Audio Development Framework Handbook

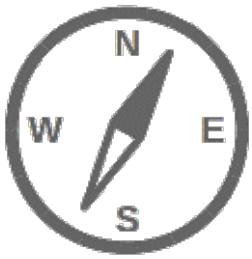
Release latest

Espressif Systems

Sep 20, 2024

CONTENTS

This is the documentation for Espressif Audio Development Framework (ADF).

		
Get Started	API Reference	Design Guide

CHAPTER

ONE

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 family of chips by Espressif. After that, a simple example will show you how to use ESP-ADF (Espressif Audio Development Framework).

1.1 Development Board Overview

For easier start with ESP-ADF, Espressif designed ESP32, ESP32-S2, and ESP32-S3 based development boards intended for audio applications. Click the links below to learn more about the available boards.

- *ESP32-LyraT*
- *ESP32-LyraT-Mini*
- *ESP32-LyraTD-MSC*
- ESP32-S2-Kaluga-1
- *ESP32-Korvo-DU1906*
- *ESP32-S3-Korvo-2*
- *ESP32-C3-Lyra*

If you do not have any of the above boards, you can still use ESP-ADF for the ESP32 and ESP32-S2 based audio applications. For this, your board needs to have a compatible audio codec or DSP chip; alternatively, you can develop a driver to support communication with your specific 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.

Note: ESP-ADF provides support for specific ESP-IDF versions. If your have already set up another version, please switch to a supported ESP-IDF version, or you may not be able to compile ESP-ADF applications. In addition, the python version needs to be between 3.7 and 3.11.

1.3 Quick Start

This section provides quick steps to run a simple ADF sample project on an ESP device for experienced users. For beginners, please go through the complete steps from *Step 1. Set up ESP-IDF* to *Step 9. Monitor* to build a project.

Note: If you encounter issues in the following steps, you could refer to the complete steps from *Step 1. Set up ESP-IDF* to *Step 9. Monitor* or describe them in GitHub Issues or ESP Forum.

1.3.1 Linux and macOS

The operating environment below is on Linux Ubuntu 18.04 and above.

1. Download the full ESP-ADF repository from GitHub by running:

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

   For users located in China, it is faster to download from Gitee:

```
git clone --recursive https://gitee.com/EspressifSystems/esp-adf.git
```

2. Configure the `$ESP-IDF` and `$ESP-ADF` compilation environment by running:

```
cd esp-adf
./install.sh
. ./export.sh
```

3. After completing the above environment variable configuration, you can compile the ADF sample project `$ADF_PATH/examples/get-started/play_mp3_control`. Switch to the project's directory, compile, and flash it onto your ESP device by running the following command. Then, you will see the serial port of the routine is printed.

```
cd $ADF_PATH/examples/get-started/play_mp3_control
idf.py build flash monitor
```

1.3.2 Windows

1. Download the full ESP-ADF repository from GitHub by running:

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

   For users located in China, it is faster to download from Gitee:

```
git clone --recursive https://gitee.com/EspressifSystems/esp-adf.git
```

2. Install the `$ESP-IDF` compilation environment in the command prompt window:

```
cd esp-adf
.\install.bat
```

   Or first download the full ESP-IDF Windows Installer from ESP-IDF Windows Installer (Please download the ESP-IDF versions supported by ESP-ADF). And then turn off the antivirus software (Because it may prevent the installation as the software writes the Windows system regedit) and install the downloaded file. After the installation

is complete, open the ESP-IDF CMD shortcut icon on the desktop, the script will automatically help you download submodules, and set environment variables such as `IDF_PATH`.

3. Set the `ADF_PATH` by running the following commands:

```
.\export.bat
echo %ADF_PATH%
```

4. If your `ADF_PATH` variable prints correctly, it's time to compile the ADF routines:

```
cd %ADF_PATH%\examples\get-started\play_mp3_control
idf.py build flash monitor
```

1.4 Installation Step by Step

This is a detailed roadmap to walk you through the installation process.

1.4.1 Setting up Development Environment

- *Step 1. Set up ESP-IDF* for Windows, Linux or Mac OS
- *Step 2. Get ESP-ADF*
- *Step 3. Set up the environment*

1.4.2 Creating Your First Project

- *Step 4. Start a Project*
- *Step 5. Connect Your Device*
- *Step 6. Configure*
- *Step 7. Build the Project*
- *Step 8. Flash onto the Device*
- *Step 9. Monitor*

1.5 Step 1. Set up ESP-IDF

Configure your PC according to **Getting Started** section of **ESP-IDF Programming Guide**. Windows, Linux and Mac OS operating systems are supported. Please select and follow the guide specific to ESP32 or ESP32-S2 chip. The chip name is provided in the board name.

Note: This guide uses the directory `~/esp` on Linux and macOS or `%userprofile%\esp` on Windows as an installation folder for ESP-ADF. You can use any directory, but you will need to adjust paths for the commands accordingly. Keep in mind that ESP-ADF does not support spaces in paths.

To make the installation easier and less prone to errors, use the `~/esp` default directory for the installation.

If this is your first exposure to the ESP-IDF, then it is recommended to get familiar with **hello_world** or **blink** example first.

After getting familiar with ESP-IDF, decide on which ESP-IDF version to use for your application depending on the Espressif chip that you have and your project type. For this, consult Versions section of ESP-IDF Programming Guide.

Once you successfully build, upload, and run examples for your version of ESP-IDF, you can proceed to the next step.

1.6 Step 2. Get ESP-ADF

Now you can start installing audio-specific API / libraries provided in ESP-ADF repository.

1.6.1 Windows

Open Command Prompt and run the following commands:

```
cd %userprofile%\esp
git clone --recursive https://github.com/espressif/esp-adf.git
```

1.6.2 Linux and macOS

Open Terminal, and run the following commands:

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

1.7 Step 3. Set up the environment

Before being able to compile ESP-ADF projects, on each new session, ESP-IDF tools should be added to the PATH environment variable. To make the tools usable from the command line, some environment variables must be set. ESP-ADF provides a script which does that.

1.7.1 Windows

ESP-IDF Tools Installer for Windows creates an "ESP-IDF Command Prompt" shortcut in the Start Menu. This shortcut opens the Command Prompt and sets up all the required environment variables. You can open this shortcut and proceed to the next step.

Alternatively, if you want to use ESP-IDF in an existing Command Prompt window, you can run:

```
%userprofile%\esp\esp-adf\export.bat
```

1.7.2 Linux and macOS

In the terminal where you have installed ESP-IDF, run:

```
. $HOME/esp/esp-adf/export.sh
```

Note the space between the leading dot and the path!

You can also create an alias for the export script to your `.profile` or `.bash_profile` script. This way you can set up the environment in a new terminal window by typing `get_idf`:

```
alias get_idf='. $HOME/esp/esp-adf/export.sh'
```

Note that it is not recommended to source `export.sh` from the profile script directly. Doing so activates IDF virtual environment in every terminal session (even in those where IDF is not needed), defeating the purpose of the virtual environment and likely affecting other software.

1.8 Step 4. Start a Project

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

To demonstrate how to build an application, we will use get-started/play_mp3_control project from examples directory in the ADF.

1.8.1 Windows

```
cd %userprofile%\esp
xcopy /e /i %ADF_PATH%\examples\get-started\play_mp3_control play_mp3_control
```

1.8.2 Linux and macOS

```
cd ~/esp
cp -r $ADF_PATH/examples/get-started/play_mp3_control .
```

There is a range of example projects in the examples directory in ESP-ADF. You can copy any project in the same way as presented above and run it.

It is also possible to build examples in-place, without copying them first.

Important: The ESP-IDF build system does not support spaces in the paths to either ESP-IDF or to projects.

1.9 Step 5. Connect Your Device

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

Note: Keep the port name handy as you will need it in the next steps.

1.10 Step 6. Configure

Navigate to your `play_mp3_control` directory from *Step 4. Start a Project* and configure the project:

1.10.1 Windows

```
cd %userprofile%\esp\play_mp3_control
idf.py set-target esp32
idf.py menuconfig
```

1.10.2 Linux and macOS

```
cd ~/esp/play_mp3_control
idf.py set-target esp32
idf.py menuconfig
```

Note: If you are using an ESP32-S2 based board, then the second command above should be `idf.py set-target esp32s2`.

Setting the target with `idf.py set-target <target>` should be done once, after opening a new project. If the project contains some existing builds and configuration, they will be cleared and initialized. The target may be saved in environment variable to skip this step at all. See Selecting the Target in ESP-IDF Programming Guide for additional information.

If the previous steps have been done correctly, the following menu appears:

You are using this menu to set up your board type and other project specific variables, e.g. Wi-Fi network name and password, the processor speed, etc.

Select your board from the menu, press `S` to save configuration and then `Q` to exit.

Note: The colors of the menu could be different in your terminal. You can change the appearance with the option `--style`. Please run `idf.py menuconfig --help` for further information.

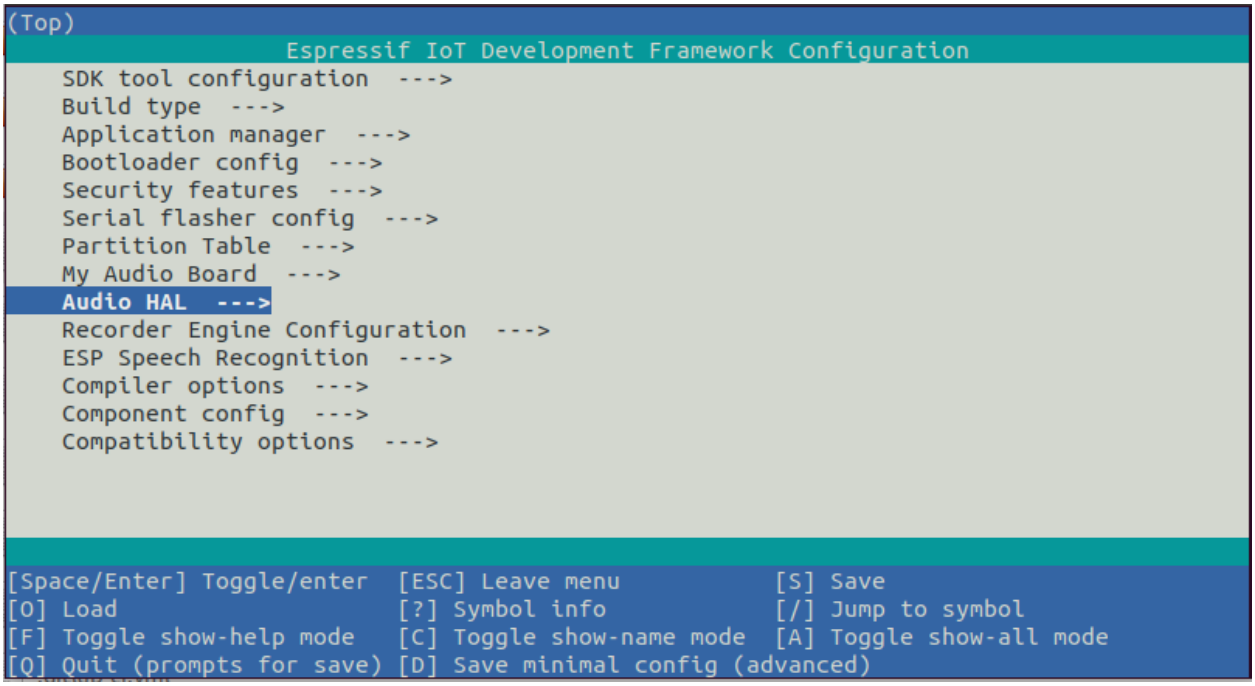


Fig. 1: Project configuration - Home window

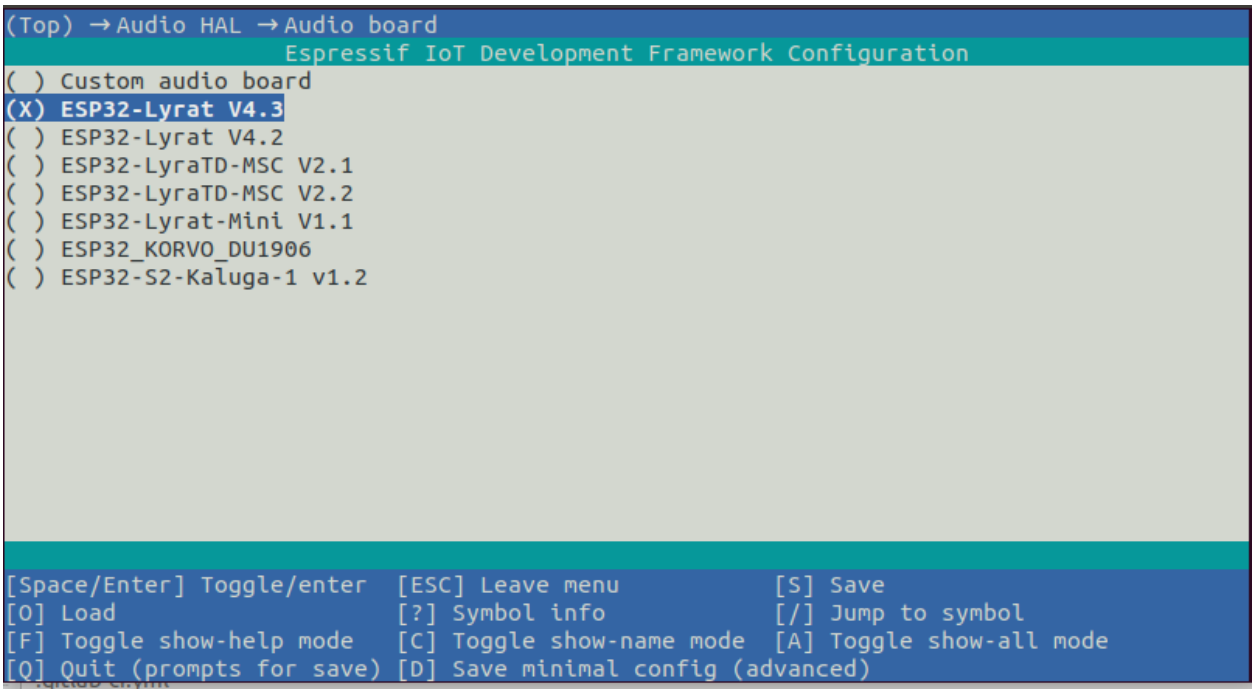


Fig. 2: Project configuration - Board selection

1.11 Step 7. Build the Project

Build the project by running:

```
idf.py build
```

This command will compile the application and all ESP-IDF and ESP-ADF components, then it will generate the bootloader, partition table, and application binaries.

```
$ idf.py build
 Executing action: all (aliases: build)
 Running ninja in directory /path/to/esp/play_mp3_control/build
 Executing "ninja all"...
 [0/1] Re-running CMake...

... (more lines of build system output)

 [1064/1064] Generating binary image from built executable
 esptool.py v3.0-dev
 Generated /path/to/esp/play_mp3_control/build/play_mp3_control.bin

 Project build complete. To flash it, run this command:
 /path/to/.espressif/python_env/idf4.2_py2.7_env/bin/python ../esp-idf/components/
↪esptool_py/esptool/esptool.py -p (PORT) -b 460800 --before default_reset --after␣
↪hard_reset --chip esp32  write_flash --flash_mode dio --flash_size detect --flash_
↪freq 40m 0x1000 build/bootloader/bootloader.bin 0x8000 build/partition_table/
↪partition-table.bin 0x10000 build/play_mp3_control.bin
 or run 'idf.py -p (PORT) flash'
```

If there are no errors, the build will finish by generating the firmware binary .bin file.

1.12 Step 8. Flash onto the Device

Flash the binaries that you just built onto your board by running:

```
idf.py -p PORT [-b BAUD] flash monitor
```

Replace PORT with your board's serial port name from *Step 5. Connect Your Device*.

You can also change the flasher baud rate by replacing BAUD with the baud rate you need. The default baud rate is `460800`.

For more information on idf.py arguments, see Using the Build System in ESP-IDF Programming Guide.

Note: The option `flash` automatically builds and flashes the project, so running `idf.py build` is not necessary.

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

```
...

esptool.py v3.0-dev
```

(continues on next page)

(continued from previous page)

```
Serial port /dev/ttyUSB0
Connecting........_____....
```

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

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

```
...

Leaving...
Hard resetting via RTS pin...
Executing action: monitor
Running idf_monitor in directory /path/to/esp/play_mp3_control
Executing "/path/to/.espressif/python_env/idf4.2_py2.7_env/bin/python /path/to/esp/
↪esp-idf/tools/idf_monitor.py -p /dev/ttyUSB0 -b 115200 --toolchain-prefix xtensa-
↪esp32-elf- /path/to/esp/play_mp3_control/build/play_mp3_control.elf -m '/path/to/.
↪espressif/python_env/idf4.2_py2.7_env/bin/python' '/path/to/esp/esp-idf/tools/idf.py
↪'"...
--- idf_monitor on /dev/ttyUSB0 115200 ---
--- Quit: Ctrl+] | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
```

If there are no issues by the end of the flash process, the board will reboot and start up the "play_mp3_control" application.

1.13 Step 9. Monitor

At this point press the **Reset** button to start the application. Following several lines of start up log, the `play_mp3_control` application specific messages should be displayed:

```
...

I (397) PLAY_FLASH_MP3_CONTROL: [ 1 ] Start audio codec chip
I (427) PLAY_FLASH_MP3_CONTROL: [ 2 ] Create audio pipeline, add all elements to␣
↪pipeline, and subscribe pipeline event
I (427) PLAY_FLASH_MP3_CONTROL: [2.1] Create mp3 decoder to decode mp3 file and set␣
↪custom read callback
I (437) PLAY_FLASH_MP3_CONTROL: [2.2] Create i2s stream to write data to codec chip
I (467) PLAY_FLASH_MP3_CONTROL: [2.3] Register all elements to audio pipeline
I (467) PLAY_FLASH_MP3_CONTROL: [2.4] Link it together [mp3_music_read_cb]-->mp3_
↪decoder-->i2s_stream-->[codec_chip]
I (477) PLAY_FLASH_MP3_CONTROL: [ 3 ] Set up  event listener
I (477) PLAY_FLASH_MP3_CONTROL: [3.1] Listening event from all elements of pipeline
I (487) PLAY_FLASH_MP3_CONTROL: [ 4 ] Start audio_pipeline
I (507) PLAY_FLASH_MP3_CONTROL: [ * ] Receive music info from mp3 decoder, sample_
↪rates=44100, bits=16, ch=2
I (7277) PLAY_FLASH_MP3_CONTROL: [ 5 ] Stop audio_pipeline
```

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_control/Makefile and you should see

```
include($ENV{ADF_PATH}/CMakeLists.txt)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
```

The first line contains `ADF_PATH` to point the toolchain to another file in ESP-ADF directory that provides configuration variables and path to ESP-ADF components reacquired by the toolchain. You need similar `Makefile` in your own applications developed with the ESP-ADF.

1.14 VS Code Extension

1. Follow VS Code Extension Quick Installation Guide to install ESP-IDF Visual Studio Code Extension. If the previous steps have been done correctly, the following toolbar appears:

Fig. 3: VS Code Extension Toolbar

2. To install the ESP-ADF extension, open `Command Palette` and enter `install adf`. Then, a progress bar shows up in the lower right corner.

   If you have cloned the ESP-ADF repository before, please enter `open settings(ui)` in `Command Palette`. Go to `User > Extensions > ESP_IDF` and manually set the ESP-ADF path in `idf.espAdfPath` or `idf.espAdfPathWin` (for Windows). You can also set the ESP-ADF path in `.vscode/settings.json`.

3. In `Command Palette`, enter `show examples project`, and then a window will be opened with a list of example projects.

4. Select an example, click `Create project using example XX`, and select the directory to save the current example.

5. On the toolbar at the bottom of VS Code, click the gear symbol `menuconfig` to configure the example and click the column symbol `Build` to build the example. See available shortcut keys for VS code extensions.

6. On the toolbar at the bottom of VS Code, click the plug-in symbol `Select Port` to configure the serial port and click the lightning symbol `Flash Device` to flash firmware. After the firmware is flashed successfully, click `Monitor Device` to start the monitor function. Or, you can also use the flame symbol to build, flash, and monitor the example at the same time.

1.15 IDF Eclipse Plugin and Espressif IDE

1.15.1 Install and Set up Environment Variables

1. Follow IDF Eclipse Plugin Quick Installation Guide to install IDF Eclipse Plugin or download and install Espressif IDE from Espressif IDE Download Link. If the previous steps have been done correctly, you can create, build and flash IDF project in the Eclipse environment.

2. To install ESP-ADF, follow section *Step 2. Get ESP-ADF*.

3. To set `ADF_PATH` environment variable, open `Window` > `Preferences` > `C/C++` > `Build` > `Environment` panel, click **Add** button and fill in `ADF_PATH`. After you complete the above steps, select `ADF_PATH` in `Environment variables` table and click **Edit** and **OK** button without changing any value (There is a bug in Eclipse CDT that is appending a null value before the path hence we need to click on edit and save it.).

   If this step does not work, you can delete `ADF_PATH` set in Eclipse and set `ADF_PATH` as system environment variable. For Windows, set environment variable in `Advanced System Setting` panel. For Linux and macOS, add `export ADF_PATH=your adf path` in file `/etc/profile`. However,

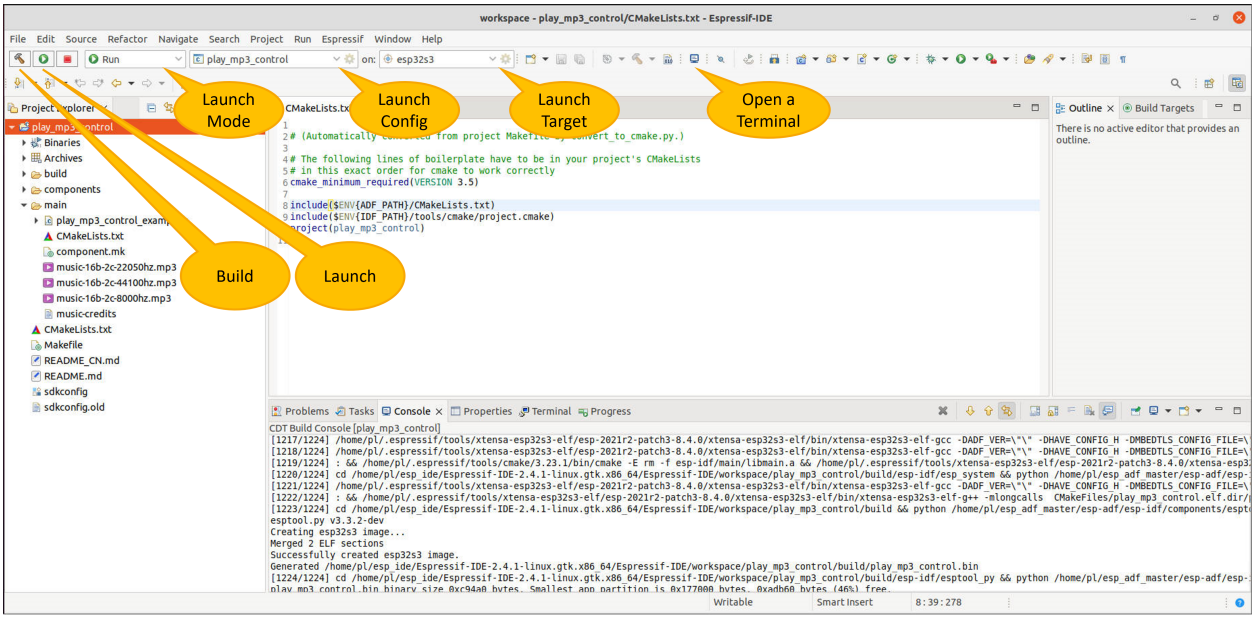


Fig. 4: Espressif IDE (Reskinned Eclipse)

it is not recommended. Doing so activates ADF virtual environment in every terminal session (including those where ADF is not needed), defeating the purpose of the virtual environment and likely affecting other software.

1.15.2 Create a New Project

1. To create new project, go to `File` > `New` > `Espressif IDF Project` and provide a project name.
2. Click **Finish** to create an empty project. Or click **Next** and check `Create a project using one of the templates` to create a project using ESP-IDF templates.

After creating a new project, you can use ESP-IDF and ESP-ADF to develop the project.

1.15.3 Import Existing Project

To import existing ESP-ADF examples, go to `File` > `Import` > `Espressif` > `Existing IDF Project` and select an ESP-ADF example (Opening an existing project directly may not be able to set the ESP target).

1.15.4 Quick Start

1. Select a project from `Project Explorer`.
2. In the **Launch Mode** drop-down menu, select `Run`.
3. In the **Launch Configuration** (auto-detected) drop-down menu, select your application.
4. Select ESP target from the third drop-down, which is called **Launch Target**. Click gear symbol **Edit** button of **Launch Target** to set `Serial Port`.
5. Double click `sdkconfig` file to launch the `SDK Configuration Editor`.
6. Click **Build** button to build the project.

7. Click **Launch** button to flash the project.
8. Click **Open a Terminal** button and select **ESP-IDF Serial Monitor** to view serial output.

For more information about IDF Eclipse Plugin and Espressif IDE, please refer to ESP-IDF Eclipse Plugin.

1.16 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 *Step 2. 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.

CHAPTER

TWO

API REFERENCE

This API provides a way to develop audio applications using *Elements* like *Codecs* (Decoders and Encoders), *Streams* or *Audio Processing* functions.

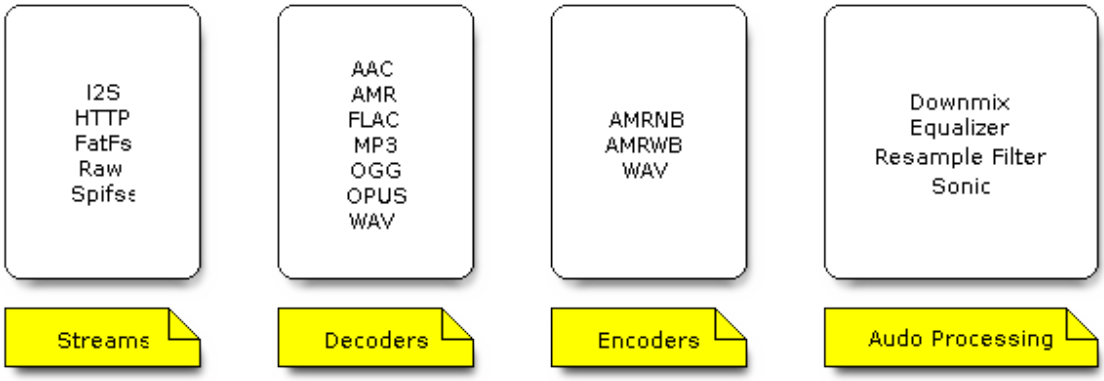


Fig. 1: **Elements** of the Audio Development Framework

The application is developed by combining the *Elements* into a *Pipeline*. A diagram below presents organization of two elements, MP3 decoder and I2S stream, in the Audio Pipeline, that has been used in get-started/play_mp3_control example.

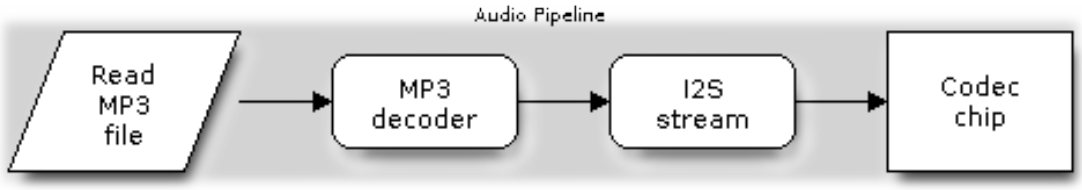


Fig. 2: Sample Organization of Elements in Audio Pipeline

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

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

2.1 Audio Framework

2.1.1 Audio Element

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

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

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

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

API Reference

Header File

- components/audio_pipeline/include/audio_element.h

Functions

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

Initialize audio element with config.

Parameters **`config`** – The configuration

Returns

- audio_elemenent handle object
- NULL

esp_err_t **`audio_element_deinit`** (*audio_element_handle_t* el)

Destroy audio element handle object, stop, clear, deletel all.

Parameters **`el`** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **`audio_element_setdata`** (*audio_element_handle_t* el, void *data)

Set context data to element handle object. It can be retrieved by calling `audio_element_getdata`.

Parameters

- **`el`** – **[in]** The audio element handle
- **`data`** – The data pointer

Returns

- ESP_OK
- ESP_FAIL

void *`audio_element_getdata` (*audio_element_handle_t* el)

Get context data from element handle object.

Parameters `el` – **[in]** The audio element handle

Returns data pointer

esp_err_t `audio_element_set_tag` (*audio_element_handle_t* el, const char *tag)

Set elemenet tag name, or clear if tag = NULL.

Parameters

- `el` – **[in]** The audio element handle
- `tag` – **[in]** The tag name pointer

Returns

- ESP_OK
- ESP_FAIL

char *`audio_element_get_tag` (*audio_element_handle_t* el)

Get element tag name.

Parameters `el` – **[in]** The audio element handle

Returns Element tag name pointer

esp_err_t `audio_element_setinfo` (*audio_element_handle_t* el, *audio_element_info_t* *info)

Set audio element infomation.

Parameters

- `el` – **[in]** The audio element handle
- `info` – The information pointer

Returns

- ESP_OK
- ESP_FAIL

esp_err_t `audio_element_getinfo` (*audio_element_handle_t* el, *audio_element_info_t* *info)

Get audio element infomation.

Parameters

- `el` – **[in]** The audio element handle
- `info` – The information pointer

Returns

- ESP_OK
- ESP_FAIL

esp_err_t `audio_element_set_uri` (*audio_element_handle_t* el, const char *uri)

Set audio element URI.

Parameters

- **el** – **[in]** The audio element handle
- **uri** – **[in]** The uri pointer

Returns

- ESP_OK
- ESP_FAIL

char ***audio_element_get_uri** (*audio_element_handle_t* el)

Get audio element URI.

Parameters **el** – **[in]** The audio element handle

Returns URI pointer

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.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_terminate** (*audio_element_handle_t* el)

Terminate Audio Element. With this function, audio_element will exit the task function. Note: this API only sends request. It does not actually terminate immediately when this function returns.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_terminate_with_ticks** (*audio_element_handle_t* el, TickType_t ticks_to_wait)

Terminate Audio Element with specific ticks for timeout. With this function, audio_element will exit the task function. Note: this API only sends request. It does not actually terminate immediately when this function returns.

Parameters

- **el** – **[in]** The audio element handle
- **ticks_to_wait** – **[in]** The maximum amount of time to blocking

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK

- ESP_FAIL

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

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

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK, Success
- ESP_FAIL, The state is not AEL_STATE_RUNNING
- ESP_ERR_TIMEOUT, Timeout

esp_err_t **audio_element_wait_for_stop_ms**(*audio_element_handle_t* el, TickType_t ticks_to_wait)

After the `audio_element_stop` function is called, the Element task will perform some abort procedures. The maximum amount of time should block waiting for Element task has stopped.

Parameters

- **el** – **[in]** The audio element handle
- **ticks_to_wait** – **[in]** The maximum amount of time to wait for stop

Returns

- ESP_OK, Success
- ESP_FAIL, The state is not AEL_STATE_RUNNING
- ESP_ERR_TIMEOUT, Timeout

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.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters

- **el** – **[in]** The audio element handle
- **wait_for_rb_threshold** – **[in]** The wait for rb threshold (0 .. 1)
- **timeout** – **[in]** The timeout

Returns

- ESP_OK
- ESP_FAIL

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`.

Parameters

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

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters

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

Returns

- ESP_OK
- ESP_FAIL

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

Remove listener out of el. No new events will be sent to the listener.

Parameters

- **el** – **[in]** The audio element handle
- **listener** – The listener

Returns

- ESP_OK
- ESP_FAIL

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

Set Element input ringbuffer.

Parameters

- **el** – **[in]** The audio element handle
- **rb** – **[in]** The ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

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

Get Element input ringbuffer.

Parameters **el** – **[in]** The audio element handle

Returns ringbuf_handle_t

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

Set Element output ringbuffer.

Parameters

- **el** – **[in]** The audio element handle
- **rb** – **[in]** The ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

ringbuf_handle_t **audio_element_get_output_ringbuf** (*audio_element_handle_t* el)

Get Element output ringbuffer.

Parameters **el** – **[in]** The audio element handle

Returns ringbuf_handle_t

audio_element_state_t **audio_element_get_state** (*audio_element_handle_t* el)

Get current Element state.

Parameters **el** – **[in]** The audio element handle

Returns audio_element_state_t

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.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

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`

Parameters

- **el** – **[in]** The audio element handle
- **size_expect** – **[in]** The size expect
- **timeout** – **[in]** The timeout

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **el** – **[in]** The audio element handle
- **status** – **[in]** The status

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_report_pos** (*audio_element_handle_t* el)

Element will sendout event with a duplicate information by this function.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_NO_MEM

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

Set input read timeout (default is `portMAX_DELAY`).

Parameters

- **el** – **[in]** The audio element handle

- **timeout** – **[in]** The timeout

Returns

- ESP_OK
- ESP_FAIL

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

Set output read timeout (default is `portMAX_DELAY`).

Parameters

- **el** – **[in]** The audio element handle
- **timeout** – **[in]** The timeout

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_reset_input_ringbuf**(*audio_element_handle_t* el)

Reset inputbuffer.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_finish_state**(*audio_element_handle_t* el)

Set element finish state.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_change_cmd**(*audio_element_handle_t* el, *audio_element_msg_cmd_t* cmd)

Change element running state with specific command.

Parameters

- **el** – **[in]** The audio element handle
- **cmd** – **[in]** Specific command from audio_element_msg_cmd_t

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG Element handle is null

esp_err_t **audio_element_reset_output_ringbuf**(*audio_element_handle_t* el)

Reset outputbuffer.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

audio_element_err_t **audio_element_input** (*audio_element_handle_t* el, char *buffer, int wanted_size)

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

Parameters

- **el** – **[in]** The audio element handle
- **buffer** – The buffer pointer
- **wanted_size** – **[in]** The wanted size

Returns

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

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

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

Parameters

- **el** – **[in]** The audio element handle
- **buffer** – The buffer pointer
- **write_size** – **[in]** The write size

Returns

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

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.

Parameters

- **el** – **[in]** The audio element handle
- **fn** – **[in]** 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** – **[in]** An optional context which will be passed to callback function on every invocation

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **el** – **[in]** The audio element
- **fn** – **[in]** 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** – **[in]** An optional context which will be passed to callback function on every invocation

Returns

- ESP_OK
- ESP_FAIL

stream_func **audio_element_get_write_cb** (*audio_element_handle_t* el)

Get callback write function that register to the element.

Parameters **el** – **[in]** The audio element

Returns

- Callback write function pointer
- NULL Failed

stream_func **audio_element_get_read_cb** (*audio_element_handle_t* el)

Get callback read function that register to the element.

Parameters **el** – **[in]** The audio element

Returns

- Callback read function pointer
- NULL Failed

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`.

Parameters **el** – **[in]** The audio element handle

Returns QueueHandle_t

esp_err_t **audio_element_set_ringbuf_done** (*audio_element_handle_t* el)

Set inputbuffer and outputbuffer have finished.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_reset_state** (*audio_element_handle_t* el)

Enforce 'AEL_STATE_INIT' state.

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK

- ESP_FAIL

int **audio_element_get_output_ringbuf_size**(*audio_element_handle_t* el)

Get Element output ringbuffer size.

Parameters **el** – **[in]** The audio element handle

Returns

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

esp_err_t **audio_element_set_output_ringbuf_size**(*audio_element_handle_t* el, int rb_size)

Set Element output ringbuffer size.

Parameters

- **el** – **[in]** The audio element handle
- **rb_size** – **[in]** Size of the ringbuffer

Returns

- ESP_OK
- ESP_FAIL

int **audio_element_multi_input**(*audio_element_handle_t* el, char *buffer, int wanted_size, int index, TickType_t ticks_to_wait)

Call this function to read data from multi input ringbuffer by given index.

Parameters

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

Returns

- ESP_ERR_INVALID_SIZE
- Others are same as the `rb_read`

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

Call this function write data by multi output ringbuffer.

Parameters

- **el** – **[in]** The audio element handle
- **buffer** – The buffer pointer
- **wanted_size** – **[in]** The wanted size
- **ticks_to_wait** – Timeout of ringbuffer

Returns

- The return value is same as the `rb_write`

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

Set multi input ringbuffer Element.

Parameters

- **el** – **[in]** The audio element handle
- **rb** – **[in]** The ringbuffer handle
- **index** – **[in]** Index of multi ringbuffer, starts from 0, should be less than NUMBER_OF_MULTI_RINGBUF

Returns

- ESP_OK
- ESP_FAIL

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

Set multi output ringbuffer Element.

Parameters

- **el** – **[in]** The audio element handle
- **rb** – **[in]** The ringbuffer handle
- **index** – **[in]** Index of multi ringbuffer, starts from 0, should be less than NUMBER_OF_MULTI_RINGBUF

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

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

Get handle of multi input ringbuffer Element by index.

Parameters

- **el** – **[in]** The audio element handle
- **index** – **[in]** Index of multi ringbuffer, starts from 0, should be less than NUMBER_OF_MULTI_RINGBUF

Returns

- NULL Error
- Others ringbuf_handle_t

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

Get handle of multi output ringbuffer Element by index.

Parameters

- **el** – **[in]** The audio element handle
- **index** – **[in]** Index of multi ringbuffer, starts from 0, should be less than NUMBER_OF_MULTI_RINGBUF

Returns

- NULL Error

- Others ringbuf_handle_t

esp_err_t **audio_element_process_init** (*audio_element_handle_t* el)

Provides a way to call element's `open`

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_process_deinit** (*audio_element_handle_t* el)

Provides a way to call element's `close`

Parameters **el** – **[in]** The audio element handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_seek** (*audio_element_handle_t* el, void *in_data, int in_size, void *out_data, int *out_size)

Call element's `seek`

Parameters

- **el** – **[in]** The audio element handle
- **in_data** – **[in]** A pointer to in data
- **in_size** – **[in]** The size of the `in_data`
- **out_data** – **[out]** A pointer to the out data
- **out_size** – **[out]** The size of the `out_data`

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_NOT_SUPPORTED

bool **audio_element_is_stopping** (*audio_element_handle_t* el)

Get Element stopping flag.

Parameters **el** – **[in]** The audio element handle

Returns element's stopping flag

esp_err_t **audio_element_update_byte_pos** (*audio_element_handle_t* el, int pos)

Update the byte position of element information.

Parameters

- **el** – **[in]** The audio element handle
- **pos** – **[in]** The byte_pos accumulated by this value

Returns

- ESP_OK

- ESP_FAIL

esp_err_t **audio_element_set_byte_pos**(*audio_element_handle_t* el, int pos)

Set the byte position of element information.

Parameters

- **el** – **[in]** The audio element handle
- **pos** – **[in]** This value is assigned to byte_pos

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_update_total_bytes**(*audio_element_handle_t* el, int total_bytes)

Update the total bytes of element information.

Parameters

- **el** – **[in]** The audio element handle
- **total_bytes** – **[in]** The total_bytes accumulated by this value

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_total_bytes**(*audio_element_handle_t* el, int total_bytes)

Set the total bytes of element information.

Parameters

- **el** – **[in]** The audio element handle
- **total_bytes** – **[in]** This value is assigned to total_bytes

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_bps**(*audio_element_handle_t* el, int bit_rate)

Set the bps of element information.

Parameters

- **el** – **[in]** The audio element handle
- **bit_rate** – **[in]** This value is assigned to bps

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_codec_fmt**(*audio_element_handle_t* el, int format)

Set the codec format of element information.

Parameters

- **el** – **[in]** The audio element handle

- **format** – **[in]** This value is assigned to codec_fmt

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_music_info**(*audio_element_handle_t* el, int sample_rates, int channels, int bits)

Set the sample_rate, channels, bits of element information.

Parameters

- **el** – **[in]** The audio element handle
- **sample_rates** – **[in]** Sample_rates of music information
- **channels** – **[in]** Channels of music information
- **bits** – **[in]** Bits of music information

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_duration**(*audio_element_handle_t* el, int duration)

Set the duration of element information.

Parameters

- **el** – **[in]** The audio element handle
- **duration** – **[in]** This value is assigned to duration

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_reserve_user0**(*audio_element_handle_t* el, int user_data0)

Set the user_data_0 of element information.

Parameters

- **el** – **[in]** The audio element handle
- **user_data0** – **[in]** This value is assigned to user_data_0

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_reserve_user1**(*audio_element_handle_t* el, int user_data1)

Set the user_data_1 of element information.

Parameters

- **el** – **[in]** The audio element handle
- **user_data1** – **[in]** This value is assigned to user_data_1

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_reserve_user2**(*audio_element_handle_t* el, int user_data2)

Set the user_data_2 of element information.

Parameters

- **el** – **[in]** The audio element handle
- **user_data2** – **[in]** This value is assigned to user_data_2

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_reserve_user3**(*audio_element_handle_t* el, int user_data3)

Set the user_data_3 of element information.

Parameters

- **el** – **[in]** The audio element handle
- **user_data3** – **[in]** This value is assigned to user_data_3

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_element_set_reserve_user4**(*audio_element_handle_t* el, int user_data4)

Set the user_data_4 of element information.

Parameters

- **el** – **[in]** The audio element handle
- **user_data4** – **[in]** This value is assigned to user_data_4

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **audio_element_reserve_data_t**

Audio Element user reserved data.

Public Members

int **user_data_0**
user data 0

int **user_data_1**
user data 1

int **user_data_2**
user data 2

int **user_data_3**
user data 3

int **user_data_4**
user data 4

struct **audio_element_info_t**
Audio Element informations.

Public Members

int **sample_rates**
Sample rates in Hz

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

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

int **bps**
Bit per second

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

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

int **duration**
The duration for an element (optional)

char ***uri**
URI (optional)

esp_codec_type_t **codec_fmt**

Music format (optional)

audio_element_reserve_data_t **reserve_data**

This value is reserved for user use (optional)

struct **audio_element_cfg_t**

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

Public Members

el_io_func **open**

Open callback function

ctrl_func **seek**

Seek callback function

process_func **process**

Process callback function

el_io_func **close**

Close callback function

el_io_func **destroy**

Destroy callback function

stream_func **read**

Read callback function

stream_func **write**

Write callback function

int **buffer_len**

Buffer length use for an Element

int **task_stack**

Element task stack

int **task_prio**

Element task priority (based on freeRTOS priority)

int **task_core**

Element task running in core (0 or 1)

int **out_rb_size**
: Output ringbuffer size

void ***data**
: User context

const char ***tag**
: Element tag

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

int **multi_in_rb_num**
: The number of multiple input ringbuffer

int **multi_out_rb_num**
: The number of multiple output ringbuffer

Macros

AUDIO_ELEMENT_INFO_DEFAULT()

DEFAULT_ELEMENT_RINGBUF_SIZE

DEFAULT_ELEMENT_BUFFER_LENGTH

DEFAULT_ELEMENT_STACK_SIZE

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 (***el_io_func**)(*audio_element_handle_t* self)

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

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

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

typedef esp_err_t (***ctrl_func**)(*audio_element_handle_t* self, void *in_data, int in_size, void *out_data, int *out_size)

Enumerations

enum **audio_element_err_t**

Values:

enumerator **AEL_IO_OK**

enumerator **AEL_IO_FAIL**

enumerator **AEL_IO_DONE**

enumerator **AEL_IO_ABORT**

enumerator **AEL_IO_TIMEOUT**

enumerator **AEL_PROCESS_FAIL**

enum **audio_element_state_t**

Audio element state.

Values:

enumerator **AEL_STATE_NONE**

enumerator **AEL_STATE_INIT**

enumerator **AEL_STATE_INITIALIZING**

enumerator **AEL_STATE_RUNNING**

enumerator **AEL_STATE_PAUSED**

enumerator **AEL_STATE_STOPPED**

enumerator **AEL_STATE_FINISHED**

enumerator **AEL_STATE_ERROR**

enum **audio_element_msg_cmd_t**

Audio element action command, process on dispatcher

Values:

enumerator **AEL_MSG_CMD_NONE**

enumerator **AEL_MSG_CMD_FINISH**

enumerator **AEL_MSG_CMD_STOP**

enumerator **AEL_MSG_CMD_PAUSE**

enumerator **AEL_MSG_CMD_RESUME**

enumerator **AEL_MSG_CMD_DESTROY**

enumerator **AEL_MSG_CMD_REPORT_STATUS**

enumerator **AEL_MSG_CMD_REPORT_MUSIC_INFO**

enumerator **AEL_MSG_CMD_REPORT_CODEC_FMT**

enumerator **AEL_MSG_CMD_REPORT_POSITION**

enum **audio_element_status_t**

Audio element status report

Values:

enumerator **AEL_STATUS_NONE**

enumerator **AEL_STATUS_ERROR_OPEN**

enumerator **AEL_STATUS_ERROR_INPUT**

enumerator **AEL_STATUS_ERROR_PROCESS**

enumerator **AEL_STATUS_ERROR_OUTPUT**

enumerator **AEL_STATUS_ERROR_CLOSE**

enumerator **AEL_STATUS_ERROR_TIMEOUT**

enumerator **AEL_STATUS_ERROR_UNKNOWN**

enumerator **AEL_STATUS_INPUT_DONE**

enumerator **AEL_STATUS_INPUT_BUFFERING**

enumerator **AEL_STATUS_OUTPUT_DONE**

enumerator **AEL_STATUS_OUTPUT_BUFFERING**

enumerator **AEL_STATUS_STATE_RUNNING**

enumerator **AEL_STATUS_STATE_PAUSED**

enumerator **AEL_STATUS_STATE_STOPPED**

enumerator **AEL_STATUS_STATE_FINISHED**

enumerator **AEL_STATUS_MOUNTED**

enumerator **AEL_STATUS_UNMOUNTED**

2.1.2 Audio Pipeline

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

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

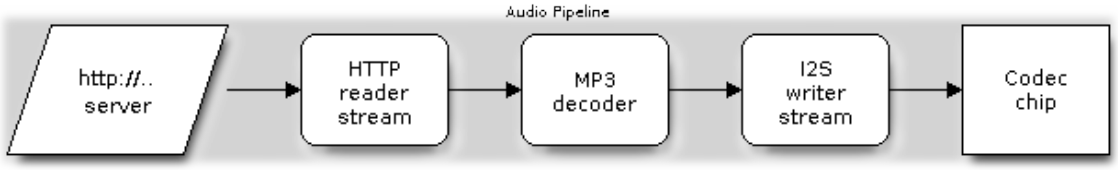


Fig. 3: Sample Organization of Elements in Audio Pipeline

API Reference

Header File

- components/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.

Parameters **config** – The configuration - audio_pipeline_cfg_t

Returns

- audio_pipeline_handle_t on success
- NULL when any errors

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.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns ESP_OK

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

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

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

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **el** – **[in]** The Audio Element Handle
- **name** – **[in]** The name identifier of the audio_element in this audio_pipeline

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **el** – **[in]** The Audio Element Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Start Audio Pipeline.

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

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Stop Audio Pipeline.

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

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

esp_err_t **audio_pipeline_terminate_with_ticks**(*audio_pipeline_handle_t* pipeline, TickType_t ticks_to_wait)

Stop Audio Pipeline with specific ticks for timeout.

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

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **ticks_to_wait** – **[in]** The maximum amount of time to block wait for element destroy

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

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

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

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

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

esp_err_t **audio_pipeline_wait_for_stop_with_ticks**(*audio_pipeline_handle_t* pipeline, TickType_t ticks_to_wait)

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 `ticks_to_wait` until all the Elements in the pipeline actually stop.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle

- **ticks_to_wait** – **[in]** The maximum amount of time to block wait for stop

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **link_tag** – Array of element `name` was registered by `audio_pipeline_register`
- **link_num** – **[in]** Total number of elements of the `link_tag` array

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

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

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Find un-kept element from registered pipeline by tag.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **tag** – **[in]** A char pointer

Returns

- NULL when any errors
- Others on success

audio_element_handle_t **audio_pipeline_get_el_once**(*audio_pipeline_handle_t* pipeline, const *audio_element_handle_t* start_el, const char *tag)

Based on beginning element to find un-kept element from registered pipeline by tag.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **start_el** – **[in]** Specific beginning element
- **tag** – **[in]** A char pointer

Returns

- NULL when any errors
- Others on success

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

Remove event listener from this audio_pipeline.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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`

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **evt** – **[in]** The Event Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

audio_event_iface_handle_t **audio_pipeline_get_event_iface**(*audio_pipeline_handle_t* pipeline)

Get the event iface using by this pipeline.

Parameters **pipeline** – **[in]** The pipeline

Returns The Event Handle

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **first** – **[in]** Previous element is first input element, need to set `true`
- **prev** – **[in]** Previous element
- **conect_rb** – **[in]** Connect ring buffer
- **next** – **[in]** Next element

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **element_1** – **[in]** The element to add to the audio_pipeline.
- ... – **[in]** Additional elements to add to the audio_pipeline.

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **element_1** – **[in]** The element to add to the audio_pipeline.
- ... – **[in]** Additional elements to add to the audio_pipeline.

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **element_1** – **[in]** The element to add to the audio_pipeline.
- ... – **[in]** Additional elements to add to the audio_pipeline.

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **element_1** – **[in]** The element event to subscribe to the audio_pipeline.
- ... – **[in]** Additional elements event to subscribe to the audio_pipeline.

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **dest_el** – **[in]** Destination element
- **status** – **[in]** The new status

Returns

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

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

Reset pipeline element items state to `AEL_STATUS_NONE`

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Reset pipeline element ringbuffer.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

esp_err_t **audio_pipeline_reset_elements**(*audio_pipeline_handle_t* pipeline)

Reset Pipeline linked elements state.

Parameters **pipeline** – **[in]** The Audio Pipeline Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Reset the specific element kept state.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **el** – **[in]** The Audio element Handle

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

Break up all the linked elements of specific `pipeline`. The include and before `kept_ctx_el` working (AEL_STATE_RUNNING or AEL_STATE_PAUSED) elements and connected ringbuffer will be reserved.

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

Parameters

- **pipeline** – **[in]** The audio pipeline handle
- **kept_ctx_el** – **[in]** Destination keep elements

Returns

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

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

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

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

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **link_tag** – Array of elements `name` that was registered by `audio_pipeline_register`
- **link_num** – **[in]** Total number of elements of the `link_tag` array

Returns

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

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

Adds a NULL-terminated list of elements to audio_pipeline.

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

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **element_1** – **[in]** The element to add to the audio_pipeline.
- **...** – **[in]** Additional elements to add to the audio_pipeline.

Returns

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

esp_err_t **audio_pipeline_change_state**(*audio_pipeline_handle_t* pipeline, *audio_element_state_t* new_state)

Set the pipeline state.

Parameters

- **pipeline** – **[in]** The Audio Pipeline Handle
- **new_state** – **[in]** The new state will be set

Returns

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

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_control.

API Reference

Header File

- components/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.

Parameters **config** – The configurations

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **audio_event_iface_destroy**(*audio_event_iface_handle_t* evt)

Cleanup event, it doesn't free evt pointer.

Parameters **evt** – The event

Returns

- ESP_OK
- ESP_FAIL

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`

Parameters

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

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

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

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **evt** – The event
- **wait_time** – **[in]** The wait time

Returns

- ESP_OK
- ESP_FAIL

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

Waiting internal queue message.

Parameters **evt** – The event

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **evt** – The event
- **msg** – The message

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **evt** – **[in]** The event
- **msg** – The message

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **evt** – The event
- **msg** – The message

Returns

- ESP_OK
- ESP_FAIL

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

Discard all ongoing event message.

Parameters **evt** – The event

Returns

- ESP_OK
- ESP_FAIL

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.

Parameters

- **evt** – The event
- **msg** – The message
- **wait_time** – The wait time

Returns

- ESP_OK
- ESP_FAIL

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

Get External queue handle of Emmitter.

Parameters **evt** – **[in]** The external queue

Returns External QueueHandle_t

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.

Parameters

- **evt** – **[in]** The event interface
- **msg** – **[out]** The pointer to structure in which event is to be received
- **wait_time** – **[in]** Timeout for receiving event

Returns

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

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

Get Internal queue handle of Emmitter.

