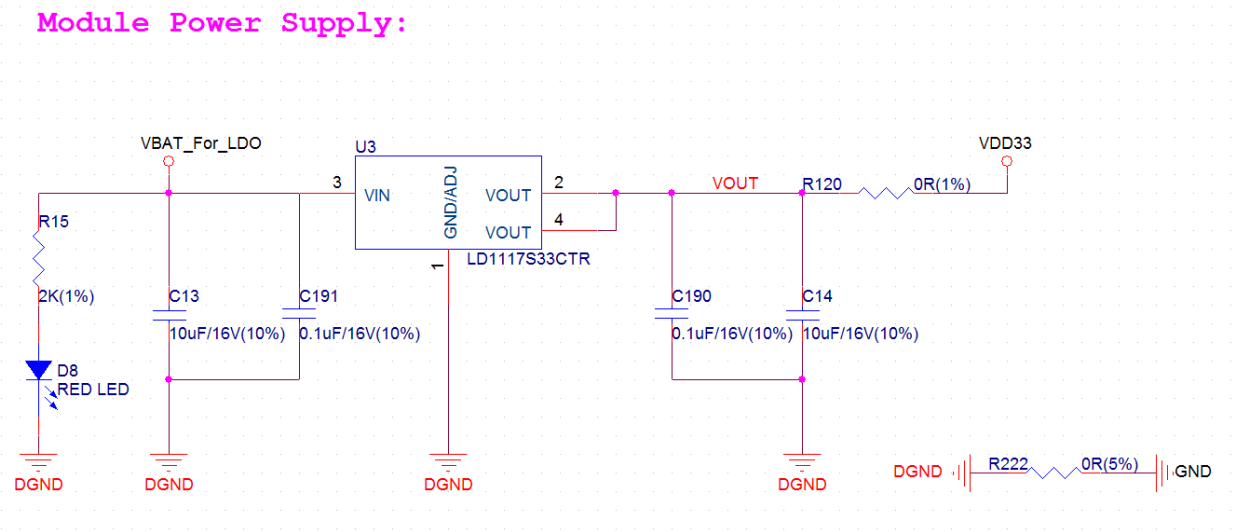

Fig. 3.7: ESP32 LyraT V4.3 - Dedicated LDO for the ESP32 Module

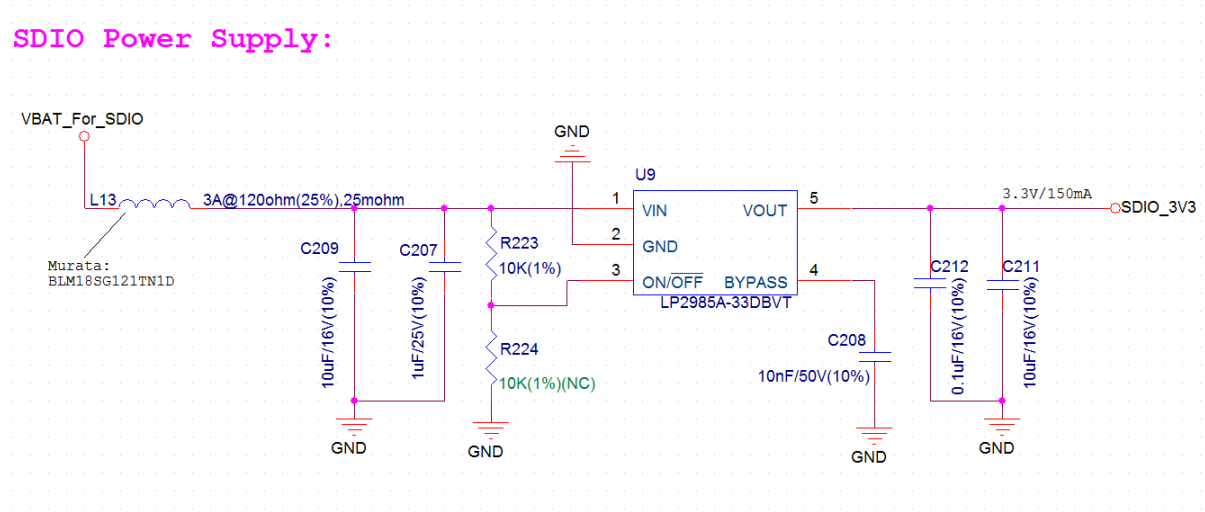

Fig. 3.8: ESP32 LyraT V4.3 - Dedicated LDO for the MicroSD Card

development board adds a set of LC circuits at the front of the PA power supply, where L uses 1.5A magnetic beads and C uses 10uF aluminum electrolytic capacitors, to effectively filter out power crosstalk.

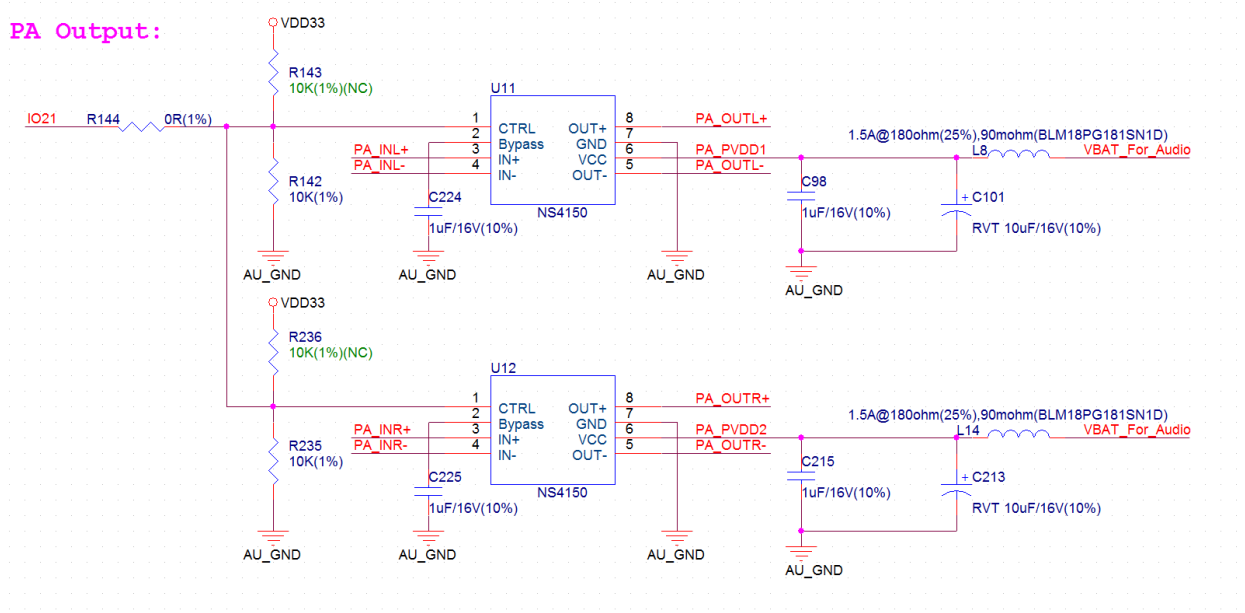

Fig. 3.9: ESP32 LyraT V4.3 - Power Supply for the PAs

Selecting of the Audio Output

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

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

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

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

Other Versions of LyraT

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

Related Documents

- ESP32 LyraT V4.3 schematic (PDF)
- *ESP32-LyraT V4.3 Getting Started Guide*
- ESP32 Datasheet (PDF)

- ESP32-WROVER Datasheet (PDF)
- JTAG Debugging

CHAPTER 4

Resources

- The esp32.com forum is a place to ask questions and find community resources.
- This ESP Audio Development Framework inherits from ESP IoT Development Framework and you can learn about it in ESP-IDF documentation.
- 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.
- Mirror of this documentation is available under: https://dl.espressif.com/doc/esp-adf/latest/.