Parameters **evt** – **[in]** The Internal queue

Returns Internal QueueHandle_t

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`

Parameters

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

Returns

- ESP_OK
- ESP_FAIL

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

- components/audio_pipeline/include/audio_common.h

Macros

ELEMENT_SUB_TYPE_OFFSET

mem_assert(x)

Enumerations

enum **audio_element_type_t**

Values:

enumerator **AUDIO_ELEMENT_TYPE_UNKNOW**

enumerator **AUDIO_ELEMENT_TYPE_ELEMENT**

enumerator **AUDIO_ELEMENT_TYPE_PLAYER**

enumerator **AUDIO_ELEMENT_TYPE_SERVICE**

enumerator **AUDIO_ELEMENT_TYPE_PERIPH**

enum **audio_stream_type_t**

Values:

enumerator **AUDIO_STREAM_NONE**

enumerator **AUDIO_STREAM_READER**

enumerator **AUDIO_STREAM_WRITER**

enum **audio_codec_type_t**

Values:

enumerator **AUDIO_CODEC_TYPE_NONE**

enumerator **AUDIO_CODEC_TYPE_DECODER**

enumerator **AUDIO_CODEC_TYPE_ENCODER**

2.1.5 ESP Audio

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

API Reference

Header File

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

Structures

struct **esp_audio_state_t**

esp_audio status information parameters

Public Members

esp_audio_status_t **status**

Status of esp_audio

audio_err_t **err_msg**

Status is `AUDIO_STATUS_ERROR`, err_msg will be setup

media_source_type_t **media_src**

Media source type

Macros

ESP_ERR_AUDIO_BASE

Starting number of ESP audio error codes

Type Definitions

typedef void (***esp_audio_event_callback***)(*esp_audio_state_t* *audio, void *ctx)

typedef esp_err_t (***audio_volume_set***)(void *hd, int vol)

typedef esp_err_t (***audio_volume_get***)(void *hd, int *vol)

Enumerations

enum **audio_err_t**

Values:

enumerator **ESP_ERR_AUDIO_NO_ERROR**

enumerator **ESP_ERR_AUDIO_FAIL**

enumerator **ESP_ERR_AUDIO_NO_INPUT_STREAM**

enumerator **ESP_ERR_AUDIO_NO_OUTPUT_STREAM**

enumerator **ESP_ERR_AUDIO_NO_CODEC**

enumerator **ESP_ERR_AUDIO_HAL_FAIL**

enumerator **ESP_ERR_AUDIO_MEMORY_LACK**

enumerator **ESP_ERR_AUDIO_INVALID_URI**

enumerator **ESP_ERR_AUDIO_INVALID_PATH**

enumerator **ESP_ERR_AUDIO_INVALID_PARAMETER**

enumerator **ESP_ERR_AUDIO_NOT_READY**

enumerator **ESP_ERR_AUDIO_NOT_SUPPORT**

enumerator **ESP_ERR_AUDIO_TIMEOUT**

enumerator **ESP_ERR_AUDIO_ALREADY_EXISTS**

enumerator **ESP_ERR_AUDIO_LINK_FAIL**

enumerator **ESP_ERR_AUDIO_UNKNOWN**

enumerator **ESP_ERR_AUDIO_OUT_OF_RANGE**

enumerator **ESP_ERR_AUDIO_STOP_BY_USER**

enumerator **ESP_ERR_AUDIO_OPEN**

enumerator **ESP_ERR_AUDIO_INPUT**

enumerator **ESP_ERR_AUDIO_PROCESS**

enumerator **ESP_ERR_AUDIO_OUTPUT**

enumerator **ESP_ERR_AUDIO_CLOSE**

enum **esp_audio_status_t**

Values:

enumerator **AUDIO_STATUS_UNKNOWN**

enumerator **AUDIO_STATUS_RUNNING**

enumerator **AUDIO_STATUS_PAUSED**

enumerator **AUDIO_STATUS_STOPPED**

enumerator **AUDIO_STATUS_FINISHED**

enumerator **AUDIO_STATUS_ERROR**

enum **audio_termination_type_t**

Values:

enumerator **TERMINATION_TYPE_NOW**

Audio operation will be terminated immediately

enumerator **TERMINATION_TYPE_DONE**

Audio operation will be stopped when finished

enumerator **TERMINATION_TYPE_MAX**

enum **esp_audio_prefer_t**

Values:

enumerator **ESP_AUDIO_PREFER_MEM**

enumerator **ESP_AUDIO_PREFER_SPEED**

enum **media_source_type_t**

Values:

enumerator **MEDIA_SRC_TYPE_NULL**

enumerator **MEDIA_SRC_TYPE_MUSIC_BASE**

enumerator **MEDIA_SRC_TYPE_MUSIC_SD**

enumerator **MEDIA_SRC_TYPE_MUSIC_HTTP**

enumerator **MEDIA_SRC_TYPE_MUSIC_FLASH**

enumerator **MEDIA_SRC_TYPE_MUSIC_A2DP**

enumerator **MEDIA_SRC_TYPE_MUSIC_DLNA**

enumerator **MEDIA_SRC_TYPE_MUSIC_RAW**

enumerator **MEDIA_SRC_TYPE_MUSIC_MAX**

enumerator **MEDIA_SRC_TYPE_TONE_BASE**

enumerator **MEDIA_SRC_TYPE_TONE_SD**

enumerator **MEDIA_SRC_TYPE_TONE_HTTP**

enumerator **MEDIA_SRC_TYPE_TONE_FLASH**

enumerator **MEDIA_SRC_TYPE_TONE_MAX**

enumerator **MEDIA_SRC_TYPE_RESERVE_BASE**

enumerator **MEDIA_SRC_TYPE_RESERVE_MAX**

Header File

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

Functions

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

Create esp_audio instance according to 'cfg' parameter.

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

Parameters **cfg** – **[in]** Provide esp_audio initialization configuration

Returns

- NULL: Error
- Others: esp_audio instance fully certifying

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

Specific esp_audio instance will be destroyed.

Parameters **handle** – **[in]** The esp_audio instance

Returns

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

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

Add audio input stream to specific esp_audio instance.

Parameters

- **handle** – **[in]** The esp_audio instance
- **in_stream** – **[in]** Audio stream instance

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_MEMORY_LACK: allocate memory fail

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

Add audio output stream to specific esp_audio instance.

Parameters

- **handle** – **[in]** The esp_audio instance
- **out_stream** – **[in]** The audio stream element instance

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_MEMORY_LACK: allocate memory fail

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

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

Parameters

- **handle** – **[in]** The esp_audio instance
- **type** – **[in]** The audio codec type(encoder or decoder)
- **lib** – **[in]** To provide audio stream element

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_MEMORY_LACK: allocate memory fail

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

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

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

Parameters

- **handle** – **[in]** The esp_audio instance
- **type** – **[in]** The CODEC_ENCODER or CODEC_DECODER
- **extension** – **[in]** Such as “mp3”, “wav”, “aac”

Returns

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

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

Play the given uri.

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

Rule of URI field are as follow.

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

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

Now esp_audio_play support follow URIs.

- ”https://dl.espressif.com/dl/audio/mp3_music.mp3”

- ”http://media-ice.musicradio.com/ClassicFMMP3”
- ”file://sdcard/test.mp3”
- ”iis://16000:2@from.pcm/rec.wav#file”
- ”iis://16000:1@record.pcm/record.wav#raw”
- ”aadp://44100:2@bt/sink/stream.pcm”
- ”hfp://8000:1@bt/hfp/stream.pcm”

Note:

- The URI parse by `http_parser_parse_url`,any illegal string will be return `ESP_ERR_AUDIO_INVALID_URI`.
- If the esp_decoder codec is added to `handle`, then the `handle` of esp_decoder will be set as the default decoder, even if other decoders are added.
- Enabled `CONFIG_FATFS_API_ENCODING_UTF_8`, the URI can be support Chinese characters.
- Asynchronous interface
- The maximum of block time can be modify by `esp_audio_play_timeout_set`, default value is 25 seconds.

Parameters

- **`handle`** – The esp_audio_handle_t instance
- **`uri`** – Such as “file://sdcard/test.wav” or “http://iot.espressif.com/file/example.mp3”. If NULL to be set, the uri setup by`esp_audio_setup` will used.
- **`type`** – Specific handle type decoder or encoder
- **`pos`** – Specific starting position by bytes

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_TIMEOUT: timeout the play activity
- ESP_ERR_AUDIO_NOT_SUPPORT: Currently status is AUDIO_STATUS_RUNNING
- ESP_ERR_AUDIO_INVALID_URI: URI is illegal
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_STOP_BY_USER: Exit without play due to esp_audio_stop has been called.

audio_err_t **`esp_audio_sync_play`** (*esp_audio_handle_t* handle, const char *uri, int pos)

Play the given uri until music finished or error occured.

Note:

- All features are same with `esp_audio_play`
- Synchronous interface
- Support decoder mode only

- No any events post during playing

Parameters

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

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_TIMEOUT: timeout the play activity
- ESP_ERR_AUDIO_NOT_SUPPORT: Currently status is AUDIO_STATUS_RUNNING
- ESP_ERR_AUDIO_INVALID_URI: URI is illegal
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

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

A synchronous interface for stop the esp_audio. The maximum of block time is 8000ms.

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

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

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

Parameters

- **handle** – **[in]** The esp_audio instance
- **type** – **[in]** Stop immediately or done

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_NOT_READY: The status is not AUDIO_STATUS_RUNNING or AUDIO_STATUS_PAUSED or element has not created
- ESP_ERR_AUDIO_TIMEOUT: timeout(8000ms) the stop activity.

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

Pause the esp_audio.

Note: 1. Only support music and without live stream. If user queue has been registered by evt_que, AUDIO_STATUS_PAUSED event for success or AUDIO_STATUS_ERROR event for error will be received.

a. The Paused music must be stoped by `esp_audio_stop` before new playing, otherwise got block on new play.

Parameters **handle** – **[in]** The esp_audio instance

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments
- ESP_ERR_AUDIO_NOT_READY: the status is not running
- ESP_ERR_AUDIO_TIMEOUT: timeout the pause activity.

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

Resume the music paused.

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

Parameters **handle** – **[in]** The esp_audio instance

Returns

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

audio_err_t **esp_audio_eq_gain_set** (*esp_audio_handle_t* handle, int band_index, int nch, int eq_gain)

Set the audio gain to be processed by the equalizer.

Parameters

- **handle** – **[in]** The esp_audio instance
- **band_index** – **[in]** The position of center frequencies of equalizer. The range of eq band index is [0 - 9].
- **nch** – **[in]** The number of channel. As for mono, the nch can only set to 1. As for dual, thc nch can set to 1 and 2.
- **eq_gain** – **[in]** The value of audio gain which in `band_index`.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_eq_gain_get** (*esp_audio_handle_t* handle, int band_index, int nch, int *eq_gain)

Get the audio gain to be processed by the equalizer.

Parameters

- **handle** – **[in]** Audio element handle
- **band_index** – **[in]** The position of center frequencies of equalizer. The range of eq band index is [0 - 9].
- **nch** – **[in]** The number of channel. As for mono, the nch can only set to 1. As for dual, thc nch can set to 1 and 2.
- **eq_gain** – **[out]** The pointer of the gain processed by equalizer

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_speed_get** (*esp_audio_handle_t* handle, *esp_audio_play_speed_t* *speed_index)

Getting esp_audio play speed index, index value is from "esp_audio_speed_t" enum.

Parameters

- **handle** – **[in]** The esp_audio instance
- **speed_index** – **[out]** Current audio play speed index.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_speed_set** (*esp_audio_handle_t* handle, *esp_audio_play_speed_t* speed_index)

Use speed_index which is from "esp_audio_speed_t" enum to set esp_audio play speed.

Parameters

- **handle** – **[in]** The esp_audio instance
- **speed_index** – **[in]** Value from "esp_audio_speed_t" enum.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_speed_idx_to_float** (*esp_audio_handle_t* handle, *esp_audio_play_speed_t* speed_index, float *speed)

Use speed_index which is from "esp_audio_speed_t" enum to get esp_audio play speed which is float type.

Parameters

- **handle** – **[in]** The esp_audio instance
- **speed_index** – **[in]** Current audio play speed index.
- **speed** – **[out]** Current audio play speed.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_vol_set** (*esp_audio_handle_t* handle, int vol)

Setting esp_audio volume.

Parameters

- **handle** – **[in]** The esp_audio instance
- **vol** – **[in]** Specific volume will be set. 0-100 is legal. 0 will be mute.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_CTRL_HAL_FAIL: error with hardware.

- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_vol_get** (*esp_audio_handle_t* handle, int *vol)

Get esp_audio volume.

Parameters

- **handle** – **[in]** The esp_audio instance
- **vol** – **[out]** A pointer to int that indicates esp_audio volume.

Returns

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

audio_err_t **esp_audio_state_get** (*esp_audio_handle_t* handle, *esp_audio_state_t* *state)

Get esp_audio status.

Parameters

- **handle** – **[in]** The esp_audio instance
- **state** – **[out]** A pointer to *esp_audio_state_t* that indicates esp_audio status.

Returns

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

audio_err_t **esp_audio_pos_get** (*esp_audio_handle_t* handle, int *pos)

Get the position in bytes of currently played music.

Note: This function works only with decoding music.

Parameters

- **handle** – **[in]** The esp_audio instance
- **pos** – **[out]** A pointer to int that indicates esp_audio decoding position.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_NOT_READY: no codec element

audio_err_t **esp_audio_time_get** (*esp_audio_handle_t* handle, int *time)

Get the position in microseconds of currently played music.

Note: This function works only with decoding music.

Parameters

- **handle** – **[in]** The esp_audio instance
- **time** – **[out]** A pointer to int that indicates esp_audio decoding position.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_NOT_READY: no out stream

audio_err_t **esp_audio_setup**(*esp_audio_handle_t* handle, *esp_audio_setup_t* *sets)

Choose the `in_stream`, `codec` and `out_stream` definitely, and set `uri`.

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

Parameters

- **handle** – **[in]** The esp_audio instance
- **sets** – **[in]** A pointer to *esp_audio_setup_t*.

Returns

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

audio_err_t **esp_audio_media_type_set**(*esp_audio_handle_t* handle, *media_source_type_t* type)

audio_err_t **esp_audio_music_info_get**(*esp_audio_handle_t* handle, *esp_audio_music_info_t* *info)

audio_err_t **esp_audio_info_get**(*esp_audio_handle_t* handle, *esp_audio_info_t* *info)

audio_err_t **esp_audio_info_set**(*esp_audio_handle_t* handle, *esp_audio_info_t* *info)

audio_err_t **esp_audio_callback_set**(*esp_audio_handle_t* handle, *esp_audio_event_callback* cb, void *cb_ctx)

audio_err_t **esp_audio_seek**(*esp_audio_handle_t* handle, int seek_time_sec)

Seek the position in second of currently played music.

Note: This function works only with decoding music.

Parameters

- **handle** – **[in]** The esp_audio instance
- **seek_time_sec** – **[out]** A pointer to int that indicates esp_audio decoding position.

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_FAIL: codec or allocation fail

- ESP_ERR_AUDIO_TIMEOUT: timeout for sync the element status
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance
- ESP_ERR_AUDIO_NOT_SUPPORT: codec has finished
- ESP_ERR_AUDIO_OUT_OF_RANGE: the seek_time_ms is out of the range
- ESP_ERR_AUDIO_NOT_READY: the status is neither running nor paused

audio_err_t **esp_audio_duration_get** (*esp_audio_handle_t* handle, int *duration)

Get the duration in microseconds of playing music.

Note: This function works only with decoding music.

Parameters

- **handle** – **[in]** The esp_audio instance
- **duration** – **[out]** A pointer to int that indicates decoding total time.

Returns

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

audio_err_t **esp_audio_play_timeout_set** (*esp_audio_handle_t* handle, int time_ms)

Setting the maximum amount of time to waiting for `esp_audio_play` only.

Parameters

- **handle** – **[in]** The esp_audio instance
- **time_ms** – **[in]** The maximum amount of time

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: invalid arguments

audio_err_t **esp_audio_prefer_type_get** (*esp_audio_handle_t* handle, *esp_audio_prefer_t* *type)

Get the type of `esp_audio_prefer_t`

Parameters

- **handle** – **[in]** The esp_audio instance
- **type** – **[out]** A pointer to esp_audio_prefer_t

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance

audio_err_t **esp_audio_event_que_set** (*esp_audio_handle_t* handle, QueueHandle_t que)

Set event queue to notify the esp_audio status.

Parameters

- **handle** – **[in]** The esp_audio instance
- **que** – **[out]** A pointer to QueueHandle_t

Returns

- ESP_ERR_AUDIO_NO_ERROR: on success
- ESP_ERR_AUDIO_INVALID_PARAMETER: no esp_audio instance

Structures

struct **esp_audio_cfg_t**

esp_audio configuration parameters

Public Members

int **in_stream_buf_size**

Input buffer size

int **out_stream_buf_size**

Output buffer size

int **resample_rate**

Destination sample rate, 0: disable resample; others: 44.1K, 48K, 32K, 16K, 8K has supported It should be make sure same with I2S stream `sample_rate`

int **component_select**

The select of audio forge component. eg. To choose equalizer and ALC together, please enter ESP_AUDIO_COMPONENT_SELECT_ALC | ESP_AUDIO_COMPONENT_SELECT_EQUALIZER.

QueueHandle_t **evt_que**

For received esp_audio events (optional)

esp_audio_event_callback **cb_func**

esp_audio events callback (optional)

void ***cb_ctx**

esp_audio callback context (optional)

esp_audio_prefer_t **prefer_type**

esp_audio works on sepcific type, default memory is preferred.

- `ESP_AUDIO_PREFER_MEM` mode stopped the previous linked elements before the new pipeline starting, except out stream element.
- `ESP_AUDIO_PREFER_SPEED` mode kept the previous linked elements before the new pipeline starting, except out stream element.

void ***vol_handle***

Volume change instance

audio_volume_set ***vol_set***

Set volume callback

audio_volume_get ***vol_get***

Get volume callback

int ***task_prio***

esp_audio task priority

int ***task_stack***

Size of esp_audio task stack

struct ***esp_audio_setup_t***

esp_audio setup parameters by manual

Public Members

audio_codec_type_t ***set_type***

Set codec type

int ***set_sample_rate***

Set music sample rate

int ***set_channel***

Set music channels

int ***set_pos***

Set starting position

int ***set_time***

Set starting position of the microseconds time (optional)

char ***set_uri***

Set URI

char ***set_in_stream***

Tag of in_stream

char ***set_codec***

Tag of the codec

char *`set_out_stream`
: Tag of out_stream

struct `esp_audio_info_t`
: esp_audio information

Public Members

audio_element_info_t `codec_info`
: Codec information

audio_element_handle_t `in_el`
: Handle of the in stream

audio_element_handle_t `out_el`
: Handle of the out stream

audio_element_handle_t `codec_el`
: Handle of the codec

audio_element_handle_t `filter_el`
: Handle of the filter

esp_audio_state_t `st`
: The state of esp_audio

int `time_pos`
: Position of the microseconds time

float `audio_speed`
: Play speed of audio

int64_t `in_stream_total_size`
: Total size of in stream

struct `esp_audio_music_info_t`
: The music 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 **bps**
Bit per second

esp_codec_type_t **codec_fmt**
Music format

Macros

ESP_AUDIO_COMPONENT_SELECT_DEFAULT
Default selected

ESP_AUDIO_COMPONENT_SELECT_ALC
ALC selected

ESP_AUDIO_COMPONENT_SELECT_EQUALIZER
Equalizer selected

DEFAULT_ESP_AUDIO_CONFIG()

Type Definitions

typedef void ***esp_audio_handle_t**

Enumerations

enum **esp_audio_play_speed_t**
esp_audio play speed
Values:

enumerator **ESP_AUDIO_PLAY_SPEED_UNKNOW**

enumerator **ESP_AUDIO_PLAY_SPEED_0_50**

enumerator **ESP_AUDIO_PLAY_SPEED_0_75**

enumerator **ESP_AUDIO_PLAY_SPEED_1_00**

enumerator **ESP_AUDIO_PLAY_SPEED_1_25**

enumerator **ESP_AUDIO_PLAY_SPEED_1_50**

enumerator **ESP_AUDIO_PLAY_SPEED_1_75**

enumerator **ESP_AUDIO_PLAY_SPEED_2_00**

enumerator **ESP_AUDIO_PLAY_SPEED_MAX**

2.2 Audio Streams

The Audio Stream refers to an *Audio Element* that is responsible for acquiring of audio data and then sending the data out after processing.

The following stream types are supported:

- *Algorithm Stream*
- *FatFs Stream*
- *HTTP Stream*
- *I2S Stream*
- *PWM Stream*
- *Raw Stream*
- *SPIFFS Stream*
- *TCP Client Stream*
- *Tone Stream*
- *Flash-Embedding Stream*
- *TTS Stream*

Each stream is initialized with a structure as an input, and the returned `audio_element_handle_t` handle is used to call the functions in `audio_element.h`. Most streams have two types, `AUDIO_STREAM_READER` (reader) and `AUDIO_STREAM_WRITER` (writer). For example, to set the I2S stream type, use *`i2s_stream_init()`* and *`i2s_stream_cfg_t`*.

See description below for the API details.

2.2.1 Algorithm Stream

The algorithm stream integrates front-end algorithms such as acoustic echo cancellation (AEC), automatic gain control (AGC), and noise suppression (NS) to process the received audio. It is often used in audio preprocessing scenarios, including VoIP, speech recognition, and keyword wake-up. The stream calls esp-sr and thus occupies large memory. The stream only supports the `AUDIO_STREAM_READER` type.

Application Example

- advanced_examples/algorithm/main/algorithm_examples.c
- protocols/voip/main/voip_app.c

Header File

- components/audio_stream/include/algorithm_stream.h

Functions

audio_element_handle_t **algo_stream_init**(*algorithm_stream_cfg_t* *config)

Initialize algorithm stream.

Parameters **config** – The algorithm Stream configuration

Returns The audio element handle

audio_element_err_t **algo_stream_set_delay**(*audio_element_handle_t* el, *ringbuf_handle_t* ringbuf, int delay_ms)

Set playback signal or recording signal delay when use type2.

Note: The AEC internal buffering mechanism requires that the recording signal is delayed by around 0 - 10 ms compared to the corresponding reference (playback) signal.

Parameters

- **el** – Handle of element
- **ringbuf** – Handle of ringbuf
- **delay_ms** – The delay between playback and recording in ms This delay_ms can be debugged by yourself, you can set the configuration debug_input to true, then get the original input data (left channel is the signal captured from the microphone, right channel is the signal played to the speaker), and check the delay with an audio analysis tool.

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **algorithm_mono_fix**(uint8_t *sbuff, uint32_t len)

Fix I2S mono noise issue.

Note: This API only for ESP32 with I2S 16bits

Parameters

- **sbuff** – I2S data buffer
- **len** – I2S data len

Returns ESP_OK

Structures

struct **algorithm_stream_cfg_t**

Algorithm stream configurations.

Public Members

algorithm_stream_input_type_t **input_type**

Input type of stream

int **task_stack**

Task stack size

int **task_prio**

Task peroid

int **task_core**

The core that task to be created

int **out_rb_size**

Size of output ringbuffer

bool **stack_in_ext**

Try to allocate stack in external memory

int **rec_linear_factor**

The linear amplication factor of record signal

int **ref_linear_factor**

The linear amplication factor of reference signal

bool **debug_input**

debug algorithm input data

bool **swap_ch**

Swap left and right channels

int8_t **algo_mask**

Choose algorithm to use

int **sample_rate**

The sampling rate of the input PCM (in Hz)

int **mic_ch**

MIC channel num

int **agc_gain**

AGC gain(dB) for voice communication

bool **aec_low_cost**

AEC uses less cpu and ram resources, but has poor suppression of nonlinear distortion

char ***partition_label**

Partition label which stored the model data

Macros

ALGORITHM_STREAM_PINNED_TO_CORE

ALGORITHM_STREAM_TASK_PERIOD

ALGORITHM_STREAM_RINGBUFFER_SIZE

ALGORITHM_STREAM_TASK_STACK_SIZE

ALGORITHM_STREAM_DEFAULT_SAMPLE_RATE_HZ

ALGORITHM_STREAM_DEFAULT_SAMPLE_BIT

ALGORITHM_STREAM_DEFAULT_MIC_CHANNELS

ALGORITHM_STREAM_DEFAULT_AGC_GAIN_DB

ALGORITHM_STREAM_DEFAULT_MASK

ALGORITHM_STREAM_CFG_DEFAULT( )

Enumerations

enum **algorithm_stream_input_type_t**

Two types of algorithm stream input method.

Values:

enumerator **ALGORITHM_STREAM_INPUT_TYPE1**

Type 1 is default used by mini-board, the reference signal and the recording signal are respectively read in from the left channel and the right channel of the same I2S

enumerator **ALGORITHM_STREAM_INPUT_TYPE2**

As the simple diagram above shows, when type2 is choosen, the recording signal and reference signal should be input by users.

enum **algorithm_stream_mask_t**

Choose the algorithm to be used.

Values:

enumerator **ALGORITHM_STREAM_USE_AEC**

Use AEC

enumerator **ALGORITHM_STREAM_USE_AGC**

Use AGC

enumerator **ALGORITHM_STREAM_USE_NS**

Use NS

enumerator **ALGORITHM_STREAM_USE_VAD**

Use VAD

2.2.2 FatFs Stream

The FatFs stream reads and writes data from FatFs. It has two types: "reader" and "writer". The type is defined by *audio_stream_type_t*.

Application Example

- Reader example: player/pipeline_play_sdcard_music
- Writer example: recorder/pipeline_recording_to_sdcard

Header File

- components/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.

Parameters **config** – The configuration

Returns The Audio Element handle

Structures

struct **fatfs_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**

Stream type

int **buf_sz**

Audio Element Buffer size

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **ext_stack**

Allocate stack on extern ram

bool **write_header**

Choose to write amrnb/amrwb header in fatfs whether or not (true or false, true means choose to write amrnb header)

Macros

FATFS_STREAM_BUF_SIZE

FATFS_STREAM_TASK_STACK

FATFS_STREAM_TASK_CORE

FATFS_STREAM_TASK_PRIO

FATFS_STREAM_RINGBUFFER_SIZE

FATFS_STREAM_CFG_DEFAULT()

2.2.3 HTTP Stream

The HTTP stream obtains and sends data through `esp_http_client()`. The stream has two types: "reader" and "writer", and the type is defined by *`audio_stream_type_t`*. `AUDIO_STREAM_READER` supports HTTP, HTTPS, HTTP Live Stream, and other protocols. Make sure the network is connected before using the stream.

Application Example

- Reader example
  - player/pipeline_living_stream
  - player/pipeline_http_mp3
- Writer example
  - recorder/pipeline_raw_http

Header File

- components/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.

Parameters **config** – The configuration

Returns The Audio Element handle

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

Connect to next track in the playlist.

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

Parameters **el** – The http_stream element handle

Returns

- ESP_OK on success
- ESP_FAIL on errors

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

esp_err_t **http_stream_fetch_again** (*audio_element_handle_t* el)

Try to fetch the tracks again.

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

Parameters **el** – The http_stream element handle

Returns

- ESP_OK on success
- ESP_ERR_NOT_SUPPORTED if playlist is finished

esp_err_t **http_stream_set_server_cert** (*audio_element_handle_t* el, const char *cert)

Set SSL server certification.

Note: EM format as string, if the client requires to verify server

Parameters

- **el** – The http_stream element handle
- **cert** – server certification

Returns

- ESP_OK on success

Structures

struct **http_stream_event_msg_t**

Stream event message.

Public Members

http_stream_event_id_t **event_id**

Event ID

void ***http_client**

Reference to HTTP Client using by this HTTP Stream

void ***buffer**

Reference to Buffer using by the Audio Element

int **buffer_len**

Length of buffer

void ***user_data**

User data context, from *http_stream_cfg_t*

audio_element_handle_t **el**

Audio element context

struct **http_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**

Type of stream

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

http_stream_event_handle_t **event_handle**

The hook function for HTTP Stream

void ***user_data**

User data context

bool **auto_connect_next_track**

connect next track without open/close

bool **enable_playlist_parser**

Enable playlist parser

int **multi_out_num**

The number of multiple output

const char ***cert_pem**

SSL server certification, PEM format as string, if the client requires to verify server

esp_err_t (***crt_bundle_attach**)(void *conf)

Function pointer to esp_crt_bundle_attach. Enables the use of certification bundle for server verification, must be enabled in menuconfig

int **request_size**

Request data size each time from `http_client` Defaults use DEFAULT_ELEMENT_BUFFER_LENGTH if set to 0 Need care this setting if audio frame size is small and want low latency playback

int **request_range_size**

Range size setting for header `Range: bytes=start-end` Request full range of resource if set to 0 Range size bigger than request size is recommended

const char ***user_agent**

The User Agent string to send with HTTP requests

Macros

HTTP_STREAM_TASK_STACK

HTTP_STREAM_TASK_CORE

HTTP_STREAM_TASK_PRIO

HTTP_STREAM_RINGBUFFER_SIZE

HTTP_STREAM_CFG_DEFAULT()

Type Definitions

typedef int (*__http_stream_event_handle_t__)(*http_stream_event_msg_t* *msg)

Enumerations

enum **http_stream_event_id_t**

HTTP Stream hook type.

Values:

enumerator **HTTP_STREAM_PRE_REQUEST**

The event handler will be called before HTTP Client making the connection to the server

enumerator **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)

enumerator **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

enumerator **HTTP_STREAM_POST_REQUEST**

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

enumerator **HTTP_STREAM_FINISH_REQUEST**

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

enumerator **HTTP_STREAM_RESOLVE_ALL_TRACKS**

enumerator **HTTP_STREAM_FINISH_TRACK**

enumerator **HTTP_STREAM_FINISH_PLAYLIST**

2.2.4 I2S Stream

The I2S stream receives and transmits audio data through the chip's I2S, PDM, ADC, and DAC interfaces. To use the ADC and DAC functions, the chip needs to define `SOC_I2S_SUPPORTS_ADC_DAC`. The stream integrates automatic level control (ALC) to adjust volume, multi-channel output, and sending audio data with extended bit width. The relevant control bits are defined in *i2s_stream_cfg_t*.

Application Example

- Reader example: recorder/pipeline_wav_amr_sdcard
- Writer example: get-started/play_mp3_control

Header File

- components/audio_stream/include/i2s_stream.h

Functions

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

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

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

Parameters **config** – The configuration

Returns The Audio Element handle

esp_err_t **i2s_stream_set_channel_type** (*i2s_stream_cfg_t* *config, *i2s_channel_type_t* type)

Set I2S stream channel format type.

Note: : This function only updates *i2s_stream_cfg_t*, so it needs to be called before i2s_stream_init.

Parameters

- **config** – **[in]** The I2S stream configuration
- **type** – **[in]** I2S channel format type

Returns

- ESP_OK
- ESP_ERR_INVALID_ARG

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`

Parameters

- **i2s_stream** – **[in]** The i2s element handle
- **rate** – **[in]** Clock rate (in Hz)
- **bits** – **[in]** Audio bit width (8, 16, 24, 32)
- **ch** – **[in]** Number of Audio channels (1: Mono, 2: Stereo). But when set to tdm mode, ch is slot mask.(ex: I2S_TDM_SLOT0 | I2S_TDM_SLOT1 | I2S_TDM_SLOT2 | I2S_TDM_SLOT3)

Returns

- ESP_OK
- ESP_FAIL

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

Set the volume of input audio stream with ALC. Positive value indicates an increase in volume, negative value indicates a decrease in volume, 0 indicates the volume level remains unchanged.

Parameters

- **i2s_stream** – **[in]** The i2s element handle
- **volume** – **[in]** The gain of input audio stream:
  - Supported range [-64, 63], unit: dB

Returns

- ESP_OK
- ESP_FAIL

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

Get volume of stream.

Parameters

- **i2s_stream** – **[in]** The i2s element handle
- **volume** – **[in]** The volume of stream

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **i2s_stream_sync_delay** (*audio_element_handle_t* i2s_stream, int delay_ms)

Set sync delay of stream.

Parameters

- **i2s_stream** – **[in]** The i2s element handle
- **delay_ms** – **[in]** The delay of stream

Returns

- ESP_OK
- ESP_FAIL

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_comm_mode_t **transmit_mode**

I2S transmit mode

i2s_chan_config_t **chan_cfg**

I2S controller channel configuration

i2s_std_config_t **std_cfg**

I2S standard mode major configuration that including clock/slot/gpio configuration

i2s_data_bit_width_t **expand_src_bits**

The source bits per sample when data expand

bool **use_alc**

It is a flag for ALC. If use ALC, the value is true. Or the value is false

int **volume**

The volume of audio input data will be set.

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

int **multi_out_num**

The number of multiple output

bool **uninstall_drv**

whether uninstall the i2s driver when stream destroyed

bool **need_expand**

whether to expand i2s data

int **buffer_len**

Buffer length use for an Element. Note: when 'bits_per_sample' is 24 bit, the buffer length must be a multiple of 3. The recommended value is 3600

Macros

I2S_STREAM_TASK_STACK

I2S_STREAM_BUF_SIZE

I2S_STREAM_TASK_PRIO

I2S_STREAM_TASK_CORE

I2S_STREAM_RINGBUFFER_SIZE

I2S_STREAM_CFG_DEFAULT()

I2S_STREAM_CFG_DEFAULT_WITH_PARA(port, rate, bits, stream_type)

I2S_STD_PHILIPS_SLOT_DEFAULT_ADF_CONFIG(bits_per_sample, mono_or_stereo)

I2S_STREAM_CFG_DEFAULT_WITH_TYLE_AND_CH(port, rate, bits, stream_type, channel)

Enumerations

enum **i2s_channel_type_t**

Values:

enumerator **I2S_CHANNEL_TYPE_RIGHT_LEFT**

Separated left and right channel

enumerator **I2S_CHANNEL_TYPE_ALL_RIGHT**

Load right channel data in both two channels

enumerator **I2S_CHANNEL_TYPE_ALL_LEFT**

Load left channel data in both two channels

enumerator **I2S_CHANNEL_TYPE_ONLY_RIGHT**

Only load data in right channel (mono mode)

enumerator **I2S_CHANNEL_TYPE_ONLY_LEFT**

Only load data in left channel (mono mode)

2.2.5 PWM Stream

In some cost-sensitive scenarios, the audio signal is not converted by the DAC but is modulated by the PWM (pulse width modulation) and then implemented by a filter circuit. The PWM stream modulates the audio signal with the chip's PWM and sends out the processed audio. It only has the `AUDIO_STREAM_WRITER` type. Note that the digital-to-analog conversion by PWM has a lower signal-to-noise ratio.

Application Example

- Writer example: player/pipeline_play_mp3_with_dac_or_pwm

Header File

- components/audio_stream/include/pwm_stream.h

Functions

audio_element_handle_t **pwm_stream_init** (*pwm_stream_cfg_t* *config)

Initialize PWM stream Only support AUDIO_STREAM_READER type.

Parameters **config** – The PWM Stream configuration

Returns The audio element handle

esp_err_t **pwm_stream_set_clk** (*audio_element_handle_t* pwm_stream, int rate, int bits, int ch)

Setup clock for PWM Stream, this function is only used with handle created by `pwm_stream_init`

Parameters

- **pwm_stream** – **[in]** The pwm element handle
- **rate** – **[in]** Clock rate (in Hz)
- **bits** – **[in]** Audio bit width (16, 32)
- **ch** – **[in]** Number of Audio channels (1: Mono, 2: Stereo)

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **audio_pwm_config_t**

PWM audio configurations.

Public Members

timer_group_t **tg_num**

timer group number (0 - 1)

timer_idx_t **timer_num**

timer number (0 - 1)

int **gpio_num_left**

the LEDC output gpio_num, Left channel

int **gpio_num_right**

the LEDC output gpio_num, Right channel

ledc_channel_t **ledc_channel_left**

LEDC channel (0 - 7), Corresponding to left channel

ledc_channel_t **ledc_channel_right**

LEDC channel (0 - 7), Corresponding to right channel

ledc_timer_t **ledc_timer_sel**

Select the timer source of channel (0 - 3)

ledc_timer_bit_t **duty_resolution**

ledc pwm bits

uint32_t **data_len**

ringbuffer size

struct **pwm_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**
Type of stream

audio_pwm_config_t **pwm_config**
driver configurations

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

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

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

int **buffer_len**
pwm_stream buffer length

bool **ext_stack**
Allocate stack on extern ram

Macros

PWM_STREAM_GPIO_NUM_LEFT

PWM_STREAM_GPIO_NUM_RIGHT

PWM_STREAM_TASK_STACK

PWM_STREAM_BUF_SIZE

PWM_STREAM_TASK_PRIO

PWM_STREAM_TASK_CORE

PWM_STREAM_RINGBUFFER_SIZE

PWM_CONFIG_RINGBUFFER_SIZE

PWM_STREAM_CFG_DEFAULT()

2.2.6 Raw Stream

The raw stream is used to obtain the output data of the previous element of the connection or to provide the data for the next element of the connection. It does not create a thread. For `AUDIO_STREAM_READER`, the connection is [i2s] -> [filter] -> [raw] or [i2s] -> [codec-amr] -> [raw]. For `AUDIO_STREAM_WRITER`, the connection is [raw] ->[codec-mp3]->[i2s].

Application Example

- Reader example: protocols/voip
- Writer example: advanced_examples/downmix_pipeline

Header File

- components/audio_stream/include/raw_stream.h

Functions

audio_element_handle_t **raw_stream_init** (*raw_stream_cfg_t* *cfg)

Initialize RAW stream.

Parameters **cfg** – The RAW Stream configuration

Returns The audio element handle

int **raw_stream_read** (*audio_element_handle_t* pipeline, char *buffer, int buf_size)

Read data from Stream.

Parameters

- **pipeline** – The audio pipeline handle
- **buffer** – The buffer
- **buf_size** – Maximum number of bytes to be read.

Returns Number of bytes actually read.

int **raw_stream_write** (*audio_element_handle_t* pipeline, char *buffer, int buf_size)

Write data to Stream.

Parameters

- **pipeline** – The audio pipeline handle
- **buffer** – The buffer
- **buf_size** – Number of bytes to write

Returns Number of bytes written

Structures

struct **raw_stream_cfg_t**

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

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

Public Members

audio_stream_type_t **type**

Type of stream

int **out_rb_size**

Size of output ringbuffer

Macros

RAW_STREAM_RINGBUFFER_SIZE

RAW_STREAM_CFG_DEFAULT ( )

2.2.7 SPIFFS Stream

The SPIFFS stream reads and writes audio data from or into SPIFFS.

Application Example

- player/pipeline_spiffs_mp3

Header File

- components/audio_stream/include/spiffs_stream.h

Functions

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

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

Parameters **config** – The configuration

Returns The Audio Element handle

Structures

struct **spiffs_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**

Stream type

int **buf_sz**

Audio Element Buffer size

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **write_header**

Choose to write amrnb/armwb header in spiffs whether or not (true or false, true means choose to write amrnb header)

Macros

SPIFFS_STREAM_BUF_SIZE

SPIFFS_STREAM_TASK_STACK

SPIFFS_STREAM_TASK_CORE

SPIFFS_STREAM_TASK_PRIO

SPIFFS_STREAM_RINGBUFFER_SIZE

SPIFFS_STREAM_CFG_DEFAULT()

2.2.8 TCP Client Stream

The TCP client stream reads and writes server data over TCP.

Application Example

- get-started/pipeline_tcp_client

Header File

- components/audio_stream/include/tcp_client_stream.h

Functions

audio_element_handle_t **tcp_stream_init**(*tcp_stream_cfg_t* *config)

Initialize a TCP stream to/from an audio element This function creates a TCP stream to/from an audio element depending on the stream type configuration (e.g., AUDIO_STREAM_READER or AUDIO_STREAM_WRITER). The handle of the audio element is the returned.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **tcp_stream_event_msg**

TCP Stream massage configuration.

Public Members

void *`source`
Element handle

void *`data`
Data of input/output

int `data_len`
Data length of input/output

esp_transport_handle_t `sock_fd`
handle of socket

struct `tcp_stream_cfg_t`
TCP Stream configuration, if any entry is zero then the configuration will be set to default values.

Public Members

audio_stream_type_t `type`
Type of stream

int `timeout_ms`
time timeout for read/write

int `port`
TCP port>

char *`host`
TCP host>

int `task_stack`
Task stack size

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

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

bool `ext_stack`
Allocate stack on extern ram

tcp_stream_event_handle_cb `event_handler`
TCP stream event callback

void ***event_ctx**

User context

Macros

TCP_STREAM_DEFAULT_PORT

TCP stream parameters.

TCP_STREAM_TASK_STACK

TCP_STREAM_BUF_SIZE

TCP_STREAM_TASK_PRIO

TCP_STREAM_TASK_CORE

TCP_SERVER_DEFAULT_RESPONSE_LENGTH

TCP_STREAM_CFG_DEFAULT()

Type Definitions

typedef struct *tcp_stream_event_msg* **tcp_stream_event_msg_t**

TCP Stream massage configuration.

typedef esp_err_t (***tcp_stream_event_handle_cb**)(*tcp_stream_event_msg_t* *msg, *tcp_stream_status_t* state, void *event_ctx)

Enumerations

enum **tcp_stream_status_t**

Values:

enumerator **TCP_STREAM_STATE_NONE**

enumerator **TCP_STREAM_STATE_CONNECTED**

2.2.9 Tone Stream

The tone stream reads the data generated by tools/audio_tone/mk_audio_tone.py. It only supports the `AUDIO_STREAM_READER` type.

Application Example

- player/pipeline_flash_tone

Header File

- components/audio_stream/include/tone_stream.h

Functions

audio_element_handle_t **tone_stream_init** (*tone_stream_cfg_t* *config)

Create an Audio Element handle to stream data from flash to another Element, only support AUDIO_STREAM_READER type.

Parameters **config** – The configuration

Returns The Audio Element handle

Structures

struct **tone_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**

Stream type

int **buf_sz**

Audio Element Buffer size

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

const char ***label**

Label of tone stored in flash. The default value is `flash_tone`

bool **extern_stack**

Task stack allocate on the extern ram

bool **use_delegate**

Read tone partition with esp_delegate. If task stack is on extern ram, this MUST be TRUE

Macros

TONE_STREAM_BUF_SIZE

TONE_STREAM_TASK_STACK

TONE_STREAM_TASK_CORE

TONE_STREAM_TASK_PRIO

TONE_STREAM_RINGBUFFER_SIZE

TONE_STREAM_EXT_STACK

TONE_STREAM_USE_DELEGATE

TONE_STREAM_CFG_DEFAULT()

2.2.10 Flash-Embedding Stream

The flash-embedding stream reads the data generated by tools/audio_tone/mk_embed_flash.py. It only supports the `AUDIO_STREAM_READER` type.

Application Example

- player/pipeline_embed_flash_tone

Header File

- components/audio_stream/include/embed_flash_stream.h

Functions

audio_element_handle_t **embed_flash_stream_init** (*embed_flash_stream_cfg_t* *config)

Create an Audio Element handle to stream data from flash to another Element, only support AUDIO_STREAM_READER type.

Parameters **config** – The configuration

Returns The Audio Element handle

esp_err_t **embed_flash_stream_set_context** (*audio_element_handle_t* embed_stream, const *embed_item_info_t* *context, int max_num)

Set the embed flash context.

```
This function mainly provides information about embed flash data
```

Parameters

- **embed_stream** – **[in]** The embed flash element handle
- **context** – **[in]** The embed flash context
- **max_num** – **[in]** The number of embed flash context

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **embed_flash_stream_cfg_t**

Flash-embedding stream configurations, if any entry is zero then the configuration will be set to default values.

Public Members

int **buf_sz**

Audio Element Buffer size

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **extern_stack**

At present, task stack can only be placed on `SRAM`, so it should always be set to `false`

struct **embed_item_info**

Embed tone information in flash.

Public Members

const uint8_t ***address**

The corresponding address in flash

int **size**

Size of corresponding data

Macros

EMBED_FLASH_STREAM_BUF_SIZE

EMBED_FLASH_STREAM_TASK_STACK

EMBED_FLASH_STREAM_TASK_CORE

EMBED_FLASH_STREAM_TASK_PRIO

EMBED_FLASH_STREAM_RINGBUFFER_SIZE

EMBED_FLASH_STREAM_EXT_STACK

EMBED_FLASH_STREAM_CFG_DEFAULT()

Type Definitions

typedef struct *embed_item_info* **embed_item_info_t**

Embed tone information in flash.

2.2.11 TTS Stream

The tex-to-speech stream (TTS stream) obtains the `esp_tts_voice` data of esp-sr. It only supports the `AUDIO_STREAM_READER` type.

Application Example

- Reader example: player/pipeline_tts_stream

Header File

- components/audio_stream/include/tts_stream.h

Functions

audio_element_handle_t **tts_stream_init** (*tts_stream_cfg_t* *config)

Create a handle to an Audio Element to stream data from TTS to another Element, the stream type only support AUDIO_STREAM_READER for now.

Parameters **config** – The configuration

Returns The Audio Element handle

esp_err_t **tts_stream_set_strings** (*audio_element_handle_t* el, const char *strings)

Set tts stream strings.

Parameters

- **el** – **[in]** The audio element handle
- **strings** – **[in]** The string pointer

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **tts_stream_set_speed** (*audio_element_handle_t* el, *tts_voice_speed_t* speed)

Setting tts stream voice speed.

Parameters

- **el** – **[in]** The esp_audio instance
- **speed** – **[in]** Speed will be set. 0-5 is legal. 0 is the slowest speed.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **tts_stream_get_speed**(*audio_element_handle_t* el, *tts_voice_speed_t* *speed)

Get tts stream voice speed.

Parameters

- **el** – **[in]** The esp_audio instance
- **speed** – **[in]** Return tts stream Speed will be [0,5]

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **tts_stream_cfg_t**

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

Public Members

audio_stream_type_t **type**

Stream type

int **buf_sz**

Audio Element Buffer size

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **ext_stack**

Allocate stack on extern ram

Macros

TTS_STREAM_BUF_SIZE

TTS_STREAM_TASK_STACK

TTS_STREAM_TASK_CORE

TTS_STREAM_TASK_PRIO

TTS_STREAM_RINGBUFFER_SIZE

TTS_STREAM_CFG_DEFAULT()

Enumerations

enum **tts_voice_speed_t**

Values:

enumerator **TTS_VOICE_SPEED_0**

enumerator **TTS_VOICE_SPEED_1**

enumerator **TTS_VOICE_SPEED_2**

enumerator **TTS_VOICE_SPEED_3**

enumerator **TTS_VOICE_SPEED_4**

enumerator **TTS_VOICE_SPEED_5**

enumerator **TTS_VOICE_SPEED_MAX**

2.3 Playlist

A playlist is a list of audio files that can be played back either sequentially or in a specified order.

The *sdcard_scan()* function in the playlist/include/sdcard_scan.h scans the audio files in a microSD card and generate a playlist of files. You can specify file depth and filter out file types when scanning. The playlist instances can be saved to a variety of storage media. The following are the supported storage media:

- *Saving to MicroSD Card*
- *Saving to DRAM*

- *Saving to NVS Partition in Flash*
- *Saving to DATA_UNDEFINED Partition in Flash*

After scanning the audio files, you can use the `playlist_operator_handle_t` handle to call the corresponding functions to create, save, print playlists, and obtain the path corresponding to the audio ID. Currently, most of the storage media mentioned in this document support the above functions.

See the description below for the API details.

2.3.1 Scanning MicroSD Card

The *`sdcard_scan()`* function can scan audio files in a specified path and generate playlists. It supports the scanning of files at a specified depth and filtering of file types. Then, the playlist can be saved to the specified storage medium using a callback function.

Application Example

- player/pipeline_sdcard_mp3_control
- cli

Header File

- components/playlist/include/sdcard_scan.h

Functions

esp_err_t **sdcard_scan**(*sdcard_scan_cb_t* cb, const char *path, int depth, const char *file_extension[], int filter_num, void *user_data)

Scan files in SD card and use callback function to save files that meet filtering conditions.

Note: example sdcard_scan(callback, "/sdcard", 5, const char *[]{"mp3", "aac"}, 2, user_data); Scan 5 levels folder in sdcard and save mp3 files and aac files.

Parameters

- **cb** – The callback function
- **path** – The path to be scanned
- **depth** – The depth of file scanning // .e.g. if you only want to save files in "/test" , depth = 0. // if you want to save files in "/test/scan_test/", depth = 1
- **file_extension** – File extension of files that are supposed to be saved // .e.g. const char *[]{"mp3", "aac"}
- **filter_num** – Number of filters
- **user_data** – The data to be used by callback function

Returns

- ESP_OK success
- ESP_FAIL failed

Type Definitions

typedef void (***sdcard_scan_cb_t**)(void *user_data, char *url)

2.3.2 Saving Playlist

Saving to MicroSD Card

The playlist can be stored in the microSD card. Functions, such as those to save and display the playlist, can be called through the `playlist_operator_handle_t` handle.

Application Example

- player/pipeline_sdcard_mp3_control
- cli

Header File

- components/playlist/include/sdcard_list.h

Functions

esp_err_t **sdcard_list_create**(*playlist_operator_handle_t* *handle)

Create a playlist in sdcard by list id.

Parameters **handle** – **[out]** The playlist handle from application layer

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_show**(*playlist_operator_handle_t* handle)

Show all the URLs in sdcard playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_next**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The following URLs in sdcard playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_prev**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The previous URLs in sdcard playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

bool **sdcard_list_exist**(*playlist_operator_handle_t* handle, const char *url)

Judge whether the url exists in sdcard playlist.

Parameters

- **handle** – Playlist handle
- **url** – The url to be checked

Returns

- true existence
- false Non-existent

esp_err_t **sdcard_list_reset**(*playlist_operator_handle_t* handle)

Reset sdcard playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_current**(*playlist_operator_handle_t* handle, char **url_buff)

Get current URL in sdcard playlist.

Parameters

- **handle** – Playlist handle
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_choose**(*playlist_operator_handle_t* handle, int url_id, char **url_buff)

Choose a url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url in sdcard list
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

int **sdcard_list_get_url_num**(*playlist_operator_handle_t* handle)

Get URLs number in sdcard playlist.

Parameters **handle** – Playlist handle

Returns

- URLs number in sdcard playlist
- ESP_FAIL Fail to get number of urls

int **sdcard_list_get_url_id**(*playlist_operator_handle_t* handle)

Get current url id in the sdcard playlist.

Parameters **handle** – Playlist handle

Returns

- Current url id in partition playlist
- ESP_FAIL Fail to get url id

esp_err_t **sdcard_list_destroy**(*playlist_operator_handle_t* handle)

Destroy sdcard playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **sdcard_list_save**(*playlist_operator_handle_t* handle, const char *url)

Save URL to sdcard playlist.

Parameters

- **handle** – Playlist handle
- **url** – URL to be saved

Returns

- ESP_OK success
- ESP_FAIL failed

Saving to DRAM

The playlist can be stored in DRAM. Functions, such as those to save and display the playlist, can be called through the `playlist_operator_handle_t` handle.

Header File

- components/playlist/include/dram_list.h

Functions

esp_err_t **dram_list_create**(*playlist_operator_handle_t* *handle)

Create a playlist in dram.

Parameters **handle** – **[out]** The playlist handle from application layer

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_save**(*playlist_operator_handle_t* handle, const char *url)

Save URL to dram playlist.

Parameters

- **handle** – Playlist handle
- **url** – URL to be saved

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_next**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The following URLs in dram playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_prev**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The previous URLs in dram playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL

- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

bool **dram_list_exist**(*playlist_operator_handle_t* handle, const char *url)

Judge whether the url exists in dram playlist.

Parameters

- **handle** – Playlist handle
- **url** – The url to be checked

Returns

- true existence
- false Non-existent

esp_err_t **dram_list_reset**(*playlist_operator_handle_t* handle)

Reset dram playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_current**(*playlist_operator_handle_t* handle, char **url_buff)

The current URL in current playlist.

Parameters

- **handle** – Playlist handle
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_choose**(*playlist_operator_handle_t* handle, int url_id, char **url_buff)

Choose a url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url in dram list
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

int **dram_list_get_url_num**(*playlist_operator_handle_t* handle)

Get URLs number in the dram playlist.

Parameters **handle** – Playlist handle

Returns

- URLs number in dram playlist
- ESP_FAIL Fail to get number of urls

int **dram_list_get_url_id**(*playlist_operator_handle_t* handle)

Get current url id in the dram playlist.

Parameters **handle** – Playlist handle

Returns

- Current url id in dram playlist
- ESP_FAIL Fail to get url id

esp_err_t **dram_list_show**(*playlist_operator_handle_t* handle)

Show all the URLs in the dram playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_remove_by_url**(*playlist_operator_handle_t* handle, const char *url)

Remove corrsponding url in dram list.

Parameters

- **handle** – Playlist handle
- **url** – The url to be removed

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_remove_by_url_id**(*playlist_operator_handle_t* handle, uint16_t url_id)

Remove url by id.

Parameters

- **handle** – Playlist handle
- **url_id** – The url id to be removed

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **dram_list_destroy**(*playlist_operator_handle_t* handle)

Destroy the dram playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

Saving to NVS Partition in Flash

The playlist can be stored in the NVS partition in flash. Functions, such as those to save and display the playlist, can be called through the `playlist_operator_handle_t` handle.

Header File

- components/playlist/include/flash_list.h

Functions

esp_err_t **flash_list_create**(*playlist_operator_handle_t* *handle)

Create a playlist in nvs flash.

Parameters **handle** – **[out]** Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_save**(*playlist_operator_handle_t* handle, const char *url)

Save URL to nvs flash list.

Parameters

- **handle** – Playlist handle
- **url** – URL to be saved

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_show**(*playlist_operator_handle_t* handle)

Show all the URLs in nvs flash list.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_next**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The following URLs in nvs flash playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL

- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_prev** (*playlist_operator_handle_t* handle, int step, char **url_buff)

The previous URLs in nvs flash playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_current** (*playlist_operator_handle_t* handle, char **url_buff)

The current URL in nvs flash playlist.

Parameters

- **handle** – Playlist handle
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

bool **flash_list_exist** (*playlist_operator_handle_t* handle, const char *url)

Judge whether the url exists in flash playlist.

Parameters

- **handle** – Playlist handle
- **url** – The url to be checked

Returns

- true existence
- false Non-existent

esp_err_t **flash_list_reset** (*playlist_operator_handle_t* handle)

Reset flash playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **flash_list_choose**(*playlist_operator_handle_t* handle, int url_id, char **url_buff)

Choose a url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url in flash list
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

int **flash_list_get_url_num**(*playlist_operator_handle_t* handle)

Get URLs number in the flash playlist.

Parameters **handle** – Playlist handle

Returns

- URLs number in flash playlist
- ESP_FAIL Fail to get number of urls

int **flash_list_get_url_id**(*playlist_operator_handle_t* handle)

Get current url id in the flash playlist.

Parameters **handle** – Playlist handle

Returns

- Curernt url id in flash playlist
- ESP_FAIL Fail to get url id

esp_err_t **flash_list_destroy**(*playlist_operator_handle_t* handle)

Destroy the nvs flash playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

Saving to DATA_UNDEFINED Partition in Flash

The playlist can be stored in the `DATA_UNDEFINED` partition (see Partition Tables for details) in flash. Functions, such as those to save and display the playlist, can be called through the `playlist_operator_handle_t` handle. Please add the two partitions whose subtypes are 0x06 and 0x07 to the flash partition table first.

Header File

- components/playlist/include/partition_list.h

Functions

esp_err_t **partition_list_create**(*playlist_operator_handle_t* *handle)

Create a playlist in flash partition by list id.

Note: Please add 2 partitions to partition table whose subtype are 0x06 and 0x07 first

---

Parameters **handle** – **[out]** The playlist handle from application layer

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_save**(*playlist_operator_handle_t* handle, const char *url)

Save URL to partition playlist.

Parameters

- **handle** – Playlist handle
- **url** – URL to be saved

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_next**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The following URLs in partition playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_prev**(*playlist_operator_handle_t* handle, int step, char **url_buff)

The previous URLs in partition playlist.

Parameters

- **handle** – Playlist handle
- **step** – The offset of URL from current URL
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

bool **partition_list_exist**(*playlist_operator_handle_t* handle, const char *url)

Judge whether the url exists in partition playlist.

Parameters

- **handle** – Playlist handle
- **url** – The url to be checked

Returns

- true existence
- false Non-existent

esp_err_t **partition_list_reset**(*playlist_operator_handle_t* handle)

Reset partition playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_current**(*playlist_operator_handle_t* handle, char **url_buff)

Get current URL in the partition playlist.

Parameters

- **handle** – Playlist handle
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_choose**(*playlist_operator_handle_t* handle, int url_id, char **url_buff)

Choose a url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url in partition list
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

int **partition_list_get_url_num**(*playlist_operator_handle_t* handle)

Get URLs number in the partition playlist.

Parameters **handle** – Playlist handle

Returns

- URLs number in partition playlist
- ESP_FAIL Fail to get number of urls

int **partition_list_get_url_id**(*playlist_operator_handle_t* handle)

Get curernt url id in the partition playlist.

Parameters **handle** – Playlist handle

Returns

- Current url id in partition playlist
- ESP_FAIL Fail to get url id

esp_err_t **partition_list_show**(*playlist_operator_handle_t* handle)

Show all the URLs in the partition playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **partition_list_destroy**(*playlist_operator_handle_t* handle)

Destroy the partition playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

2.3.3 Playlist Manager

Playlist Manager manages the above playlists and can add multiple playlist instances to the `playlist_handle_t` handle.

Header File

- components/playlist/include/playlist.h

Functions

playlist_handle_t **playlist_create**(void)

Create a playlist manager handle.

Returns

- playlist handle success
- NULL failed

esp_err_t **playlist_add**(*playlist_handle_t* handle, *playlist_operator_handle_t* list_handle, uint8_t list_id)

Create a playlist manager and add playlist handle to it.

Note: The partition playlist can only be added once, or it will be overwrited by the newest partiiton playlist

---

Note: Different lists must use different IDs, because even if they are in different handles, list_id is the only indicator that distinguishes them.

Parameters

- **handle** – Playlist manager handle
- **list_handle** – The playlist handle to be added
- **list_id** – The playlist id to be registered

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_checkout_by_id**(*playlist_handle_t* handle, uint8_t id)

Playlist checkout by list id.

Parameters

- **handle** – Playlist handle
- **id** – Specified list id

Returns

- ESP_OK success
- ESP_FAIL failed

int **playlist_get_list_num**(*playlist_handle_t* handle)

Get number of playlists in the handle.

Parameters **handle** – Playlist handle

Returns

- success Number of playlists in handle
- failed -1

playlist_type_t **playlist_get_current_list_type**(*playlist_handle_t* handle)

Get current playlist type.

Parameters **handle** – Playlist handle

Returns

- success Type of current playlist
- failed -1

int **playlist_get_current_list_id**(*playlist_handle_t* handle)

Get current playlist id.

Parameters **handle** – Playlist handle

Returns

- success Current playlist id
- failed -1

esp_err_t **playlist_get_current_list_url**(*playlist_handle_t* handle, char **url_buff)

Get current URL in current playlist.

Parameters

- **handle** – Playlist handle
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

int **playlist_get_current_list_url_num**(*playlist_handle_t* handle)

Get number of URLs in current playlist.

Parameters **handle** – Playlist handle

Returns Number of URLS in current playlsit

int **playlist_get_current_list_url_id**(*playlist_handle_t* handle)

Get current url id in current playlist.

Parameters **handle** – Playlist handle

Returns Current url's id in current playlsit

esp_err_t **playlist_save**(*playlist_handle_t* handle, const char *url)

Save a URL to the current playlist.

Parameters

- **handle** – Playlist handle
- **url** – The URL to be saved ot sdcard

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_next**(*playlist_handle_t* handle, int step, char **url_buff)

Next URl in current playlist.

Parameters

- **handle** – Playlist handle
- **step** – Next steps from current position
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_prev**(*playlist_handle_t* handle, int step, char **url_buff)

Previous URL in current playlist.

Parameters

- **handle** – Playlist handle
- **step** – Previous steps from current position
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_choose**(*playlist_handle_t* handle, int url_id, char **url_buff)

Choose a url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url in current list
- **url_buff** – **[out]** A second rank pointer to get a address of URL

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_show**(*playlist_handle_t* handle)

Show URLs in current playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_reset**(*playlist_handle_t* handle)

Reset current playlist.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_remove_by_url**(*playlist_handle_t* handle, const char *url)

Remove corresponding url.

Parameters

- **handle** – Playlist handle
- **url** – The url to be removed

Returns

- ESP_OK success
- ESP_FAIL failed

esp_err_t **playlist_remove_by_url_id**(*playlist_handle_t* handle, uint16_t url_id)

Remove url by url id.

Parameters

- **handle** – Playlist handle
- **url_id** – The id of url to be removed

Returns

- ESP_OK success
- ESP_FAIL failed

bool **playlist_exist**(*playlist_handle_t* handle, const char *url)

Judge whether the url exists in current playlist.

Parameters

- **handle** – Playlist handle
- **url** – The url to be checked

Returns

- true existence
- false Non-existent

esp_err_t **playlist_destroy**(*playlist_handle_t* handle)

Destroy all playlists in the handle.

Parameters **handle** – Playlist handle

Returns

- ESP_OK success
- ESP_FAIL failed

Structures

struct **playlist_operation_t**

All types of Playlists' operation.

Public Members

esp_err_t (*__show__)(void *playlist)
: Show all the URLs in playlist

esp_err_t (*__save__)(void *playlist, const char *url)
: Save URLs to playlist

esp_err_t (*__next__)(void *playlist, int step, char **url_buff)
: Get next URL in playlist

esp_err_t (*__prev__)(void *playlist, int step, char **url_buff)
: Get previous URL in playlist

esp_err_t (*__reset__)(void *playlist)
: Reset the playlist

esp_err_t (*__choose__)(void *playlist, int url_id, char **url_buff)
: Get url by url id

esp_err_t (*__current__)(void *playlist, char **url_buff)
: Get current URL in playlist

esp_err_t (*__destroy__)(void *playlist)
: Destroy playlist

bool (*__exist__)(void *playlist, const char *url)
: Judge whether the url exists

int (*__get_url_num__)(void *playlist)
: Get number of URLS in current playlist

int (*__get_url_id__)(void *playlist)
: Get current url id in playlist

playlist_type_t __type__
: Type of playlist

esp_err_t (*__remove_by_url__)(void *playlist, const char *url)
: Remove the corresponding url

esp_err_t (*__remove_by_id__)(void *playlist, uint16_t url_id)
: Remove url by id

struct __playlist_operator_t__
: Information of playlist manager node.

Public Members

void ***playlist**

Specific playlist's pointer

esp_err_t (***get_operation**)(*playlist_operation_t* *operation)

Function pointer to get playlists' handle

Type Definitions

typedef *playlist_operator_t* ***playlist_operator_handle_t**

typedef struct playlist_handle ***playlist_handle_t**

Enumerations

enum **playlist_type_t**

Type of playlist.

Values:

enumerator **PLAYLIST_UNKNOWN**

Unknown type

enumerator **PLAYLIST_SDCARD**

Playlist in sdcard

enumerator **PLAYLIST_FLASH**

Playlist in nvs

enumerator **PLAYLIST_DRAM**

Playlist in ram

enumerator **PLAYLIST_PARTITION**

Playlist in partition

2.4 Codecs

2.4.1 AAC Decoder

Decode an audio data stream provided in AAC format.

API Reference

Header File

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

Functions

audio_element_handle_t **aac_decoder_init** (*aac_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming AAC data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **aac_decoder_cfg_t**

AAC Decoder configuration.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

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

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

bool **plus_enable**

Dynamically enable HE-AAC (v1 v2) decoding

Macros

AAC_DECODER_TASK_STACK_SIZE

AAC_DECODER_TASK_CORE

AAC_DECODER_TASK_PRIO

AAC_DECODER_RINGBUFFER_SIZE

DEFAULT_AAC_DECODER_CONFIG()

2.4.2 AMR Decoder and Encoder

Decode and encode an audio data stream from / to AMR format. Encoders cover both AMR-NB and AMR-WB formats.

Application Examples

Implementation of this API is demonstrated in the following examples:

- player/pipeline_play_sdcard_music
- recorder/pipeline_wav_amr_sdcard

API Reference - Decoder

Header File

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

Functions

audio_element_handle_t **amr_decoder_init**(*amr_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming AMR data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **amr_decoder_cfg_t**

AMR Decoder configuration.

Public Members

int **out_rb_size**
Size of output ringbuffer

int **task_stack**
Task stack size

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

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

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

Macros

AMR_DECODER_TASK_STACK_SIZE

AMR_DECODER_TASK_CORE

AMR_DECODER_TASK_PRIO

AMR_DECODER_RINGBUFFER_SIZE

DEFAULT_AMR_DECODER_CONFIG()

API Reference - AMR-NB Encoder

Header File

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

Functions

esp_err_t **amrnb_encoder_set_bitrate**(*audio_element_handle_t* self, *amrnb_encoder_bitrate_t* bitrate_mode)

Set AMRNB encoder bitrate.

Parameters

- **self** – Audio element handle
- **bitrate_mode** – Bitrate choose, value from amrnb_encoder_bitrate_t

Returns

ESP_OK ESP_FAIL

audio_element_handle_t **amrnb_encoder_init** (*amrnb_encoder_cfg_t* *config)

Create an Audio Element handle to encode incoming AMRNB data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **amrnb_encoder_cfg_t**

AMRNB Encoder configurations.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

amrnb_encoder_bitrate_t **bitrate_mode**

AMRNB Encoder bitrate choose

bool **contain_amrnb_header**

Choose to contain amrnb header in amrnb encoder whether or not (true or false, true means choose to contain amrnb header)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

AMRNB_ENCODER_TASK_STACK

AMRNB_ENCODER_TASK_CORE

AMRNB_ENCODER_TASK_PRIO

AMRNB_ENCODER_RINGBUFFER_SIZE

DEFAULT_AMRNB_ENCODER_CONFIG()

Enumerations

enum **amrnb_encoder_bitrate_t**

Enum of AMRNB Encoder bitrate choose.

Values:

enumerator **AMRNB_ENC_BITRATE_UNKNOW**

Invalid mode

enumerator **AMRNB_ENC_BITRATE_MR475**

enumerator **AMRNB_ENC_BITRATE_MR515**

enumerator **AMRNB_ENC_BITRATE_MR59**

enumerator **AMRNB_ENC_BITRATE_MR67**

enumerator **AMRNB_ENC_BITRATE_MR74**

enumerator **AMRNB_ENC_BITRATE_MR795**

enumerator **AMRNB_ENC_BITRATE_MR102**

enumerator **AMRNB_ENC_BITRATE_MR122**

enumerator **AMRNB_ENC_BITRATE_MRDTX**

enumerator **AMRNB_ENC_BITRATE_N_MODES**

API Reference - AMR-WB Encoder

Header File

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

Functions

esp_err_t **amrwb_encoder_set_bitrate**(*audio_element_handle_t* self, *amrwb_encoder_bitrate_t* bitrate_mode)

Set AMRWB encoder bitrate.

Parameters

- **self** – Audio element handle
- **bitrate_mode** – Bitrate choose, value from amrwb_encoder_bitrate_t

Returns

ESP_OK ESP_FAIL

audio_element_handle_t **amrwb_encoder_init**(*amrwb_encoder_cfg_t* *config)

Create an Audio Element handle to encode incoming amrwb data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **amrwb_encoder_cfg_t**

AMRWB Encoder configurations.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

amrwb_encoder_bitrate_t **bitrate_mode**

AMRWB Encoder bitrate choose

bool **contain_amrwb_header**

Choose to contain amrwb header in amrwb encoder whether or not (true or false, true means choose to contain amrwb header)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

AMRWB_ENCODER_TASK_STACK

AMRWB_ENCODER_TASK_CORE

AMRWB_ENCODER_TASK_PRIO

AMRWB_ENCODER_RINGBUFFER_SIZE

DEFAULT_AMRWB_ENCODER_CONFIG()

Enumerations

enum **amrwb_encoder_bitrate_t**

Enum of AMRWB Encoder bitrate choose.

Values:

enumerator **AMRWB_ENC_BITRATE_MDNONE**

Invalid mode

enumerator **AMRWB_ENC_BITRATE_MD66**

6.60kbps

enumerator **AMRWB_ENC_BITRATE_MD885**

8.85kbps

enumerator **AMRWB_ENC_BITRATE_MD1265**

12.65kbps

enumerator **AMRWB_ENC_BITRATE_MD1425**

14.25kbps

enumerator **AMRWB_ENC_BITRATE_MD1585**

15.85bps

enumerator **AMRWB_ENC_BITRATE_MD1825**
18.25bps

enumerator **AMRWB_ENC_BITRATE_MD1985**
19.85kbps

enumerator **AMRWB_ENC_BITRATE_MD2305**
23.05kbps

enumerator **AMRWB_ENC_BITRATE_MD2385**
23.85kbps>

enumerator **AMRWB_ENC_BITRATE_N_MODES**
Invalid mode

2.4.3 FLAC Decoder

Decode an audio data stream provided in FLAC format.

API Reference

Header File

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

Functions

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

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **flac_decoder_cfg_t**
FLAC Decoder configuration.

Public Members

int **out_rb_size**
: Size of output ringbuffer

int **task_stack**
: Task stack size

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

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

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

Macros

FLAC_DECODER_TASK_STACK_SIZE

FLAC_DECODER_TASK_CORE

FLAC_DECODER_TASK_PRIO

FLAC_DECODER_RINGBUFFER_SIZE

DEFAULT_FLAC_DECODER_CONFIG()

2.4.4 MP3 Decoder

Decode an audio data stream provided in MP3 format.

Application Examples

Implementation of this API is demonstrated in the following examples:

- get-started/play_mp3_control
- player/pipeline_play_sdcard_music
- player/pipeline_play_mp3_with_dac_or_pwm

API Reference

Header File

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

Functions

audio_element_handle_t **mp3_decoder_init** (*mp3_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming MP3 data.

Parameters **config** – The configuration

Returns The audio element handle

const esp_id3_info_t ***mp3_decoder_get_id3_info** (*audio_element_handle_t* self)

Get ID3 information.

Parameters **self** – The audio element handle

Returns esp_id3_info_t: success NULL: ID3 is not exist or memory aloocation failed.

Structures

struct **mp3_decoder_cfg_t**

Mp3 Decoder configuration.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

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

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

bool **id3_parse_enable**

True: parse ID3. False: Don't parse ID3

Macros

MP3_DECODER_TASK_STACK_SIZE

MP3_DECODER_TASK_CORE

MP3_DECODER_TASK_PRIO

MP3_DECODER_RINGBUFFER_SIZE

DEFAULT_MP3_DECODER_CONFIG()

2.4.5 OGG Decoder

Decode an audio data stream provided in OGG format.

API Reference

Header File

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

Functions

audio_element_handle_t **ogg_decoder_init**(*ogg_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming OGG data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **ogg_decoder_cfg_t**

OGG Decoder configuration.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

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

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

OGG_DECODER_TASK_STACK_SIZE

OGG_DECODER_TASK_CORE

OGG_DECODER_TASK_PRIO

OGG_DECODER_RINGBUFFER_SIZE

DEFAULT_OGG_DECODER_CONFIG()

2.4.6 OPUS Decoder

Decode an audio data stream provided in OPUS format.

API Reference

Header File

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

Functions

audio_element_handle_t **decoder_opus_init**(*opus_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming OPUS data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **opus_decoder_cfg_t**

OPUS Decoder configuration.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

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

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

OPUS_DECODER_TASK_STACK_SIZE

OPUS_DECODER_TASK_CORE

OPUS_DECODER_TASK_PRIO

OPUS_DECODER_RINGBUFFER_SIZE

DEFAULT_OPUS_DECODER_CONFIG()

2.4.7 WAV Decoder and Encoder

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

Application Examples

Implementation of this API is demonstrated in the following examples:

- player/pipeline_play_sdcard_music
- recorder/pipeline_wav_amr_sdcard

API Reference - Decoder

Header File

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

Functions

audio_element_handle_t **wav_decoder_init** (*wav_decoder_cfg_t* *config)

Create an Audio Element handle to decode incoming WAV data.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **wav_decoder_cfg_t**

brief WAV Decoder configurations

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

WAV_DECODER_TASK_STACK

WAV_DECODER_TASK_CORE

WAV_DECODER_TASK_PRIO

WAV_DECODER_RINGBUFFER_SIZE

DEFAULT_WAV_DECODER_CONFIG()

API Reference - Encoder

Header File

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

Functions

audio_element_handle_t **wav_encoder_init**(*wav_encoder_cfg_t* *config)

Create a handle to an Audio Element to encode incoming data using WAV format.

Parameters **config** – The configuration

Returns The audio element handle

Structures

struct **wav_encoder_cfg_t**

WAV Encoder configurations.

Public Members

int **out_rb_size**

Size of output ringbuffer

int **task_stack**

Task stack size

int **task_core**

Task running in core (0 or 1)

int **task_prio**

Task priority (based on freeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

WAV_ENCODER_TASK_STACK

WAV_ENCODER_TASK_CORE

WAV_ENCODER_TASK_PRIO

WAV_ENCODER_RINGBUFFER_SIZE

DEFAULT_WAV_ENCODER_CONFIG()

2.5 Audio Processing

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

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

Please refer to description of respective APIs below.

2.5.1 Downmix

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

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

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

The downmix process has 3 states:

- Bypass Downmixing – Only the base audio file will be processed;
- Switch on Downmixing – The base audio file and the target audio file will first enter the transition period, during which the gains of these two files will be changed from the original level to the target level; then enter the stable period, sharing a same target gain;

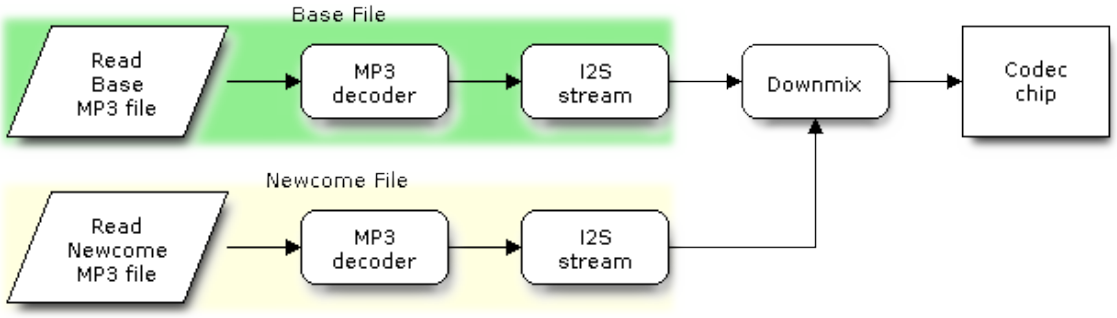


Fig. 4: Illustration of Downmixing Process

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

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

Application Example

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

API Reference

Header File

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

Functions

void **downmix_set_input_rb_timeout** (*audio_element_handle_t* self, int ticks_to_wait, int index)

Sets the downmix timeout.

Parameters

- **self** – Audio element handle
- **ticks_to_wait** – Input ringbuffer timeout
- **index** – The index of multi input ringbuffer

void **downmix_set_input_rb** (*audio_element_handle_t* self, *ringbuf_handle_t* rb, int index)

Sets the downmix input ringbuffer. refer to `ringbuf.h`

Parameters

- **self** – Audio element handle
- **rb** – Handle of ringbuffer
- **index** – The index of multi input ringbuffer.

esp_err_t **downmix_set_output_type** (*audio_element_handle_t* self, esp_downmix_output_type_t output_type)

Set channel mode of output data. Only supported mono and dual.

Parameters

- **self** – Audio element handle
- **output_type** – Down-mixer output type.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **downmix_set_work_mode** (*audio_element_handle_t* self, esp_downmix_work_mode_t mode)

Sets BYPASS, ON or OFF status of down-mixer.

Parameters

- **self** – Audio element handle
- **mode** – Down-mixer work mode.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **downmix_set_out_ctx_info** (*audio_element_handle_t* self, esp_downmix_out_ctx_type_t out_ctx)

Passes content of per channel output stream by down-mixer.

Parameters

- **self** – Audio element handle
- **out_ctx** – Content of output stream.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **downmix_set_source_stream_info** (*audio_element_handle_t* self, int rate, int ch, int index)

Sets the sample rate and the number of channels of input stream to be processed.

Parameters

- **self** – Audio element handle
- **rate** – Sample rate of the input stream
- **ch** – Channel number of the input stream. Only supported mono and dual.
- **index** – The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **downmix_set_gain_info** (*audio_element_handle_t* self, float *gain, int index)

Sets the audio gain to be processed.

Parameters

- **self** – Audio element handle
- **gain** – The gain array of downmix which the array size is 2 and the gain data range is [-100, 100], unit: dB.
- **index** – The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **downmix_set_transit_time_info** (*audio_element_handle_t* self, int transit_time, int index)

Sets the audio `transit_time` to be processed.

Parameters

- **self** – Audio element handle
- **transit_time** – The reset value of `transit_time`
- **index** – The index of input stream. The index must be in [0, SOURCE_NUM_MAX - 1] range

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **source_info_init** (*audio_element_handle_t* self, esp_downmix_input_info_t *source_info)

Initializes information of the source streams for downmixing.

Parameters

- **self** – Audio element handle
- **source_info** – The information array of source streams

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

audio_element_handle_t **downmix_init** (*downmix_cfg_t* *config)

Initializes the Audio Element handle for downmixing.

Parameters **config** – the configuration

Returns The audio element handler

Structures

struct **downmix_cfg_t**

Downmix configuration.

Public Members

esp_downmix_info_t **downmix_info**

Downmix information

int **max_sample**

The number of samples per downmix processing

int **out_rb_size**

Size of ring buffer

int **task_stack**

Size of task stack

int **task_core**

Task running in core…

int **task_prio**

Task priority (based on the FreeRTOS priority)

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

DOWNMIX_TASK_STACK

DOWNMIX_TASK_CORE

DOWNMIX_TASK_PRIO

DOWNMIX_RINGBUFFER_SIZE

DM_BUF_SIZE

DEFAULT_DOWNMIX_CONFIG()

2.5.2 Equalizer

Provided in this API equalizer supports:

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

The center frequencies of bands are shown in table below.

Band Index	0	1	2	3	4	5	6	7	8	9
Frequency	31 Hz	62 Hz	125 Hz	250 Hz	500 Hz	1 kHz	2 kHz	4 kHz	8 kHz	16 kHz

Default gain of each band is -13 dB. To set the gains of all bands use structure `equalizer_cfg`. To set the gain of individual band use function `equalizer_set_gain_info()`.

Application Example

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

API Reference

Header File

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

Functions

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

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

Parameters

- **self** – Audio element handle
- **rate** – Audio sample rate
- **ch** – Audio channel

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

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

Set the audio gain to be processed by the equalizer.

Parameters

- **self** – Audio element handle
- **index** – The position of center frequencies of equalizer. If channel is mono, the index range is [0, 9]; If channel is stereo and `is_channels_gain_equal` is true, the index range is [0, 9]; If channel is stereo and `is_channels_gain_equal` is false, the index range is [0, 19];

- **`value_gain`** – The value of audio gain which in `index`
- **`is_channels_gain_equal`** – If audio channel is stereo, the audio gain values of two channels are equal when `is_channels_gain_equal` is `true`, otherwise it means unequal.

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

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

Create an Audio Element handle that equalizes incoming data.

Parameters **`config`** – The configuration

Returns The audio element handler

Structures

struct **`equalizer_cfg`**

Equalizer Configuration.

+——+ | MONO | +———+ | Only Left channel/Only Right channel | +——+ |band0|band1|band2|band3|band4|band5|band6|band7|band8|band9| +———+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | +——+

+——+———+ | DUAL | +——+———+ | Left channel | Right channel | +———+——+ |band0|band1|band2|band3|band4|band5|band6|band7|band8|band9|band0|band1|band2|band3|band4|band5|band6|band7|band8|band +——+——+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | +———+——+

3) Different sample rates support different EQ frequency bands. 11025: {31, 62, 125, 250, 500, 1000, 2000, 3000, 4000, 5500} 22050: {31, 62, 125, 250, 500, 1000, 2000, 4000, 8000, 11000} 44100/48000: {31, 62, 125, 250, 500, 1000, 2000, 4000, 8000, 16000}

Note: 1) This figure indicate the default eq gain of every band in current equalizer. 2) Every channel have 10 band to set.

Public Members

int **samplerate**

Audio sample rate. Supported samplerate: 11025, 22050, 44100, 48000, unit: Hz

int **channel**

Number of audio channels. Supported channel: mono, stereo

int ***set_gain**

Equalizer gain

int **out_rb_size**

Size of output ring buffer

int **task_stack**

Task stack size

int **task_core**

Task running in core…

int **task_prio**

Task priority

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

EQUALIZER_TASK_STACK

EQUALIZER_TASK_CORE

EQUALIZER_TASK_PRIO

EQUALIZER_RINGBUFFER_SIZE

DEFAULT_EQUALIZER_CONFIG()

Type Definitions

typedef struct *equalizer_cfg* **`equalizer_cfg_t`**

Equalizer Configuration.

+—————————————+ | MONO | +—————————————+ | Only Left channel/Only Right channel | +—————————————+ |band0|band1|band2|band3|band4|band5|band6|band7|band8|band9| +—————————————+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | +—————————————+

+—————————————+—————————————+ | DUAL | +—————————————+—————————————+ | Left channel | Right channel | +—————————————+—————————————+ |band0|band1|band2|band3|band4|band5|band6|band7|band8|band9|band0|band1|band2|band3|band4|band5|band6|band7|band8|band +—————————————+—————————————+ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | +—————————————+—————————————+

3) Different sample rates support different EQ frequency bands. 11025: {31, 62, 125, 250, 500, 1000, 2000, 3000, 4000, 5500} 22050: {31, 62, 125, 250, 500, 1000, 2000, 4000, 8000, 11000} 44100/48000: {31, 62, 125, 250, 500, 1000, 2000, 4000, 8000, 16000}

Note: 1) This figure indicate the default eq gain of every band in current equalizer. 2) Every channel have 10 band to set.

2.5.3 Resample Filter

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

Application Example

Implementation of this API is demonstrated in the following examples:

- audio_processing/pipeline_resample
- audio_processing/pipeline_spiffs_amr_resample
- get-started/play_mp3_control

API Reference

Header File

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

Functions

esp_err_t **rsp_filter_set_src_info** (*audio_element_handle_t* self, int src_rate, int src_ch)

Set the source audio sample rate and the number of channels to be processed by the resample. If need change bits or not ensure source data infomation, please use rsp_filter_change_src_info to instead this function.

Parameters

- **self** – Audio element handle
- **src_rate** – The sample rate of stream data
- **src_ch** – The number channels of stream data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **rsp_filter_change_src_info** (*audio_element_handle_t* self, int src_rate, int src_ch, int src_bit)

Set the source audio sample rate, the number of channels and bits per sample to be processed by the resample.

Parameters

- **self** – Audio element handle
- **src_rate** – The sample rate of stream data
- **src_ch** – The number channels of stream data
- **src_bit** – The bit per sample of stream data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

audio_element_handle_t **rsp_filter_init** (*rsp_filter_cfg_t* *config)

Create an Audio Element handle to resample incoming data.

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

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

Parameters **config** – The configuration

Returns The audio element handler

Structures

struct **rsp_filter_cfg_t**

Resample Filter Configuration.

Public Members

int **src_rate**

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

int **src_ch**

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

int **dest_rate**

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

int **dest_bits**

The bit for sample of the destination PCM data. Currently, supported bit width :16 bits.

int **dest_ch**

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

int **src_bits**

The bit for sample of the source PCM data. Currently, supported bit width :8bits 16 bits 24bits 32bits.

esp_resample_mode_t **mode**

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

int **max_indata_bytes**

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

int **out_len_bytes**

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

esp_resample_type_t **type**

The resampling type (Automatic, Upsampling and Downsampling)

int **complexity**

Indicates the complexity of the resampling. This parameter is only valid when a FIR filter is used. Range:[1, 5]; 1 indicates the lowest complexity, which means the accuracy is the lowest and the speed is the fastest; Meanwhile, 5 indicates the highest complexity, which means the accuracy is the highest and the speed is the slowest.If user set `complexity` less than 1, `complexity` can be set 1. If user set `complexity` more than 5, `complexity` can be set 5.

int **down_ch_idx**

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

esp_rsp_prefer_type_t **prefer_flag**

The select flag about lesser CPU usage or lower INRAM usage, refer to esp_resample.h

int **out_rb_size**

Output ringbuffer size

int **task_stack**

Task stack size

int **task_core**

Task running on core

int **task_prio**

Task priority

bool **stack_in_ext**

Try to allocate stack in external memory

Macros

RSP_FILTER_BUFFER_BYTE

RSP_FILTER_TASK_STACK

RSP_FILTER_TASK_CORE

RSP_FILTER_TASK_PRIO

RSP_FILTER_RINGBUFFER_SIZE

DEFAULT_RESAMPLE_FILTER_CONFIG()

2.5.4 Sonic

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

The following parameters can be adjusted:

- speed
- pitch
- interpolation type

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

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

Application Example

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

API Reference

Header File

- components/esp-adf-libs/esp_codec/include/codec/audio_sonic.h

Functions

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

Sets the audio sample rate and the number of channels to be processed by the sonic.

Parameters

- **self** – Audio element handle
- **rate** – The sample rate of stream data
- **ch** – The channel numbers of stream data

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

esp_err_t **sonic_set_pitch_and_speed_info**(*audio_element_handle_t* self, float pitch, float speed)

Sets the audio pitch and speed to be processed by the sonic.

Parameters

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

Returns

- ESP_OK on success
- ESP_ERR_INVALID_ARG invalid arguments

audio_element_handle_t **sonic_init** (*sonic_cfg_t* *config)

Creates an Audio Element handle for sonic.

Parameters **config** – The sonic configuration

Returns The audio element handler

Structures

struct **sonic_info_t**

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

Public Members

int **samplerate**

Audio file sample rate (in Hz)

int **channel**

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

int **resample_linear_interpolate**

Flag of using simple linear interpolation. 1 indicates using simple linear interpolation. 0 indicates not using simple linear interpolation.

float **pitch**

Scale factor of pitch of audio file. If the value of 'pitch' is 0.3, the pitch of audio file processed by sonic islower than the original. If the value of 'pitch' is 1.3, the pitch of audio file processed by sonic is 30% higher than the original.

float **speed**

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

struct **sonic_cfg_t**

Sonic configuration.

Public Members

sonic_info_t **sonic_info**
Information of sonic

int **out_rb_size**
Size of output ring buffer

int **task_stack**
Task stack size

int **task_core**
Task running in core

int **task_prio**
Task priority

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

Macros

SONIC_SET_VALUE_FOR_INITIALIZATION

SONIC_TASK_STACK

SONIC_TASK_CORE

SONIC_TASK_PRIO

SONIC_RINGBUFFER_SIZE

DEFAULT_SONIC_CONFIG()

2.6 Services

A service is a software implementation of specific product functions, such as input keys, network configuration management, and battery check. Each service is also an abstraction of hardware. For example, the input key service supports ADC keys and GPIO keys. Services can be reused on different products, and high-level APIs and events allow easy and convenient product development.

For details please refer to descriptions listed below.

2.6.1 Bluetooth Service

The Bluetooth service is dedicated to interface with Bluetooth devices and provides support for the following protocols:

- HFP (Hands-Free Profile): remotely controlling the mobile phone by the Hands-Free device and the voice connections between them
- A2DP (Advanced Audio Distribution Profile): implementing streaming of multimedia audio using a Bluetooth connection
- AVRCP (Audio Video Remote Control Profile): used together with A2DP for remote control of devices such as headphones, car audio systems, or speakers

Application Example

Implementation of this API is demonstrated in the following example:

- player/pipeline_bt_sink

Header File

- components/bluetooth_service/include/bluetooth_service.h

Functions

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

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

Parameters **config** – The configuration

Returns

- ESP_OK
- ESP_FAIL

audio_element_handle_t **bluetooth_service_create_stream**()

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

Returns The Audio Element handle

esp_periph_handle_t **bluetooth_service_create_periph**()

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

Returns The Peripheral handle

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

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK

- ESP_FAIL

esp_err_t **periph_bluetooth_pause**(*esp_periph_handle_t* periph)

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_stop**(*esp_periph_handle_t* periph)

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_next**(*esp_periph_handle_t* periph)

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_prev**(*esp_periph_handle_t* periph)

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_rewind**(*esp_periph_handle_t* periph)

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_fast_forward**(*esp_periph_handle_t* periph)

Send the AVRC passthrough command (FAST FORWARD) to the Bluetooth device.

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **periph_bluetooth_discover**(*esp_periph_handle_t* periph)

Start device discovery.

Parameters **periph** – **[in]** The periph

Returns

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

esp_err_t **periph_bluetooth_cancel_discover**(*esp_periph_handle_t* periph)

Cancel device discovery.

Parameters **periph** – **[in]** The periph

Returns

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

esp_err_t **periph_bluetooth_connect**(*esp_periph_handle_t* periph, *bluetooth_addr_t* remote_bda)

Connect remote Device.

Parameters

- **periph** – **[in]** The periph
- **remote_bda** – **[in]** remote Bluetooth device address

Returns

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

esp_err_t **bluetooth_service_destroy**()

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

Returns

- ESP_OK
- ESP_FAIL

int **periph_bluetooth_get_a2dp_sample_rate**()

Get a2dp sample rate.

Returns

- sample rate

Structures

struct **bluetooth_service_user_cb_t**

brief Bluetooth service user callback

Public Members

esp_a2d_cb_t **user_a2d_cb**

callback for a2dp

esp_a2d_sink_data_cb_t **user_a2d_sink_data_cb**

callback for a2dp sink data

esp_a2d_source_data_cb_t **user_a2d_source_data_cb**

callback for a2dp source data

esp_avrc_ct_cb_t **user_avrc_ct_cb**

callback for avrc ct

struct **bluetooth_service_cfg_t**

brief Bluetooth service configuration

Public Members

const char ***device_name**

Bluetooth local device name

const char ***remote_name**

Bluetooth remote device name

bluetooth_service_mode_t **mode**

Bluetooth working mode

bluetooth_service_user_cb_t **user_callback**

Bluetooth user callback

Macros

ESP_A2DP_SAMPLE_RATE

BLUETOOTH_ADDR_LEN

brief Bluetooth address length

Type Definitions

typedef uint8_t **bluetooth_addr_t**[BLUETOOTH_ADDR_LEN]

brief Bluetooth device address

Enumerations

enum **bluetooth_service_mode_t**

brief Bluetooth service working mode

Values:

enumerator **BLUETOOTH_A2DP_SINK**

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

enumerator **BLUETOOTH_A2DP_SOURCE**

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

Header File

- components/bluetooth_service/include/bt_keycontrol.h

Header File

- components/bluetooth_service/include/hfp_stream.h

Header File

- components/bluetooth_service/include/a2dp_stream.h

2.6.2 Input Key Service

The input key service provides GPIO input interrupts and ADC key functions in the form of events. For what common key functions are defined by the events for audio products, see *input_key_user_id_t*.

Application Example

Implementation of this API is demonstrated in the following example:

- checks/check_board_buttons
- player/pipeline_sdcard_mp3_control
- protocols/voip

Header File

- components/input_key_service/include/input_key_service.h

Functions

periph_service_handle_t **input_key_service_create** (*input_key_service_cfg_t* *input_key_config)

Initialize and start the input key service.

Parameters **input_key_config** – Configuration of input key service

Returns NULL failed others input key service handle

periph_service_state_t **get_input_key_service_state** (*periph_service_handle_t* input_handle)

Get the state of input key service.

Parameters **input_handle** – The handle of input key service

Returns state of input key service

esp_err_t **input_key_service_add_key** (*periph_service_handle_t* input_key_handle, *input_key_service_info_t* *input_key_info, int add_key_num)

Add input key's information to service list.

Parameters

- **input_key_handle** – handle of service
- **input_key_info** – input key's information
- **add_key_num** – number of keys

Returns ESP_OK success ESP_FAIL failed

Structures

struct **input_key_service_info_t**

input key's infomation

Public Members

esp_periph_id_t **type**

ID of peripherals

int **user_id**

The key's user id

int **act_id**

The key's action id

struct **input_key_service_cfg_t**

input key's configuration

Public Members

periph_service_config_t **based_cfg**

Peripheral service configuration

esp_periph_set_handle_t **handle**

Peripheral set handle

Macros

INPUT_KEY_SERVICE_TASK_STACK_SIZE

INPUT_KEY_SERVICE_TASK_PRIORITY

INPUT_KEY_SERVICE_TASK_ON_CORE

INPUT_KEY_SERVICE_DEFAULT_CONFIG()

Enumerations

enum **input_key_service_action_id_t**

input key action id

Values:

enumerator **INPUT_KEY_SERVICE_ACTION_UNKNOWN**

unknown action id

enumerator **INPUT_KEY_SERVICE_ACTION_CLICK**

click action id

enumerator **INPUT_KEY_SERVICE_ACTION_CLICK_RELEASE**

click release action id

enumerator **INPUT_KEY_SERVICE_ACTION_PRESS**

long press action id

enumerator **INPUT_KEY_SERVICE_ACTION_PRESS_RELEASE**

long press release id

Header File

- components/input_key_service/include/input_key_com_user_id.h

Enumerations

enum **input_key_user_id_t**

input key user user-defined id

Values:

enumerator **INPUT_KEY_USER_ID_UNKNOWN**

unknown user id

enumerator **INPUT_KEY_USER_ID_REC**

user id for recording

enumerator **INPUT_KEY_USER_ID_SET**

user id for settings

enumerator **INPUT_KEY_USER_ID_PLAY**

user id for playing

enumerator **INPUT_KEY_USER_ID_MODE**
user id for mode

enumerator **INPUT_KEY_USER_ID_VOLDOWN**
user id for volume down

enumerator **INPUT_KEY_USER_ID_VOLUP**
user id for volume up

enumerator **INPUT_KEY_USER_ID_MUTE**
user id for mute

enumerator **INPUT_KEY_USER_ID_CAPTURE**
user id for capture photo

enumerator **INPUT_KEY_USER_ID_MSG**
user id for message

enumerator **INPUT_KEY_USER_ID_BATTERY_CHARGING**
user id for battery charging

enumerator **INPUT_KEY_USER_ID_WAKEUP**
user id for GPIO wakeup

enumerator **INPUT_KEY_USER_ID_COLOR**
user id for color

enumerator **INPUT_KEY_USER_ID_MAX**

2.6.3 Wi-Fi Service

The Wi-Fi service can configure and manage the network. SmartConfig, BluFi, and AirKiss are supported for network configuration.

Application Example

Implementation of this API is demonstrated in the following example:

- dueros

Header File

- components/wifi_service/include/wifi_service.h

Functions

periph_service_handle_t **wifi_service_create** (*wifi_service_config_t* *config)

esp_err_t **wifi_service_destroy** (*periph_service_handle_t* handle)

esp_err_t **wifi_service_register_setting_handle** (*periph_service_handle_t* handle, *esp_wifi_setting_handle_t* setting, int *out_index)

esp_err_t **wifi_service_setting_start** (*periph_service_handle_t* handle, int index)

esp_err_t **wifi_service_update_sta_info** (*periph_service_handle_t* handle, wifi_config_t *wifi_conf)

esp_err_t **wifi_service_setting_stop** (*periph_service_handle_t* handle, int index)

esp_err_t **wifi_service_connect** (*periph_service_handle_t* handle)

esp_err_t **wifi_service_disconnect** (*periph_service_handle_t* handle)

esp_err_t **wifi_service_set_sta_info** (*periph_service_handle_t* handle, wifi_config_t *info)

periph_service_state_t **wifi_service_state_get** (*periph_service_handle_t* handle)

wifi_service_disconnect_reason_t **wifi_service_disconnect_reason_get** (*periph_service_handle_t* handle)

esp_err_t **wifi_service_erase_ssid_manager_info** (*periph_service_handle_t* handle)

esp_err_t **wifi_service_get_last_ssid_cfg** (*periph_service_handle_t* handle, wifi_config_t *wifi_cfg)

Structures

struct **wifi_service_config_t**

WiFi service configurations.

Public Members

int **task_stack**

>0 Service task stack; =0 with out task created

int **task_prio**

Service task priority (based on freeRTOS priority)

int **task_core**

Service task running in core (0 or 1)

bool **extern_stack**

Task stack allocate on the extern ram

periph_service_cb **evt_cb**

Service callback function

void ***cb_ctx**

Callback context

char ***user_data**

User data

int **setting_timeout_s**

Timeout of setting WiFi

int **max_retry_time**

Maximum times of reconnection

int **max_prov_retry_time**

Maximum times of reconnection after wifi provision

uint8_t **max_ssid_num**

Maximum ssid that can be stored

Macros

WIFI_SERVICE_DEFAULT_CONFIG()

Enumerations

enum **wifi_service_event_t**

WiFi STA service status.

Values:

enumerator **WIFI_SERV_EVENT_UNKNOWN**

enumerator **WIFI_SERV_EVENT_CONNECTING**

enumerator **WIFI_SERV_EVENT_CONNECTED**

enumerator **WIFI_SERV_EVENT_DISCONNECTED**

enumerator **WIFI_SERV_EVENT_SETTING_TIMEOUT**

enumerator **WIFI_SERV_EVENT_SETTING_FAILED**

enumerator **WIFI_SERV_EVENT_SETTING_FINISHED**

enum **wifi_service_disconnect_reason_t**

WiFi STA disconnection reasons.

Values:

enumerator **WIFI_SERV_STA_UNKNOWN**

enumerator **WIFI_SERV_STA_COM_ERROR**

enumerator **WIFI_SERV_STA_AUTH_ERROR**

enumerator **WIFI_SERV_STA_AP_NOT_FOUND**

enumerator **WIFI_SERV_STA_BY_USER**

enumerator **WIFI_SERV_STA_SET_INFO**

Header File

- components/wifi_service/include/esp_wifi_setting.h

Functions

esp_wifi_setting_handle_t **esp_wifi_setting_create**(const char *tag)

brief Create wifi setting handle instance

Parameters **tag** – Tag of the wifi setting handle

Returns

- NULL, Fail
- Others, Success

esp_err_t **esp_wifi_setting_destroy**(*esp_wifi_setting_handle_t* handle)

brief Destroy wifi setting handle instance

Parameters **handle** – The wifi setting handle instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_register_function**(*esp_wifi_setting_handle_t* handle, *wifi_setting_func* start, *wifi_setting_func* stop, *wifi_setting_teardown_func* teardown)

Register the wifi setting execute functions.

Parameters

- **handle** – The wifi setting handle instance
- **start** – The start wifi setting
- **stop** – The stop
- **teardown** – The destroy

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_register_notify_handle**(*esp_wifi_setting_handle_t* handle, void *on_handle)

Register the notify execute handle.

Parameters

- **handle** – The peripheral handle
- **on_handle** – The notify execute handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_info_notify**(*esp_wifi_setting_handle_t* handle, wifi_config_t *info)

Call this to notify the `wifi_config_t` to `on_handle`

Parameters

- **handle** – The wifi setting handle instance
- **info** – The `wifi_config_t`

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_set_data**(*esp_wifi_setting_handle_t* handle, void *data)

Set the user data.

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

Parameters

- **handle** – **[in]** The wifi setting handle instance
- **data** – The user data

Returns

- ESP_OK
- ESP_FAIL

void ***esp_wifi_setting_get_data**(*esp_wifi_setting_handle_t* handle)

Get the user data stored on `handle`

Parameters **handle** – **[in]** The wifi setting handle instance

Returns user data pointer

esp_err_t **esp_wifi_setting_start**(*esp_wifi_setting_handle_t* handle)

Call this to execute `start` function of wifi setting instance.

Parameters **handle** – The wifi setting handle instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_stop**(*esp_wifi_setting_handle_t* handle)

Call this to execute `stop` function of wifi setting instance.

Parameters **handle** – The wifi setting handle instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_wifi_setting_teardown**(*esp_wifi_setting_handle_t* handle, wifi_config_t *info)

Call this to execute `teardown` function of wifi setting instance.

Parameters

- **handle** – The wifi setting handle instance
- **info** – The `wifi_config_t`

Returns

- ESP_OK
- ESP_FAIL

Type Definitions

typedef struct esp_wifi_setting ***esp_wifi_setting_handle_t**

typedef esp_err_t (***wifi_setting_func**)(*esp_wifi_setting_handle_t* handle)

typedef esp_err_t (***wifi_setting_teardown_func**)(*esp_wifi_setting_handle_t* handle, wifi_config_t *info)

Header File

- components/wifi_service/include/smart_config.h

Functions

esp_wifi_setting_handle_t **smart_config_create**(*smart_config_info_t* *info)

brief Create smartconfig setting handle instance

Parameters **info** – Configuration of the smartconfig

Returns

- NULL, Fail
- Others, Success

Structures

struct **smart_config_info_t**

esp smartconfig configuration

Public Members

smartconfig_type_t **type**

Type of smartconfig

Macros

SMART_CONFIG_INFO_DEFAULT()

Header File

- components/wifi_service/include/blufi_config.h

Functions

esp_wifi_setting_handle_t **blufi_config_create**(void *info)

Create blufi setting handle instance.

Parameters **info** – **[in]** A pointer to void

Returns

- NULL, Fail
- Others, Success

esp_err_t **blufi_set_sta_connected_flag**(*esp_wifi_setting_handle_t* handle, bool flag)

Set flag to judge whether the station has connected to the AP.

Parameters

- **handle** – **[in]** Wifi setting handle
- **flag** – **[in]** bool type of station connection state

Returns

- NULL, Fail
- Others, Success

esp_err_t **blufi_set_customized_data**(*esp_wifi_setting_handle_t* handle, char *data, int data_len)

Set customized data to be sent after configurate wifi successfully.

Parameters

- **handle** – **[in]** Wifi setting handle
- **data** – **[in]** Customized data
- **data_len** – **[in]** Customized data length

Returns

- ESP_FAIL, Fail
- ESP_OK, Success

esp_err_t **blufi_send_customized_data**(*esp_wifi_setting_handle_t* handle)

Send customized data that be set before.

Parameters **handle** – **[in]** Wifi setting handle

Returns

- ESP_FAIL, Fail
- ESP_OK, Success

Header File

- components/wifi_service/include/airkiss_config.h

Functions

esp_wifi_setting_handle_t **airkiss_config_create**(*airkiss_config_info_t* *info)

brief Create airkiss setting handle instance

Parameters **info** – Configuration of the airkiss

Returns

- NULL, Fail
- Others, Success

Structures

struct **airkiss_lan_pack_param_t**

airkiss lan data pack

Public Members

void ***appid**

APP identifer data

void ***deviceid**

Device identifer data

struct **airkiss_config_info_t**

airkiss configurations

Public Members

airkiss_lan_pack_param_t **lan_pack**

User lan pack parameter

bool **ssdp_notify_enable**

Notify enable flag

char ***aes_key**

Airkiss aes key data

Macros

AIRKISS_CONFIG_INFO_DEFAULT()

Header File

- components/wifi_service/include/wifi_ssid_manager.h

Functions

wifi_ssid_manager_handle_t **wifi_ssid_manager_create**(uint8_t max_ssid_num)

esp_err_t **wifi_ssid_manager_get_latest_config**(*wifi_ssid_manager_handle_t* handle, wifi_config_t *config)

esp_err_t **wifi_ssid_manager_save**(*wifi_ssid_manager_handle_t* handle, const char *ssid, const char *pwd)

esp_err_t **wifi_ssid_manager_get_best_config**(*wifi_ssid_manager_handle_t* handle, wifi_config_t *config)

int **wifi_ssid_manager_get_ssid_num**(*wifi_ssid_manager_handle_t* handle)

esp_err_t **wifi_ssid_manager_list_show**(*wifi_ssid_manager_handle_t* handle)

esp_err_t **wifi_ssid_manager_erase_all**(*wifi_ssid_manager_handle_t* handle)

esp_err_t **wifi_ssid_manager_destroy**(*wifi_ssid_manager_handle_t* handle)

Type Definitions

typedef struct wifi_ssid_manager ***wifi_ssid_manager_handle_t**

2.6.4 OTA Service

OTA service can upgrade firmware over the air (OTA). The firmware can be fetched from local files or the network.

Application Example

Implementation of this API is demonstrated in the following example:

- ota

Header File

- components/ota_service/include/ota_service.h

Functions

periph_service_handle_t **ota_service_create**(*ota_service_config_t* *config)

Create the OTA service instance.

Parameters **config** – configuration of the OTA service

Returns

- NULL: Failed
- Others: Success

esp_err_t **ota_service_set_upgrade_param**(*periph_service_handle_t* handle, *ota_upgrade_ops_t* *list, int list_len)

Configure the upgrade parameter This function is not thread safe.

This function will set the parameter table to ota service, and the ota service will upgrade the partitions defined in the table one by one,

Parameters

- **handle** – **[in]** pointer to 'periph_service_handle_t' structure
- **list** – **[in]** pointer to '*ota_upgrade_ops_t*' structure
- **list_len** – **[in]** length of the 'list'

Returns

- ESP_OK: Success
- Others: Failed

Structures

struct **ota_service_config_t**

The OTA service configuration.

Public Members

int **task_stack**

>0 Service task stack; =0 with out task created

int **task_prio**

Service task priority (based on freeRTOS priority)

int **task_core**

Service task running in core (0 or 1)

periph_service_cb **evt_cb**

Service callback function

void ***cb_ctx**

Callback context

struct **ota_node_attr_t**

The OTA node attributions.

Public Members

esp_partition_type_t **type**

Partition type

char ***label**

Partition label

char ***uri**

The upgrade URL

char ***cert_pem**

SSL server certification, PEM format as string, if the client requires to verify server

struct **ota_upgrade_ops_t**

The upgrade operation.

Public Members

ota_node_attr_t **node**

The OTA node

ota_service_err_reason_t (***prepare**)(void **handle, *ota_node_attr_t* *node)

Functions ready for upgrade

ota_service_err_reason_t (***need_upgrade**)(void *handle, *ota_node_attr_t* *node)

Detect whether an upgrade is required

ota_service_err_reason_t (***execute_upgrade**)(void *handle, *ota_node_attr_t* *node)

For execute upgrade

ota_service_err_reason_t (***finished_check**)(void *handle, *ota_node_attr_t* *node, *ota_service_err_reason_t* result)

Check result of upgrade

bool **reboot_flag**

Reboot or not after upgrade

bool **break_after_fail**

Abort upgrade when got failed

struct **ota_result_t**

The result of the OTA upgrade.

Public Members

uint8_t **id**

The result ID

ota_service_err_reason_t **result**

The error reason

Macros

OTA_SERVICE_ERR_REASON_BASE

OTA_SERVICE_DEFAULT_CONFIG()

Enumerations

enum **ota_service_event_type_t**

The OTA service event type.

Values:

enumerator **OTA_SERV_EVENT_TYPE_RESULT**

enumerator **OTA_SERV_EVENT_TYPE_FINISH**

enum **ota_service_err_reason_t**

The OTA service error reasons.

Values:

enumerator **OTA_SERV_ERR_REASON_UNKNOWN**

enumerator **OTA_SERV_ERR_REASON_SUCCESS**

enumerator **OTA_SERV_ERR_REASON_NULL_POINTER**

enumerator **OTA_SERV_ERR_REASON_URL_PARSE_FAIL**

enumerator **OTA_SERV_ERR_REASON_ERROR_VERSION**

enumerator **OTA_SERV_ERR_REASON_NO_HIGHER_VERSION**

enumerator **OTA_SERV_ERR_REASON_ERROR_MAGIC_WORD**

enumerator **OTA_SERV_ERR_REASON_ERROR_PROJECT_NAME**

enumerator **OTA_SERV_ERR_REASON_FILE_NOT_FOUND**

enumerator **OTA_SERV_ERR_REASON_PARTITION_NOT_FOUND**

enumerator **OTA_SERV_ERR_REASON_PARTITION_WT_FAIL**

enumerator **OTA_SERV_ERR_REASON_PARTITION_RD_FAIL**

enumerator **OTA_SERV_ERR_REASON_STREAM_INIT_FAIL**

enumerator **OTA_SERV_ERR_REASON_STREAM_RD_FAIL**

enumerator **OTA_SERV_ERR_REASON_GET_NEW_APP_DESC_FAIL**

Header File

- components/ota_service/include/ota_proc_default.h

Functions

void **ota_app_get_default_proc**(*ota_upgrade_ops_t* *ops)

get the default process of `app partition` upgrade

Parameters **ops** – **[in]** pointer to *`ota_upgrade_ops_t`* structure

void **ota_data_get_default_proc**(*ota_upgrade_ops_t* *ops)

get the default process of `data partition` upgrade

Parameters **ops** – **[in]** pointer to *`ota_upgrade_ops_t`* structure

ota_service_err_reason_t **ota_data_image_stream_read**(void *handle, char *buf, int wanted_size)

read from the stream of upgrading

Parameters

- **handle** – **[in]** pointer to upgrade handle
- **buf** – **[in]** pointer to receive buffer
- **wanted_size** – **[in]** bytes to read

Returns

- OTA_SERV_ERR_REASON_SUCCESS: Success
- Others: Failed

ota_service_err_reason_t **ota_data_partition_write**(void *handle, char *buf, int size)

write to the data partition under upgrading

Parameters

- **handle** – **[in]** pointer to upgrade handle
- **buf** – **[in]** pointer to data buffer
- **size** – **[in]** bytes to write

Returns

- OTA_SERV_ERR_REASON_SUCCESS: Success
- Others: Failed

void **ota_data_partition_erase_mark**(void *handle)

Indicates that the ota partition has been erased By default, this part of flash will be erased during ota. If the behavior of erasing is called in applition, this API needs to be called.

Parameters **handle** – **[in]** pointer to upgrade handle

int **ota_get_version_number**(char *version)

Convert string of version to integer The version should be (V0.0.0 - V255.255.255)

Parameters **version** – **[in]** pointer to the string of version

Returns

- -1: Failed
- Others: version number

Header File

- components/ota_service/include/esp_fs_ota.h

Functions

esp_err_t **esp_fs_ota**(*esp_fs_ota_config_t* *ota_config)

Upgrade the firmware from filesystem.

Note: This API handles the entire OTA operation, so if this API is being used then no other APIs from `esp_fs_ota` component should be called. If more information and control is needed during the FS OTA process, then one can use `esp_fs_ota_begin` and subsequent APIs. If this API returns successfully, esp_restart() must be called to boot from the new firmware image.

Parameters **ota_config** – **[in]** pointer to *esp_fs_ota_config_t* structure.

Returns

- ESP_OK: OTA data updated, next reboot will use specified partition.
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image

- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in esp-idf's app_update component.

esp_err_t **esp_fs_ota_begin**(*esp_fs_ota_config_t* *ota_config, *esp_fs_ota_handle_t* *handle)

Start FS OTA Firmware upgrade.

This function initializes ESP FS OTA context and open the firmware file. This function must be invoked first. If this function returns successfully, then `esp_fs_ota_perform` should be called to continue with the OTA process and there should be a call to `esp_fs_ota_finish` on completion of OTA operation or on failure in subsequent operations.