CHAPTER 5

Copyrights and Licenses

5.1 Software Copyrights

All original source code in this repository is Copyright (C) 2015-2018 Espressif Systems. This source code is licensed under the ESPRESSIF MIT License as described in the file LICENSE.

Additional third party copyrighted code is included under the following licenses:

- esp-stagefright is Copyright (c) 2005-2008, The Android Open Source Project, and is licensed under the Apache License Version 2.0.

Please refer to the COPYRIGHT in ESP-IDF Programming Guide

Where source code headers specify Copyright & License information, this information takes precedence over the summaries made here.

CHAPTER 6

About

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

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

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

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

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

- genindex

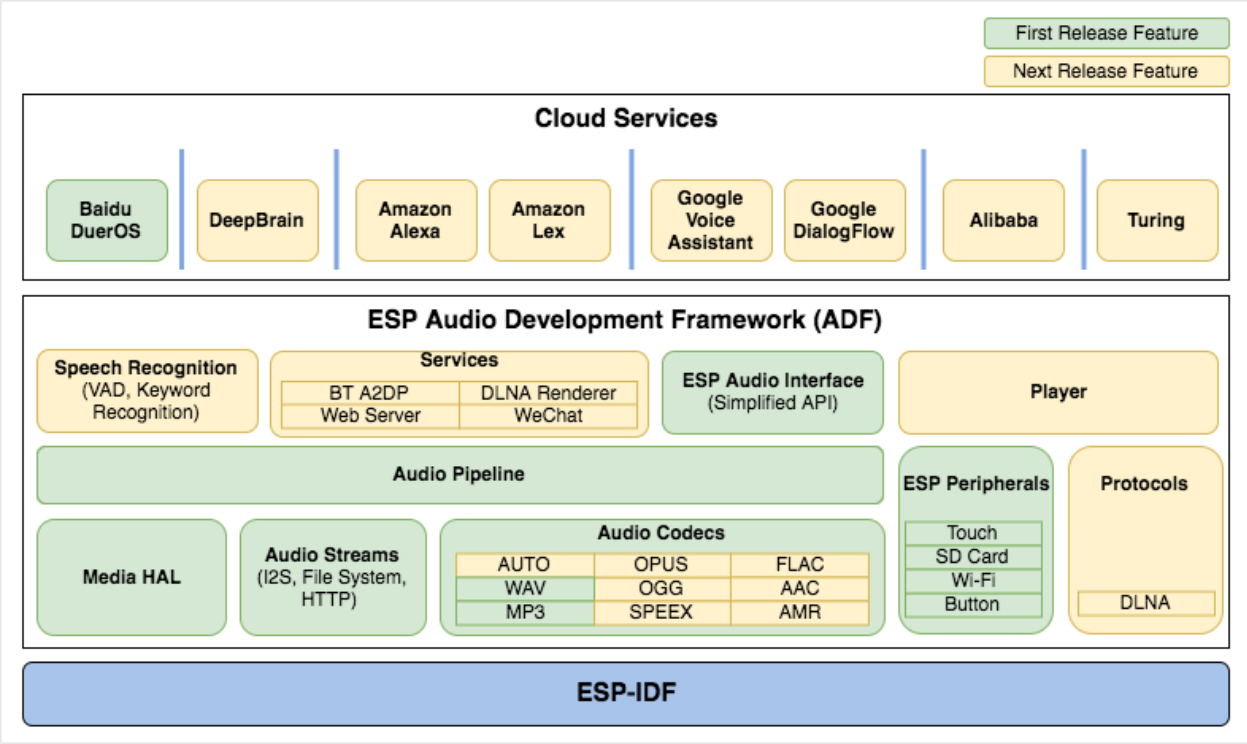

Fig. 6.1: Espressif Audio Development Framework

Index

A

D

E

F

H

I

M

O

P

R

S