Parameters

- **`ota_config`** – **[in]** pointer to *esp_fs_ota_config_t* structure
- **`handle`** – **[out]** pointer to an allocated data of type `esp_fs_ota_handle_t` which will be initialised in this function

Returns

- ESP_OK: FS OTA Firmware upgrade context initialised and file opened successful
- ESP_FAIL: For generic failure.
- ESP_ERR_INVALID_ARG: Invalid argument (missing/incorrect config, etc.)
- For other return codes, refer documentation in app_update component and esp_http_client component in esp-idf.

esp_err_t **esp_fs_ota_perform**(*esp_fs_ota_handle_t* fs_ota_handle)

Read image data from file stream and write it to OTA partition.

This function reads image data from file stream and writes it to OTA partition. This function must be called only if esp_fs_ota_begin() returns successfully. This function must be called in a loop since it returns after every file read operation thus giving you the flexibility to stop OTA operation midway.

Parameters **`fs_ota_handle`** – **[in]** pointer to esp_fs_ota_handle_t structure

Returns

- ESP_ERR_FS_OTA_IN_PROGRESS: OTA update is in progress, call this API again to continue.
- ESP_OK: OTA update was successful
- ESP_FAIL: OTA update failed
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image
- ESP_ERR_NO_MEM: Cannot allocate memory for OTA operation.
- ESP_ERR_FLASH_OP_TIMEOUT or ESP_ERR_FLASH_OP_FAIL: Flash write failed.
- For other return codes, refer OTA documentation in esp-idf's app_update component.

esp_err_t **esp_fs_ota_finish**(*esp_fs_ota_handle_t* fs_ota_handle)

Clean-up FS OTA Firmware upgrade and close the file stream.

This function closes the file stream and frees the ESP FS OTA context. This function switches the boot partition to the OTA partition containing the new firmware image.

Note: If this API returns successfully, esp_restart() must be called to boot from the new firmware image

Parameters **fs_ota_handle** – **[in]** pointer to esp_fs_ota_handle_t structure

Returns

- ESP_OK: Clean-up successful
- ESP_ERR_INVALID_STATE
- ESP_ERR_INVALID_ARG: Invalid argument
- ESP_ERR_OTA_VALIDATE_FAILED: Invalid app image

esp_err_t **esp_fs_ota_get_img_desc**(*esp_fs_ota_handle_t* fs_ota_handle, esp_app_desc_t *new_app_info)

Reads app description from image header. The app description provides information like the "Firmware version" of the image.

Note: This API can be called only after esp_fs_ota_begin() and before esp_fs_ota_perform(). Calling this API is not mandatory.

Parameters

- **fs_ota_handle** – **[in]** pointer to esp_fs_ota_handle_t structure
- **new_app_info** – **[out]** pointer to an allocated esp_app_desc_t structure

Returns

- ESP_ERR_INVALID_ARG: Invalid arguments
- ESP_FAIL: Failed to read image descriptor
- ESP_OK: Successfully read image descriptor

int **esp_fs_ota_get_image_len_read**(*esp_fs_ota_handle_t* fs_ota_handle)

Structures

struct **esp_fs_ota_config_t**

ESP FS OTA configuration.

Public Members

const char ***path**

file path

const int **buffer_size**

Size of buffer

Macros

ESP_ERR_FS_OTA_BASE

ESP_ERR_FS_OTA_IN_PROGRESS

Type Definitions

typedef void ***esp_fs_ota_handle_t**

2.6.5 DuerOS Service

The DuerOS service allows voice interaction with DuerOS.

Application Example

Implementation of this API is demonstrated in the following example:

- dueros

Header File

- components/dueros_service/include/dueros_service.h

Functions

audio_service_handle_t **dueros_service_create**()

Create the dueros service.

Returns

- NULL, Fail
- Others, Success

service_state_t **dueros_service_state_get**()

Get dueros service state.

Returns The state of service

esp_err_t **dueros_voice_upload**(*audio_service_handle_t* handle, void *buf, int len)

Upload voice to backend server.

Parameters

- **handle** – dueros service handle
- **buf** – Data buffer
- **len** – Size of buffer

Returns ESP_OK ESP_FAIL

esp_err_t **dueros_voice_cancel**(*audio_service_handle_t* handle)

Cancel the current session.

Parameters **handle** – dueros service handle

Returns ESP_OK ESP_FAIL

esp_err_t **dueros_start_wifi_cfg**(*audio_service_handle_t* handle, duer_wifi_cfg_t *cfg)

Start the wifi configure process.

Parameters

- **handle** – Dueros service handle
- **cfg** – Configuration

Returns ESP_OK ESP_FAIL

esp_err_t **dueros_stop_wifi_cfg**(*audio_service_handle_t* handle)

Stop the wifi configure process.

Parameters **handle** – Dueros service handle

Returns ESP_OK ESP_FAIL

esp_err_t **dueros_wifi_status_report**(*audio_service_handle_t* handle, *dueros_wifi_st_t* *st)

Report the wifi status to dipb.

Parameters

- **handle** – Dueros service handle
- **st** – WiFi status and error code

Returns ESP_OK ESP_FAIL

Structures

struct **dueros_wifi_st_t**

Status of WiFi connection.

Public Members

int **status**

Please refer to: `enum duer_dipb_client_status_e`

int **err**

Please refer to: `enum duer_wifi_connect_error_code_e`

2.6.6 Display Service

The display service defines enumeration values for some common display patterns in `display_pattern_t`, with which you can set the corresponding patterns of LEDs or LED bars.

The currently supported LED drivers are AW2013, WS2812, and IS31x.

Application Example

Implementation of this API is demonstrated in the following example:

- checks/check_display_led

Header File

- components/display_service/include/display_service.h

Functions

display_service_handle_t **display_service_create**(*display_service_config_t* *cfg)

esp_err_t **display_service_set_pattern**(void *handle, int disp_pattern, int value)

esp_err_t **display_destroy**(*display_service_handle_t* handle)

Structures

struct **display_service_config_t**

Display service configurations.

Public Members

periph_service_config_t **based_cfg**

Peripheral service configuration

void ***instance**

Sub-instance

Type Definitions

typedef struct display_service_impl ***display_service_handle_t**

Enumerations

enum **display_pattern_t**

Values:

enumerator **DISPLAY_PATTERN_UNKNOWN**

enumerator **DISPLAY_PATTERN_WIFI_SETTING**

enumerator **DISPLAY_PATTERN_WIFI_CONNECTTING**

enumerator **DISPLAY_PATTERN_WIFI_CONNECTED**

enumerator **DISPLAY_PATTERN_WIFI_DISCONNECTED**

enumerator **DISPLAY_PATTERN_WIFI_SETTING_FINISHED**

enumerator **DISPLAY_PATTERN_BT_CONNECTTING**

enumerator **DISPLAY_PATTERN_BT_CONNECTED**

enumerator **DISPLAY_PATTERN_BT_DISCONNECTED**

enumerator **DISPLAY_PATTERN_RECORDING_START**

enumerator **DISPLAY_PATTERN_RECORDING_STOP**

enumerator **DISPLAY_PATTERN_RECOGNITION_START**

enumerator **DISPLAY_PATTERN_RECOGNITION_STOP**

enumerator **DISPLAY_PATTERN_WAKEUP_ON**

enumerator **DISPLAY_PATTERN_WAKEUP_FINISHED**

enumerator **DISPLAY_PATTERN_MUSIC_ON**

enumerator **DISPLAY_PATTERN_MUSIC_FINISHED**

enumerator **DISPLAY_PATTERN_VOLUME**

enumerator **DISPLAY_PATTERN_MUTE_ON**

enumerator **DISPLAY_PATTERN_MUTE_OFF**

enumerator **DISPLAY_PATTERN_TURN_ON**

enumerator **DISPLAY_PATTERN_TURN_OFF**

enumerator **DISPLAY_PATTERN_BATTERY_LOW**

enumerator **DISPLAY_PATTERN_BATTERY_CHARGING**

enumerator **DISPLAY_PATTERN_BATTERY_FULL**

enumerator **DISPLAY_PATTERN_POWERON_INIT**

enumerator **DISPLAY_PATTERN_WIFI_NO_CFG**

enumerator **DISPLAY_PATTERN_SPEECH_BEGIN**

enumerator **DISPLAY_PATTERN_SPEECH_OVER**

enumerator **DISPLAY_PATTERN_MAX**

Header File

- components/display_service/led_bar/include/led_bar_aw2013.h

Functions

void **aw2013_led_bar_task**(void *parameters)

esp_periph_handle_t **led_bar_aw2013_init**(void)

Initialize led bar instance.

Returns

- NULL Error
- others Success

esp_err_t **led_bar_aw2013_pattern**(void *handle, int pat, int value)

Set led bar display pattern.

Parameters

- **handle** – led bar instance
- **pat** – display pattern
- **value** – value of pattern

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **led_bar_aw2013_set_blink_time**(void *handle, uint8_t time, int period)

Set blinking period and times.

Parameters

- **handle** – led bar instance
- **time** – times of blink
- **period** – period of blink

Returns

- ESP_OK
- ESP_FAIL

void **led_bar_aw2013_deinit**(*esp_periph_handle_t* handle)

Destroy esp_periph_handle_t instance.

Parameters **handle** – led bar instance

Header File

- components/display_service/led_bar/include/led_bar_is31x.h

Functions

esp_periph_handle_t **led_bar_is31x_init**()

Initialize esp_periph_handle_t instance.

Returns

- NULL, Fail
- Others, Success

esp_err_t **led_bar_is31x_pattern**(void *handle, int pat, int value)

Set led bar display pattern.

Parameters

- **handle** – led bar instance
- **pat** – display pattern

- **value** – value of pattern

Returns

- ESP_OK
- ESP_FAIL

void **led_bar_is31x_deinit**(*esp_periph_handle_t* handle)

Destroy esp_periph_handle_t instance.

Parameters **handle** – led bar instance

Header File

- components/display_service/led_bar/include/led_bar_ws2812.h

Functions

led_bar_ws2812_handle_t **led_bar_ws2812_init**(gpio_num_t gpio_num, int led_num)

Initialize led_bar_ws2812_handle_t instance.

Parameters

- **gpio_num** – The GPIO number of ws2812
- **led_num** – The number of all ws2812

Returns

- led_bar_ws2812_handle_t

esp_err_t **led_bar_ws2812_pattern**(void *handle, int pat, int value)

Set ws2812 pattern.

Parameters

- **handle** – ws2812 indicator instance
- **pat** – display pattern
- **value** – value of pattern

Returns

- ESP_OK, success
- Others, fail

esp_err_t **led_bar_ws2812_deinit**(*led_bar_ws2812_handle_t* handle)

Destroy led_bar_ws2812_handle_t instance.

Parameters **handle** – ws2812 indicator instance

Returns

Returns

- ESP_OK, success
- Others, fail

Type Definitions

typedef struct led_bar_ws2812_impl ***led_bar_ws2812_handle_t**

Header File

- components/display_service/led_indicator/include/led_indicator.h

Functions

led_indicator_handle_t **led_indicator_init**(gpio_num_t num)

Initialize led_indicator_handle_t instance.

Parameters **num** – led gpio number

Returns

- NULL, Fail
- Others, Success

esp_err_t **led_indicator_pattern**(void *handle, int pat, int value)

Set led indicator display pattern.

Parameters

- **handle** – led indicator instance
- **pat** – display pattern
- **value** – value of pattern

Returns

- ESP_OK
- ESP_FAIL

void **led_indicator_deinit**(*led_indicator_handle_t* handle)

Destroy led_indicator_handle_t instance.

Parameters **handle** – led indicator instance

Type Definitions

typedef struct led_indicator_impl ***led_indicator_handle_t**

2.6.7 Battery Service

The battery service can monitor and manage battery voltage. You can access battery voltage and the events defined by *battery_service_event_t* through callback functions.

Application Example

Implementation of this API is demonstrated in the following example:

- system/battery

Header File

- components/battery_service/include/battery_service.h

Functions

periph_service_handle_t **battery_service_create** (*battery_service_config_t* *config)

Create the battery service instance.

Parameters **config** – configuration of the battery service

Returns

- NULL: Failed
- Others: Success

esp_err_t **battery_service_vol_report_switch** (*periph_service_handle_t* handle, bool on_off)

Start or stop the battery voltage report.

Parameters

- **handle** – **[in]** pointer to 'periph_service_handle_t' structure
- **on_off** – **[in]** 'true' to start, 'false' to stop

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG handle is NULL

esp_err_t **battery_service_set_vol_report_freq** (*periph_service_handle_t* handle, int freq)

Set voltage report freqency.

Parameters

- **handle** – **[in]** pointer to 'periph_service_handle_t' structure
- **freq** – **[in]** voltage report freqency

Returns

- ESP_OK: Success
- ESP_ERR_INVALID_ARG handle is NULL

Structures

struct **battery_service_config_t**

Battery service configure.

Public Members

int **task_stack**

>0 Service task stack; =0 with out task created

int **task_prio**

Service task priority (based on freeRTOS priority)

int **task_core**

Service task running in core (0 or 1)

bool **extern_stack**

Task stack allocate on the extern ram

periph_service_cb **evt_cb**

Service callback function

void ***cb_ctx**

Callback context

vol_monitor_handle_t **vol_monitor**

Battery monitor

void ***charger_monitor**

Charger monitor. Not supported yet

Macros

BATTERY_SERVICE_DEFAULT_CONFIG( )

Enumerations

enum **battery_service_event_t**

Battery service event.

Values:

enumerator **BAT_SERV_EVENT_UNKNOWN**

enumerator **BAT_SERV_EVENT_VOL_REPORT**

enumerator **BAT_SERV_EVENT_BAT_FULL**

enumerator **BAT_SERV_EVENT_BAT_LOW**

enumerator **BAT_SERV_EVENT_CHARGING_BEGIN**

enumerator **BAT_SERV_EVENT_CHARGING_STOP**

Header File

- components/battery_service/monitors/include/voltage_monitor.h

Functions

vol_monitor_handle_t **vol_monitor_create**(*vol_monitor_param_t* *config)

Create the voltage monitor instance.

Parameters **config** – **[in]** pointer to '*vol_monitor_param_t*' structure

Returns

- NULL: Failed
- Others: Success

esp_err_t **vol_monitor_destroy**(*vol_monitor_handle_t* handle)

Destroy the voltage monitor.

Parameters **handle** – **[in]** pointer to 'vol_monitor_handle_t' structure

Returns

- ESP_OK: Success
- Others: Failed

esp_err_t **vol_monitor_set_event_cb**(*vol_monitor_handle_t* handle, *vol_monitor_event_cb* event_cb, void *user_ctx)

Set the event callback.

Parameters

- **handle** – **[in]** pointer to 'vol_monitor_handle_t' structure
- **event_cb** – **[in]** callback used to handle the events of voltage monitor
- **user_ctx** – **[in]** user's data

Returns

- ESP_OK: Success
- Others: Failed

esp_err_t **vol_monitor_start_freq_report**(*vol_monitor_handle_t* handle)

Start the voltage report with the configured frequency.

Parameters **handle** – **[in]** pointer to 'vol_monitor_handle_t' structure

Returns

- ESP_OK: Success
- Others: Failed

esp_err_t **vol_monitor_stop_freq_report**(*vol_monitor_handle_t* handle)

Stop the voltage frequency report.

Parameters **handle** – **[in]** pointer to 'vol_monitor_handle_t' structure

Returns

- ESP_OK: Success
- Others: Failed

esp_err_t **vol_monitor_set_report_freq**(*vol_monitor_handle_t* handle, int freq)

Set the voltage report frequency.

Parameters

- **handle** – **[in]** pointer to 'vol_monitor_handle_t' structure
- **freq** – **[in]** voltage report freqency

Returns

- ESP_OK: Success
- Others: Failed

Structures

struct **vol_monitor_param_t**

Battery adc configure.

Public Members

bool (***init**)(void*)

Voltage read init

bool (***deinit**)(void*)

Voltage read deinit

int (***vol_get**)(void*)

Voltage read interface

void ***user_data**

Parameters for callbacks

int **read_freq**

Voltage read frequency, unit: s

int **report_freq**

Voltage report frequency, voltage will be report with a interval calculate by `read_freq * report_freq`

int **vol_full_threshold**

Voltage threshold to report, unit: mV

int **vol_low_threshold**

Voltage threshold to report, unit: mV

Type Definitions

typedef void ***vol_monitor_handle_t**

voltage monitor handle

typedef void (***vol_monitor_event_cb**)(int msg_id, void *data, void *user_ctx)

callback define

Enumerations

enum **vol_monitor_event_t**

Battery adc configure.

Values:

enumerator **VOL_MONITOR_EVENT_FREQ_REPORT**

enumerator **VOL_MONITOR_EVENT_BAT_FULL**

enumerator **VOL_MONITOR_EVENT_BAT_LOW**

2.6.8 Core Dump Upload Service

To investigate crashes in some sold devices, the backtrace is needed for analysis. The core dump upload service transmits the backtrace stored in the device partition over HTTP. To enable this feature, select `ESP_COREDUMP_ENABLE_TO_FLASH`.

Application Example

Implementation of this API is demonstrated in the following example:

- system/coredump

Header File

- components/coredump_upload_service/include/coredump_upload_service.h

Functions

bool **coredump_need_upload**()

This function will check the reset code and determine whether to upload the coredump.

Returns

- true: last reboot is a abnormal reset.
- false

esp_err_t **coredump_upload**(*periph_service_handle_t* handle, char *url)

Upload the core dump image to the url. This function will block the current task until the upload process finished.

Parameters

- **handle** – **[in]** the 'periph_service_handle_t'
- **url** – **[in]** server addr

Returns

- ESP_OK
- ESP_FAIL

periph_service_handle_t **coredump_upload_service_create**(*coredump_upload_service_config_t* *config)

Create the core dump upload service instance.

Parameters **config** – configuration of the OTA service

Returns

- NULL: Failed
- Others: Success

Structures

struct **coredump_upload_service_config_t**

coredump service configuration parameters

Public Members

int **task_stack**

>0 Service task stack; =0 with out task created

int **task_prio**

Service task priority (based on freeRTOS priority)

int **task_core**

Service task running in core (0 or 1)

periph_service_cb **evt_cb**

Service callback function

void ***cb_ctx**

Callback context

bool (***do_post**)(char *url, uint8_t *data, size_t len)

POST interface, users can override this to customize the http client. if left NULL, the service will use the default one

Macros

COREDUMP_UPLOAD_SERVICE_DEFAULT_CONFIG()

2.6.9 Header File

- components/esp_dispatcher/include/periph_service.h

2.6.10 Functions

periph_service_handle_t **periph_service_create**(*periph_service_config_t* *config)

brief Create peripheral service instance

Parameters **config** – **[in]** Configuration of the peripheral service instance

Returns

- NULL, Fail
- Others, Success

esp_err_t **periph_service_destroy**(*periph_service_handle_t* handle)

brief Destroy peripheral service instance

Parameters **handle** – **[in]** The peripheral service instance

Returns

- ESP_OK

- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **periph_service_start** (*periph_service_handle_t* handle)

brief Start the specific peripheral service

Parameters **handle** – **[in]** The peripheral service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **periph_service_stop** (*periph_service_handle_t* handle)

brief Stop the specific peripheral service

Parameters **handle** – **[in]** The peripheral service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **periph_service_set_callback** (*periph_service_handle_t* handle, *periph_service_cb* cb, void *ctx)

brief Set the specific peripheral service callback function

Parameters

- **handle** – **[in]** The peripheral service instance
- **cb** – **[in]** A pointer to service_callback
- **ctx** – **[in]** A pointer to user context

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **periph_service_callback** (*periph_service_handle_t* handle, *periph_service_event_t* *evt)

brief Called peripheral service by configurations

Parameters

- **handle** – **[in]** The peripheral service instance
- **evt** – **[in]** A pointer to *periph_service_event_t*

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **periph_service_set_data**(*periph_service_handle_t* handle, void *data)

brief Set user data to specific peripheral service instance

Parameters

- **handle** – **[in]** The peripheral service instance
- **data** – **[in]** A pointer to user data

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

void ***periph_service_get_data**(*periph_service_handle_t* handle)

brief Get user data by specific peripheral service instance

Parameters **handle** – **[in]** The peripheral service instance

Returns A pointer to user data

esp_err_t **periph_service_ioctl**(*periph_service_handle_t* handle, void *ioctl_handle, int cmd, int value)

brief In/out control by peripheral service instance

Parameters

- **handle** – **[in]** The peripheral service instance
- **ioctl_handle** – **[in]** Sub-instance handle
- **cmd** – **[in]** Command of value
- **value** – **[in]** Data of the command

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

2.6.11 Structures

struct **periph_service_event_t**

Peripheral service event informations.

Public Members

int **type**

Type of event

void ***source**

Event source

void *data
: Event data

int len
: Length of data

struct periph_service_config_t
: Peripheral service configurations.

Public Members

int task_stack
: >0 Service task stack; =0 with out task created

int task_prio
: Service task priority (based on freeRTOS priority)

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

TaskFunction_t task_func
: Service task function

bool extern_stack
: Task stack allocate on the extern ram

periph_service_ctrl service_start
: Start function

periph_service_ctrl service_stop
: Stop function

periph_service_ctrl service_destroy
: Destroy function

periph_service_io service_ioctl
: In out control function

char *service_name
: Name of peripheral service

void *user_data
: User data

2.6.12 Type Definitions

typedef struct periph_service_impl ***periph_service_handle_t**

typedef esp_err_t (***periph_service_ctrl**)(*periph_service_handle_t* handle)

typedef esp_err_t (***periph_service_io**)(void *ioctl_handle, int cmd, int value)

typedef esp_err_t (***periph_service_cb**)(*periph_service_handle_t* handle, *periph_service_event_t* *evt, void *ctx)

2.6.13 Enumerations

enum **periph_service_state_t**

Peripheral service state.

Values:

enumerator **PERIPH_SERVICE_STATE_UNKNOWN**

enumerator **PERIPH_SERVICE_STATE_IDLE**

enumerator **PERIPH_SERVICE_STATE_RUNNING**

enumerator **PERIPH_SERVICE_STATE_STOPPED**

2.6.14 Header File

- components/esp_dispatcher/include/audio_service.h

2.6.15 Functions

audio_service_handle_t **audio_service_create**(*audio_service_config_t* *config)

brief Create audio service instance

Parameters **config** – **[in]** Configuration of the audio service instance

Returns

- NULL, Fail
- Others, Success

esp_err_t **audio_service_destroy**(*audio_service_handle_t* handle)

brief Destroy audio service instance

Parameters **handle** – **[in]** The audio service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_start**(*audio_service_handle_t* handle)

brief Start the specific audio service

Parameters **handle** – **[in]** The audio service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_stop**(*audio_service_handle_t* handle)

brief Stop the specific audio service

Parameters **handle** – **[in]** The audio service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_set_callback**(*audio_service_handle_t* handle, *service_callback* cb, void *ctx)

brief Set the specific audio service callback function.

Parameters

- **handle** – **[in]** The audio service instance
- **cb** – **[in]** A pointer to service_callback
- **ctx** – **[in]** A pointer to user context

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_callback**(*audio_service_handle_t* handle, *service_event_t* *evt)

brief Called audio service by configurations

Parameters

- **handle** – **[in]** The audio service instance
- **evt** – **[in]** A pointer to *service_event_t*

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_connect** (*audio_service_handle_t* handle)

brief Connect the specific audio service

Parameters **handle** – **[in]** The audio service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_disconnect** (*audio_service_handle_t* handle)

brief Disconnect the specific audio service

Parameters **handle** – **[in]** The audio service instance

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

esp_err_t **audio_service_set_data** (*audio_service_handle_t* handle, void *data)

brief Set user data to specific audio service instance

Parameters

- **handle** – **[in]** The audio service instance
- **data** – **[in]** A pointer to user data

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG

void ***audio_service_get_data** (*audio_service_handle_t* handle)

brief Get user data by specific audio service instance

Parameters **handle** – **[in]** The audio service instance

Returns A pointer to user data

2.6.16 Structures

struct **service_event_t**

Audio service event informations.

Public Members

int **type**
Type of event

void ***source**
Event source

void ***data**
Event data

int **len**
Length of data

struct **audio_service_config_t**
Audio service configurations.

Public Members

int **task_stack**
>0 Service task stack; =0 with out task created

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

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

TaskFunction_t **task_func**
A pointer to TaskFunction_t for service task function

service_ctrl **service_start**
Start function

service_ctrl **service_stop**
Stop function

service_ctrl **service_connect**
Connect function

service_ctrl **service_disconnect**
Disconnect function

service_ctrl **service_destroy**
Destroy function

const char ***service_name**

Name of audio service

void ***user_data**

User context

2.6.17 Type Definitions

typedef struct audio_service_impl ***audio_service_handle_t**

typedef esp_err_t (***service_ctrl**)(*audio_service_handle_t* handle)

typedef esp_err_t (***service_callback**)(*audio_service_handle_t* handle, *service_event_t* *evt, void *ctx)

2.6.18 Enumerations

enum **service_state_t**

Audio service state.

Values:

enumerator **SERVICE_STATE_UNKNOWN**

enumerator **SERVICE_STATE_IDLE**

enumerator **SERVICE_STATE_CONNECTING**

enumerator **SERVICE_STATE_CONNECTED**

enumerator **SERVICE_STATE_RUNNING**

enumerator **SERVICE_STATE_STOPPED**

2.7 Speech Recognition

ESP-ADF offers a comprehensive range of speech recognition processing functions, such as front-end speech processing, TTS, voice wake-up, and command word recognition. ESP-ADF's Element-based *audio recorder* integrates speech recognition and audio signal processing into an event-driven High-level API. This approach enables users to maintain flexibility in their configurations while making it easier to use.

2.7.1 Voice Activity Detection

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

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

Application Example

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

API Reference

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

2.7.2 WakeNet Interface

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

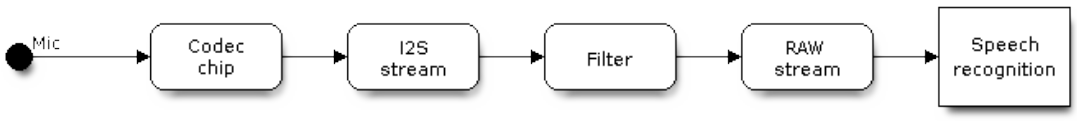


Fig. 5: Sample Speech Recognition Pipeline

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

The above introduction is the primary guidance. ESP-ADF offers users a more flexible and convenient module, namely the *audio recorder*, which is strongly recommended for use.

API Reference

For the latest speech recognition API reference, please refer to ESP-SR Speech Recognition Framework.

2.7.3 Audio Recorder

The Audio Recorder API is a set of functions to facilitate voice recording. It combines two important functions, namely Audio Front End (AFE) and audio encoding. This allows users to customize AFE's Voice Activity Detection (VAD), Automatic Gain Control (AGC), and Acoustic Echo Cancellation (AEC) settings. The encoding function is used by users to establish the encoding audio element, which supports various formats such as AAC, AMR-NB, AMR-WB, ADPCM, WAV, OPUS, and G711. The *audio_rec_evt_t* event makes it easy for users to interact with the Audio Recorder software.

The data path of the Audio recorder is presented in the diagram below.

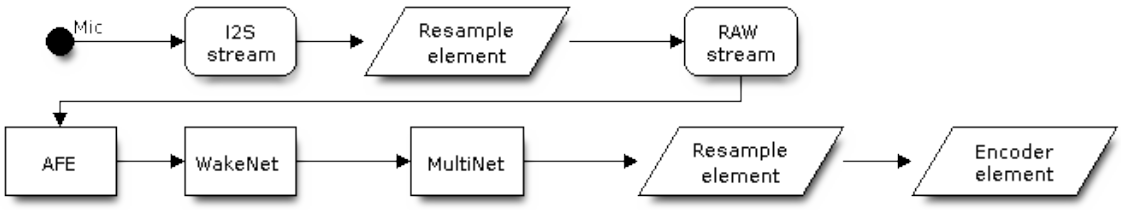


Fig. 6: Audio recorder data path

The area represented by the parallelogram is configurable by the user according to their needs, such as sampling frequency, whether to encode, and encoding format.

Application Example

The speech_recognition/wwe/ example demonstrates how to initialize the speech recognition model, determine the number of samples and the sample rate of voice data to feed to the model, detect the wake-up word and command words, and encode voice to specific audio format.

API Reference

Header File

- components/audio_recorder/include/audio_recorder.h

Functions

audio_rec_handle_t **audio_recorder_create** (*audio_rec_cfg_t* *cfg)

Initialize and start up audio recorder.

Parameters **cfg** – Configuration of audio recorder

Returns NULL failed Others audio recorder handle

esp_err_t **audio_recorder_trigger_start** (*audio_rec_handle_t* handle)

Start recording by force.

Note: If there need to read from recorder without wake word detected or read from recorder while the wake word detection is disabled, this interface can be use to force start the recorder process.

Parameters **handle** – Audio recorder handle

Returns ESP_OK ESP_FAIL

esp_err_t **audio_recorder_trigger_stop**(*audio_rec_handle_t* handle)

Stop recording by force.

Note: No matter the recorder process is triggered by wake word detected or triggered by `audio_recorder_trigger_start`, this function can be used to force stop the recorder. And if the VAD detection is disabeled, this must be invoked to stop recording after `audio_recorder_trigger_start`.

Parameters **handle** – Audio recorder handle

Returns ESP_OK ESP_FAIL

esp_err_t **audio_recorder_wakenet_enable**(*audio_rec_handle_t* handle, bool enable)

Enable or suspend wake word detection.

Parameters

- **handle** – Audio recorder handle
- **enable** – true: enable wake word detection false: disable wake word detection

Returns ESP_OK ESP_FAIL

esp_err_t **audio_recorder_multinet_enable**(*audio_rec_handle_t* handle, bool enable)

Enable or suspend speech command recognition.

Parameters

- **handle** – Audio recorder handle
- **enable** – true: enable speech command recognition false: disable speech command recognition

Returns ESP_OK ESP_FAIL

esp_err_t **audio_recorder_vad_check_enable**(*audio_rec_handle_t* handle, bool enable)

Enable or suspend voice duration check.

Parameters

- **handle** – Audio recorder handle
- **enable** – true: enable voice duration check false: disable voice duration check

Returns ESP_OK ESP_FAIL

int **audio_recorder_data_read**(*audio_rec_handle_t* handle, void *buffer, int length, TickType_t ticks)

Read data from audio recorder.

Parameters

- **handle** – Audio recorder handle
- **buffer** – Buffer to save data
- **length** – Size of buffer
- **ticks** – Timeout for reading

Returns Length of data actually read ESP_ERR_INVALID_ARG

esp_err_t **audio_recorder_destroy**(*audio_rec_handle_t* handle)

Destroy audio recorder and recycle all resource.

Parameters **handle** – Audio recorder handle

Returns ESP_OK ESP_FAIL

bool **audio_recorder_get_wakeup_state**(*audio_rec_handle_t* handle)

Get the wake up state of audio recorder.

Parameters **handle** – Audio recorder handle

Returns true false

Structures

struct **audio_rec_evt_t**

Recorder event.

Public Types

enum **[anonymous]**

Audio recorder event type.

Values:

enumerator **AUDIO_REC_WAKEUP_START**

Wakeup start

enumerator **AUDIO_REC_WAKEUP_END**

Wakeup stop

enumerator **AUDIO_REC_VAD_START**

Vad start

enumerator **AUDIO_REC_VAD_END**

Vad stop

enumerator **AUDIO_REC_COMMAND_DECT**

Form 0 is the id of the voice commands detected by Multinet

Public Members

enum *audio_rec_evt_t*::*[anonymous]* **type**

Audio recorder event type.

Event type

void ***event_data**

Event data: For `AUDIO_REC_WAKEUP_START`, event data is `recorder_sr_wakeup_result_t` For `AUDIO_REC_COMMAND_DECT` or higher, event data is `recorder_sr_mn_result_t` For other events, event data is NULL

size_t **data_len**

Length of event data

struct **audio_rec_cfg_t**

Audio recorder configuration.

Public Members

int **pinned_core**

Audio recorder task pinned to core

int **task_prio**

Audio recorder task priority

int **task_size**

Audio recorder task stack size

rec_event_cb_t **event_cb**

Event callback function, event type as *audio_rec_evt_t* shown above

void ***user_data**

Pointer to user data (optional)

recorder_data_read_t **read**

Data callback function used feed data to audio recorder

void ***sr_handle**

SR handle

recorder_sr_iface_t ***sr_iface**

SR interface

int **wakeup_time**

Unit:ms. The duration that the wakeup state remains when VAD is not triggered

int **vad_start**

Unit:ms. Consecutive speech frame will be judged to vad start

int **vad_off**

Unit:ms. When the silence time exceeds this value, it is determined as AUDIO_REC_VAD_END state

int **wakeup_end**

Unit:ms. When the silence time after AUDIO_REC_VAD_END state exceeds this value, it is determined as AUDIO_REC_WAKEUP_END

void ***encoder_handle**

Encoder handle

recorder_encoder_iface_t ***encoder_iface**

Encoder interface

Macros

AUDIO_REC_DEF_TASK_SZ

Stack size of recorder task

AUDIO_REC_DEF_TASK_PRIO

Priority of recoder task

AUDIO_REC_DEF_TASK_CORE

Pinned to core

AUDIO_REC_DEF_WAKEUP_TM

Default wake up time (ms)

AUDIO_REC_DEF_WAKEEND_TM

Duration after vad off (ms)

AUDIO_REC_VAD_START_SPEECH_MS

Consecutive speech frame will be judged to vad start (ms)

AUDIO_REC_DEF_VAD_OFF_TM

Default vad off time (ms)

AUDIO_RECORDER_DEFAULT_CFG()

Type Definitions

typedef esp_err_t (***rec_event_cb_t**)(*audio_rec_evt_t* *event, void *user_data)

Event Notification.

typedef struct __audio_recorder ***audio_rec_handle_t**

Audio recorder handle.

Header File

- components/audio_recorder/include/recorder_sr.h

Functions

recorder_sr_handle_t **recorder_sr_create** (*recorder_sr_cfg_t* *cfg, recorder_sr_iface_t **iface)

Initialize sr processor, and the sr is disabled as default.

Parameters

- **cfg** – Configuration of sr
- **iface** – User interface provide by recorder sr

Returns NULL failed Others SR handle

esp_err_t **recorder_sr_destroy** (*recorder_sr_handle_t* handle)

Destroy SR processor and recycle all resource.

Parameters **handle** – SR processor handle

Returns ESP_OK ESP_FAIL

esp_err_t **recorder_sr_reset_speech_cmd** (*recorder_sr_handle_t* handle, char *command_str, char *err_phrase_id)

Reset the speech commands.

Parameters

- **handle** – SR processor handle
- **command_str** – String of the commands. more details on `#2reset-api-on-the-fly`
- **err_phrase_id** – error string output

Returns ESP_OK ESP_FAIL

Structures

struct **recorder_sr_cfg_t**

SR processor configuration.

Note: Since the detection of command words requires a clear starting point, the moment the wake word is detected is taken as the default start of detection. Therefore, if the wake word detection is disabled, the detection will use the `vad_state` detected by `esp-sr` as the start of detection. However, due to the fluctuation of this `vad_state`, the effectiveness of command word detection will be limited.

Public Members

afe_config_t **afe_cfg**

Configuration of AFE

int8_t **input_order**[DAT_CH_MAX]

Channel order of the input data

bool **multinet_init**

Enable of speech command recognition

int **feed_task_core**

Core id of feed task

int **feed_task_prio**

Priority of feed task

int **feed_task_stack**

Stack size of feed task

int **fetch_task_core**

Core id of fetch task

int **fetch_task_prio**

Priority of fetch task

int **fetch_task_stack**

Stack size of fetch task

int **rb_size**

Ringbuffer size of recorder sr

char ***partition_label**

Partition label which stored the model data

char ***mn_language**

Command language for multinet to load

char ***wn_wakeword**

Wake Word for WakeNet to load. This is useful when multiple Wake Words are selected in sdkconfig. Setting this to NULL will use the first found model.

Macros

FEED_TASK_STACK_SZ

FETCH_TASK_STACK_SZ

FEED_TASK_PRIO

FETCH_TASK_PRIO

FEED_TASK_PINNED_CORE

FETCH_TASK_PINNED_CORE

SR_OUTPUT_RB_SIZE

INPUT_ORDER_DEFAULT()

DEFAULT_RECORDER_SR_CFG()

Type Definitions

typedef void ***recorder_sr_handle_t**

SR processor handle.

Header File

- components/audio_recorder/include/recorder_encoder.h

Functions

recorder_encoder_handle_t **recorder_encoder_create**(*recorder_encoder_cfg_t* *cfg, recorder_encoder_iface_t **iface)

Initialize encoder processor, and the encoder is disabled as default.

Parameters

- **cfg** – Configuration of encoder
- **iface** – User interface provide by recorder encoder

Returns NULL failed Others encoder handle

esp_err_t **recorder_encoder_destroy**(*recorder_encoder_handle_t* handle)

Destroy encoder processor and recycle all resource.

Parameters **handle** – Encoder processor handle

Returns ESP_OK ESP_FAIL

Structures

struct **recorder_encoder_cfg_t**

recorder encoder configuration parameters

Public Members

audio_element_handle_t **resample**

Handle of resample

audio_element_handle_t **encoder**

Handle of encoder

Type Definitions

typedef void ***recorder_encoder_handle_t**

encoder handle

2.8 Peripherals

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

2.8.1 ESP Peripherals

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

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

---

Examples

Please refer to player/pipeline_http_mp3/main/play_http_mp3_example.c.

API Reference

Header File

- components/esp_peripherals/include/esp_peripherals.h

Functions

esp_periph_set_handle_t **esp_periph_set_init** (*esp_periph_config_t* *config)

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

Parameters **config** – **[in]** The configurations

Returns The peripheral sets instance

esp_err_t **esp_periph_set_destroy** (*esp_periph_set_handle_t* periph_set_handle)

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

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_set_stop_all** (*esp_periph_set_handle_t* periph_set_handle)

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

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns

- ESP_OK
- ESP_FAIL

esp_periph_handle_t **esp_periph_set_get_by_id**(*esp_periph_set_handle_t* periph_set_handle, int periph_id)

Get the peripheral handle by Peripheral ID.

Parameters

- **periph_set_handle** – The esp_periph_set_handle_t instance
- **periph_id** – **[in]** as esp_periph_id_t, or any ID you use when calling `esp_periph_create`

Returns The esp_periph_handle_t

audio_event_iface_handle_t **esp_periph_set_get_event_iface**(*esp_periph_set_handle_t* periph_set_handle)

Return the event_iface used by this esp_peripherals.

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns The audio event iface handle

esp_err_t **esp_periph_set_register_callback**(*esp_periph_set_handle_t* periph_set_handle, *esp_periph_event_handle_t* cb, void *user_context)

Register peripheral sets event callback function.

Parameters

- **periph_set_handle** – The esp_periph_set_handle_t instance
- **cb** – The event handle callback function
- **user_context** – The user context pointer

Returns

- ESP_OK
- ESP_FAIL

QueueHandle_t **esp_periph_set_get_queue**(*esp_periph_set_handle_t* periph_set_handle)

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

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns The queue handle

esp_err_t **esp_periph_set_list_init**(*esp_periph_set_handle_t* periph_set_handle)

Call this function to initialize all the listed peripherals.

Note: Work with no task peripheral set only

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_set_list_run**(*esp_periph_set_handle_t* periph_set_handle, *audio_event_iface_msg_t* msg)

Call this function to run all the listed peripherals.

Note: Work with no task peripheral set only

Parameters

- **periph_set_handle** – The esp_periph_set_handle_t instance
- **msg** – The *audio_event_iface_msg_t* handle message

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_set_list_destroy**(*esp_periph_set_handle_t* periph_set_handle)

Call this function to destroy all the listed peripherals.

Note: Work with no task peripheral set only

Parameters **periph_set_handle** – The esp_periph_set_handle_t instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_remove_from_set**(*esp_periph_set_handle_t* periph_set_handle, *esp_periph_handle_t* periph)

Call this function to remove periph from periph_set.

Parameters

- **periph_set_handle** – The esp_periph_set_handle_t instance
- **periph** – The esp_periph_handle_t instance

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_alloc_periph_id**(*esp_periph_set_handle_t* periph_set_handle, int *periph_id)

Allocated a peripheral ID from the periph_set.

Note: This function can apply for a peripheral ID to realize a customized peripheral module. The requested peripheral ID will not be included in `esp_periph_id_t`

Parameters

- **periph_set_handle** – **[in]** The esp_periph_set_handle_t instance

- **periph_id** – **[out]** The peripheral identifier

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_set_change_waiting_time**(*esp_periph_set_handle_t* periph_set_handle, int time_ms)

Call this function to change periph_set waiting time.

Parameters

- **periph_set_handle** – The esp_periph_set_handle_t instance
- **time_ms** – The waiting time

Returns

- ESP_OK
- ESP_FAIL

esp_periph_handle_t **esp_periph_create**(int periph_id, const char *tag)

Call this function to initialize a new peripheral.

Parameters

- **periph_id** – **[in]** The periph identifier
- **tag** – **[in]** The tag name, we named it easy to get in debug logs

Returns The peripheral handle

esp_err_t **esp_periph_set_function**(*esp_periph_handle_t* periph, *esp_periph_func* init, *esp_periph_run_func* run, *esp_periph_func* destroy)

Each peripheral has a cycle of sequential operations from initialization, execution of commands to destroying the peripheral. These operations are represented by functions passed as call parameters to this function.

Parameters

- **periph** – **[in]** The periph
- **init** – **[in]** The initialize
- **run** – **[in]** The run
- **destroy** – **[in]** The destroy

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_start**(*esp_periph_set_handle_t* periph_set_handle, *esp_periph_handle_t* periph)

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

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

Parameters

- **periph_set_handle** – **[in]** The esp_periph_set_handle_t instance

- **periph** – **[in]** The peripheral instance

Returns

- ESP_OK on success
- ESP_FAIL when any errors

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

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

Parameters **periph** – **[in]** The periph

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters

- **periph** – **[in]** The periph
- **cmd** – **[in]** The command
- **data** – The data
- **data_len** – **[in]** The data length

Returns

- ESP_OK
- ESP_FAIL

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

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

Parameters

- **periph** – **[in]** The periph
- **cmd** – **[in]** The command
- **data** – The data
- **data_len** – **[in]** The data length

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_send_event**(*esp_periph_handle_t* periph, int event_id, void *data, int data_len)

In addition to sending an event via event_iface, this function will dispatch the `event_handle` callback if the event_handle callback is provided at `esp_periph_init`

Parameters

- **periph** – **[in]** The peripheral

- **event_id** – **[in]** The event identifier
- **data** – The data
- **data_len** – **[in]** The data length

Returns

- ESP_OK
- ESP_FAIL

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

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

Note:

- You do not need to stop or destroy the timer, when the `esp_periph_destroy` function is called, it will stop and destroy all
- This timer using FreeRTOS Timer, with autoreload = true

Parameters

- **periph** – **[in]** The peripheral
- **interval_tick** – **[in]** The interval tick
- **callback** – **[in]** The callback

Returns

- ESP_OK
- ESP_FAIL

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

Stop peripheral timer.

Parameters **periph** – **[in]** The peripheral

Returns

- ESP_OK
- ESP_FAIL

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

Set the user data.

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

---

Parameters

- **periph** – **[in]** The peripheral
- **data** – The data

Returns

- ESP_OK
- ESP_FAIL

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

Get the user data stored in the peripheral.

Parameters **periph** – **[in]** The peripheral

Returns Peripheral data pointer

esp_periph_state_t **esp_periph_get_state**(*esp_periph_handle_t* periph)

Get the current state of peripheral.

Parameters **periph** – **[in]** The handle of peripheral

Returns The peripharal working state

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

Get Peripheral identifier.

Parameters **periph** – **[in]** The peripheral

Returns The peripheral identifier

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

Set Peripheral identifier.

Parameters

- **periph** – **[in]** The peripheral
- **periph_id** – **[in]** The peripheral identifier

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_init**(*esp_periph_handle_t* periph)

Call this to execute `init` function of peripheral instance.

Parameters **periph** – The peripheral handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_run**(*esp_periph_handle_t* periph)

Call this to execute `run` function of peripheral instance.

Parameters **periph** – The peripheral handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_destroy**(*esp_periph_handle_t* periph)

Call this to execute `destroy` function of peripheral instance.

Parameters **periph** – The peripheral handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **esp_periph_register_on_events**(*esp_periph_handle_t* periph, *esp_periph_event_t* *evts)

Rigster peripheral on event handle.

Parameters

- **periph** – The peripheral handle
- **evts** – The esp_periph_event_t handle

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **esp_periph_config_t**

Common peripherals configurations.

Public Members

int **task_stack**

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

int **task_prio**

Service task priority (based on freeRTOS priority)

int **task_core**

Service task running in core (0 or 1)

bool **extern_stack**

Service task stack allocate on extern ram

struct **esp_periph_event**

peripheral events

Public Members

void ***user_ctx***
peripheral context data

esp_periph_event_handle_t ***cb***
peripheral callback function

audio_event_iface_handle_t ***iface***
peripheral event

Macros

DEFAULT_ESP_PERIPH_STACK_SIZE

DEFAULT_ESP_PERIPH_TASK_PRIO

DEFAULT_ESP_PERIPH_TASK_CORE

DEFAULT_ESP_PERIPH_SET_CONFIG()

periph_tick_get

Type Definitions

typedef struct esp_periph_sets ***esp_periph_set_handle_t**

typedef struct esp_periph ***esp_periph_handle_t**

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

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

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

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

typedef struct *esp_periph_event* **esp_periph_event_t**
peripheral events

Enumerations

enum **esp_periph_id_t**

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

Values:

enumerator **PERIPH_ID_BUTTON**

enumerator **PERIPH_ID_TOUCH**

enumerator **PERIPH_ID_SDCARD**

enumerator **PERIPH_ID_WIFI**

enumerator **PERIPH_ID_FLASH**

enumerator **PERIPH_ID_AUXIN**

enumerator **PERIPH_ID_ADC**

enumerator **PERIPH_ID_CONSOLE**

enumerator **PERIPH_ID_BLUETOOTH**

enumerator **PERIPH_ID_LED**

enumerator **PERIPH_ID_SPIFFS**

enumerator **PERIPH_ID_ADC_BTN**

enumerator **PERIPH_ID_IS31FL3216**

enumerator **PERIPH_ID_GPIO_ISR**

enumerator **PERIPH_ID_WS2812**

enumerator **PERIPH_ID_AW2013**

enumerator **PERIPH_ID_LCD**

enumerator **PERIPH_ID_CUSTOM_BASE**

enum **esp_periph_state_t**

Peripheral working state.

Values:

enumerator **PERIPH_STATE_NULL**

enumerator **PERIPH_STATE_INIT**

enumerator **PERIPH_STATE_RUNNING**

enumerator **PERIPH_STATE_PAUSE**

enumerator **PERIPH_STATE_STOPPING**

enumerator **PERIPH_STATE_ERROR**

enumerator **PERIPH_STATE_STATUS_MAX**

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

2.8.2 Wi-Fi Peripheral

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

Application Example

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

API Reference

Header File

- components/esp_peripherals/include/periph_wifi.h

Functions

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

Create the wifi peripheral handle for esp_peripherals.

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

Parameters **config** – The configuration

Returns The esp peripheral handle

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

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

Parameters

- **periph** – **[in]** The periph
- **tick_to_wait** – **[in]** The tick to wait

Returns

- ESP_OK
- ESP_FAIL

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

Check the Wi-Fi connection status.

Parameters **periph** – **[in]** The periph

Returns Wi-Fi network status

esp_err_t **esp_wifi_set_listen_interval** (*esp_periph_handle_t* periph, int interval)

Set Wi-Fi listen interval for ESP32 station to receive beacon.

Parameters

- **periph** – **[in]** The wifi periph
- **interval** – **[in]** listen interval. units: AP beacon intervals(see BcnInt, default: 100ms)

Returns

- ESP_OK
- ESP_FAIL

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

Start Wi-Fi network setup in `mode`

Parameters

- **periph** – **[in]** The periph
- **mode** – **[in]** The mode

Returns

- ESP_OK
- ESP_FAIL

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

Wait for Wi-Fi setup done.

Parameters

- **periph** – **[in]** The periph
- **tick_to_wait** – **[in]** The tick to wait

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **periph_wpa2_enterprise_cfg_t**

The WPA2 enterprise peripheral configuration.

Public Members

bool **diasble_wpa2_e**

Disable wpa2 enterprise

int **eap_method**

TLS: 0, PEAP: 1, TTLS: 2

char ***ca_pem_start**

binary wpa2 ca pem start

char ***ca_pem_end**

binary wpa2 ca pem end

char ***wpa2_e_cert_start**

binary wpa2 cert start

char ***wpa2_e_cert_end**

binary wpa2 cert end

char ***wpa2_e_key_start**

binary wpa2 key start

char ***wpa2_e_key_end**

binary wpa2 key end

const char ***eap_id**

Identity in phase 1 of EAP procedure

const char *__eap_username__

Username for EAP method (PEAP and TTLS)

const char *__eap_password__

Password for EAP method (PEAP and TTLS)

struct __periph_wifi_cfg_t__

The Wi-Fi peripheral configuration.

Public Members

bool __disable_auto_reconnect__

Disable Wi-Fi auto reconnect

int __reconnect_timeout_ms__

The reconnect timeout after disconnected from Wi-Fi network

wifi_config_t __wifi_config__

Wifi configure

periph_wpa2_enterprise_cfg_t __wpa2_e_cfg__

wpa2 enterprise config

Enumerations

enum __periph_wifi_state_t__

Peripheral Wi-Fi event id.

Values:

enumerator __PERIPH_WIFI_UNCHANGE__

enumerator __PERIPH_WIFI_CONNECTING__

enumerator __PERIPH_WIFI_CONNECTED__

enumerator __PERIPH_WIFI_DISCONNECTED__

enumerator __PERIPH_WIFI_SETTING__

enumerator __PERIPH_WIFI_CONFIG_DONE__

enumerator __PERIPH_WIFI_CONFIG_ERROR__

enumerator **PERIPH_WIFI_ERROR**

enum **periph_wifi_config_mode_t**

Wi-Fi setup mode type.

Values:

enumerator **WIFI_CONFIG_ESPTOUCH**

Using smartconfig with ESPTOUCH protocol

enumerator **WIFI_CONFIG_AIRKISS**

Using smartconfig with AIRKISS protocol

enumerator **WIFI_CONFIG_ESPTOUCH_AIRKISS**

Using smartconfig with ESPTOUCH_AIRKISS protocol

enumerator **WIFI_CONFIG_WPS**

Using WPS (not support)

enumerator **WIFI_CONFIG_BLUEFI**

Using BLUEFI

enumerator **WIFI_CONFIG_WEB**

Using HTTP Server (not support)

2.8.3 SD Card Peripheral

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

Application Examples

Implementation of this API is demonstrated in couple of examples:

- player/pipeline_play_sdcard_music
- player/pipeline_sdcard_mp3_control
- recorder/pipeline_recording_to_sdcard
- recorder/pipeline_wav_amr_sdcard

API Reference

Header File

- components/esp_peripherals/include/periph_sdcard.h

Functions

esp_periph_handle_t **periph_sdcard_init** (*periph_sdcard_cfg_t* *sdcard_config)

Create the sdcard peripheral handle for esp_peripherals.

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

Parameters **sdcard_config** – The sdcard configuration

Returns The esp peripheral handle

bool **periph_sdcard_is_mounted** (*esp_periph_handle_t* periph)

Check the sdcard is mounted or not.

Parameters **periph** – **[in]** The periph

Returns SDCARD mounted state

Structures

struct **periph_sdcard_cfg_t**

The SD Card Peripheral configuration.

Public Members

int **card_detect_pin**

Card detect gpio number

const char ***root**

Base path for vfs

periph_sdcard_mode_t **mode**

card mode

Enumerations

enum **periph_sdcard_event_id_t**

Peripheral sdcard event id.

Values:

enumerator **SDCARD_STATUS_UNKNOWN**

No event

enumerator **SDCARD_STATUS_CARD_DETECT_CHANGE**

Detect changes in the card_detect pin

enumerator **SDCARD_STATUS_MOUNTED**

SDCARD mounted successfully

enumerator **SDCARD_STATUS_UNMOUNTED**

SDCARD unmounted successfully

enumerator **SDCARD_STATUS_MOUNT_ERROR**

SDCARD mount error

enumerator **SDCARD_STATUS_UNMOUNT_ERROR**

SDCARD unmount error

enum **periph_sdcard_mode_t**

SD card mode, SPI, 1-line SD mode, 4-line SD mode.

Values:

enumerator **SD_MODE_SPI**

sd_card SPI

enumerator **SD_MODE_1_LINE**

sd_card 1-line SD mode

enumerator **SD_MODE_4_LINE**

sd_card 4-line SD mode

enumerator **SD_MODE_8_LINE**

sd_card 8-line SD mode

enumerator **SD_MODE_MAX**

2.8.4 Spiffs Peripheral

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

Application Example

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

API Reference

Header File

- components/esp_peripherals/include/periph_spiffs.h

Functions

esp_periph_handle_t **periph_spiffs_init** (*periph_spiffs_cfg_t* *spiffs_config)

Create the spiffs peripheral handle for esp_peripherals.

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

---

Parameters **spiffs_config** – The spiffs configuration

Returns The esp peripheral handle

bool **periph_spiffs_is_mounted** (*esp_periph_handle_t* periph)

Check if the SPIFFS is mounted or not.

Parameters **periph** – **[in]** The periph

Returns SPIFFS mounted state

Structures

struct **periph_spiffs_cfg_t**

The SPIFFS Peripheral configuration.

Public Members

const char ***root**

Base path for vfs

const char ***partition_label**

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

size_t **max_files**

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

bool **format_if_mount_failed**

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

Enumerations

enum **periph_spiffs_event_id_t**

Peripheral spiffs event id.

Values:

enumerator **SPIFFS_STATUS_UNKNOWN**

No event

enumerator **SPIFFS_STATUS_MOUNTED**

SPIFFS mounted successfully

enumerator **SPIFFS_STATUS_UNMOUNTED**

SPIFFS unmounted successfully

enumerator **SPIFFS_STATUS_MOUNT_ERROR**

SPIFFS mount error

enumerator **SPIFFS_STATUS_UNMOUNT_ERROR**

SPIFFS unmount error

2.8.5 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

Please refer to cli/main/console_example.c.

API Reference

Header File

- components/esp_peripherals/include/periph_console.h

Functions

esp_periph_handle_t **periph_console_init** (*periph_console_cfg_t* *config)

Initialize Console Peripheral.

Parameters **config** – The configuration

Returns The esp peripheral handle

Structures

struct **periph_console_cmd_t**

Command structure.

Public Members

const char ***cmd**

Name of command, must be unique

int **id**

Command ID will be sent together when the command is matched

const char ***help**

Explanation of the command

console_cmd_callback_t **func**

Function callback for the command

struct **periph_console_cfg_t**

Console Peripheral configuration.

Public Members

int **command_num**

Total number of commands

const *periph_console_cmd_t* ***commands**

Pointer to array of commands

int **task_stack**

Console task stack, using default if the value is zero

int **task_prio**

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

int **buffer_size**

Size of console input buffer

const char ***prompt_string**

Console prompt string, using default CONSOLE_PROMPT_STRING if the pointer is NULL

Macros

CONSOLE_DEFAULT_TASK_PRIO

CONSOLE_DEFAULT_TASK_STACK

CONSOLE_DEFAULT_BUFFER_SIZE

CONSOLE_DEFAULT_PROMPT_STRING

Type Definitions

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

2.8.6 Touch Peripheral

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

Application Example

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

API Reference

Header File

- components/esp_peripherals/include/periph_touch.h

Functions

esp_periph_handle_t **periph_touch_init** (*periph_touch_cfg_t* *config)

Create the touch peripheral handle for esp_peripherals.

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

Parameters **config** – The configuration

Returns The esp peripheral handle

Structures

struct **periph_touch_cfg_t**

The Touch peripheral configuration.

Public Members

int **touch_mask**

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

int **tap_threshold_percent**

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

int **long_tap_time_ms**

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

Enumerations

enum **esp_touch_pad_sel_t**

Touch pad selection.

Values:

enumerator **TOUCH_PAD_SEL0**

enumerator **TOUCH_PAD_SEL1**

enumerator **TOUCH_PAD_SEL2**

enumerator **TOUCH_PAD_SEL3**

enumerator **TOUCH_PAD_SEL4**

enumerator **TOUCH_PAD_SEL5**

enumerator **TOUCH_PAD_SEL6**

enumerator **TOUCH_PAD_SEL7**

enumerator **TOUCH_PAD_SEL8**

enumerator **TOUCH_PAD_SEL9**

enum **periph_touch_event_id_t**

Peripheral touch event id.

Values:

enumerator **PERIPH_TOUCH_UNCHANGE**

No event

enumerator **PERIPH_TOUCH_TAP**

When touch pad is tapped

enumerator **PERIPH_TOUCH_RELEASE**

When touch pad is released after tap

enumerator **PERIPH_TOUCH_LONG_TAP**

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

enumerator **PERIPH_TOUCH_LONG_RELEASE**

When touch pad is released after long tap

2.8.7 Button Peripheral

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

Application Example

Implementation of this API is demonstrated in cloud_services/google_translate_device example.

API Reference

Header File

- components/esp_peripherals/include/periph_button.h

Functions

esp_periph_handle_t **periph_button_init** (*periph_button_cfg_t* *but_cfg)

Create the button peripheral handle for esp_peripherals.

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

Parameters **but_cfg** – The but configuration

Returns The esp peripheral handle

Structures

struct **periph_button_cfg_t**

The Button peripheral configuration.

Public Members

uint64_t **gpio_mask**

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

int **long_press_time_ms**

Long press duration in milliseconds, default is 2000ms

Enumerations

enum **periph_button_event_id_t**

Peripheral button event id.

Values:

enumerator **PERIPH_BUTTON_UNCHANGE**

No event

enumerator **PERIPH_BUTTON_PRESSED**

When button is pressed

enumerator **PERIPH_BUTTON_RELEASE**

When button is released

enumerator **PERIPH_BUTTON_LONG_PRESSED**

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

enumerator **PERIPH_BUTTON_LONG_RELEASE**

When button is released and event PERIPH_BUTTON_LONG_PRESSED happened

2.8.8 LED Peripheral

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

Application Examples

Implementation of this API is demonstrated in couple of examples:

- /cloud_services/google_translate_device
- /dueros

API Reference

Header File

- components/esp_peripherals/include/periph_led.h

Functions

esp_periph_handle_t **periph_led_init** (*periph_led_cfg_t* *config)

Create the LED peripheral handle for esp_peripherals.

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

Parameters **config** – The configuration

Returns The esp peripheral handle

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

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

Parameters

- **periph** – **[in]** The LED periph
- **gpio_num** – **[in]** The gpio number
- **time_on_ms** – **[in]** The time on milliseconds
- **time_off_ms** – **[in]** The time off milliseconds
- **fade** – **[in]** Fading enabled
- **loop** – **[in]** Loop
- **level** – **[in]** idle level

Returns

- ESP_OK
- ESP_FAIL

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

Stop Blink the LED.

Parameters

- **periph** – **[in]** The periph
- **gpio_num** – **[in]** The gpio number

Returns

- ESP_OK
- ESP_FAIL

Structures

struct **periph_led_cfg_t**

The LED peripheral configuration.

Public Members

ledc_mode_t **led_speed_mode**

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

ledc_timer_bit_t **led_duty_resolution**

LEDC channel duty resolution

ledc_timer_t **led_timer_num**

Select the timer source of channel (0 - 3)

uint32_t **led_freq_hz**

LEDC timer frequency (Hz)

int **gpio_num**

Optional, < 0 invalid gpio number

Enumerations

enum **periph_led_event_id_t**

Peripheral LED event id.

Values:

enumerator **PERIPH_LED_UNCHANGE**

No event

enumerator **PERIPH_LED_BLINK_FINISH**

When LED blink is finished

enum **periph_led_idle_level_t**

Peripheral LED idle output level.

Values:

enumerator **PERIPH_LED_IDLE_LEVEL_LOW**

Low level output

enumerator **PERIPH_LED_IDLE_LEVEL_HIGH**

High level output

2.8.9 ADC Button Peripheral

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

Application Examples

Implementation of this API is demonstrated in the following example:

- checks/check_board_buttons

API Reference

Header File

- components/esp_peripherals/include/periph_adc_button.h

Functions

esp_periph_handle_t **periph_adc_button_init** (*periph_adc_button_cfg_t* *btn_cfg)

Create the button peripheral handle for esp_peripherals.

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

Parameters **btn_cfg** – The button configuration.

Returns The esp peripheral handle.

Structures

struct **periph_adc_button_cfg_t**

The configuration of ADC Button.

Public Members

adc_arr_t ***arr**

An array with configuration of buttons

int **arr_size**

The array size

adc_btn_task_cfg_t **task_cfg**

Adc button task configuration

Macros

ADC_BUTTON_STACK_SIZE

ADC_BUTTON_TASK_PRIORITY

ADC_BUTTON_TASK_CORE_ID

PERIPH_ADC_BUTTON_DEFAULT_CONFIG()

ADC_DEFAULT_ARR()

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

Enumerations

enum **periph_adc_button_event_id_t**

Values:

enumerator **PERIPH_ADC_BUTTON_IDLE**

enumerator **PERIPH_ADC_BUTTON_PRESSED**

enumerator **PERIPH_ADC_BUTTON_RELEASE**

enumerator **PERIPH_ADC_BUTTON_LONG_PRESSED**

enumerator **PERIPH_ADC_BUTTON_LONG_RELEASE**

2.8.10 LED Controller Peripheral

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

Application Examples

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

API Reference

Header File

- components/esp_peripherals/include/periph_is31fl3216.h

Functions

esp_periph_handle_t **periph_is31fl3216_init** (*periph_is31fl3216_cfg_t* *is31fl3216_config)

Initializate the is31fl3216.

Parameters **is31fl3216_config** –

Returns

- ESP_OK Success
- ESP_FAIL Fail

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

Set the state of all the channels.

Parameters

- **periph** – The is31fl3216 handle
- **state** – The state of all channels

Returns

- ESP_OK Success
- ESP_FAIL Fail

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

Set the current enable channels.

Parameters

- **periph** – The is31fl3216 handle
- **blink_pattern** – The bit pattern of enabled channels

Returns

- ESP_OK Success
- ESP_FAIL Fail

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

Set the duty of the channel.

Parameters

- **periph** – The is31fl3216 handle
- **index** – The channel number

- **value** – The value of the channel's duty to be set

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **periph_is31fl3216_set_duty_step**(*esp_periph_handle_t* periph, uint8_t step)

Set the duty step of flash.

Parameters

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

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **periph_is31fl3216_set_interval**(*esp_periph_handle_t* periph, uint16_t interval_ms)

Set the internval time.

Parameters

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

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **periph_is31fl3216_set_shift_mode**(*esp_periph_handle_t* periph, *periph_is31_shift_mode_t* mode)

Set the shift mode.

Parameters

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

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **periph_is31fl3216_set_light_on_num**(*esp_periph_handle_t* periph, uint16_t light_on_num, uint16_t max_light_num)

Set the light on numbers.

Parameters

- **periph** – The is31fl3216 handle
- **light_on_num** – Enabled led number
- **max_light_num** – Maximum led number

Returns

- ESP_OK Success
- ESP_FAIL Fail

esp_err_t **periph_is31fl3216_set_act_time**(*esp_periph_handle_t* periph, uint16_t act_ms)

Set the action time.

Parameters

- **periph** – The is31fl3216 handle
- **act_ms** – Action time, unit is millisecond, 0 is infinite

Returns

- ESP_OK Success
- ESP_FAIL Fail

Structures

struct **periph_is31fl3216_cfg_t**

The configuration of is31fl3216.

Public Members

uint32_t **duty**[IS31FL3216_CH_NUM]

An array of the is31fl3216's duty

uint16_t **is31fl3216_pattern**

Current enable channel

periph_is31fl3216_state_t **state**

The state of all the channels

Macros

IS31FL3216_CH_NUM

BLUE_LED_MAX_NUM

Enumerations

enum **periph_is31fl3216_state_t**

Values:

enumerator **IS31FL3216_STATE_UNKNOWN**

enumerator **IS31FL3216_STATE_OFF**

enumerator **IS31FL3216_STATE_ON**

enumerator **IS31FL3216_STATE_FLASH**

enumerator **IS31FL3216_STATE_BY_AUDIO**

enumerator **IS31FL3216_STATE_SHIFT**

enum **periph_is31_shift_mode_t**

Values:

enumerator **PERIPH_IS31_SHIFT_MODE_UNKNOWN**

enumerator **PERIPH_IS31_SHIFT_MODE_ACC**

accumulation mode

enumerator **PERIPH_IS31_SHIFT_MODE_SINGLE**

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

2.9 Abstraction Layer

2.9.1 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. 7: 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/pipeline_sdcard_mp3_control example.

API Reference

Header File

- components/audio_pipeline/include/ringbuf.h

Functions

ringbuf_handle_t **rb_create** (int block_size, int n_blocks)

Create ringbuffer with total size = block_size * n_blocks.

Parameters

- **block_size** – **[in]** Size of each block
- **n_blocks** – **[in]** Number of blocks

Returns ringbuf_handle_t

esp_err_t **rb_destroy** (*ringbuf_handle_t* rb)

Cleanup and free all memory created by ringbuf_handle_t.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_abort**(*ringbuf_handle_t* rb)

Abort waiting until there is space for reading or writing of the ringbuffer.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_reset**(*ringbuf_handle_t* rb)

Reset ringbuffer, clear all values as initial state.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_reset_is_done_write**(*ringbuf_handle_t* rb)

Reset is_done_write flag.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

int **rb_bytes_available**(*ringbuf_handle_t* rb)

Get total bytes available of Ringbuffer.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns total bytes available

int **rb_bytes_filled**(*ringbuf_handle_t* rb)

Get the number of bytes that have filled the ringbuffer.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns The number of bytes that have filled the ringbuffer

int **rb_get_size**(*ringbuf_handle_t* rb)

Get total size of Ringbuffer (in bytes)

Parameters **rb** – **[in]** The Ringbuffer handle

Returns total size of Ringbuffer

int **rb_read**(*ringbuf_handle_t* rb, char *buf, int len, TickType_t ticks_to_wait)

Read from Ringbuffer to `buf` with len and wait `tick_to_wait` ticks until enough bytes to read if the ringbuffer bytes available is less than `len`. If `buf` argument provided is `NULL`, then ringbuffer do pseudo reads by simply advancing pointers.

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **buf** – The buffer pointer to read out data
- **len** – **[in]** The length request

- **ticks_to_wait** – **[in]** The ticks to wait

Returns Number of bytes read

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`

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **buf** – The buffer
- **len** – **[in]** The length
- **ticks_to_wait** – **[in]** The ticks to wait

Returns Number of bytes written

esp_err_t **rb_done_write**(*ringbuf_handle_t* rb)

Set status of writing to ringbuffer is done.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_unblock_reader**(*ringbuf_handle_t* rb)

Unblock from rb_read.

Parameters **rb** – **[in]** The Ringbuffer handle

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_set_reader_holder**(*ringbuf_handle_t* rb, void *holder)

Set the owner of the 'rb_read'.

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **holder** – **[in]** The owner of the 'rb_read'

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_get_reader_holder**(*ringbuf_handle_t* rb, void **holder)

Get the owner of the 'rb_read'.

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **holder** – **[out]** The owner of the 'rb_read'

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_set_writer_holder**(*ringbuf_handle_t* rb, void *holder)

Set the owner of the 'rb_write'.

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **holder** – **[in]** The owner of the 'rb_write'

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **rb_get_writer_holder**(*ringbuf_handle_t* rb, void **holder)

Get the owner of the 'rb_write'.

Parameters

- **rb** – **[in]** The Ringbuffer handle
- **holder** – **[out]** The owner of the 'rb_write'

Returns

- ESP_OK
- ESP_FAIL

Macros

RB_OK

RB_FAIL

RB_DONE

RB_ABORT

RB_TIMEOUT

Type Definitions

typedef struct ringbuf ***ringbuf_handle_t**

2.9.2 Audio HAL

Abstraction layer for audio board hardware, serves as an interface between the user application and the hardware driver for specific audio board like *ESP32 LyraT*.

The API provides data structures to configure sampling rates of ADC and DAC signal conversion, data bit widths, I2C stream parameters, and selection of signal channels connected to ADC and DAC. It also contains several specific functions to e.g. initialize the audio board, `audio_hal_init()`, control the volume, `audio_hal_get_volume()` and `audio_hal_set_volume()`.

API Reference

Header File

- components/audio_hal/include/audio_hal.h

Functions

audio_hal_handle_t **audio_hal_init** (*audio_hal_codec_config_t* *audio_hal_conf, *audio_hal_func_t* *audio_hal_func)

Initialize media codec driver.

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

Parameters

- **audio_hal_conf** – Configure structure audio_hal_config_t
- **audio_hal_func** – Structure containing functions used to operate audio the codec chip

Returns int, 0—success, others—fail

esp_err_t **audio_hal_deinit** (*audio_hal_handle_t* audio_hal)

Uninitialize media codec driver.

Parameters **audio_hal** – reference function pointer for selected audio codec

Returns int, 0—success, others—fail

esp_err_t **audio_hal_ctrl_codec** (*audio_hal_handle_t* audio_hal, *audio_hal_codec_mode_t* mode, *audio_hal_ctrl_t* audio_hal_ctrl)

Start/stop codec driver.

Parameters

- **audio_hal** – reference function pointer for selected audio codec
- **mode** – select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t
- **audio_hal_ctrl** – select start stop state for specific mode

Returns int, 0—success, others—fail

esp_err_t **audio_hal_codec_iface_config**(*audio_hal_handle_t* audio_hal, *audio_hal_codec_mode_t* mode, *audio_hal_codec_i2s_iface_t* *iface)

Set codec I2S interface samples rate & bit width and format either I2S or PCM/DSP.

Parameters

- **audio_hal** – reference function pointer for selected audio codec
- **mode** – select media hal codec mode either encode/decode/or both to start from audio_hal_codec_mode_t
- **iface** – I2S sample rate (ex: 16000, 44100), I2S bit width (16, 24, 32),I2s format (I2S, PCM, DSP).

Returns

- 0 Success
- -1 Error

esp_err_t **audio_hal_set_mute**(*audio_hal_handle_t* audio_hal, bool mute)

Set voice mute. Enables or disables DAC mute of a codec.

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

Parameters

- **audio_hal** – reference function pointer for selected audio codec
- **mute** – true/false. If true speaker will be muted and if false speaker will be unmuted.

Returns int, 0—success, others—fail

esp_err_t **audio_hal_set_volume**(*audio_hal_handle_t* audio_hal, int volume)

Set voice volume.

Note: if volume is 0, mute is enabled,range is 0-100.

Parameters

- **audio_hal** – reference function pointer for selected audio codec
- **volume** – value of volume in percent(%)

Returns int, 0—success, others—fail

esp_err_t **audio_hal_get_volume**(*audio_hal_handle_t* audio_hal, int *volume)

get voice volume.

Note: if volume is 0, mute is enabled, range is 0-100.

Parameters

- **audio_hal** – reference function pointer for selected audio codec

- **volume** – value of volume in percent returned(%)

Returns int, 0—success, others—fail

esp_err_t **audio_hal_enable_pa**(*audio_hal_handle_t* audio_hal, bool enable)

Enables or disables PA.

Parameters

- **audio_hal** – reference function pointer for selected audio codec
- **enable** – true/false.

Returns int, 0—success, others—fail

Structures

struct **audio_hal_codec_i2s_iface_t**

I2s interface configuration for audio codec chip.

Public Members

audio_hal_iface_mode_t **mode**

audio codec chip mode

audio_hal_iface_format_t **fmt**

I2S interface format

audio_hal_iface_samples_t **samples**

I2S interface samples per second

audio_hal_iface_bits_t **bits**

i2s interface number of bits per sample

struct **audio_hal_codec_config_t**

Configure media hal for initialization of audio codec chip.

Public Members

audio_hal_adc_input_t **adc_input**

set adc channel

audio_hal_dac_output_t **dac_output**

set dac channel

audio_hal_codec_mode_t **codec_mode**

select codec mode: adc, dac or both

audio_hal_codec_i2s_iface_t **i2s_iface**

set I2S interface configuration

struct **audio_hal**

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

Public Members

esp_err_t (***audio_codec_initialize**)(*audio_hal_codec_config_t* *codec_cfg)

initialize codec

esp_err_t (***audio_codec_deinitialize**)(void)

deinitialize codec

esp_err_t (***audio_codec_ctrl**)(*audio_hal_codec_mode_t* mode, *audio_hal_ctrl_t* ctrl_state)

control codec mode and state

esp_err_t (***audio_codec_config_iface**)(*audio_hal_codec_mode_t* mode, *audio_hal_codec_i2s_iface_t* *iface)

configure i2s interface

esp_err_t (***audio_codec_set_mute**)(bool mute)

set codec mute

esp_err_t (***audio_codec_set_volume**)(int volume)

set codec volume

esp_err_t (***audio_codec_get_volume**)(int *volume)

get codec volume

esp_err_t (***audio_codec_enable_pa**)(bool enable)

enable pa

xSemaphoreHandle **audio_hal_lock**

semaphore of codec

void ***handle**

handle of audio codec

Macros

AUDIO_HAL_VOL_DEFAULT

Type Definitions

typedef struct *audio_hal* ***audio_hal_handle_t**

typedef struct *audio_hal* **audio_hal_func_t**

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

Enumerations

enum **audio_hal_codec_mode_t**

Select media hal codec mode.

Values:

enumerator **AUDIO_HAL_CODEC_MODE_ENCODE**

select adc

enumerator **AUDIO_HAL_CODEC_MODE_DECODE**

select dac

enumerator **AUDIO_HAL_CODEC_MODE_BOTH**

select both adc and dac

enumerator **AUDIO_HAL_CODEC_MODE_LINE_IN**

set adc channel

enum **audio_hal_adc_input_t**

Select adc channel for input mic signal.

Values:

enumerator **AUDIO_HAL_ADC_INPUT_LINE1**

mic input to adc channel 1

enumerator **AUDIO_HAL_ADC_INPUT_LINE2**

mic input to adc channel 2

enumerator **AUDIO_HAL_ADC_INPUT_ALL**

mic input to both channels of adc

enumerator **AUDIO_HAL_ADC_INPUT_DIFFERENCE**

mic input to adc difference channel

enum **audio_hal_dac_output_t**

Select channel for dac output.

Values:

enumerator **AUDIO_HAL_DAC_OUTPUT_LINE1**

dac output signal to channel 1

enumerator **AUDIO_HAL_DAC_OUTPUT_LINE2**

dac output signal to channel 2

enumerator **AUDIO_HAL_DAC_OUTPUT_ALL**

dac output signal to both channels

enum **audio_hal_ctrl_t**

Select operating mode i.e. start or stop for audio codec chip.

Values:

enumerator **AUDIO_HAL_CTRL_STOP**

set stop mode

enumerator **AUDIO_HAL_CTRL_START**

set start mode

enum **audio_hal_iface_mode_t**

Select I2S interface operating mode i.e. master or slave for audio codec chip.

Values:

enumerator **AUDIO_HAL_MODE_SLAVE**

set slave mode

enumerator **AUDIO_HAL_MODE_MASTER**

set master mode

enum **audio_hal_iface_samples_t**

Select I2S interface samples per second.

Values:

enumerator **AUDIO_HAL_08K_SAMPLES**

set to 8k samples per second

enumerator **AUDIO_HAL_11K_SAMPLES**
set to 11.025k samples per second

enumerator **AUDIO_HAL_16K_SAMPLES**
set to 16k samples in per second

enumerator **AUDIO_HAL_22K_SAMPLES**
set to 22.050k samples per second

enumerator **AUDIO_HAL_24K_SAMPLES**
set to 24k samples in per second

enumerator **AUDIO_HAL_32K_SAMPLES**
set to 32k samples in per second

enumerator **AUDIO_HAL_44K_SAMPLES**
set to 44.1k samples per second

enumerator **AUDIO_HAL_48K_SAMPLES**
set to 48k samples per second

enum **audio_hal_iface_bits_t**
Select I2S interface number of bits per sample.
Values:

enumerator **AUDIO_HAL_BIT_LENGTH_16BITS**
set 16 bits per sample

enumerator **AUDIO_HAL_BIT_LENGTH_24BITS**
set 24 bits per sample

enumerator **AUDIO_HAL_BIT_LENGTH_32BITS**
set 32 bits per sample

enum **audio_hal_iface_format_t**
Select I2S interface format for audio codec chip.
Values:

enumerator **AUDIO_HAL_I2S_NORMAL**
set normal I2S format

enumerator **AUDIO_HAL_I2S_LEFT**
set all left format

enumerator **AUDIO_HAL_I2S_RIGHT**

set all right format

enumerator **AUDIO_HAL_I2S_DSP**

set dsp/pcm format

2.9.3 ES8388 Driver

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

API Reference

Header File

- components/audio_hal/driver/es8388/es8388.h

Functions

esp_err_t **es8388_init**(*audio_hal_codec_config_t* *cfg)

Initialize ES8388 codec chip.

Parameters **cfg** – configuration of ES8388

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_deinit**(void)

Deinitialize ES8388 codec chip.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_config_fmt**(es_module_t mod, es_i2s_fmt_t cfg)

Configure ES8388 I2S format.

Parameters

- **mod** – set ADC or DAC or both
- **cfg** – ES8388 I2S format

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_i2s_config_clock**(es_i2s_clock_t cfg)

Configure I2s clock in MSATER mode.

Parameters **cfg** – set bits clock and WS clock

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_set_bits_per_sample**(es_module_t mode, es_bits_length_t bit_per_sample)

Configure ES8388 data sample bits.

Parameters

- **mode** – set ADC or DAC or both
- **bit_per_sample** – bit number of per sample

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_start**(es_module_t mode)

Start ES8388 codec chip.

Parameters **mode** – set ADC or DAC or both

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_stop**(es_module_t mode)

Stop ES8388 codec chip.

Parameters **mode** – set ADC or DAC or both

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_set_voice_volume**(int volume)

Set voice volume.

Parameters **volume** – voice volume (0~100)

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8388_get_voice_volume**(int *volume)

Get voice volume.

Parameters ***volume** – **[out]** voice volume (0~100)

Returns

- ESP_OK

- ESP_FAIL

esp_err_t **es8388_set_voice_mute**(bool enable)

Configure ES8388 DAC mute or not. Basically you can use this function to mute the output or unmute.

Parameters **enable** – enable(1) or disable(0)

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_get_voice_mute**(void)

Get ES8388 DAC mute status.

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_set_mic_gain**(es_mic_gain_t gain)

Set ES8388 mic gain.

Parameters **gain** – db of mic gain

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_config_adc_input**(es_adc_input_t input)

Set ES8388 adc input mode.

Parameters **input** – adc input mode

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_config_dac_output**(es_dac_output_t output)

Set ES8388 dac output mode.

Parameters **output** – dac output mode

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_write_reg**(uint8_t reg_add, uint8_t data)

Write ES8388 register.

Parameters

- **reg_add** – address of register
- **data** – data of register

Returns

- ESP_FAIL Parameter error

- ESP_OK Success

void **es8388_read_all**(void)

Print all ES8388 registers.

esp_err_t **es8388_config_i2s**(*audio_hal_codec_mode_t* mode, *audio_hal_codec_i2s_iface_t* *iface)

Configure ES8388 codec mode and I2S interface.

Parameters

- **mode** – codec mode
- **iface** – I2S config

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_ctrl_state**(*audio_hal_codec_mode_t* mode, *audio_hal_ctrl_t* ctrl_state)

Control ES8388 codec chip.

Parameters

- **mode** – codec mode
- **ctrl_state** – start or stop decode or encode progress

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8388_pa_power**(bool enable)

Set ES8388 PA power.

Parameters **enable** – true for enable PA power, false for disable PA power

Returns

- ESP_ERR_INVALID_ARG
- ESP_OK

Macros

ES8388_ADDR

0x22:CE=1;0x20:CE=0

ES8388_CONTROL1

ES8388_CONTROL2

ES8388_CHIPPOWER

ES8388_ADCPOWER

ES8388_DACPOWER

ES8388_CHIPLOPOW1

ES8388_CHIPLOPOW2

ES8388_ANAVOLMANAG

ES8388_MASTERMODE

ES8388_ADCCONTROL1

ES8388_ADCCONTROL2

ES8388_ADCCONTROL3

ES8388_ADCCONTROL4

ES8388_ADCCONTROL5

ES8388_ADCCONTROL6

ES8388_ADCCONTROL7

ES8388_ADCCONTROL8

ES8388_ADCCONTROL9

ES8388_ADCCONTROL10

ES8388_ADCCONTROL11

ES8388_ADCCONTROL12

ES8388_ADCCONTROL13

ES8388_ADCCONTROL14

ES8388_DACCONTROL1

ES8388_DACCONTROL2

ES8388_DACCONTROL3

ES8388_DACCONTROL4

ES8388_DACCONTROL5

ES8388_DACCONTROL6

ES8388_DACCONTROL7

ES8388_DACCONTROL8

ES8388_DACCONTROL9

ES8388_DACCONTROL10

ES8388_DACCONTROL11

ES8388_DACCONTROL12

ES8388_DACCONTROL13

ES8388_DACCONTROL14

ES8388_DACCONTROL15

ES8388_DACCONTROL16

ES8388_DACCONTROL17

ES8388_DACCONTROL18

ES8388_DACCONTROL19

ES8388_DACCONTROL20

ES8388_DACCONTROL21

ES8388_DACCONTROL22

ES8388_DACCONTROL23

ES8388_DACCONTROL24

ES8388_DACCONTROL25

ES8388_DACCONTROL26

ES8388_DACCONTROL27

ES8388_DACCONTROL28

ES8388_DACCONTROL29

ES8388_DACCONTROL30

2.9.4 ES8374 Driver

Driver for ES8374 codec chip.

API Reference

Header File

- components/audio_hal/driver/es8374/es8374.h

Functions

esp_err_t **es8374_codec_init**(*audio_hal_codec_config_t* *cfg)

Initialize ES8374 codec chip.

Parameters **cfg** – configuration of ES8374

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_codec_deinit**(void)

Deinitialize ES8374 codec chip.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_config_fmt**(es_module_t mode, es_i2s_fmt_t fmt)

Configure ES8374 I2S format.

Parameters

- **mode** – Set ADC or DAC or both

- **fmt** – ES8374 I2S format

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_i2s_config_clock**(es_i2s_clock_t cfg)

Configure I2S clock in MSATER mode.

Parameters **cfg** – Set bits clock and WS clock

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_set_bits_per_sample**(es_module_t mode, es_bits_length_t bit_per_sample)

Configure ES8374 data sample bits.

Parameters

- **mode** – Set ADC or DAC or both
- **bit_per_sample** – Bit number of per sample

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_start**(es_module_t mode)

Start ES8374 codec chip.

Parameters **mode** – Set ADC or DAC or both

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_stop**(es_module_t mode)

Stop ES8374 codec chip.

Parameters **mode** – Set ADC or DAC or both

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_codec_set_voice_volume**(int volume)

Set voice volume.

Parameters **volume** – Voice volume (0~100)

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_codec_get_voice_volume**(int *volume)

Get voice volume.

Parameters ***volume** – **[out]** Voice volume (0~100)

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **es8374_set_voice_mute**(bool enable)

Mute or unmute ES8374 DAC. Basically you can use this function to mute or unmute the output.

Parameters **enable** – **[inout]** Enable Mute(1) or Unmute(0)

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_get_voice_mute**(void)

Get ES8374 DAC mute status.

Returns

- ESP_FAIL
- ESP_OK

esp_err_t **es8374_set_mic_gain**(es_mic_gain_t gain)

Set ES8374 mic gain.

Parameters **gain** – **[inout]** gain db of mic gain

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_config_adc_input**(es_adc_input_t input)

Set ES8374 ADC input mode.

Parameters **input** – **[in]** ADC input mode

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_config_dac_output**(es_dac_output_t output)

Set ES8374 DAC output mode.

Parameters **output** – **[in]** DAC output mode

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_write_reg**(uint8_t reg_add, uint8_t data)

Write ES8374 register.

Parameters

- **reg_add** – Address of register
- **data** – Data of register

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

void **es8374_read_all**(void)

Print all ES8374 registers.

esp_err_t **es8374_codec_config_i2s**(*audio_hal_codec_mode_t* mode, *audio_hal_codec_i2s_iface_t* *iface)

Configure ES8374 codec mode and I2S interface.

Parameters

- **mode** – Codec mode
- **iface** – I2S config

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_codec_ctrl_state**(*audio_hal_codec_mode_t* mode, *audio_hal_ctrl_t* ctrl_state)

Control ES8374 codec chip.

Parameters

- **mode** – Codec mode
- **ctrl_state** – Start or stop decode or encode progress

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **es8374_pa_power**(bool enable)

Set ES8374 PA power.

Parameters **enable** – True for enable PA power, false for disable PA power

Returns

- ESP_ERR_INVALID_ARG
- ESP_OK

Macros

ES8374_ADDR

2.9.5 ZL38063 Driver

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

API Reference

Header File

- components/audio_hal/driver/zl38063/zl38063.h

Functions

esp_err_t **zl38063_codec_init**(*audio_hal_codec_config_t* *cfg)

Initialize ZL38063 chip.

Parameters **cfg** – configuration of ZL38063

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **zl38063_codec_deinit**(void)

Deinitialize ZL38063 chip.

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **zl38063_codec_ctrl_state**(*audio_hal_codec_mode_t* mode, *audio_hal_ctrl_t* ctrl_state)

Control ZL38063 chip.

The functions zl38063_ctrl_state and zl38063_config_i2s are not used by this driver. They are kept here to maintain the uniformity and convenience of the interface of the ADF project. These settings for zl38063 are burned in firmware and configuration files. Default i2s configuration: 48000Hz, 16bit, Left-Right channels. Use resampling to be compatible with different file types.

Parameters

- **mode** – codec mode
- **ctrl_state** – start or stop decode or encode progress

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **zl38063_codec_config_i2s**(*audio_hal_codec_mode_t* mode, *audio_hal_codec_i2s_iface_t* *iface)

Configure ZL38063 codec mode and I2S interface.

Parameters

- **mode** – codec mode
- **iface** – I2S config

Returns

- ESP_FAIL Parameter error
- ESP_OK Success

esp_err_t **zl38063_codec_set_voice_mute**(bool mute)

mute or unmute the codec

Parameters **mute** – true, false

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **zl38063_codec_set_voice_volume**(int volume)

Set voice volume.

Parameters **volume** – voice volume (0~100)

Returns

- ESP_OK
- ESP_FAIL

esp_err_t **zl38063_codec_get_voice_volume**(int *volume)

Get voice volume.

Parameters ***volume** – **[out]** voice volume (0~100)

Returns

- ESP_OK
- ESP_FAIL

2.10 Configuration Options

Compile-time configuration options specific to ESP-ADF.

2.10.1 ADF Features

ESP_DISPATCHER_DELEGATE_TASK_CORE

Delegation task core

Found in: ADF Features

Pinned delegate task to core 0 or core 1.

ESP_DISPATCHER_DELEGATE_TASK_PRIO

Delegate task's prio

Found in: ADF Features

The delegate task's prio.

ESP_DISPATCHER_DELEGATE_STACK_SIZE

Delegate task's stack size

Found in: ADF Features

The delegate task's stack is located in DRAM, modify this size to make sure all the needed operation can be run success in the it.

2.10.2 Audio HAL

AUDIO_BOARD

Audio board

Found in: Audio HAL

Select an audio board to use with the ESP-ADF

Available options:

- AUDIO_BOARD_CUSTOM
- ESP_LYRAT_V4_3_BOARD
- ESP_LYRAT_V4_2_BOARD
- ESP_LYRATD_MSC_V2_1_BOARD
- ESP_LYRATD_MSC_V2_2_BOARD
- ESP_LYRAT_MINI_V1_1_BOARD
- ESP32_KORVO_DU1906_BOARD
- ESP32_S2_KALUGA_1_V1_2_BOARD
- ESP32_S3_KORVO2_V3_BOARD
- ESP32_S3_KORVO2L_V1_BOARD
- ESP32_S3_BOX_LITE_BOARD

- ESP32_S3_BOX_BOARD
- ESP32_S3_BOX_3_BOARD
- ESP32_C3_LYRA_V2_BOARD
- ESP32_C6_DEVKIT_BOARD
- ESP32_P4_FUNCTION_EV_BOARD

ESP32_KORVO_DU1906_DAC

ESP32 KORVO DU1906 Board DAC chip

Found in: Audio HAL

Select DAC chip to use on ESP32_KORVO_DU1906 board

Available options:

- ESP32_KORVO_DU1906_DAC_TAS5805M
- ESP32_KORVO_DU1906_DAC_ES7148

ESP32_KORVO_DU1906_ADC

ESP32 KORVO DU1906 Board ADC chip

Found in: Audio HAL

Select ADC chip to use on ESP32_KORVO_DU1906 board

Available options:

- ESP32_KORVO_DU1906_ADC_ES7243

2.10.3 ADF Library Configuration

MEDIA_PROTOCOL_LIB_ENABLE

Enable Media Protocol Library

Found in: ADF Library Configuration

MEDIA_LIB_MEM_AUTO_TRACE

Support trace memory automatically after media_lib_sal init

Found in: ADF Library Configuration

MEDIA_LIB_MEM_TRACE_DEPTH

Memory trace stack depth

Found in: ADF Library Configuration

Set memory trace depth

MEDIA_LIB_MEM_TRACE_NUM

Memory trace number

Found in: ADF Library Configuration

Set memory trace number

MEDIA_LIB_MEM_TRACE_MODULE

Trace for module memory usage

Found in: ADF Library Configuration

MEDIA_LIB_MEM_TRACE_LEAKAGE

Trace for memory leakage

Found in: ADF Library Configuration

MEDIA_LIB_MEM_TRACE_SAVE_HISTORY

Trace to save memory history

Found in: ADF Library Configuration

MEDIA_LIB_MEM_SAVE_CACHE_SIZE

Cache buffer size to store save history

Found in: ADF Library Configuration

Set cache size for memory history

MEDIA_LIB_MEM_TRACE_SAVE_PATH

Memory trace save path

Found in: ADF Library Configuration

Set memory trace save path

2.10.4 ESP Speech Recognition

MODEL_DATA_PATH

model data path

Found in: ESP Speech Recognition

Available options:

- MODEL_IN_FLASH
- MODEL_IN_SDCARD

USE_AFE

use afe

Found in: ESP Speech Recognition

AFE_INTERFACE_SEL

Afe interface

Found in: ESP Speech Recognition

Select the afe interface to be used.

Available options:

- AFE_INTERFACE_V1

USE_NSNET

use nsnet

Found in: ESP Speech Recognition

SR_NSN_MODEL_LOAD

Select deep noise suppression

Found in: ESP Speech Recognition

Select the deep noise suppression to be loaded.

Available options:

- SR_NSN_NONE
- SR_NSN_NSNET1
- SR_NSN_NSNET2

USE_WAKENET

use wakenet

Found in: ESP Speech Recognition

SR_WN_MODEL_LOAD

Select wake words

Found in: ESP Speech Recognition

Select the Wake Words to be loaded.

Available options:

- SR_WN_WN5_HILEXIN
- SR_WN_WN5X3_HILEXIN
- SR_WN_WN5_NIHAOXIAOZHI
- SR_WN_WN5X3_NIHAOXIAOZHI
- SR_WN_WN5X3_NIHAOXIAOXIN
- SR_WN_WN8_ALEXA
- SR_WN_WN9_HILEXIN
- SR_WN_WN9_XIAOAITONGXUE
- SR_WN_WN9_ALEXA
- SR_WN_WN9_HIESP
- SR_WN_WN9_HIMFIVE
- SR_WN_WN9_NIHAOXIAOZHI_TTS
- SR_WN_WN9_JARVIS_TTS
- SR_WN_WN9_COMPUTER_TTS
- SR_WN_WN9_HEYWILLOW_TTS
- SR_WN_WN9_SOPHIA_TTS
- SR_WN_WN9_NIHAOXIAOXIN_TTS
- SR_WN_WN9_XIAOMEITONGXUE_TTS
- SR_WN_WN9_HIXIAOXING_TTS
- SR_WN_WN9_MYCROFT_TTS
- SR_WN_WN9_HEYPRINTER_TTS
- SR_WN_WN9_XIAOLONGXIAOLONG_TTS
- SR_WN_WN9_MIAOMIAOTONGXUE_TTS
- SR_WN_WN9_HIJOY_TTS
- SR_WN_WN9_HILILI_TTS
- SR_WN_WN9_HITELLY_TTS
- SR_WN_WN9_HEYWANDA_TTS

- SR_WN_WN9_HIMIAOMIAO_TTS
- SR_WN_WN9_XIAOBINXIAOBIN_TTS
- SR_WN_WN9_ASTROLABE_TTS
- SR_WN_WN9_CUSTOMWORD
- SR_WN_LOAD_MULIT_WORD

Load Multiple Wake Words

SR_WN_WN9_HILEXIN_MULTI

Hi,□□ (wn9_hilexin)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_XIAOAITONGXUE_MULTI

□□□□ (wn9_xiaoaitongxue)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_NIHAOXIAOZHI_TTS_MULTI

□□□□ (wn9_nihaoxiaozhi_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_ALEXA_MULTI

Alexa (wn9_alexa)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HIESP_MULTI

Hi,ESP (wn9_hiesp)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_JARVIS_TTS_MULTI

Jarvis (wn9_jarvis_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_COMPUTER_TTS_MULTI

computer (wn9_computer_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HEYWILLOW_TTS_MULTI

Hey,Willow (wn9_heywillow_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_SOPHIA_TTS_MULTI

Sophia (wn9_sophia_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_NIHAOXIAOXIN_TTS_MULTI

□□□□ (wn9_nihaoxiaoxin_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_XIAOMEITONGXUE_TTS_MULTI

□□□□ (wn9_xiaomeitongxue_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HEYPRINTER_TTS_MULTI

Hey,Printer (wn9_heyprinter_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_XIAOLONGXIAOLONG_TTS_MULTI

□□□□ (wn9_xiaolongxiaolong_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_MIAOMIAOTONGXUE_TTS_MULTI

□□□□ (wn9_miaomiaotongxue_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HEYWANDA_TTS_MULTI

Hey,Wanda (wn9_heywanda_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HIMIAOMIAO_TTS_MULTI

Hi,□□ (wn9_himiaomiao_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_MYCROFT_TTS_MULTI

Mycroft (wn9_mycroft_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HIJOY_TTS_MULTI

Hi,Joy (wn9_hijoy_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HILILI_TTS_MULTI

Hi,Lily/Hi,□□ (wn9_hilili_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_HITELLY_TTS_MULTI

Hi,Telly/Hi,□□ (wn9_hitelly_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_XIAOBINXIAOBIN_TTS_MULTI

□□□□/□□□□ (wn9_xiaobinxiaobin_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

SR_WN_WN9_ASTROLABE_TTS_MULTI

Astrolabe (wn9_astrolabe_tts)

Found in: ESP Speech Recognition > Load Multiple Wake Words

USE_MULTINET

use multinet

Found in: ESP Speech Recognition

CHINESE_SR_MN_MODEL_SEL

Chinese Speech Commands Model

Found in: ESP Speech Recognition

Select the Wake Word Engine to be used.

Available options:

- SR_MN_CN_NONE
- SR_MN_CN_MULTINET2_SINGLE_RECOGNITION
- SR_MN_CN_MULTINET5_RECOGNITION_QUANT8
- SR_MN_CN_MULTINET6_QUANT
- SR_MN_CN_MULTINET6_AC_QUANT
- SR_MN_CN_MULTINET7_QUANT
- SR_MN_CN_MULTINET7_AC_QUANT

ENGLISH_SR_MN_MODEL_SEL

English Speech Commands Model

Found in: ESP Speech Recognition

Select the Wake Word Engine to be used.

Available options:

- SR_MN_EN_NONE
- SR_MN_EN_MULTINET5_SINGLE_RECOGNITION_QUANT8
- SR_MN_EN_MULTINET6_QUANT
- SR_MN_EN_MULTINET7_QUANT

Add Chinese speech commands

CN_SPEECH_COMMAND_ID0

ID0

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID1

ID1

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID2

ID2

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID3

ID3

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID4

ID4

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID5

ID5

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID6

ID6

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID7

ID7

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID8

ID8

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID9

ID9

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID10

ID10

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID11

ID11

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID12

ID12

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID13

ID13

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID14

ID14

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID15

ID15

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID16

ID16

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID17

ID17

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID18

ID18

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID19

ID19

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID20

ID20

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID21

ID21

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID22

ID22

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID23

ID23

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID24

ID24

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID25

ID25

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID26

ID26

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID27

ID27

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID28

ID28

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID29

ID29

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID30

ID30

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID31

ID31

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID32

ID32

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID33

ID33

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID34

ID34

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID35

ID35

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID36

ID36

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID37

ID37

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID38

ID38

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID39

ID39

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID40

ID40

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID41

ID41

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID42

ID42

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID43

ID43

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID44

ID44

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID45

ID45

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID46

ID46

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID47

ID47

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID48

ID48

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID49

ID49

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID50

ID50

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID51

ID51

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID52

ID52

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID53

ID53

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID54

ID54

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID55

ID55

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID56

ID56

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID57

ID57

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID58

ID58

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID59

ID59

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID60

ID60

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID61

ID61

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID62

ID62

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID63

ID63

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID64

ID64

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID65

ID65

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID66

ID66

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID67

ID67

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID68

ID68

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID69

ID69

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID70

ID70

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID71

ID71

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID72

ID72

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID73

ID73

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID74

ID74

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID75

ID75

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID76

ID76

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID77

ID77

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID78

ID78

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID79

ID79

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID80

ID80

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID81

ID81

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID82

ID82

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID83

ID83

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID84

ID84

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID85

ID85

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID86

ID86

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID87

ID87

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID88

ID88

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID89

ID89

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID90

ID90

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID91

ID91

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID92

ID92

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID93

ID93

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID94

ID94

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID95

ID95

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID96

ID96

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID97

ID97

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID98

ID98

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID99

ID99

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID100

ID100

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID101

ID101

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID102

ID102

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID103

ID103

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID104

ID104

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID105

ID105

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID106

ID106

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID107

ID107

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID108

ID108

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID109

ID109

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID110

ID110

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID111

ID111

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID112

ID112

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID113

ID113

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID114

ID114

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID115

ID115

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID116

ID116

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID117

ID117

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID118

ID118

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID119

ID119

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID120

ID120

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID121

ID121

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID122

ID122

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID123

ID123

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID124

ID124

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID125

ID125

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID126

ID126

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID127

ID127

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID128

ID128

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID129

ID129

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID130

ID130

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID131

ID131

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID132

ID132

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID133

ID133

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID134

ID134

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID135

ID135

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID136

ID136

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID137

ID137

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID138

ID138

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID139

ID139

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID140

ID140

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID141

ID141

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID142

ID142

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID143

ID143

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID144

ID144

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID145

ID145

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID146

ID146

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID147

ID147

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID148

ID148

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID149

ID149

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID150

ID150

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID151

ID151

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID152

ID152

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID153

ID153

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID154

ID154

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID155

ID155

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID156

ID156

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID157

ID157

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID158

ID158

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID159

ID159

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID160

ID160

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID161

ID161

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID162

ID162

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID163

ID163

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID164

ID164

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID165

ID165

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID166

ID166

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID167

ID167

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID168

ID168

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID169

ID169

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID170

ID170

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID171

ID171

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID172

ID172

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID173

ID173

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID174

ID174

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID175

ID175

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID176

ID176

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID177

ID177

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID178

ID178

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID179

ID179

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID180

ID180

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID181

ID181

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID182

ID182

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID183

ID183

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID184

ID184

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID185

ID185

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID186

ID186

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID187

ID187

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID188

ID188

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID189

ID189

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID190

ID190

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID191

ID191

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID192

ID192

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID193

ID193

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID194

ID194

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID195

ID195

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID196

ID196

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID197

ID197

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID198

ID198

Found in: ESP Speech Recognition > Add Chinese speech commands

CN_SPEECH_COMMAND_ID199

ID199

Found in: ESP Speech Recognition > Add Chinese speech commands

Add English speech commands

EN_SPEECH_COMMAND_ID0

ID0

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID1

ID1

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID2

ID2

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID3

ID3

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID4

ID4

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID5

ID5

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID6

ID6

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID7

ID7

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID8

ID8

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID9

ID9

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID10

ID10

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID11

ID11

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID12

ID12

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID13

ID13

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID14

ID14

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID15

ID15

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID16

ID16

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID17

ID17

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID18

ID18

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID19

ID19

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID20

ID20

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID21

ID21

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID22

ID22

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID23

ID23

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID24

ID24

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID25

ID25

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID26

ID26

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID27

ID27

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID28

ID28

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID29

ID29

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID30

ID30

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID31

ID31

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID32

ID32

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID33

ID33

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID34

ID34

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID35

ID35

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID36

ID36

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID37

ID37

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID38

ID38

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID39

ID39

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID40

ID40

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID41

ID41

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID42

ID42

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID43

ID43

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID44

ID44

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID45

ID45

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID46

ID46

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID47

ID47

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID48

ID48

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID49

ID49

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID50

ID50

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID51

ID51

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID52

ID52

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID53

ID53

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID54

ID54

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID55

ID55

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID56

ID56

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID57

ID57

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID58

ID58

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID59

ID59

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID60

ID60

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID61

ID61

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID62

ID62

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID63

ID63

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID64

ID64

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID65

ID65

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID66

ID66

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID67

ID67

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID68

ID68

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID69

ID69

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID70

ID70

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID71

ID71

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID72

ID72

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID73

ID73

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID74

ID74

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID75

ID75

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID76

ID76

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID77

ID77

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID78

ID78

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID79

ID79

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID80

ID80

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID81

ID81

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID82

ID82

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID83

ID83

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID84

ID84

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID85

ID85

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID86

ID86

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID87

ID87

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID88

ID88

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID89

ID89

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID90

ID90

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID91

ID91

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID92

ID92

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID93

ID93

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID94

ID94

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID95

ID95

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID96

ID96

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID97

ID97

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID98

ID98

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID99

ID99

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID100

ID100

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID101

ID101

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID102

ID102

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID103

ID103

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID104

ID104

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID105

ID105

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID106

ID106

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID107

ID107

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID108

ID108

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID109

ID109

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID110

ID110

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID111

ID111

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID112

ID112

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID113

ID113

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID114

ID114

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID115

ID115

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID116

ID116

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID117

ID117

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID118

ID118

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID119

ID119

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID120

ID120

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID121

ID121

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID122

ID122

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID123

ID123

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID124

ID124

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID125

ID125

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID126

ID126

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID127

ID127

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID128

ID128

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID129

ID129

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID130

ID130

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID131

ID131

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID132

ID132

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID133

ID133

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID134

ID134

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID135

ID135

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID136

ID136

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID137

ID137

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID138

ID138

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID139

ID139

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID140

ID140

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID141

ID141

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID142

ID142

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID143

ID143

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID144

ID144

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID145

ID145

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID146

ID146

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID147

ID147

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID148

ID148

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID149

ID149

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID150

ID150

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID151

ID151

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID152

ID152

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID153

ID153

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID154

ID154

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID155

ID155

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID156

ID156

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID157

ID157

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID158

ID158

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID159

ID159

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID160

ID160

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID161

ID161

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID162

ID162

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID163

ID163

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID164

ID164

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID165

ID165

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID166

ID166

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID167

ID167

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID168

ID168

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID169

ID169

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID170

ID170

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID171

ID171

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID172

ID172

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID173

ID173

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID174

ID174

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID175

ID175

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID176

ID176

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID177

ID177

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID178

ID178

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID179

ID179

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID180

ID180

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID181

ID181

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID182

ID182

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID183

ID183

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID184

ID184

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID185

ID185

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID186

ID186

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID187

ID187

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID188

ID188

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID189

ID189

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID190

ID190

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID191

ID191

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID192

ID192

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID193

ID193

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID194

ID194

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID195

ID195

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID196

ID196

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID197

ID197

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID198

ID198

Found in: ESP Speech Recognition > Add English speech commands

EN_SPEECH_COMMAND_ID199

ID199

Found in: ESP Speech Recognition > Add English speech commands

2.10.5 Audio Codec Device Configuration

CODEC_I2C_BACKWARD_COMPATIBLE

Enable backward compatibility for the I2C driver (force use of the old i2c_driver above v5.3)

Found in: Component config > Audio Codec Device Configuration

Enable this option for backward compatibility with the old I2C driver

CODEC_ES8311_SUPPORT

Support ES8311 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES8311.

CODEC_ES7210_SUPPORT

Support ES7210 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES7210.

CODEC_ES7243_SUPPORT

Support ES7243 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES7243.

CODEC_ES7243E_SUPPORT

Support ES7243E Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES7243E.

CODEC_ES8156_SUPPORT

Support ES8156 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES8156.

CODEC_AW88298_SUPPORT

Support AW88298 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec AW88298.

CODEC_ES8374_SUPPORT

Support ES8374 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES8374.

CODEC_ES8388_SUPPORT

Support ES8388 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ES8388.

CODEC_TAS5805M_SUPPORT

Support TAS5805M Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec TAS5805M.

CODEC_ZL38063_SUPPORT

Support ZL38063 Codec Chip

Found in: Component config > Audio Codec Device Configuration

Enable this option to support codec ZL38063. ZL38063 firmware only support xtensa, don't enable for RISC-V IC.

2.10.6 DuerOS Service

DUEROS_GEN_PROFILE

Generate the nvs partition which include the profile of dueros

Found in: Component config > DuerOS Service

DUEROS_FLASH_PROFILE

Flash the generated nvs partition

Found in: Component config > DuerOS Service

DUEROS_DEVICE_NAME

Duer device name

Found in: Component config > DuerOS Service

CHAPTER

THREE

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 audio signals 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
- **Wi-Fi interface** to obtain an audio data stream from the internet
- **Bluetooth interface** to obtain an audio data stream from e.g. a Bluetooth 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 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 Bluetooth 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 ESP chips (including ESP32, ESP32-S2, ESP32-S3) have all the above features or are able to support them (e.g. can drive Ethernet PHY). Considering the ESP chip cost is low, 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 a couple of examples.

3.1.2 Project Minimum

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

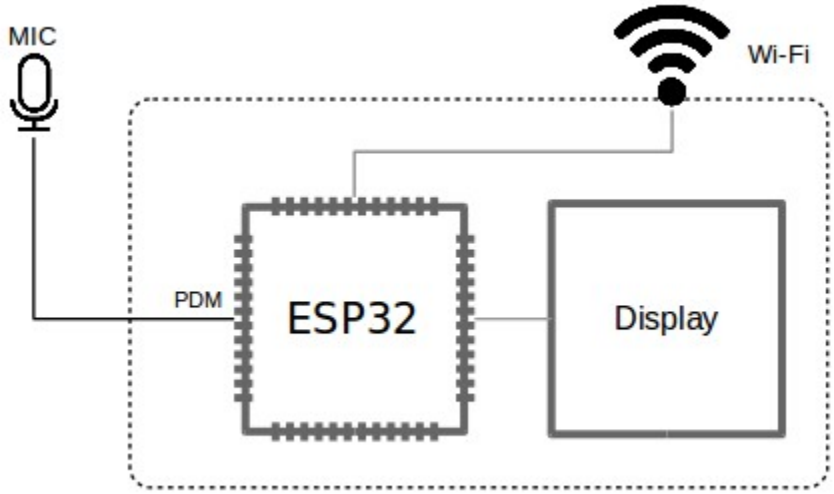


Fig. 1: Audio Project Example - Send Voice Commands to Cloud Service

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.

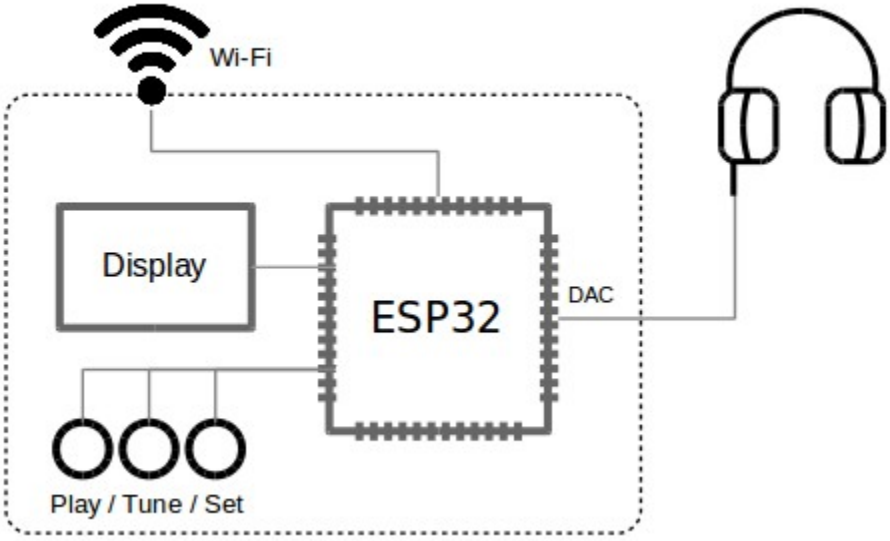


Fig. 2: Audio Project Example - Internet Connected Radio Player

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, an additional memory may be required. For such cases, please consider using ESP32-WROVER-E that provides 8 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.

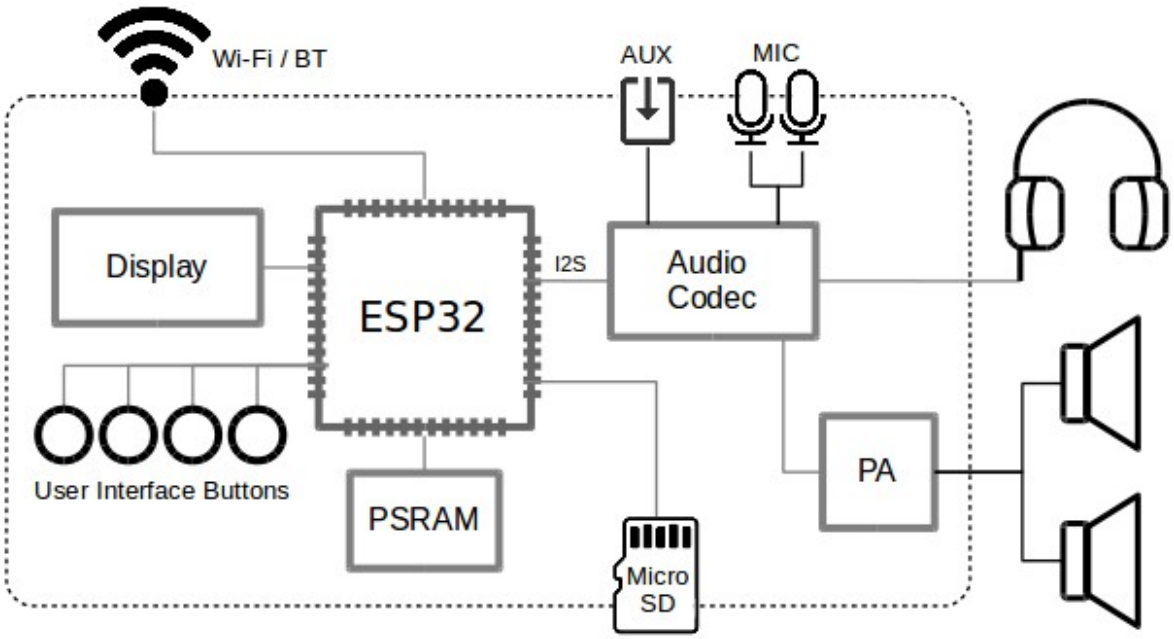


Fig. 3: Typical Audio Project Example

3.2.1 Memory

The spare internal Data-RAM is about 290kB with "hello_world" example. For audio system this may be insufficient, and therefore the ESP32 incorporates the ability to use up to 4MB of external SPI RAM (i.e. PSRAM) memory. The external memory is incorporated in the memory map and is, within certain restrictions, usable in the same way internal Data-RAM is.

Refer to External SPI-connected RAM section in IDF 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. Taking ESP32-LyraT V4.3 as an example, depending on value of *SDCARD_OPEN_FILE_NUM_MAX* in audio_board/lyrat_v4_3/board_def.h, that is then used in `sdcard_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

Please refer to audio_stream/fatfs_stream.c for an example of using an audio element.

Audio Event

Example

Please refer to player/pipeline_http_mp3/main/play_http_mp3_example.c for an example of using an audio event.

Audio Pipeline

Example

Please refer to player/pipeline_play_sdcard_music/main/play_sdcard_music_example.c for an example of linking elements into an audio pipeline.

Audio Peripheral

Example

```
ESP_LOGI(TAG, "[ 3 ] Start and wait for Wi-Fi network");
esp_periph_config_t periph_cfg = DEFAULT_ESP_PERIPH_SET_CONFIG();
esp_periph_set_handle_t set = esp_periph_set_init(&periph_cfg);
periph_wifi_cfg_t wifi_cfg = {
    .wifi_config.sta.ssid = CONFIG_WIFI_SSID,
    .wifi_config.sta.password = CONFIG_WIFI_PASSWORD,
};
esp_periph_handle_t wifi_handle = periph_wifi_init(&wifi_cfg);
esp_periph_start(set, wifi_handle);
periph_wifi_wait_for_connected(wifi_handle, portMAX_DELAY);
```

Audio Player

Example

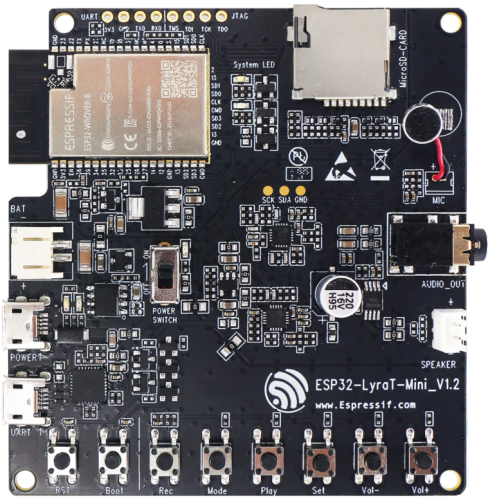
Please refer to cli for an example of initializing `esp_audio` as an audio player.

3.4 Development Boards

Below are getting started guides and hardware details of audio development boards designed by Espressif.

Getting Started with ESP32-LyraT

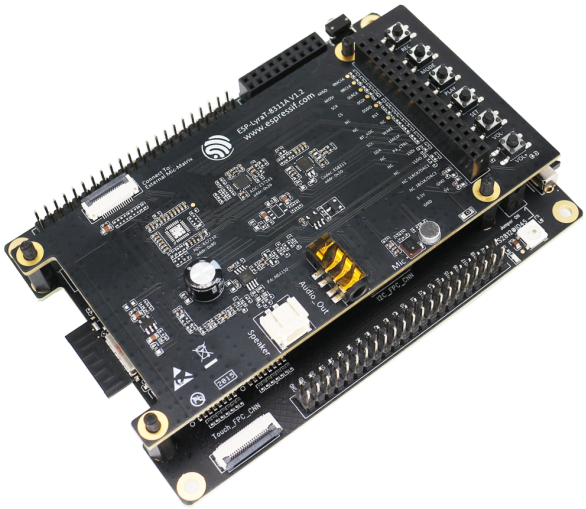
Hardware Reference of ESP32-LyraT



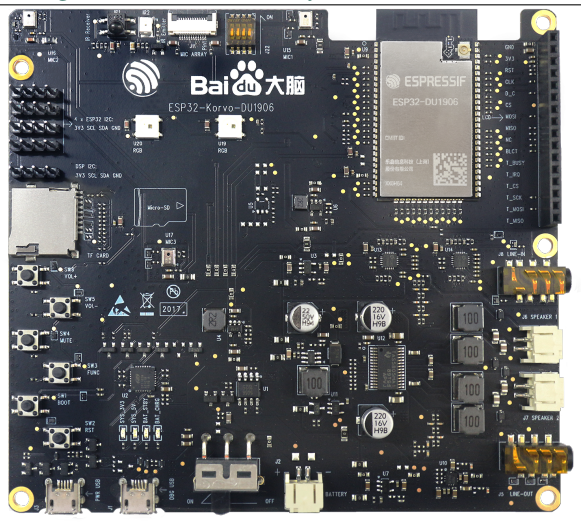
Getting Started with ESP32-LyraT-Mini

Hardware Reference of ESP32-LyraT-Mini

Getting Started with ESP32-LyraTD-MSC



Getting Started with ESP32-S2-Kaluga-1-Kit



Getting Started with ESP32-Korvo-DU1906

Getting Started with ESP32-S3-Korvo-2

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

This guide provides users with functional descriptions, configuration options for ESP32-LyraT-Mini V1.2 audio development board, as well as how to get started with the ESP32-LyraT board.

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

The ESP32-LyraT-Mini is a mono audio board. If you are looking for a stereo audio board, check *ESP32-LyraT V4.3 Getting Started Guide*.

What You Need

- *ESP32-LyraT-Mini V1.2 board*
- Speaker or headphones with a 3.5 mm jack. If you use a speaker, it is recommended to choose one no more than 3 watts, and JST PH 2.0 2-Pin plugs are needed. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.
- Two Micro-USB 2.0 cables, Type A to Micro B
- PC loaded with Windows, Linux or Mac OS

Optional components

- Micro SD-card
- Li-ion Battery

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

Overview

The ESP32-LyraT-Mini V1.2 is an audio development board produced by Espressif built around ESP32. It is intended for audio applications, by providing hardware for audio processing and additional RAM on top of what is already on-board of the ESP32 chip. The specific hardware includes:

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

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

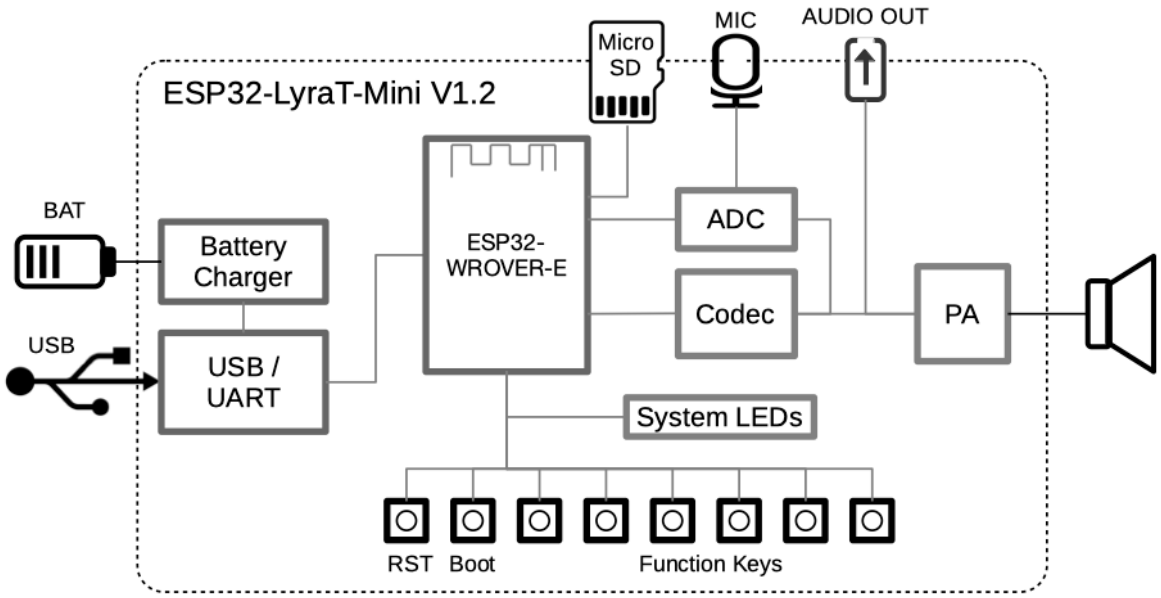


Fig. 4: ESP32-LyraT-Mini Block Diagram

Components

The following list and figure describe key components, interfaces and controls of the ESP32-LyraT-Mini used in this guide. For detailed technical documentation of this board, please refer to *ESP32-LyraT-Mini V1.2 Hardware Reference* and ESP32-LyraT-Mini V1.2 schematic (PDF). The list below provides description starting from the picture's top right corner and going clockwise.

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

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

Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

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

USB Power Port Provides the power supply for the board.

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

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

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

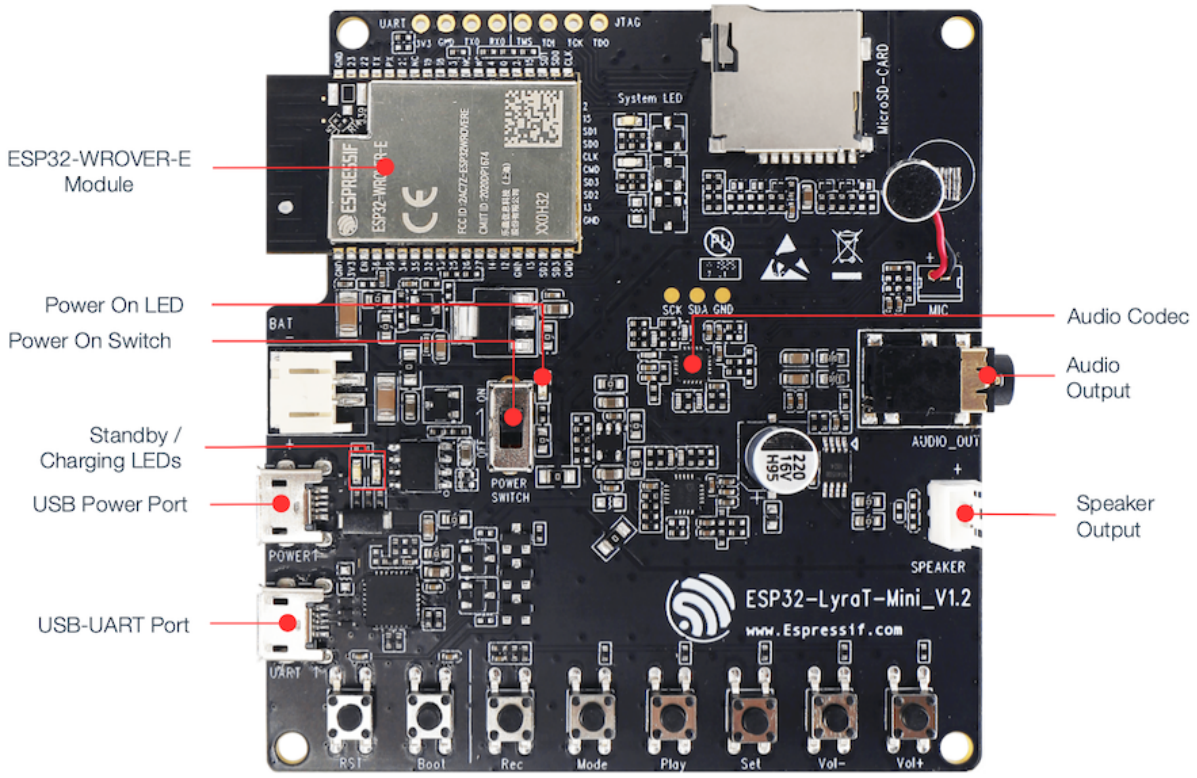


Fig. 5: ESP32 LyraT-Mini V1.2 Board Layout Overview

ESP32-WROVER-E Module The ESP32-WROVER-E module contains ESP32 chip to provide Wi-Fi / Bluetooth connectivity and data processing power as well as integrates 4 MB external SPI flash and an additional 8 MB PSRAM for flexible data storage.

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

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

Develop Applications

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Installation Step by Step* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Revision History

- Changed the integrated module to ESP32-WROVER-E from ESP32-WROVER-B.

Other Boards from LyraT Family

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

Related Documents

- ESP32-LyraT-Mini V1.2 schematic (PDF)
- ESP32-LyraT-Mini V1.2 Board Dimensions (PDF)
- *ESP32-LyraT-Mini V1.2 Hardware Reference*
- ESP32 Datasheet (PDF)
- ESP32-WROVER-E Datasheet (PDF)

3.4.2 ESP32-LyraT-Mini V1.2 Hardware Reference

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

In this Section

Overview

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

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

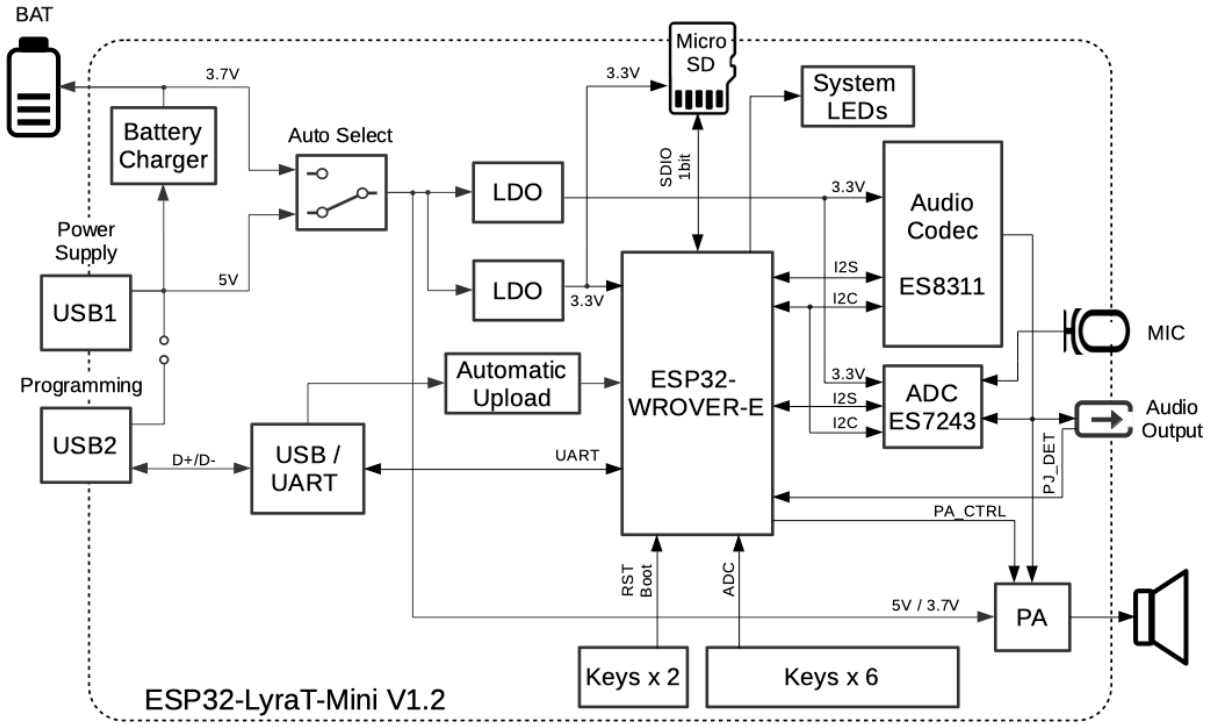


Fig. 6: ESP32-LyraT-Mini V1.2 Electrical Block Diagram

Functional Description

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

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

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

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

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

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

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

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

Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Function Press Keys Six press keys labeled **Rec**, **Mode**, **Play**, **Set**, **Vol-**, and **Vol+**. They are routed to **ESP32-WROVER-E Module** and intended for development and testing of a UI for audio applications using dedicated API.

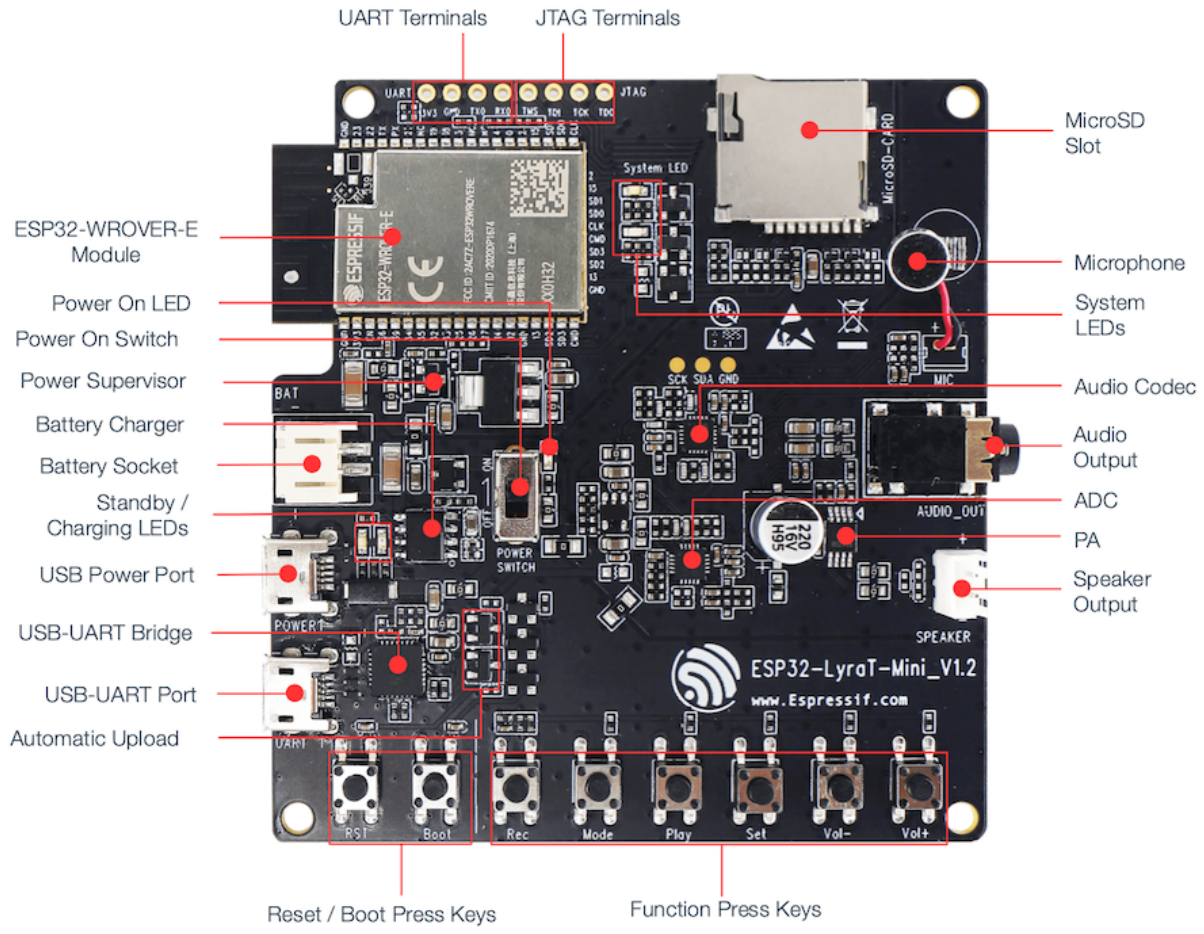


Fig. 7: ESP32 LyraT-Mini V1.2 Board Layout

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

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

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

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

USB Power Port Provides the power supply for the board.

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

Battery Socket Two-pin socket to connect a single cell Li-ion battery. The pins have a 2.00 mm / 0.08" pitch. The battery serves as an alternative power supply to the **USB Power Port** for charging the board. Make sure to use a Li-ion battery that has protection circuit and fuse. The recommended specifications of the battery: capacity > 1000 mAh, output voltage 3.7 V, input voltage 4.2 V – 5 V. 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 Constant current and constant voltage linear charger for single cell lithium-ion batteries AP5056. Used for charging of a battery connected to the **Battery Socket** over the **USB Power Port**.

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

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

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

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

ESP32-WROVER-E Module The ESP32-WROVER-E module contains ESP32 chip to provide Wi-Fi / Bluetooth connectivity and data processing power as well as integrates 4 MB external SPI flash and an additional 8 MB PSRAM for flexible data storage.

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

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

Allocation of ESP32 Pins to Test Points

This section describes allocation of test points available on the ESP32-LyraT-Mini board.

The test points are bare through hole solder pads and have standard 2.54 mm / 0.1 inch pitch. User may need to populate them with pin headers or sockets for easy connection of external hardware.

JTAG Test Point

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

UART Test Point

.	ESP32 Pin	Pin Description
1	RXD0	RX
2	TXD0	TX
3	GND	GND
4	n/a	3.3 V

MicroSD Card

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

.	ESP32 Pin	MicroSD Signal
1	MTDI / GPIO12	–
2	MTCK / GPIO13	–
3	MTDO / GPIO15	CMD
4	MTMS / GPIO14	CLK
5	GPIO2	DATA0
6	GPIO4	–
7	GPIO34	CD

GPIO Allocation Summary

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

Pin [1]	Pin Name	ES8311	ES7243	Keys	MicroSD	Other
3	EN			EN_KEY		
4	S_VP		I2S_DATA			
5	S_VN			REC, MODE, PLAY, SET, VOL-, VOL+		
6	IO34				CD	
7	IO35	I2S0_ASDOUT				
8	IO32		I2S1_SCLK			
9	IO33		I2S1_LRCK			
10	IO25	I2S0_LRCK				
11	IO26	I2S0_DSDIN				
12	IO27					Blue_LED
13	IO14				CLK	
14	IO12				NC (DATA2)	
16	IO13				NC (DATA3)	
17	SD2					
18	SD3					
19	CMD					
20	CLK					
21	SD0					
22	SD1					

continues on next page

Table 1 – continued from previous page

Pin [1]	Pin Name	ES8311	ES7243	Keys	MicroSD	Other
23	IO15				CMD	
24	IO2			IO2_KEY	DATA0	
25	IO0	I2S0_MCLK	I2S1_MCLK	IO0_KEY		
26	IO4				NC (DATA1)	
27	NC (IO16)					
28	NC (IO17)					
29	IO5	I2S0_SCLK				
30	IO18	I2C_SDA	I2C_SDA			
31	IO19					PJ_DET [2]
33	IO21					PA_CTRL [3]
34	RXD0					RXD0 [4]
35	TXD0					TXD0 [4]
36	IO22					Green_LED
37	IO23	I2C_SCK	I2C_SCL			

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

Notes on Power Distribution

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

Power Supply over USB and from Battery

There are two ways to power the development board: 5 V USB Power Port or 3.7 V optional battery. The optional battery is preferable for applications where a cleaner power supply is required.

Independent Audio and Digital Power Supply

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

Selecting of the Audio Output

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

- Power amplifier (PA) to drive an external speaker
- Phone jack socket to drive external headphones

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

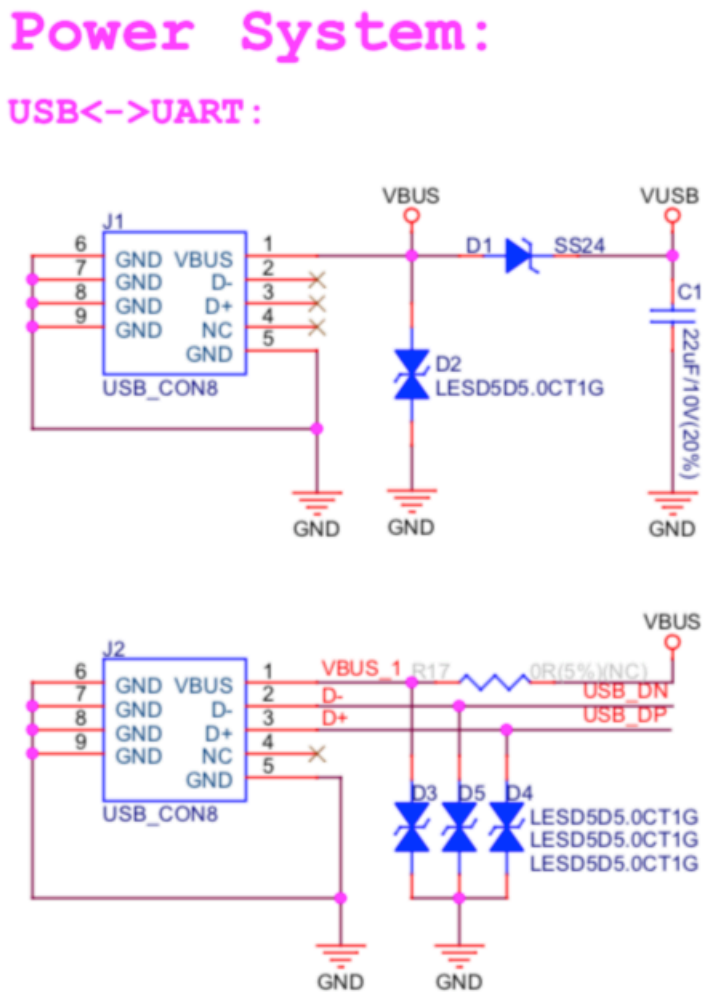


Fig. 8: ESP32-LyraT-Mini V1.2 - Dedicated USB Power Supply Socket

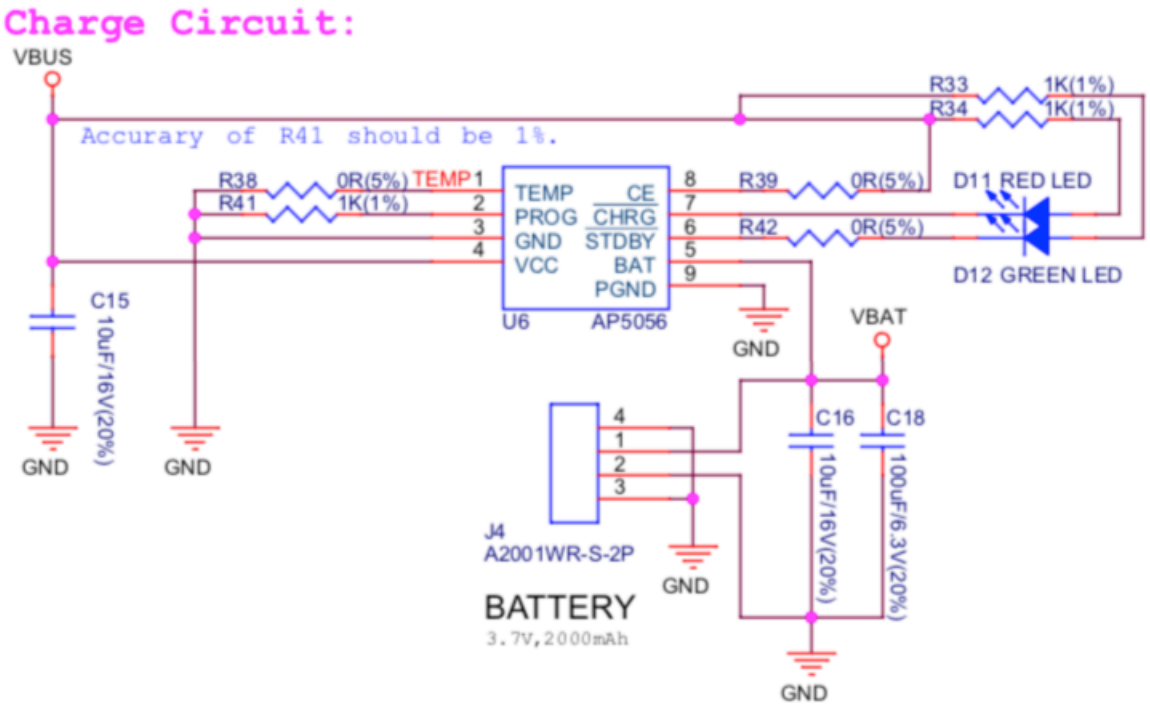


Fig. 9: ESP32-LyraT-Mini V1.2 - Power Supply from a Battery

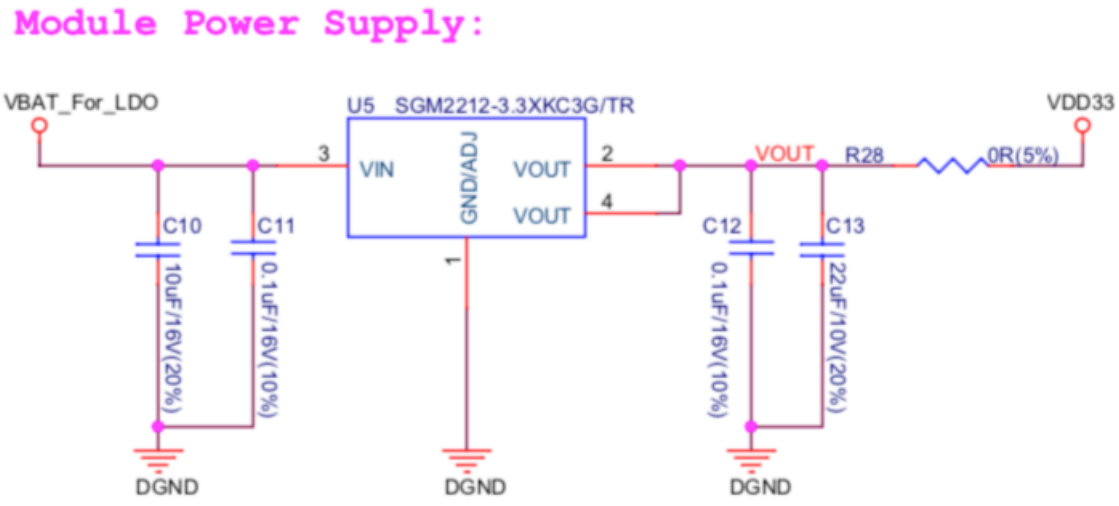


Fig. 10: ESP32-LyraT-Mini V1.2 - Digital Power Supply

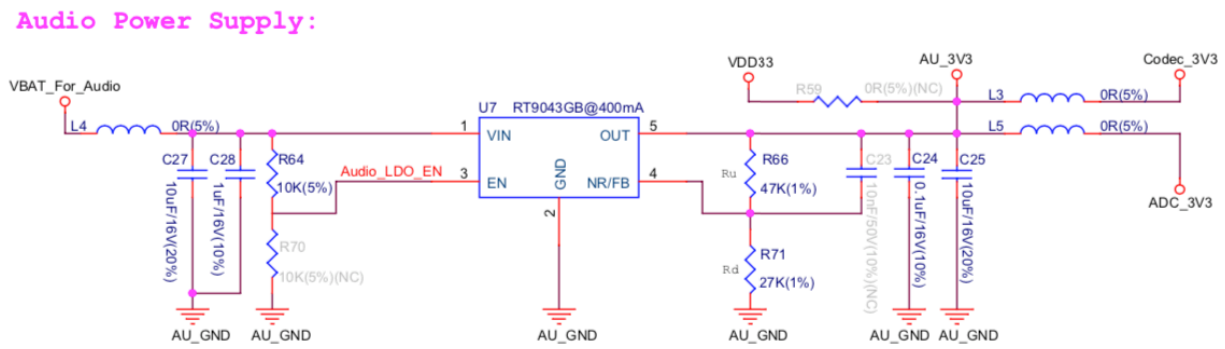


Fig. 11: ESP32-LyraT-Mini V1.2 - Audio Power Supply

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

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

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

Related Documents

- ESP32-LyraT-Mini V1.2 Schematic (PDF)
- ESP32-LyraT-Mini V1.2 Board Dimensions (PDF)
- *ESP32-LyraT-Mini V1.2 Getting Started Guide*
- ESP32 Datasheet (PDF)
- ESP32-WROVER-E Datasheet (PDF)

3.4.3 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 Bluetooth audio speakers, speech-based remote controllers, connected smart-home appliances with one or more audio functionality, etc.

The ESP32-LyraT is a stereo audio board. If you are looking for a mono audio board, intended for lower end applications, check *ESP32-LyraT-Mini V1.2 Getting Started Guide*.

What You Need

- 1 × *ESP32 LyraT V4.3 board*
- 2 x Speaker or headphones with a 3.5 mm jack. If you use a speaker, it is recommended to choose one no more than 3 watts, and JST PH 2.0 2-Pin plugs are needed. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.
- 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-E Module
- Audio Codec Chip
- Dual Microphones on board
- Headphone output
- 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-E Module The ESP32-WROVER-E module contains ESP32 chip to provide Wi-Fi / Bluetooth connectivity and data processing power as well as integrates 4 MB external SPI flash and an additional 8 MB PSRAM for flexible data storage.

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

> **Note:** The socket may be used with mobile phone headsets and is compatible with OMPT standard headsets only. It does not work with CTIA headsets. Please refer to Phone connector (audio) on Wikipedia.

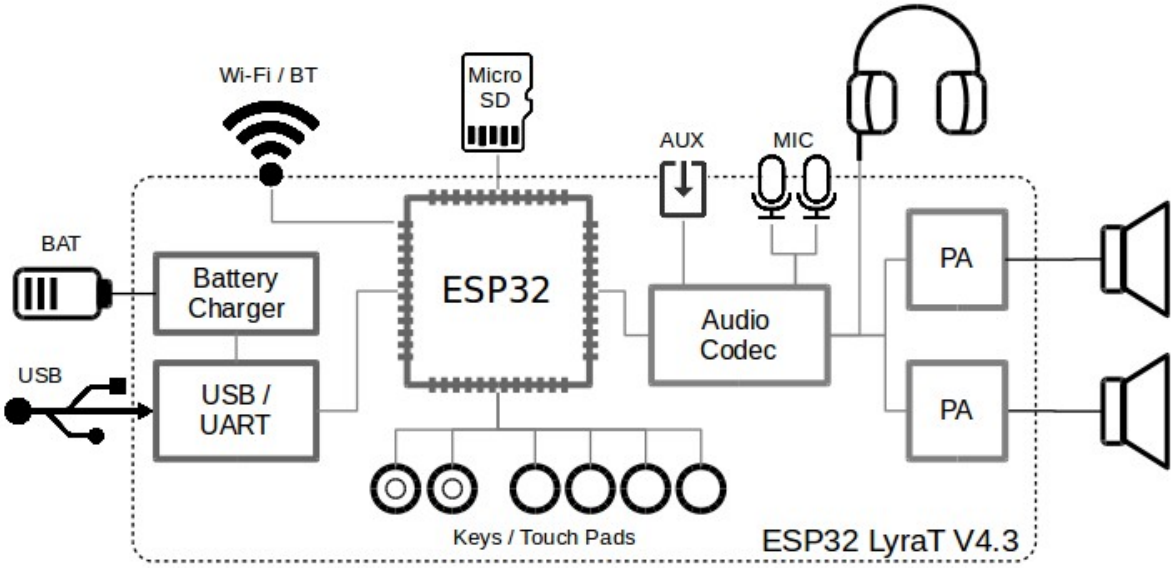


Fig. 12: ESP32-LyraT Block Diagram

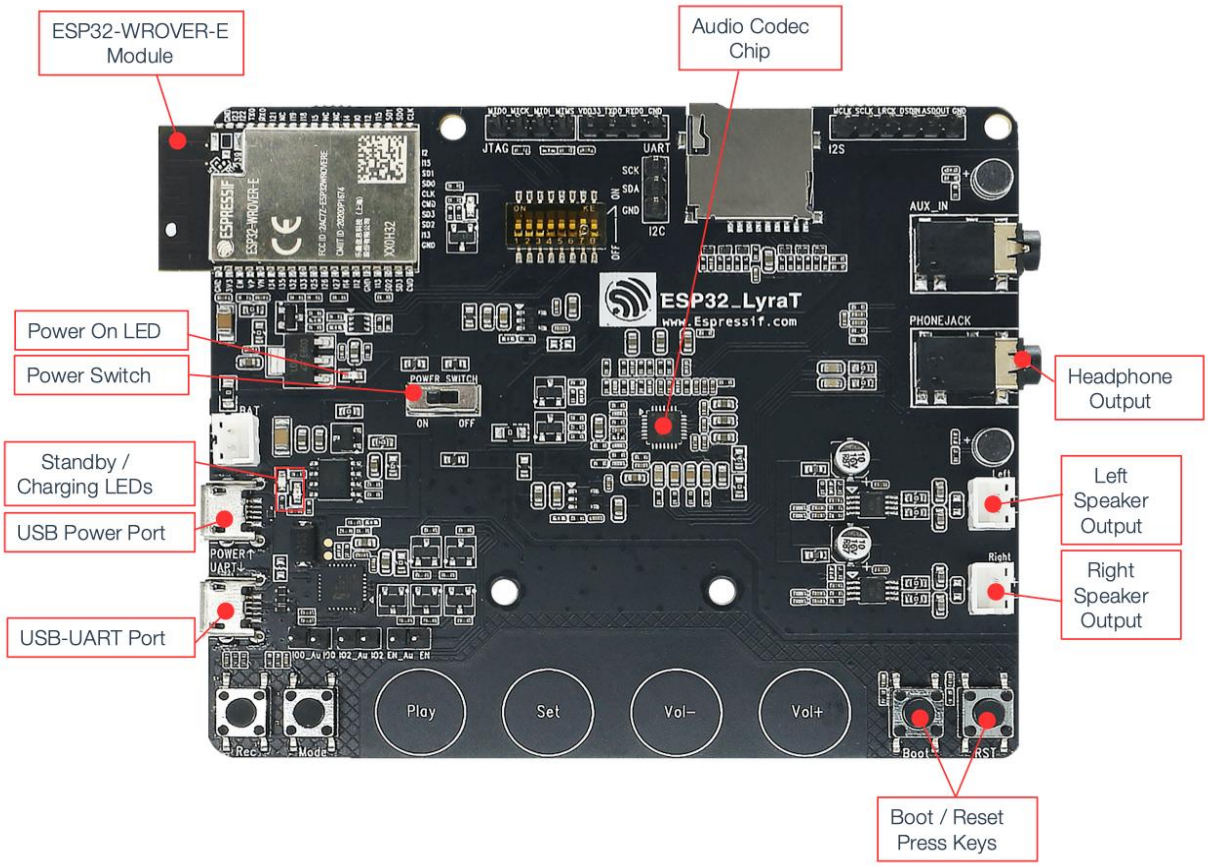


Fig. 13: ESP32-LyraT V4.3 Board Layout Overview

Left Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Right Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

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

Audio Codec Chip The Audio Codec Chip, ES8388, is a low power stereo audio codec with a headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects, analog mixing and gain functions. It is interfaced with **ESP32-WROVER-E 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-E 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.

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 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

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Installation Step by Step* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Summary of Key Changes from LyraT V4.2

- Changed the integrated module to ESP32-WROVER-E from ESP32-WROVER.
- 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.

Other Versions of LyraT

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

Other Boards from LyraT Family

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

Related Documents

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

3.4.4 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.

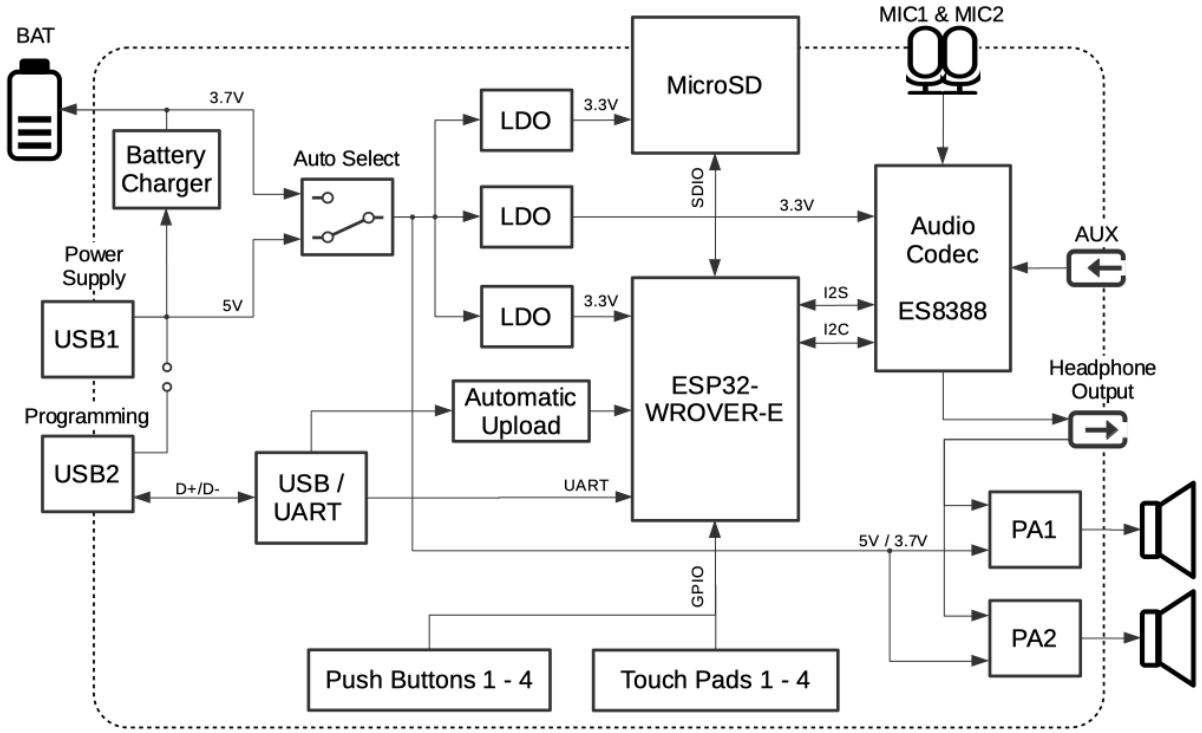


Fig. 14: ESP32-LyraT V4.3 Electrical Block Diagram

Functional Description

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

ESP32-WROVER-E Module The ESP32-WROVER-E module contains ESP32 chip to provide Wi-Fi / Bluetooth connectivity and data processing power as well as integrates 4 MB external SPI flash and an additional 8 MB PSRAM for flexible data storage.

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

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

JTAG Header Provides access to the **JTAG** interface of **ESP32-WROVER-E 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-E Module** and **USB-UART Bridge Chip**.

I2C Header Provides access to the I2C interface. Both **ESP32-WROVER-E 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-E 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.

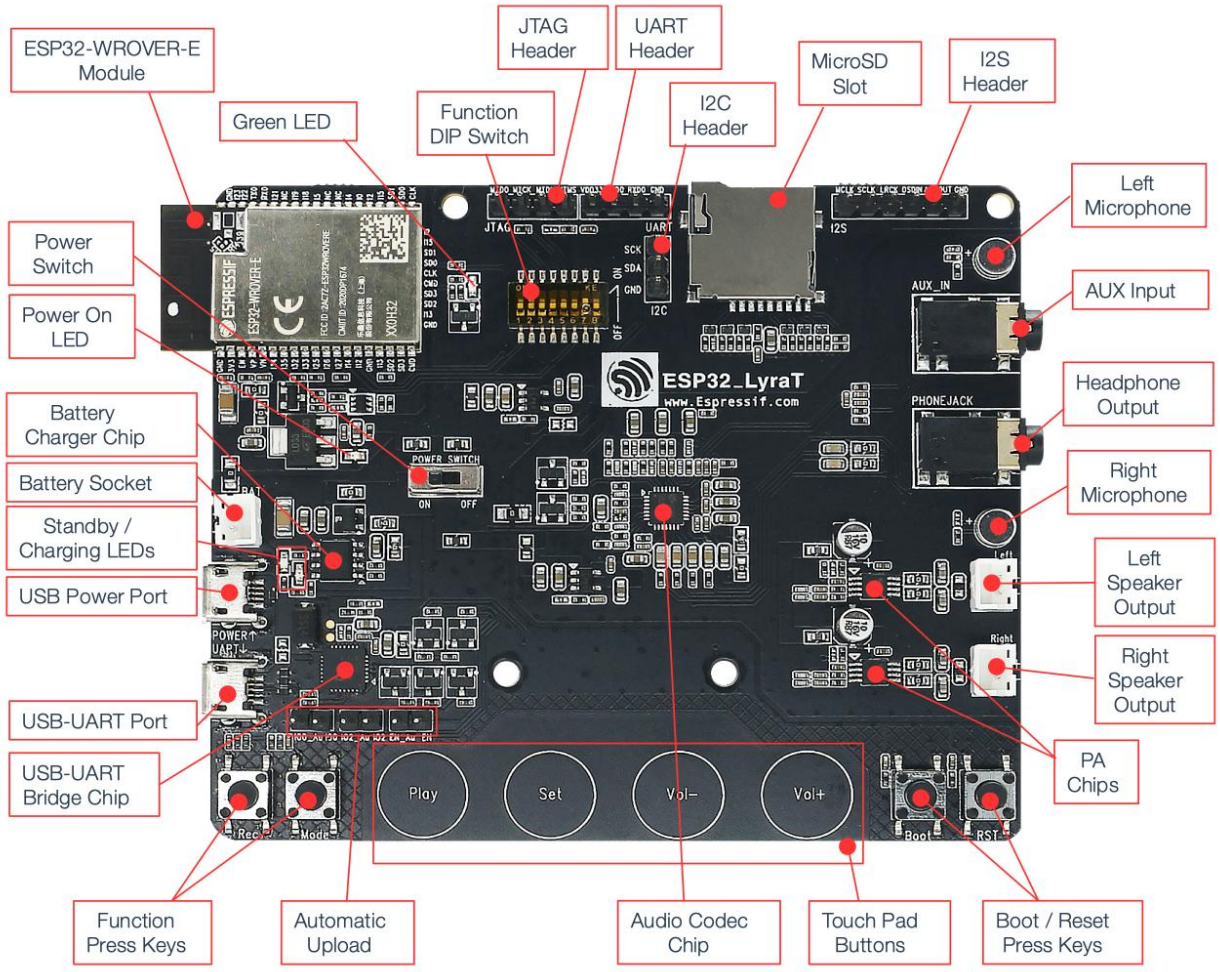


Fig. 15: ESP32-LyraT V4.3 Board Layout

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

Left Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Right Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

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

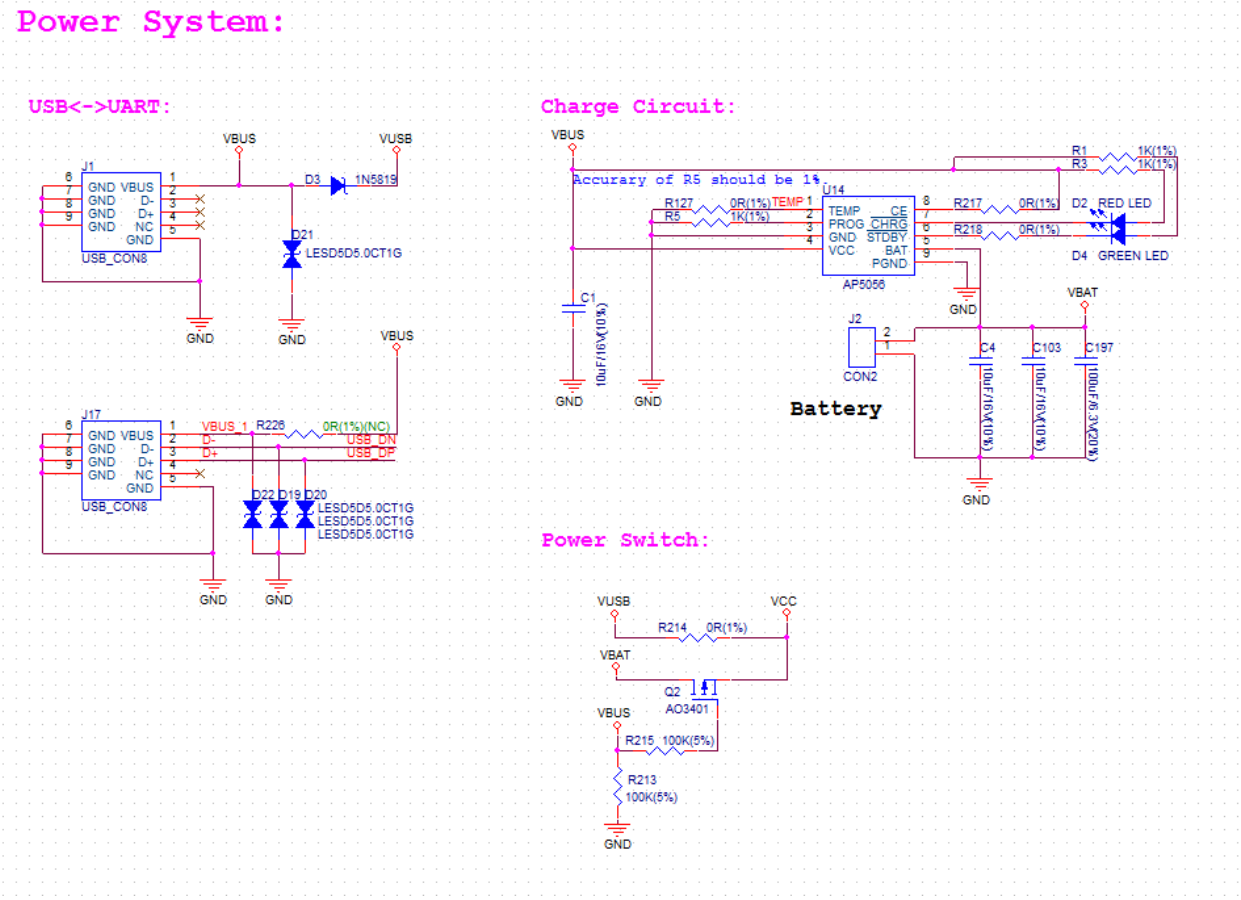


Fig. 16: ESP32 LyraT V4.3 - Power Supply Separation

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.

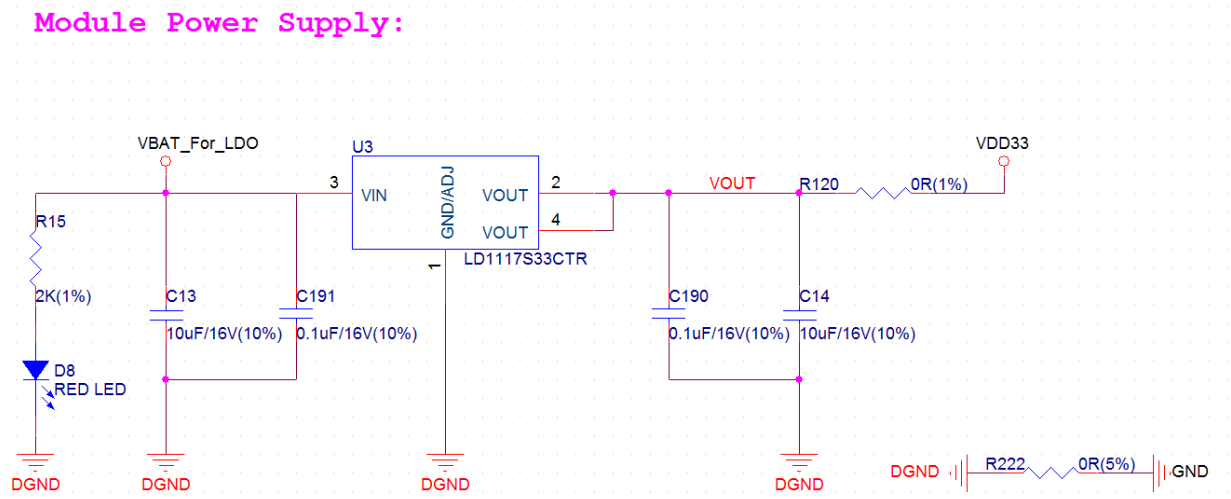


Fig. 17: ESP32 LyraT V4.3 - Dedicated LDO for the ESP32 Module

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.

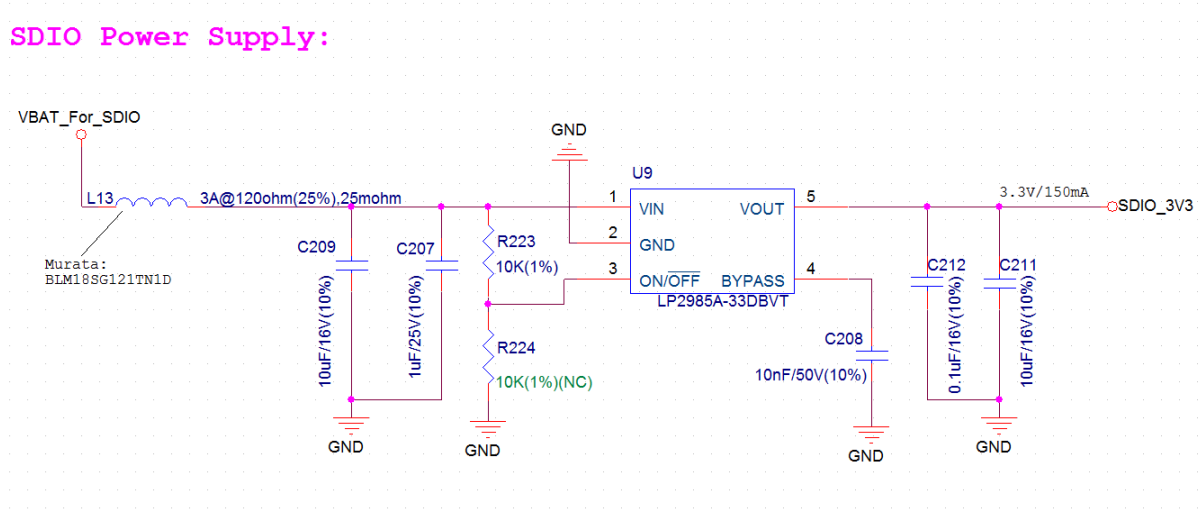


Fig. 18: ESP32 LyraT V4.3 - Dedicated LDO for the MicroSD Card

Separate Power Feed for the PAs

The audio amplifier unit features two NS4150 that require a large power supply for driving external speakers with the maximum output power of 3W. The power is supplied directly to both PAs from the battery or the USB. The development board adds a set of LC circuits at the front of the PA power supply, where L uses 1.5A magnetic beads and C uses 10uF aluminum electrolytic capacitors, to effectively filter out power crosstalk.

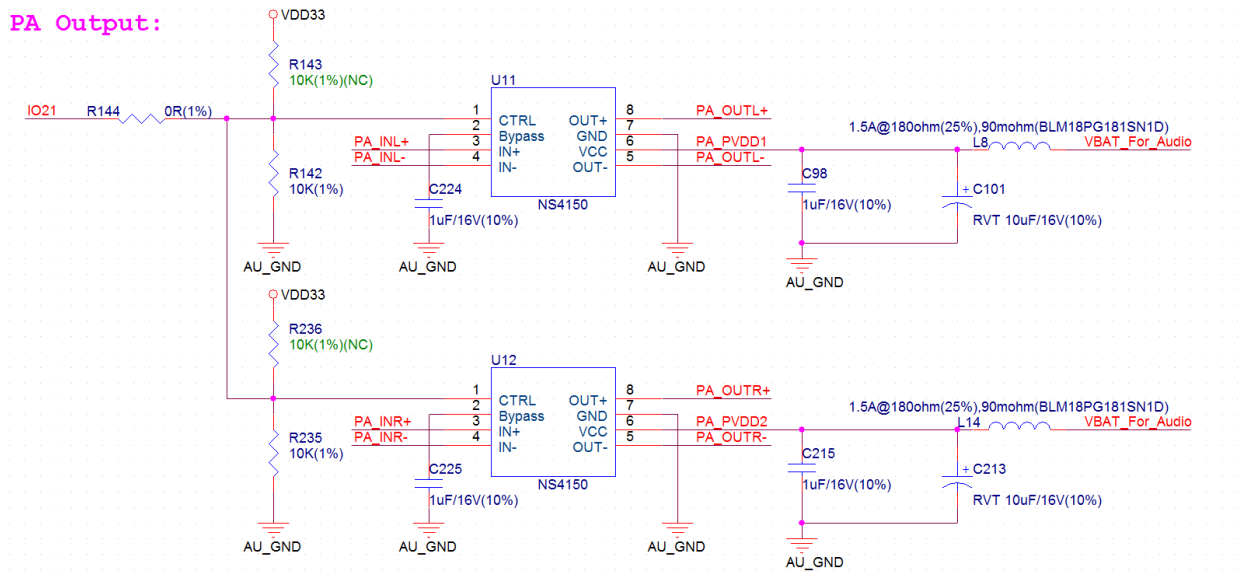


Fig. 19: ESP32 LyraT V4.3 - Power Supply for the PAs

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-E Datasheet (PDF)
- JTAG Debugging

3.4.5 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 × Speaker or headphones with a 3.5 mm jack. If you use a speaker, it is recommended to choose one no more than 3 watts, and JST PH 2.0 2-Pin plugs are needed. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.
- 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 superscript * indicates that the product is in EOL status. The same applies below.

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

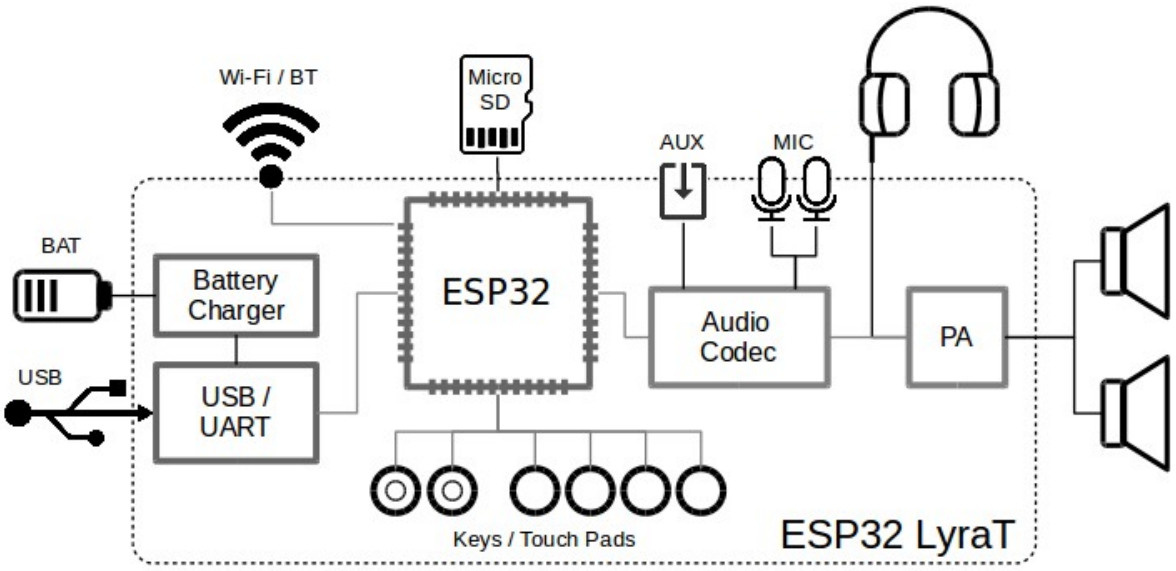


Fig. 20: ESP32-LyraT Block Diagram

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.

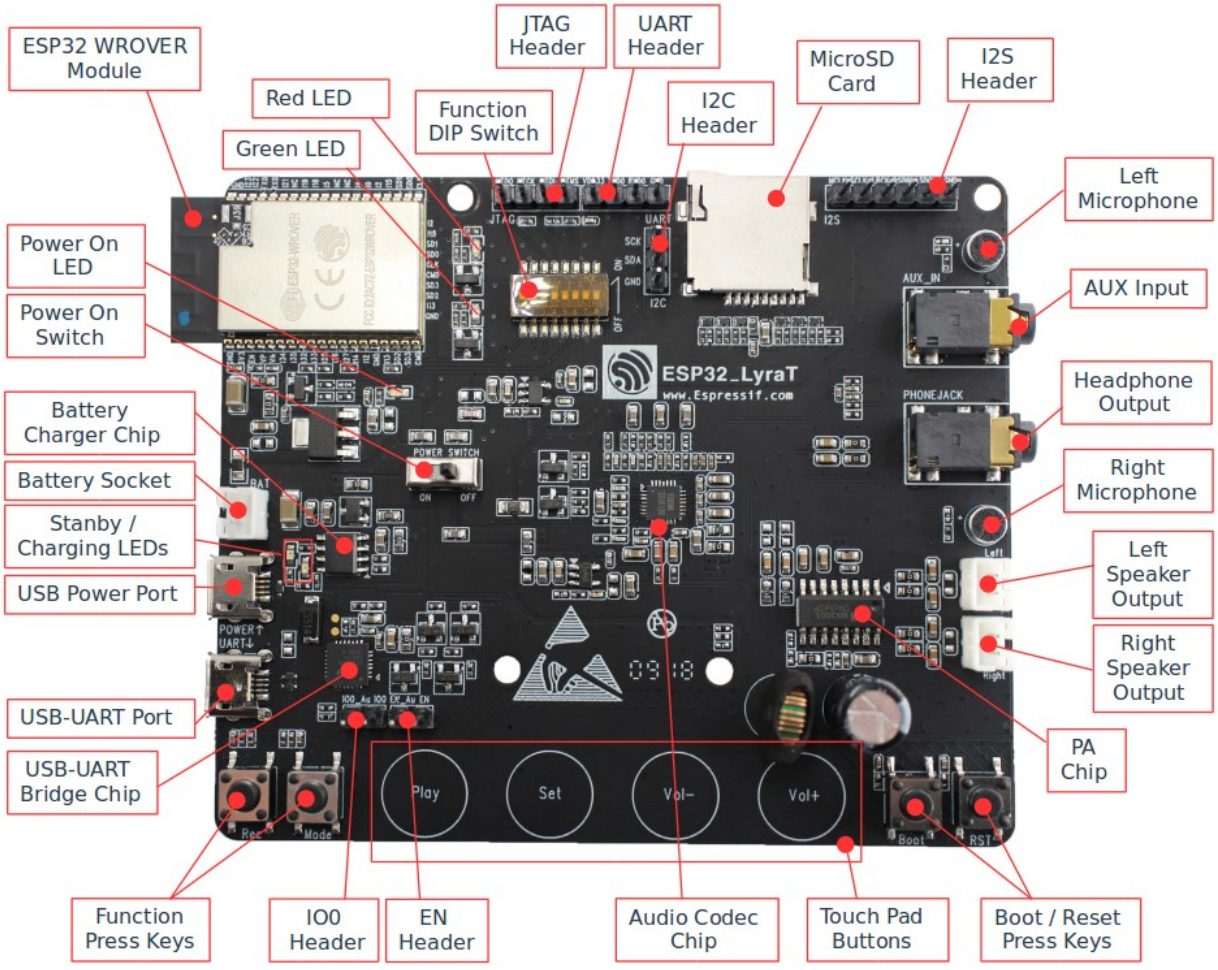


Fig. 21: ESP32-LyraT V4.2 Board Layout

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

Left Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Right Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

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

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

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

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

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

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

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

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

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

USB Power Port Provides the power supply for the board.

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

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

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

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

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

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

Hardware Setup Options

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

Enable MicroSD Card in 1-wire Mode

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

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

In this mode:

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

Enable MicroSD Card in 4-wire Mode

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

In this mode:

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

Enable JTAG

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

In this mode:

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

Allocation of ESP32 Pins

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

Red / Green LEDs

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

Touch Pads

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

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

MicroSD Card / J5

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

UART Header / JP2

	Header Pin
1	3.3V
2	TX
3	RX
4	GND

EN and IO0 Headers / JP23 and J24

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

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

I2S Header / JP4

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

I2C Header / JP5

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

JTAG Header / JP7

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

Function DIP Switch / JP8

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

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

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

1. Install jumpers on **IO0** and **EN** headers to enable automatic application upload. If there are no jumpers then upload may be triggered using **Boot / RST** buttons.
2. Connect 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

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Installation Step by Step* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Related Documents

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

3.4.6 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 × Speaker or headphones with a 3.5 mm jack. If you use a speaker, it is recommended to choose one no more than 3 watts, and JST PH 2.0 2-Pin plugs are needed. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.
- 1 × 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**

The superscript [*] indicates that the product is in EOL status. The same applies below.

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

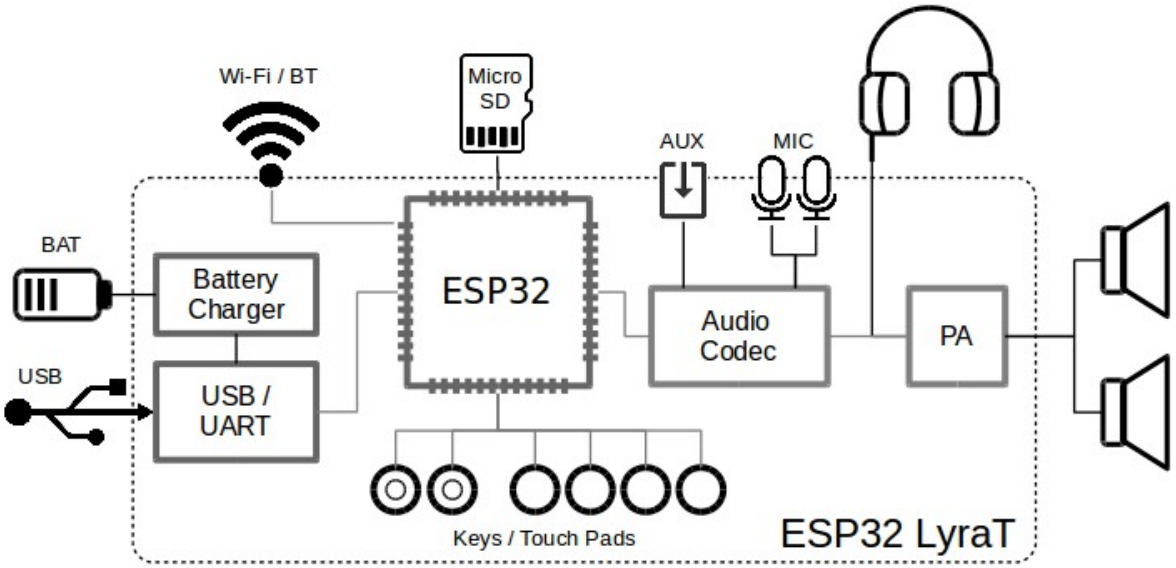


Fig. 22: ESP32-LyraT block diagram

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.

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

Left Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Right Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

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

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

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

Audio Codec Chip The Audio Codec Chip, ES8388, is a low-power stereo audio codec with headphone amplifier. It consists of 2-channel ADC, 2-channel DAC, microphone amplifier, headphone amplifier, digital sound effects,

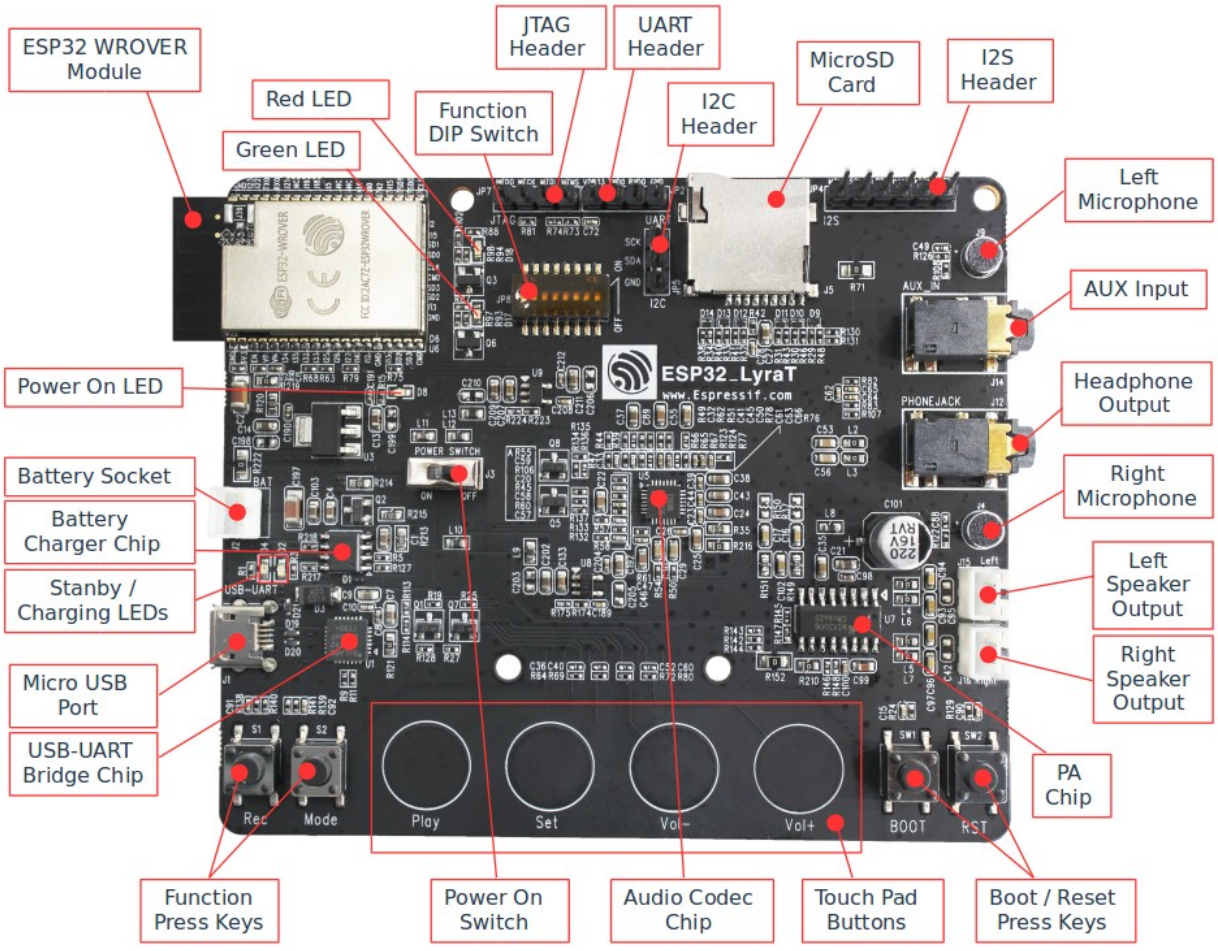


Fig. 23: ESP32 LyraT V4 board layout

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 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

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Installation Step by Step* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Related Documents

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

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

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

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

What You Need

- 1 × *ESP32-LyraTD-MSC V2.2 board*
- 2 x Speaker or headphones with a 3.5 mm jack. If you use a speaker, it is recommended to choose one no more than 3 watts, and JST PH 2.0 2-Pin plugs are needed. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.
- 2 x Micro-USB 2.0 cables, Type A to Micro B
- 1 × PC loaded with Windows, Linux or Mac OS

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

Overview

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

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

Fig. 24: ESP32-LyraTD-MSC Side View

The specific hardware includes:

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

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

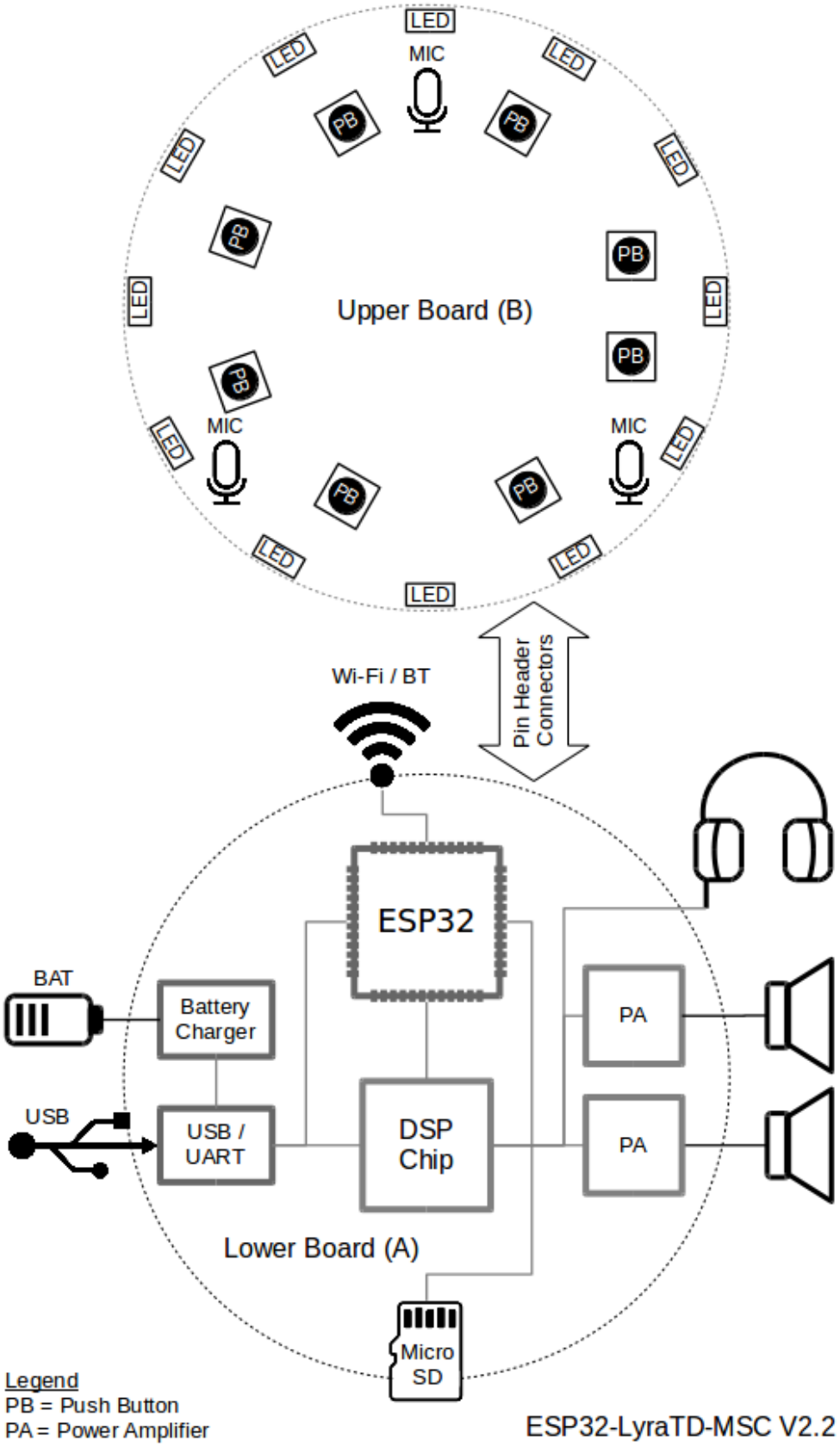


Fig. 25: ESP32-LyraTD-MSC Block Diagram

Components

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

ESP32-WROVER-E Module The ESP32-WROVER-E module contains ESP32 chip to provide Wi-Fi / Bluetooth connectivity and data processing power as well as integrates 4 MB external SPI flash and an additional 8 MB PSRAM for flexible data storage.

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

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

> **Note:** The socket may be used with mobile phone headsets and is compatible with OMPT standard headsets only. It does not work with CTIA headsets. Please refer to Phone connector (audio) on Wikipedia.

Left Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

Right Speaker Output Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm / 0.08" pitch.

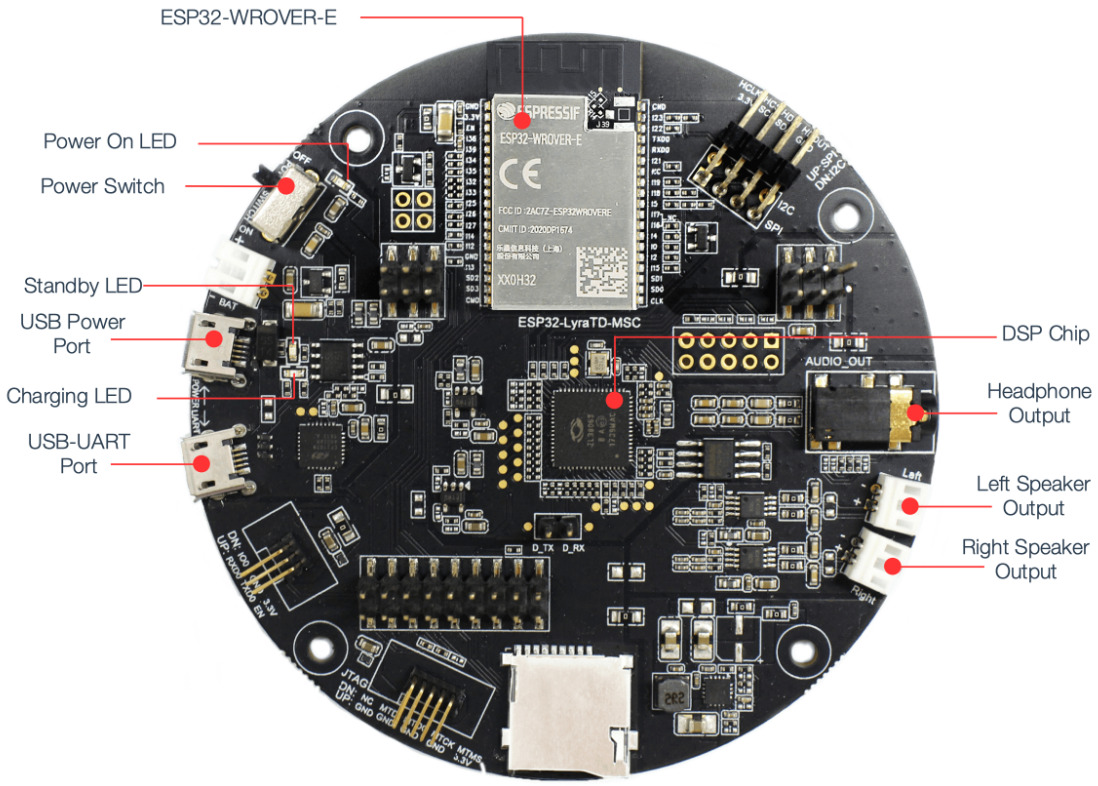


Fig. 26: ESP32-LyraTD-MSC V2.2 Lower Board (A) Components

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

USB Power Port Provides the power supply for the board.

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

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

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

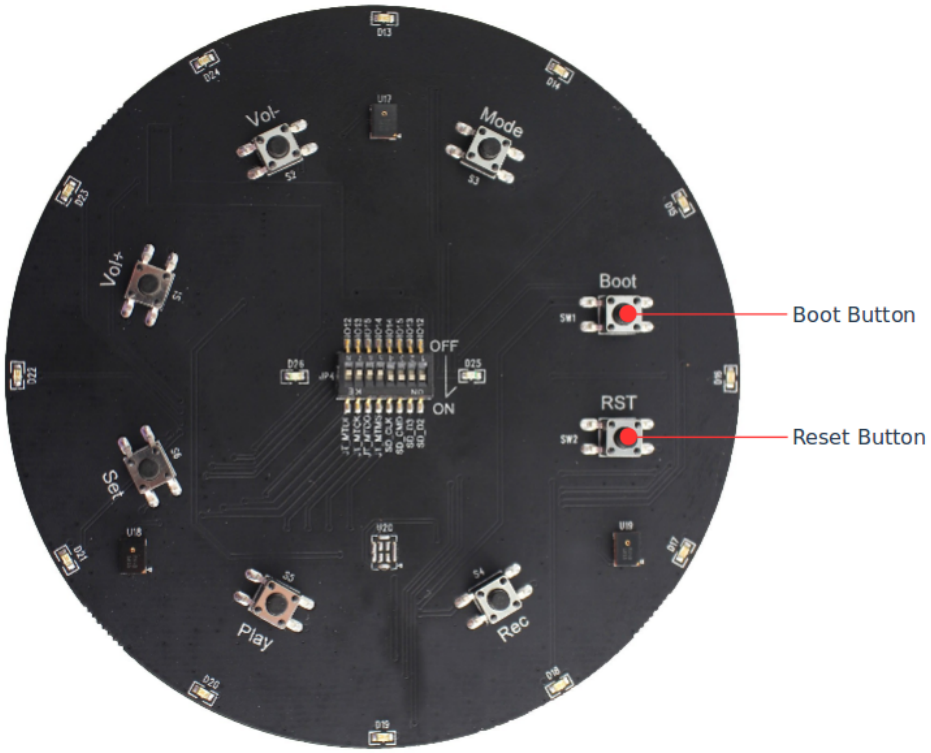


Fig. 27: ESP32-LyraTD-MSC V2.2 Upper Board (B) Components

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

Reset: pressing this button alone resets the system.

Start Application Development

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

Initial Setup

Prepare the board for loading of the first sample application:

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

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

Develop Applications

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Installation Step by Step* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Revision History

- Changed the integrated module to ESP32-WROVER-E from ESP32-WROVER-B.

Other Boards from LyraT Family

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

Related Documents

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

3.4.8 ESP32-Korvo-DU1906

This user guide provides information on ESP32-Korvo-DU1906.

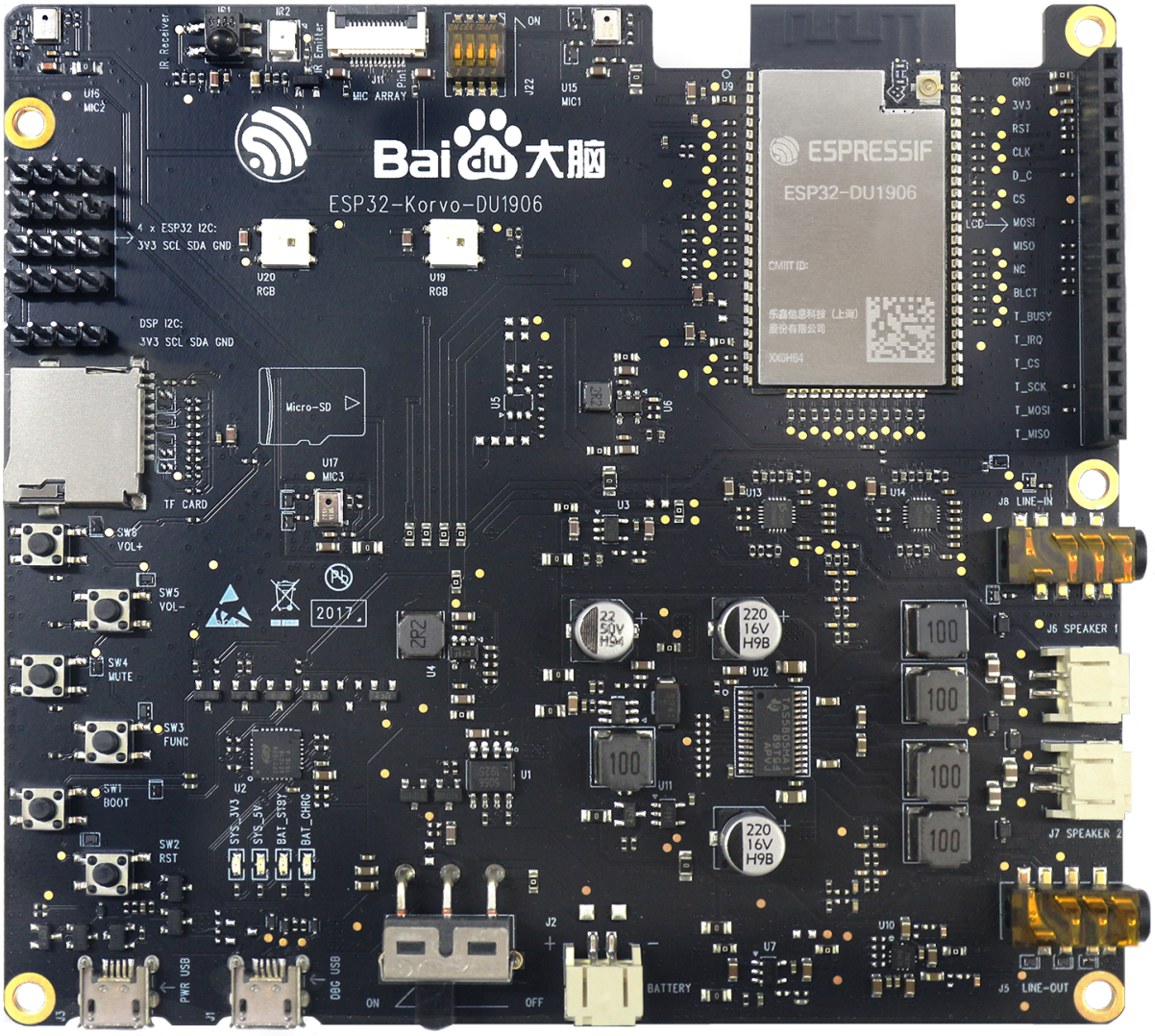


Fig. 28: ESP32-Korvo-DU1906 (click to enlarge)

The document consists of the following major sections:

- *Getting Started*: Provides an overview of the ESP32-Korvo-DU1906 and hardware/software setup instructions to get started.
- *Start Application Development*: Provides more detailed information about the ESP32-Korvo-DU1906's application development process.
- *Related Documents*: Gives links to related documentaiton.

Getting Started

The core component of ESP32-Korvo-DU1906 includes an ESP32-DU1906 Bluetooth/Wi-Fi audio module, which is able to realize noise reduction, acoustic echo cancellation (AEC), beam formation and detection. ESP32-Korvo-DU1906 integrates power management, Bluetooth/Wi-Fi audio module, Coder-Decoder (CODEC), power amplifier, and etc., supports various functions such as:

- ADC
- Microphone array
- SD card
- Functional buttons
- Indicator lights
- Battery constant-current/constant-voltage linear power management chip
- USB-to-UART conversion
- LCD connector

What You Need

- 1 x PC loaded with Windows, Mac OS and Linux (Linux Operating System is recommended)
- 1 x ESP32-Korvo-DU1906
- 2 x Micro USB cables
- 2 x Speaker (4 Ohm, 2.5 W)

Overview

The biggest advantage of this development board is the audio chip – ESP32-DU1906, the core processing module, is an powerful AI module integrating Wi-Fi+Bluetooth+Bluetooth Low Energy RF and voice/speech signal processing functions, which can be used in various fields. By providing the leading end-to-end audio solutions, high-efficient integrated AI service capabilities, and an on-device AIOT platform which integrates ends and devices, this board is able to largely reduce the threshold for AI access.

DU1906 is a voice processing flagship chip launched by Baidu. This chip has a highly integrated algorithm, which can solve the industrial needs of real-time processing of far-field array signals, and high-precision voice wake-up and real-time monitoring with ultra-low error occurs simultaneously on this single one chip.

The block diagram below presents main components of the ESP32-Korvo-DU1906 and interconnections between components.

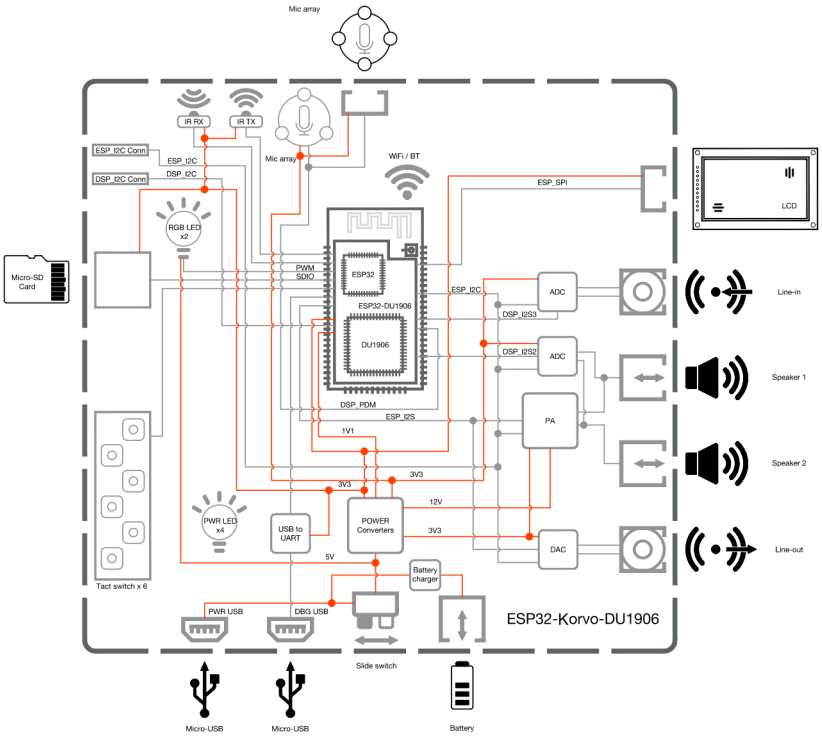


Fig. 29: ESP32-Korvo-DU1906 Block Diagram (click to enlarge)

Description of Components

The following list and figure describe key components, interfaces and controls of the ESP32-Korvo-DU1906 used in this guide. This covers just what is needed now. For additional details please refer to schematics provided in Related Documents.

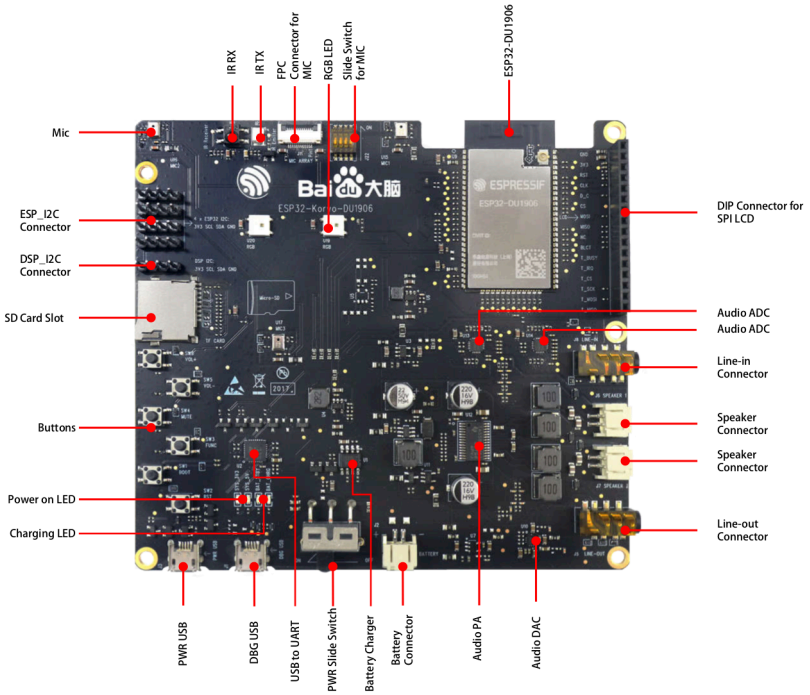


Fig. 30: ESP32-Korvo-DU1906 Components (click to enlarge)

Key Componenet	Description
ESP32-DU1906	This is a powerful, general-purpose, Wi-Fi/Bluetooth audio communication module, targeting a wide variety of applications ranging from low-power sensor networks to the most demanding tasks, such as voice encoding/decoding, music streaming and running voice assistant client SDK.
DIP Connector for SPI LCD	ESP32-Korvo-DU1906 has a 2.54 mm pitch connector to connect SPI LCD.
Audio ADC (Audio Analog-to-Digital Converter)	ESP32-Korvo-DU1906 includes two ES7243 high-efficiency ADCs, with one for the collection of Audio PA outputs, and another for the collection of Line-in outputs. Both ADCs can be used for AEC.
Line-in/out Connector (Earphone Jacks)	The two earphone jacks are used to connect to Line-out outputs of Audio DAC and Line-in inputs of Audio ADC respectively. When a device is plugged in the Line-out earphone jack of Audio DAC, the Audio PA on the ESP32-Korvo-DU1906 will be turned off.
Speaker Connector	Output sockets to connect two 4-ohm speakers to provide stereo sound via digital Audio PA.
Audio DAC (Audio Digital-to-Analog Converter)	ES7148 Stereo DAC is able to convert digital signals into analog audio outputs.
Audio PA (Digital Audio Power Amplifier)	TAS5805M is a high-efficiency stereo closed-loop D-type amplifier with low power dissipation and rich sound. It can convert audio digital signals into high-power analog audio outputs and transmit them to external speakers for playback. When the Line-out earphone jack of the audio DAC plugged into the device, the Digital Audio PA on the ESP32-Korvo-DU1906 will be turned off.
Battery Connector	Connect a battery.
Battery Charger	AP5056, a constant-current/constant-voltage linear power management chip, can be used for charging management to a single lithium-ion battery.
PWR Slide Switch	Power switch for the board, turn on/off the power supply.
USB to UART	CP2102N supports USB-to-UART conversion for easy download and debugging of software.
DBG USB (Debugging USB)	USB communication between PC and ESP32-DU1906 module.
PWR USB (Power supply USB)	Provide power supply for the whole system. It is recommended to connect to an at least 5 V / 2 A power adapter for sufficient current supply.
Charging LEDs	Indicating battery state. When a battery is connected, BAT_CHRG LED will turn red (indicating the battery is charging), then BAT_STBY LED will turn

Start Application Development

Before powering up the ESP32-Korvo-DU1906, 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 two **Speaker Connectors**. Connecting earphones to the **Line-out Connector** is an option.
2. Plug in the Micro-USB cables to the PC and to both **USB connectors** of the ESP32-Korvo-DU1906.
3. Assuming that a battery is connected, the **Charging LED** (red) will keep the lights on.
4. Toggle left the **PWR Slide 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

Once the board is initially set up and checked, you can start preparing the development tools. The Section *Development Boards* will walk you through the following steps:

- **Set up ESP-IDF** to get a common development framework for the ESP32 (and ESP32-S2) chips in C language;
- **Get ESP-ADF** to install the API specific to audio applications;
- **Set up env** to make the framework aware of the audio specific API;
- **Start a Project** that will provide a sample audio application for the board;
- **Connect Your Device** to prepare the application for loading;
- **Build the Project** to finally run the application and play some music.

Other Related Boards

- *ESP32-LyraT V4.3 Getting Started Guide*
- *ESP32-LyraT-Mini V1.2 Getting Started Guide*
- *ESP32-LyraTD-MSC V2.2 Getting Started Guide*

Contents and Packaging

Retail orders

If you order one or several samples, each board will come in a plastic package or other package chosen by the retailer.

For retail orders, please go to https://www.espressif.com/zh-hans/products/devkits/esp32-korvo-du1906.

Related Documents

- ESP32-Korvo-DU1906 Schematic (PDF)
- ESP32 Datasheet (PDF)

3.4.9 ESP32-S3-Korvo-2 V3.0

This user guide will help you get started with ESP32-S3-Korvo-2 V3.0 and will also provide more in-depth information.

The ESP32-S3-Korvo-2 is a multimedia development board based on the ESP32-S3 chip. It is equipped with a two-microphone array which is suitable for voice recognition and near/far-field voice wake-up applications. The board integrates multiple peripherals such as LCD, camera, and microSD card. It also supports JPEG video stream processing. With all of its outstanding features, the board is an ideal choice for the development of low-cost and low-power network-connected audio and video products.

This board mainly consists of the following parts:

- Main board: ESP32-S3-Korvo-2
- LCD extension board: *ESP32-S3-Korvo-2-LCD*
- Camera

This document is mostly dedicated to the main board. For detailed information on other parts, click the links above.

The document consists of the following sections:

- *Getting started*: Overview of the board and hardware/software setup instructions to get started.
- *Hardware Reference*: More detailed information about the board's hardware.
- *Hardware Revision Details*: Hardware revision history, known issues, and links to user guides for previous versions (if any) of the board.
- *Related Documents*: Links to related documentation.

Getting Started

This section provides a brief introduction of ESP32-S3-Korvo-2 V3.0, instructions on how to do the initial hardware setup and how to flash firmware onto it.

Fig. 31: ESP32-S3-Korvo-2 V3.0 with ESP32-S3-WROOM-1 module

Description of Components

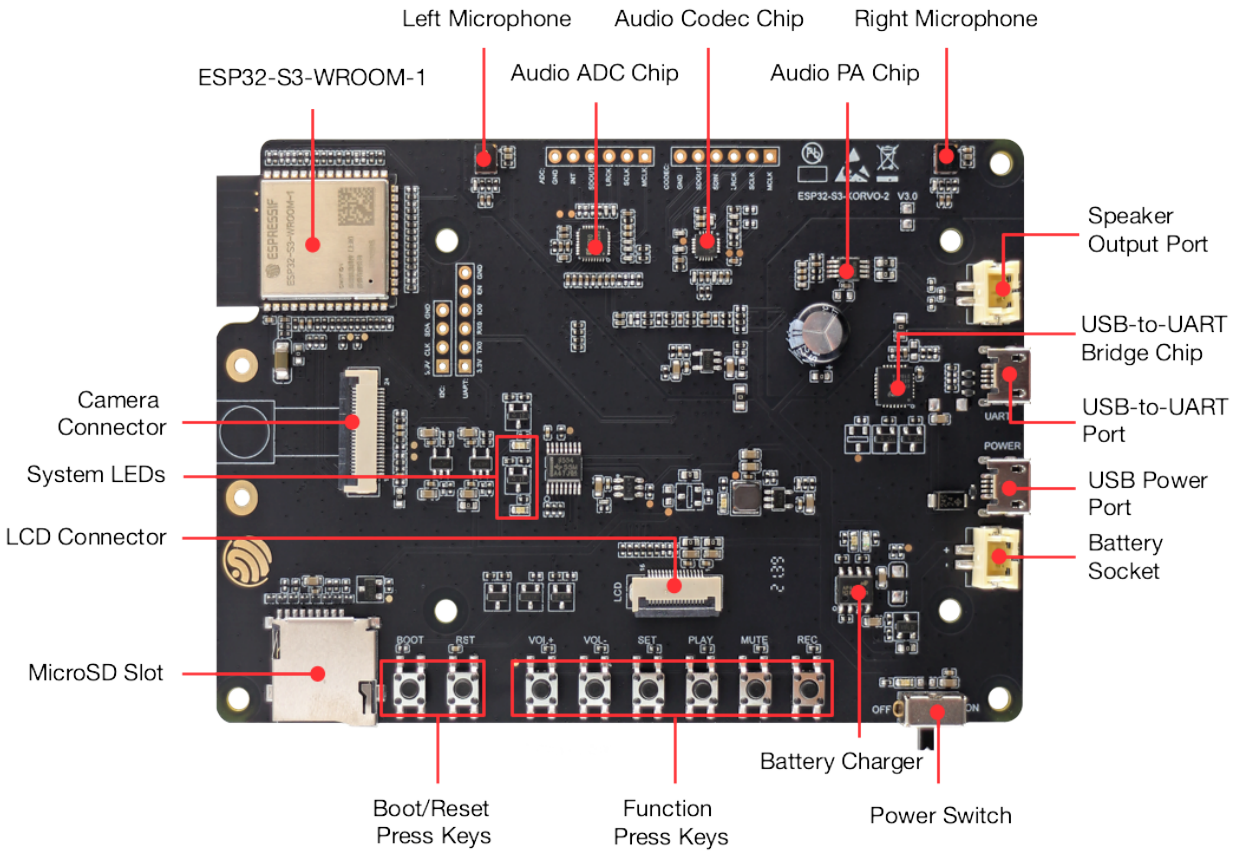


Fig. 32: ESP32-S3-Korvo-2 V3.0 (click to enlarge)

The key components of the board are described in a clockwise direction.

Key Component	Description
ESP32-S3-WROOM-1 Module	The ESP32-S3-WROOM-1 is a powerful, generic Wi-Fi + Bluetooth LE MCU module that is built around the ESP32-S3 series of SoCs. On top of a rich set of peripherals, the acceleration for neural network computing and signal processing workloads provided by the SoC make the modules an ideal choice for a wide variety of application scenarios related to AI and Artificial Intelligence of Things (AIoT), such as wake word detection, speech commands recognition, face detection and recognition, smart home, smart appliances, smart control panel, smart speaker, etc.
Left Microphone	Onboard microphone connected to ADC.
Audio ADC Chip	ES7210 is a high-performance, low-power 4-channel audio analog-to-digital converter for microphone array applications. It is very suitable for music and speech applications. In addition, ES7210 can also be used to collect acoustic echo cancellation (AEC) echo reference signals.
Audio Codec Chip	The audio codec chip, ES8311, is a low-power mono audio codec. It consists of 1-channel ADC, 1-channel DAC, low noise pre-amplifier, headphone driver, digital sound effects, analog mixing, and gain functions. It is interfaced with ESP32-S3-WROOM-1 module over I2S and I2C buses to provide audio processing in hardware independently from the audio application.
Audio PA Chip	NS4150 is an EMI, 3 W mono Class D audio power amplifier, amplifying audio signals from audio codec chips to drive speakers.
Right Microphone	Onboard microphone connected to ADC.
Speaker Output Port	Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm/0.08” pitch.
USB-to-UART Bridge Chip	A single chip USB-UART bridge CP2102N provides up to 3 Mbps transfer rates for software download and debugging.
USB-to-UART Port	Functions as the communication interface between a PC and the ESP32-S3-WROOM-1 module.
USB Power Port	Provides power to the board. It is recommended to use at least 5V/2A power adapter to ensure a stable power supply.
Battery Socket	Two pins socket to connect a single cell Li-ion battery.
Power Switch	Power on/off knob: toggling it down powers the board on; toggling it up powers the board off.
Battery Charger	AP5056 is a constant current and constant voltage linear charger for single cell lithium-ion batteries. Used for charging of a battery connected to the Battery Socket over the Micro USB Port.
Function Press Keys	Six press keys labeled REC, MUTE, PLAY, SET, VOL- and VOL+. They are routed to ESP32-S3-WROOM-1 module and intended for development and testing of a UI for audio applications using a dedicated API.
Boot/Reset Press Keys	Boot: holding down the Boot key and momentarily pressing the Reset key initiates the firmware upload mode. Then you can upload firmware through the serial port. Reset: pressing this button alone resets the system.
MicroSD Slot	The development board supports a microSD card in 1-bit mode, and can store or play audio files in the microSD card.
LCD Connector	A FPC connector with 0.5 mm pitch to connect to the LCD extension board.
System LEDs	Two general-purpose LEDs (green and red) controlled by ESP32-S3-WROOM-1 module to indicate certain operation states of the audio application using dedicated API.
Camera Connector	An external camera module that can be connected to the development board with the connector to transmit images.

Start Application Development

Before powering up your board, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- 1 x ESP32-S3-Korvo-2 V3.0
- 1 x Speaker
- 2 x USB 2.0 cable (Standard-A to Micro-B)
- 1 x Computer running Windows, Linux, or macOS

Note: Be sure to use an appropriate USB cable. Some cables are for charging only and do not provide the needed data lines nor work for programming the boards.

Optional Hardware

- 1 x MicroSD card
- 1 x Li-ion battery

Note: Be sure to use a Li-ion battery that has a built-in protection circuit.

Hardware Setup

1. Connect the speaker to the **Speaker Output**.
2. Plug in the USB cables to the PC and to both USB ports of the board.
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 the **Power Switch**.
5. The red Power On LED should turn on.

Software Setup

Please proceed to *Get Started*, where Section *Installation Step by Step* will quickly help you set up the development environment and then flash an application example onto your board.

Contents and Packaging

The main board and its accessories can be ordered separately. The accessories include:

- LCD extension board: ESP32-S3-Korvo-2-LCD
- Camera
- Connectors:
  - 20-pin FPC cable
- Fasteners:
  - Copper standoffs (x8)
  - Screws (x4)

Retail Orders

If you order a few samples, each board comes in an individual package in either antistatic bag or any packaging depending on your retailer.

For retail orders, please go to https://www.espressif.com/en/company/contact/buy-a-sample.

Wholesale Orders

If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to https://www.espressif.com/en/contact-us/sales-questions.

Hardware Reference

Block Diagram

The block diagram below shows the components of ESP32-S3-Korvo-2 V3.0 and their interconnections.

Notes on Power Distribution

Power Supply over USB and from Battery

The main power supply is 5 V and provided by a USB. The secondary power supply is 3.7 V and provided by an optional battery. The USB power itself is fed with a dedicated cable, separating 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.

As shown in the figure below, if the USB power supply and battery power supply are connected at the same time with a high VBUS, an off-state Q14, and an automatic cut-off VBAT, the USB becomes the power supply for the system.

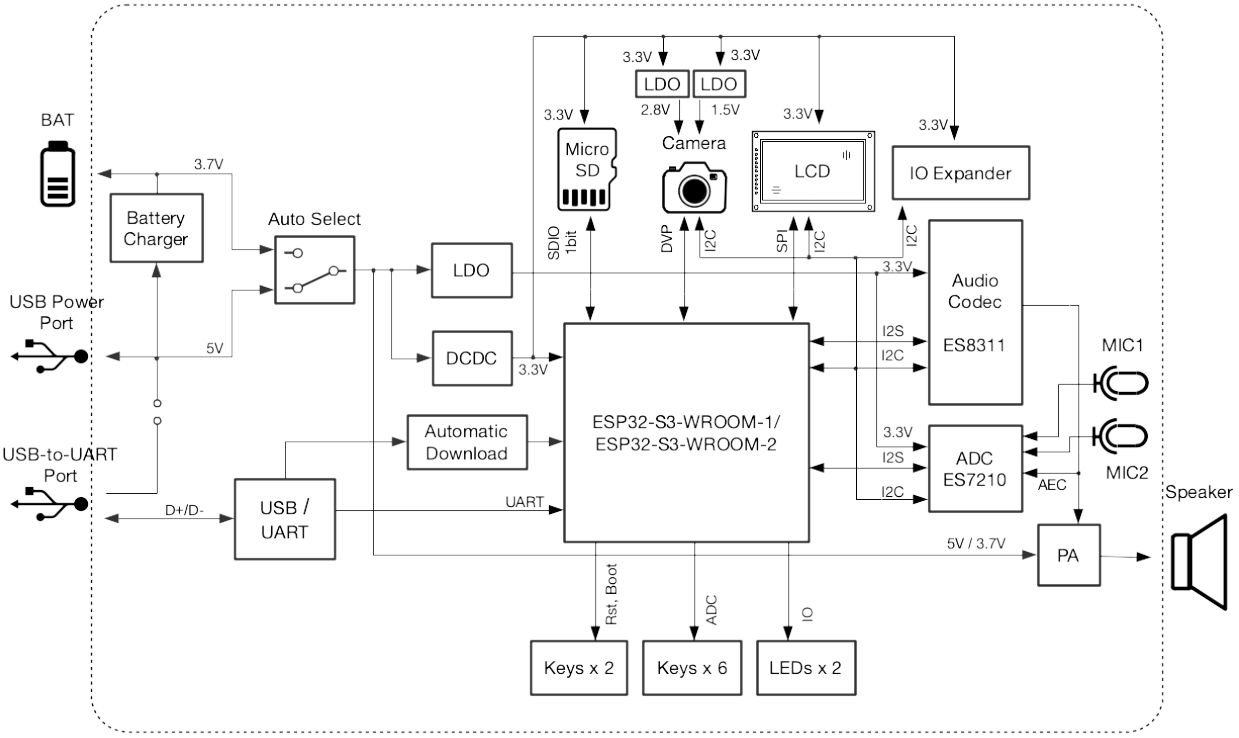


Fig. 33: ESP32-S3-Korvo-2 V3.0 Electrical Block Diagram

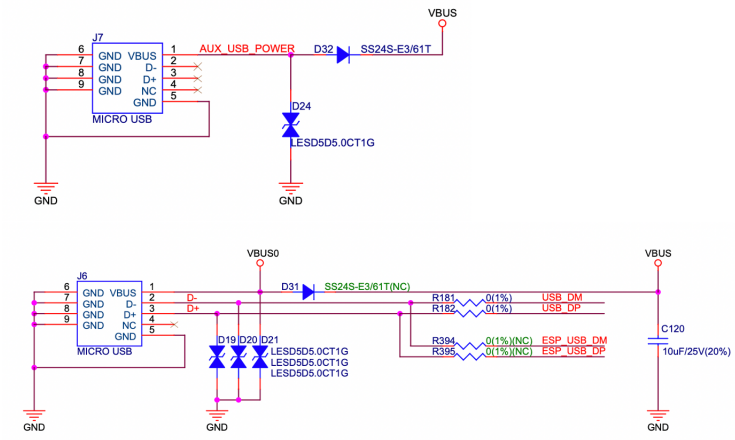


Fig. 34: ESP32-S3-Korvo-2 V3.0 - Dedicated USB Power Supply Socket

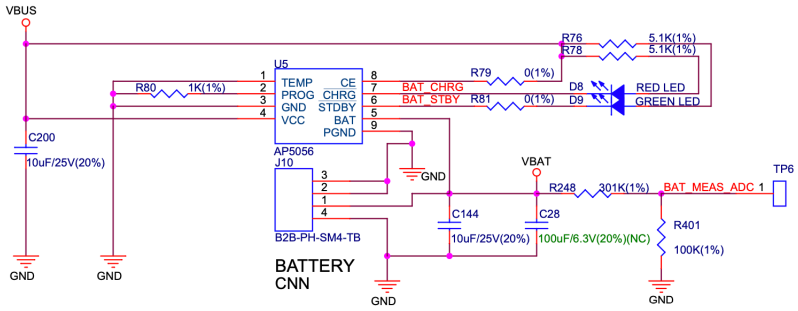


Fig. 35: ESP32-S3-Korvo-2 V3.0 - Power Supply from a Battery

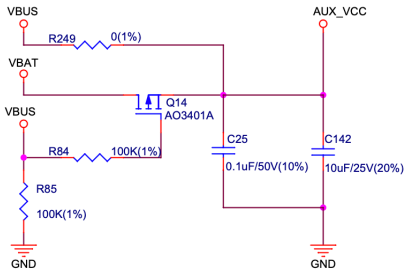


Fig. 36: ESP32-S3-Korvo-2 V3.0 - Power Supply Options

Independent Audio and Digital Power Supply

ESP32-S3-Korvo-2 V3.0 features independent power supplies to the audio components and ESP module. This should reduce noise in the audio signal from digital components and improve overall performance of the components.

Power for Digital:

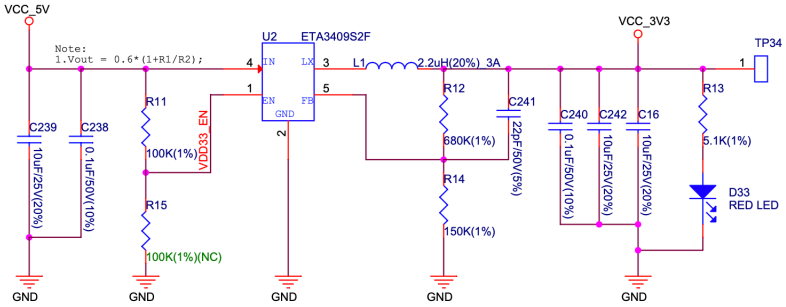


Fig. 37: ESP32-S3-Korvo-2 V3.0 - Digital Power Supply

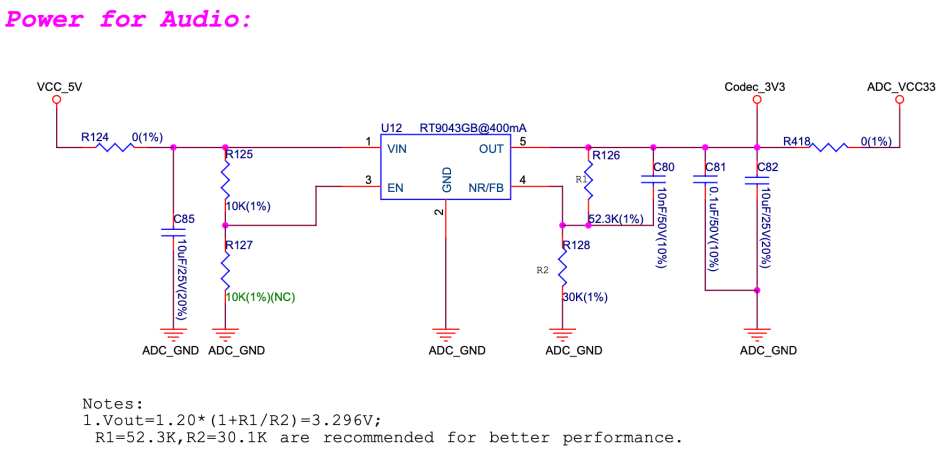


Fig. 38: ESP32-S3-Korvo-2 V3.0 - Audio Power Supply

GPIO Allocation Summary

The table below provides allocation of GPIOs exposed on terminals of ESP32-S3-WROOM-1 module to control specific components or functions of the board.

Table 2: ESP32-S3-WROOM-1 GPIO Allocation

Pin[Page 399, 1]	Pin Name	ES8311	ES7210	Camera	LCD	Keys	MicroSD	IO Expander	Other
3	EN					EN_KEY			
4	IO4						DATA0		
5	IO5					REC, MUTE, PLAY, SET, VOL-, VOL+			
6	IO6								BAT_MEAS_ADC
7	IO7						CMD		
8	IO15						CLK		
9	IO16	I2S0_MCLK	MCLK						
10	IO17	I2C_SDA	I2C_SDA	SIOD	TP_I2C_SDA			I2C_SDA	
11	IO18	I2C_CLK	I2C_CLK	SIOC	TP_I2C_CLK			I2C_CLK	
12	IO8	I2S0_DSDIN							
13	IO19								ESP_USB_DM (Reserve)
14	IO20								ESP_USB_DP (Reserve)
15	IO3			D5					
16	IO46								NC
17	IO9	I2S0_SCLK	SCLK						
18	IO10		SDOUT						

continues on next page

Table 2 – continued from previous page

Pin[1]	Pin Name	ES8311	ES7210	Camera	LCD	Keys	MicroSD	IO Expander	Other
19	IO11			PCLK					
20	IO12			D6					
21	IO13			D2					
22	IO14			D4					
23	IO21			VSYNC					
24	IO47			D3					
25	IO48								PA_CTRL
26	IO45	I2S0_LRCK	LRCK						
27	IO0				LCD_SPI_SDO	BOOT_KEY			
28	IO35								NC
29	IO36								NC
30	IO37								NC
31	IO38			HREF					
32	IO39			D9					
33	IO40			XCLK					
34	IO41			D8					
35	IO42			D7					
36	RXD0								ESP0_UART0_RX
37	TXD0								ESP0_UART0_TX
38	IO2				LCD_SPI_DC				
39	IO1				LCD_SPI_CLK				
41	EPAD								

The GPIOs allocated to the IO expander are further expanded to multiple GPIOs.

Table 3: IO Expander GPIO Allocation

IO Expander Pin	Pin Name	LCD	Other
4	P0		PA_CTRL
5	P1	LCD_CTRL	
6	P2	LCD_RST	
7	P3	LCD_CS	
9	P4	TP_INT	
10	P5		PERI_PWR_ON
11	P6		LED1
12	P7		LED2

[1] Pin - ESP32-S3-WROOM-1 module pin number, GND and power supply pins are not listed.

Connector

Camera Connector

No.	Camera Signal	ESP32-S3 Pin
1	SIOD	GPIO17
2	SIOC	GPIO18
3	D5	GPIO3
4	PCLK	GPIO11
5	D6	GPIO12
6	D2	GPIO13
7	D4	GPIO14
8	VSYNC	GPIO21
9	D3	GPIO47
10	HREF	GPIO38
11	D9	GPIO39
12	XCLK	GPIO40
13	D8	GPIO41
14	D7	GPIO42

LCD Connector

No.	LCD Signal	ESP32-S3 Pin
1	TP_I2C_SDA	GPIO17
2	TP_I2C_CLK	GPIO18
3	LCD_SPI_SDA	GPIO0
4	LCD_SPI_DC	GPIO2
5	LCD_SPI_CLK	GPIO1

No.	LCD Signal	IO Expander
1	ESP_LCD_CTRL	P1
2	ESP_LCD_RST	P2
3	ESP_LCD_CS	P3
4	ESP_TP_INT	P4

AEC Path

AEC path provides reference signals for AEC algorithm.

ESP32-S3-Korvo-2 provides two compatible echo reference signal source designs. One is Codec (ES8311) DAC output (DAC_AOUTLN/DAC_AOUTLP), the other is PA (NS4150) output (PA_OUTL+/PA_OUTL-). The former is the default and recommended selection. Resistors R132 and R140 marked NC (no component) in the figure below should not be installed.

The echo reference signal is collected by ADC_MIC3P/ADC_MIC3N of ADC (ES7210) and then sent back to ESP32-S3 for AEC algorithm.

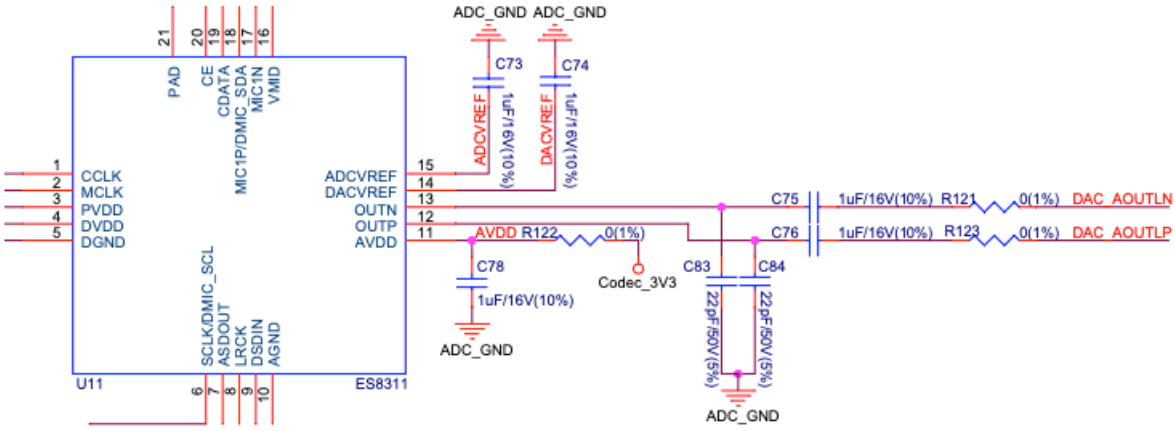


Fig. 39: ESP32-S3-Korvo-2 V3.0 - AEC Codec DAC Output (click to enlarge)

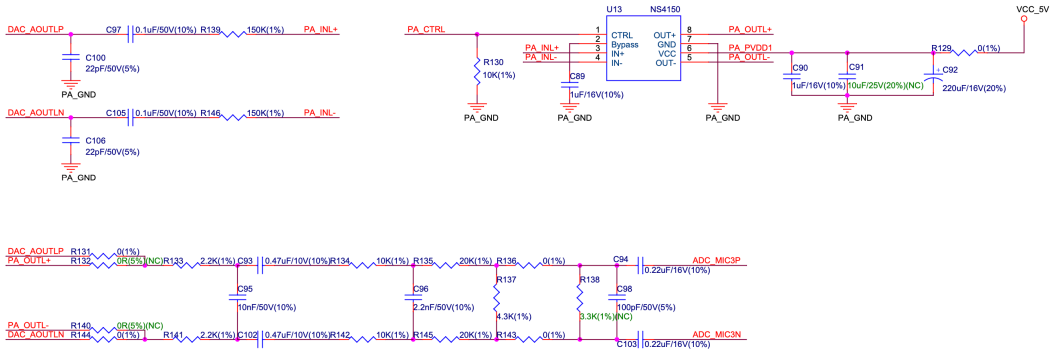


Fig. 40: ESP32-S3-Korvo-2 V3.0 - AEC PA Output (click to enlarge)

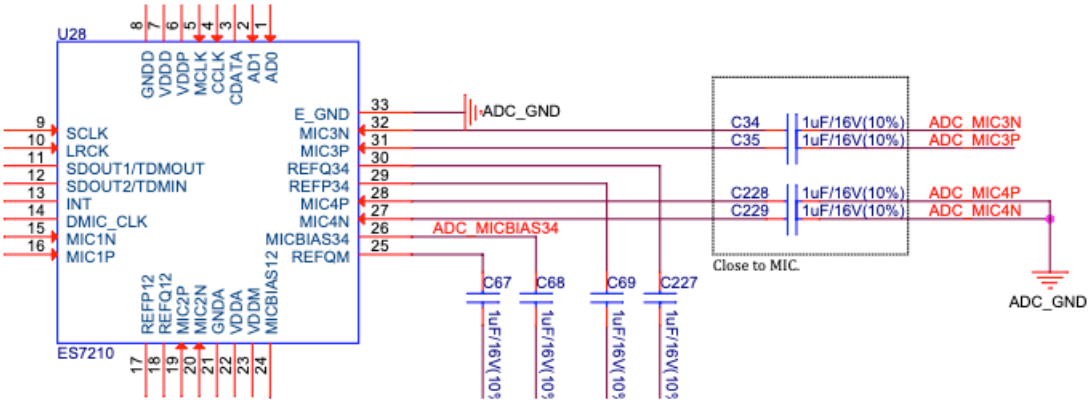


Fig. 41: ESP32-S3-Korvo-2 V3.0 - AEC Reference Signal Collection (click to enlarge)

Hardware Setup Options

Using Automatic Upload

Entering of the ESP board 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 and IO0 of the ESP board. For details see ESP32-S3-Korvo-2 V3.0 Schematic (PDF).

Allocation of ESP Pins to Test Points

This section describes the allocation of test points available on the ESP32-S3-Korvo-2 V3.0 board.

The test points are bare through hole solder pads and have a standard 2.54 mm/0.1" pitch. You may need to populate them with pin headers or sockets for easy connection of external hardware.

Codec Test Point/J15

No.	Codec Pin	ESP32-S3 Pin
1	MCLK	GPIO16
2	SCLK	GPIO9
3	LRCK	GPIO45
4	DSDIN	GPIO8
5	ASDOUT	–
6	GND	–

ADC Test Point/J16

No.	ADC Pin	ESP32-S3 Pin
1	MCLK	GPIO16
2	SCLK	GPIO9
3	LRCK	GPIO45
4	SDOUT	GPIO10
5	INT	–
6	GND	–

UART Test Point/J17

No.	UART Pin
1	3.3V
2	TXD
3	RXD
4	IO0
5	EN
6	GND

I2C Test Point/J18

No.	I2C Pin	ESP32-S3 Pin
1	3.3V	–
2	CLK	GPIO18
3	SDA	GPIO17
4	GND	–

Hardware Revision Details

This is the first revision of this board released.

Related Documents

- ESP32-S3 Series Datasheet (PDF)
- ESP32-S3-WROOM-1/1U Datasheet (PDF)
- ESP32-S3-Korvo-2 V3.0 Schematic (PDF)
- ESP32-S3-Korvo-2 V3.0 PCB Layout (PDF)

For further design documentation for the board, please contact us at sales@espressif.com.

3.4.10 ESP32-S3-Korvo-2-LCD V1.0

This user guide will help you get started with the ESP32-S3-Korvo-2-LCD extension board and will also provide more in-depth information.

This extension board cannot be bought separately and is sold as part of the *ESP32-S3-Korvo-2 V3.0 accessories*.

ESP32-S3-Korvo-2-LCD extends the functionality of ESP32-S3-Korvo-2 V3.0 (referred to as *main board* below) by adding an LCD graphic display and capacitive touchpad.

The document consists of the following major sections:

- *Getting started*: Overview of the board and hardware/software setup instructions to get started.
- *Hardware Reference*: More detailed information about the board's hardware.
- *Related Documents*: Links to related documentation.

Getting Started

This extension board adds a 2.4" LCD graphic display with the resolution of 320x240 and a 10-point capacitive touchpad. This display is connected to ESP32-S3 over the SPI bus.

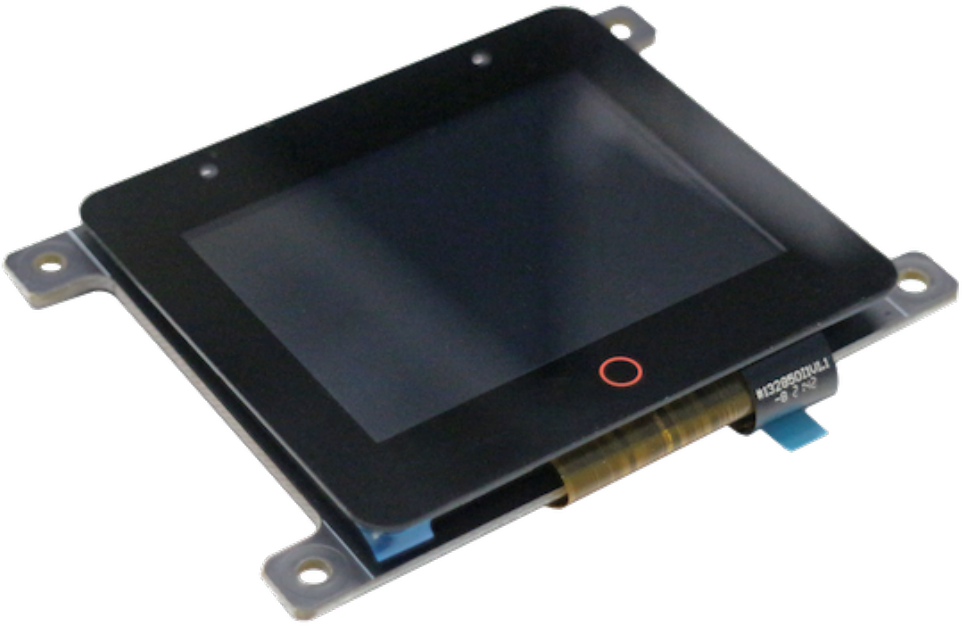


Fig. 42: ESP32-S3-Korvo-2-LCD V1.0

Description of Components

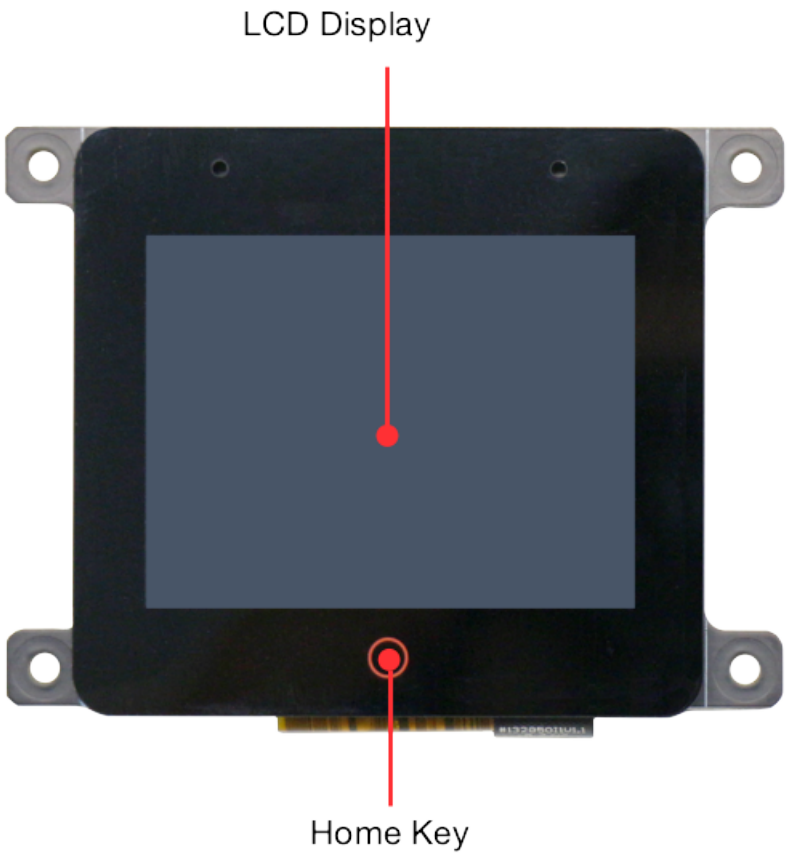


Fig. 43: ESP32-S3-Korvo-2-LCD V1.0 - front

The key components of the board are described in a clockwise direction. **Reserved** means that the functionality is available, but the current version of the board does not use it.

Key Component	Description
LCD Display	A 2.4" 320x240 SPI LCD display module; the display driver/controller is Ilitek ILI934.
Home Key	(Reserved) Returns to homepage or previous page.
Signal Connector	Connects the power, ground, and signal wires between the extension board and the main board with a FPC cable.
LCD Connector	Connects LCD display to the driver circuit of this board.
TP Connector	Connects LCD display to the touch circuit of this board.

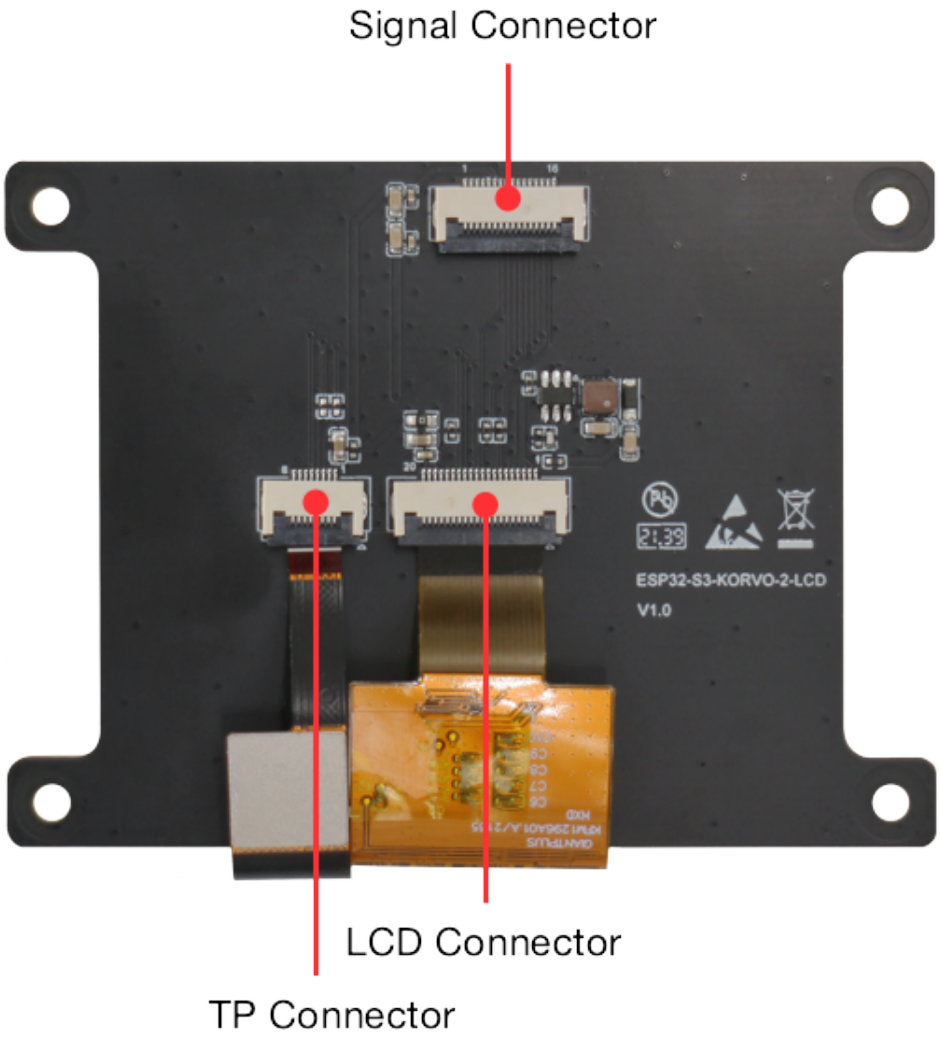


Fig. 44: ESP32-S3-Korvo-2-LCD V1.0 - back

Start Application Development

Before powering up your board, please make sure that it is in good condition with no obvious signs of damage.

Required Hardware

- Main board: ESP32-S3-Korvo-2 V3.0
- Extension board: ESP32-S3-Korvo-2-LCD V1.0
- Two USB 2.0 cables (Standard-A to Micro-B)
- Mounting copper standoffs and screws (for stable mounting)
- A FPC cable (for connecting the main board and extension board)
- Computer running Windows, Linux, or macOS

Hardware Setup

To mount your ESP32-S3-Korvo-2-LCD onto ESP32-S3-Korvo-2:

1. Connect the two boards with the FPC cable.
2. Install copper standoffs and screws for stable mounting.

Software Setup

See Section *Software Setup* of the main board user guide.

Hardware Reference

Block Diagram

The block diagram below shows the components of ESP32-S3-Korvo-2-LCD and their interconnections.

Hardware Revision Details

Initial release.

Related Documents

- *ESP32-S3-Korvo-2 V3.0*
- ESP32-S3-Korvo-2-LCD Schematic (PDF)
- ESP32-S3-Korvo-2-LCD PCB layout (PDF)

For further design documentation for the board, please contact us at sales@espressif.com.

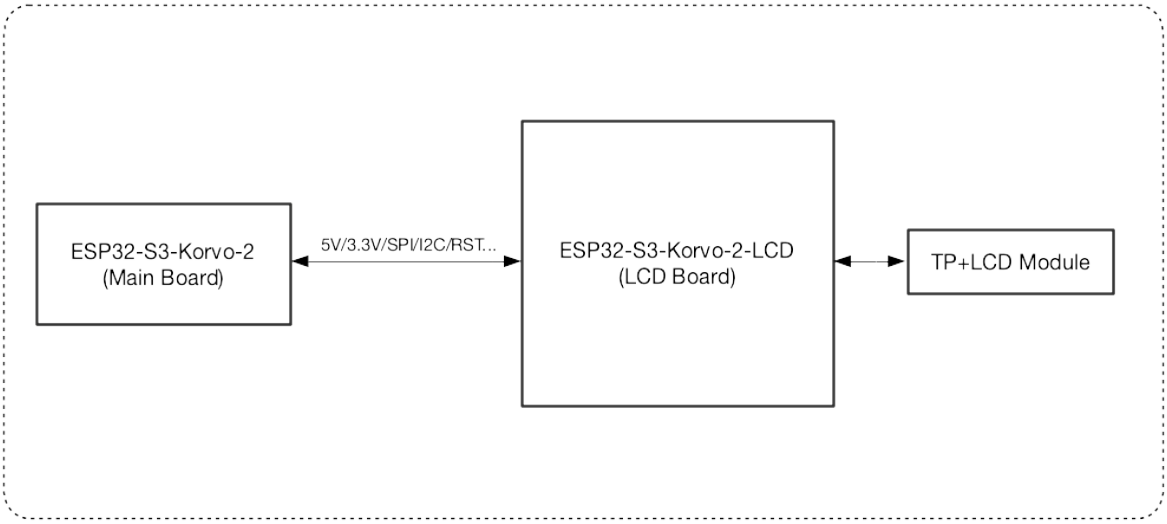


Fig. 45: ESP32-S3-Korvo-2-LCD

3.4.11 ESP32-C3-Lyra V2.0

This user guide will help you get started with ESP32-C3-Lyra V2.0 and will also provide more in-depth information.

The document consists of the following sections:

- *Board Overview*: Overview of the board hardware/software.
- *Start Application Development*: How to set up hardware/software to develop applications.
- *Hardware Reference*: More detailed information about the board's hardware.
- *Hardware Revision Details*: Hardware revision history, known issues, and links to user guides for previous versions (if any) of the board.
- *Ordering*: How to buy the board.
- *Related Documents*: Links to related documentation.

Board Overview

ESP32-C3-Lyra is an ESP32-C3-based audio development board produced by Espressif for controlling light with audio. The board has control over the microphone, speaker, and LED strip, perfectly matching customers' product development needs for ultra-low-cost and high-performance audio broadcasters and rhythm light strips.

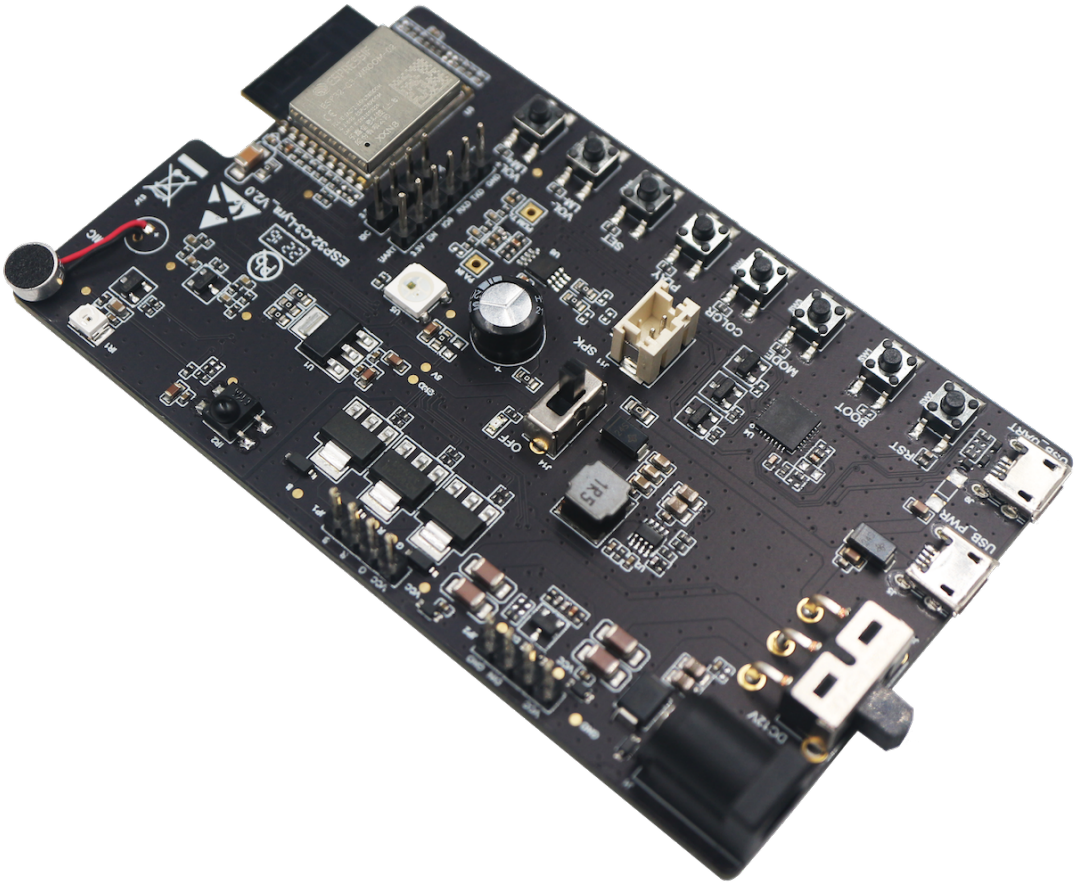


Fig. 46: ESP32-C3-Lyra with ESP32-C3-WROOM-02 module

Feature List

The main features of the board are listed below:

- **Module Embedded:** ESP32-C3-WROOM-02 module with a 4 MB external SPI flash
- **Audio:** built-in ECM microphone, speaker power amplifier, speaker connector
- **LED Strip Connector:** support for two types of connections, i.e., connections to addressable LED strips and RGB LED strips
- **Infrared Control:** support for infrared (IR) transmitting and receiving
- **Keys:** boot key, reset key, and six function keys (MODE, COLOR, PLAY/PAUSE, SET, VOL+/LM+, VOL-/LM-)
- **USB:** 1 x USB Power Port, 1 x USB-to-UART Port
- **Power Supply:** 5 V power supply over USB or 12 V DC power supply

Block Diagram

The block diagram below shows the components of ESP32-C3-Lyra and their interconnections.

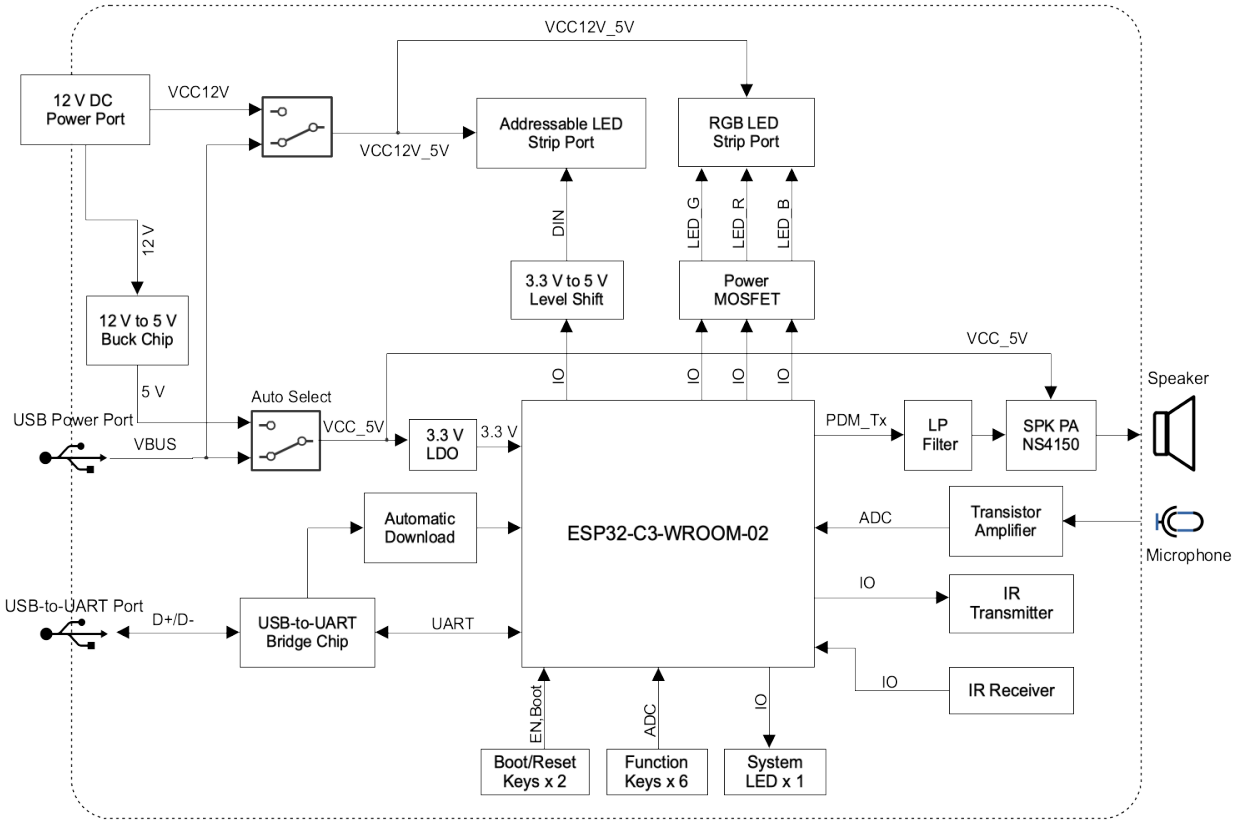


Fig. 47: ESP32-C3-Lyra Block Diagram (click to enlarge)

Description of Components

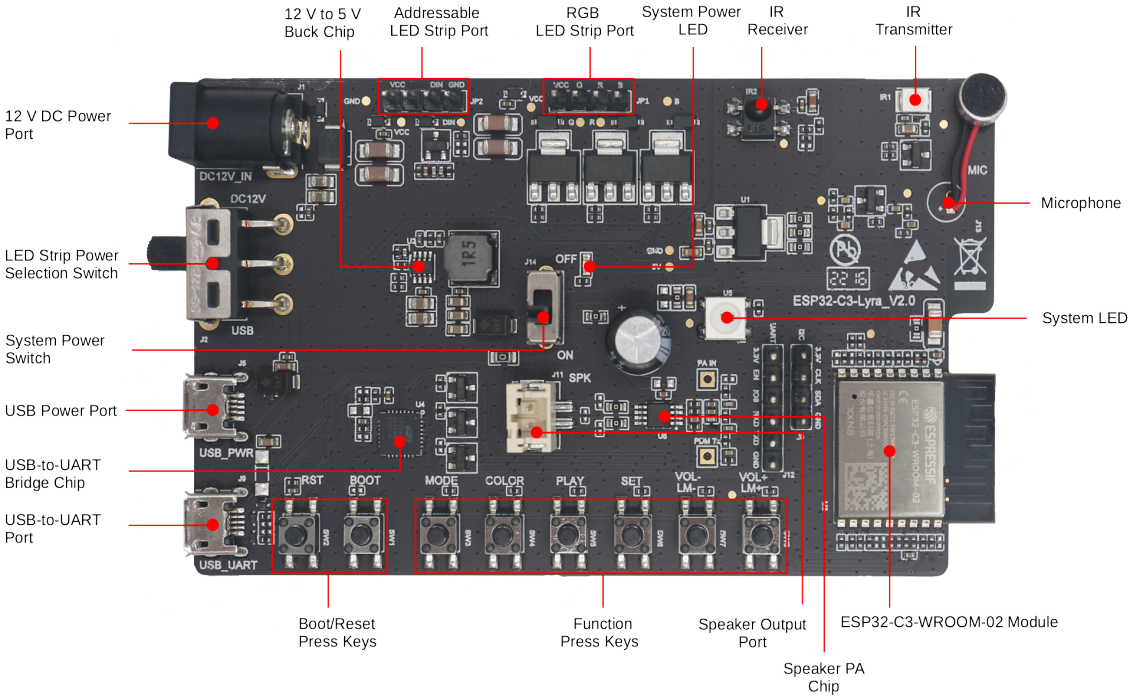


Fig. 48: ESP32-C3-Lyra - front (click to enlarge)

The key components of the board are described in a clockwise direction.

Key Component	Description
ESP32-C3-WROOM-02 Module	It is a general-purpose Wi-Fi and Bluetooth LE module developed based on ESP32-C3, a 32-bit RISC-V single-core processor that operates at up to 160 MHz. The module has a rich set of peripherals and a high performance, making it an ideal choice for smart homes, industrial automation, health care, consumer electronics, etc. It integrates a 4 MB external SPI flash and an on-board PCB antenna. The ESP32-C3-WROOM-02U module is also compatible with this board, but it needs to connect to an external antenna.
Speaker PA Chip	NS4150 is an EMI, 3 W mono Class D audio power amplifier, amplifying audio signals from ESP32-C3 PDM_TX to drive speakers.
Speaker Output Port	Output socket to connect a speaker. The 4-ohm and 3-watt speaker is recommended. The pins have a 2.00 mm/0.08" pitch.
Function Press Keys	Six function press keys, including MODE, COLOR, PLAY/PAUSE, SET, VOL+/LM+, VOL-/LM-. They are routed to the **ESP32-C3-WROOM-02 Module** and intended for the development and testing of a UI for audio applications or LED strips using dedicated APIs.
Boot/Reset Press Keys	Boot: holding down the Boot key and momentarily pressing the Reset button initiates the firmware upload mode. Then you can upload firmware through the serial port. Reset: pressing this key alone resets the system.
USB-to-UART Port	Functions as the communication interface between the PC and the ESP32-C3-WROOM-02 module.
USB-to-UART Bridge Chip	The single-chip USB-UART bridge CP2102N provides up to 3 Mbps transfer rates for software download and debugging.
USB Power Port	Provides power to the board. It is recommended to use at least 5 V/2 A power adapters to ensure a stable power supply.
System Power Switch	System Power on/off knob. Toggling it to **ON** turns the 5 V system power on; toggling it to **OFF** turns the 5 V system power off.
LED Strip Power Selection Switch	Toggle this switch to select between a 5 V power supply over USB and a 12 V DC power supply for your LED strip according to the working voltage of the LED strip and the type of the power adapter you actually use.
12 V DC Power Port	Supports 12 V DC power adapters with a maximum current of 2 A. The DC power Jack contact have an outer diameter of 5.5 mm and an inner contact diameter of 2.5 mm.
12 V to 5 V Buck Chip	The 12 V to 5 V buck chip MP2313 is a high-efficiency synchronous step-down converter that operates at 1 A and 2 MHz.
Addressable LED Strip Port	It is a male pin header connector with 4 x 1 pins and a 2.54 mm pitch. It can connect to the addressable LED strip that is controlled with a single wire. It supports 5 V and 12 V LED strips, such as WS2811 and WS2812 LED. ESP32-C3 can send commands to control the LED strips over RMT or SPI.
RGB LED Strip Port	It is a male pin header connector with 4 x 1 pins and a 2.54 mm pitch. It can connect to regular RGB LED strips (non-addressable, individual wire per color) that operate at 5 V or 12 V. ESP32-C3 can output PWM waveform via this port to control the LED strips.
System Power LED	It turns red when **System Power Switch** is toggled to ON.
IR Receiver	IRM-H638T/TR2 is a miniature SMD type infrared remote control system receiver. The demodulated output signal can directly be decoded by ESP32-C3.
IR Transmitter	IR67-21C/TR8 is an infrared emitting diode. It is spectrally matched with silicon photodiode and phototransistor.
Microphone	On-board ECM microphone. Signals picked up by it are amplified via transistors and sent to the analog-to-digital converter (ADC) of ESP32-C3-WROOM-02.
System LED	It is an RGB LED, model WS2812C, controlled by the ESP32-C3-WROOM-02 via GPIO, which can be used to indicate the operating status of the audio application.

Default Firmware and Function Test

Each ESP32-C3-Lyra comes with a pre-built default firmware that allows you to test its functions including LED control (LEDC), remote control transceiver (RMT), ADC, and pulse-density modulation (PDM_TX). This section describes how to test peripheral's function with the pre-built firmware.

Note: The default firmware of ESP32-C3-Lyra sold in the Developing IoT Projects with ESP32-C3 Package II provides a musical rhythm light ring effect.

Preparing Hardware

See Section *Required Hardware* and *Optional Hardware* for more information.

- 1 x ESP32-C3-Lyra
- 2 x USB 2.0 cable (Standard-A to Micro-B)
- 1 x Computer running Windows, Linux, or macOS
- 1 x 5 V RGB LED strip WS2812 (optional)
- 1 x Mobile phone or music player
- 1 x Speaker (optional)

Connecting Hardware

- Before powering up your board, please make sure that it is in good condition with no obvious signs of damage.
- Connect the board to the 5 V power supply through the **USB Power Port** using a USB cable. After the board is powered up, you will notice that the **System Power LED** turns on, which means the board is powered up. If the LED is not on, please toggle the **System Power Switch**.
- Toggle the **LED Strip Power Selection Switch** to the USB power side.
- Connect the board to the computer through the **USB-to-UART Port** using a USB cable.

Testing Default Firmware

Note: If you are developing with ESP32-C3-Lyra provided in the Developing IoT Projects with ESP32-C3 Package II, please skip this step.

---

1. Press the **Reset Press Key** on the board.
2. The board automatically starts the flash test. The log shown on a PC connected to **USB-to-UART Port** is as follows:

```
Step1 Flash Test Start
Step1 Flash Test OK
```

3. The board tests the **Function Press Keys**. Please press the key as the log prompts. For example, press **VOL+** when you see the following log:

```
Step2 Keys Test Start
Please press The Key: VOL+
```

4. The board tests the **System LED**. You will see the LED keep switching between red, blue, and green. Then, press the key `VOL+/LM+` to proceed to the next step.
5. The board tests LEDC (PWM). If you connect an RGB LED strip to the **RGB LED Strip Port**, you will see the LEDs breathing. Then, press the key `VOL+/LM+` to proceed to the next step.
6. The board tests ADC. If you play the 1 kHz sine audio signal close to the **Microphone** with the mobile phone or music player, the following log will be seen when the board detects the audio signal:

```
Step5 Adc Test Start
Please play 1khz audio
Step5 Adc Test OK
```

7. The board tests the PDM_TX function. Connect the speaker to the **Speaker Output Port** and you will hear the music played from flash.

Software Support

ESP-ADF is the development framework for ESP32-C3-Lyra. To see which version of ESP-ADF is supported for this board, please go to the section Hardware.

Below are other software repositories developed by Espressif that may help you experiment with the functions of ESP32-C3-Lyra.

- ESP-IDF: development framework for Espressif SoCs based on FreeRTOS with a rich set of components, including LED control (LEDC), ADC, RMT, SPI etc.

Application examples for this board can be found at application example .

Start Application Development

This section provides instructions on how to do hardware and software setup and flash firmware onto the board to develop your own application.

Required Hardware

Hardware	Qty	Note
ESP32-C3-Lyra	1	–
USB 2.0 cables (Standard-A to Micro-B)	2	One for USB power supply, the other for flashing firmware onto the board. Be sure to use an appropriate USB cable. Some cables are for charging only and do not provide the needed data lines nor work for programming the boards.
Computer running Windows, Linux, or macOS	1	–
Speaker	1	The 4-ohm 3-watt speaker is recommended. It should be fitted with JST PH 2.0 2-Pin plugs. In case you do not have this type of plug it is also fine to use Dupont female jumper wires during development.

Optional Hardware

Hardware	Qty	Note
12 V DC adapter	1	The maximum operating current of the adapter is 2 A. It provides power supply for 12 V LED strips.
5 V or 12 V addressable LED strip/ring	1	It is recommended to use WS2812/WS2811 LED strip (a female connector with 4 x 1 pins and a 2.54 mm pitch), or a WS2812 LED ring with 16 individually addressable RGB LEDs assembled (a female connector with 3 x 1 pins and a 2.54 mm pitch). This LED strip/ring should be connected to the **Addressable LED Strip Port** (*JP2*).
5 V or 12 V RGB LED strip	1	It should have a female connector with 4 x 1 pins and a 2.54 mm pitch. This LED strip should be connected to **RGB LED Strip Port** (*JP1*).

Power Supply Options

There are two ways to provide power to the board:

- **USB Power Port** (5 V)
- **12 V DC Power Port**

Hardware Setup

Prepare the board for loading of the first sample application:

1. Connect the speaker to the **Speaker Output Port**.
2. (Optional) Connect the LED strip to the development board through the **Addressable LED Strip Port** or the **RGB LED Port** depending on the type of your LED strip.
3. Connect the power supply to the development board through the **USB Power Port** (5 V) or the **12 V DC Power Port** accordingly to supply power for your LED strip.
4. (Optional) Toggle the **LED Strip Power Selection Switch** depending on the working voltage and current of your LED strip.

   Note: If you toggle the switch to the wrong side, the light strip will work abnormally. **Do not** power the 5 V LED strip with the 12 V DC adapter. Otherwise, the light strip will be damaged.

   ---

5. Toggle the **System Power Switch** to **ON**. The red **System Power LED** should turn on.
6. Connect the board to the computer through the **USB-to-UART Port** using a USB cable.

Now the board is ready for software setup.

Software Setup

After hardware setup, you can proceed to *Get Started* to prepare development tools.

For more software information on developing applications, please go to *Software Support*.

Hardware Reference

This section provides more detailed information about the board's hardware.

GPIO Allocation

The table provides the allocation of GPIOs exposed on terminals of the ESP32-C3-WROOM-02 module to control specific components or functions of the board.

Power Distribution

Power Supply over USB or from 12 V/2 A DC Input

There are two ways to power the development board: 5 V USB Power Port or 12 V/2 A DC input.

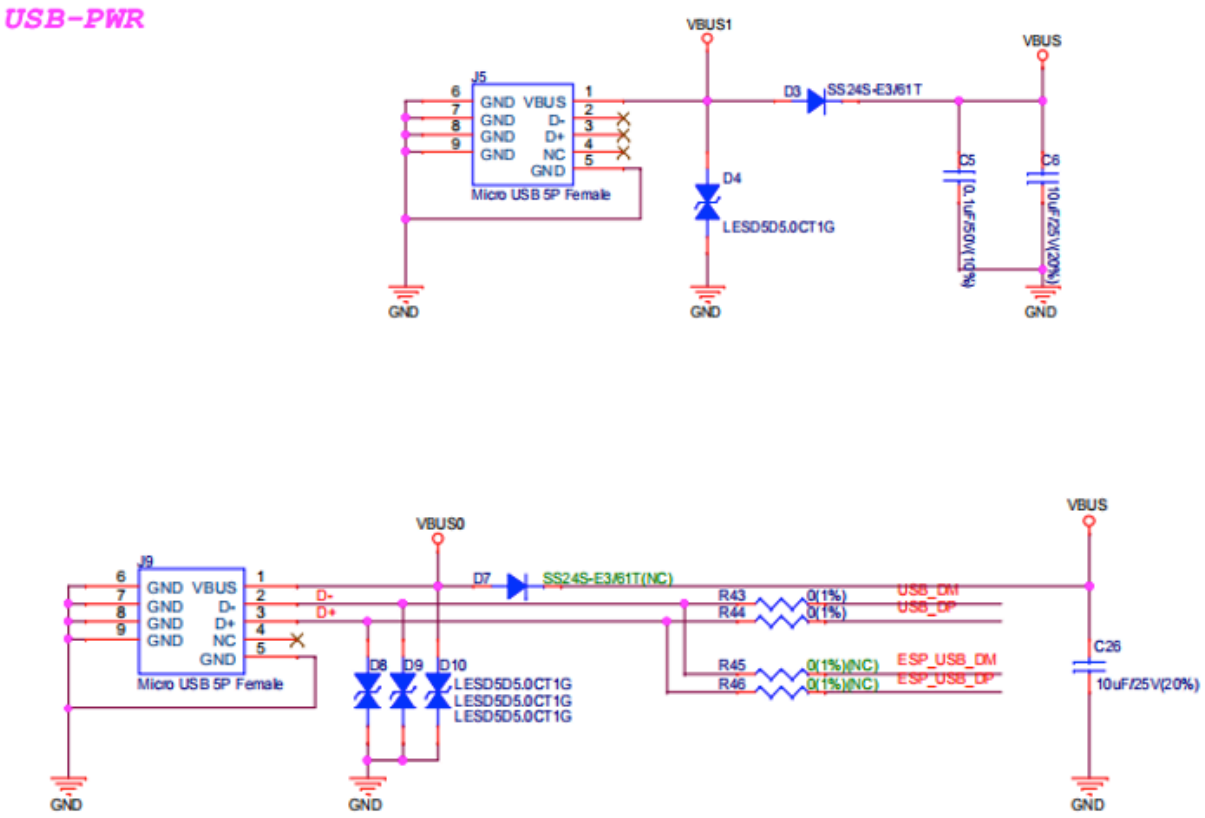


Fig. 49: ESP32-C3-Lyra - Dedicated USB Power Supply Socket

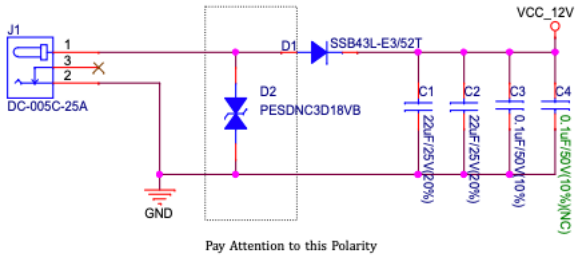


Fig. 50: ESP32-C3-Lyra - Power Supply from 12 V DC Input

LED Strip Power Selection Switch

According to the working voltage and current of your LED strip, select a proper power adapter and the port, and toggle the **LED Strip Power Selection Switch** to the corresponding side to power up the LED strip.

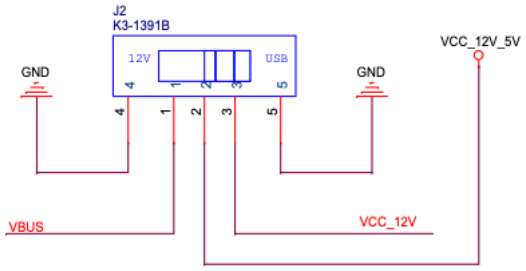


Fig. 51: LED Strip Power Selection Switch

12 V to 5 V Buck Power

System 3.3 V Power

Connector

RGB LED Strip Connector (JP1)

No.	Signal Name	ESP32-C3 Pin
1	VCC_12V_5V	–
2	LED_G	GPIO6
3	LED_R	GPIO5
4	LED_B	GPIO4

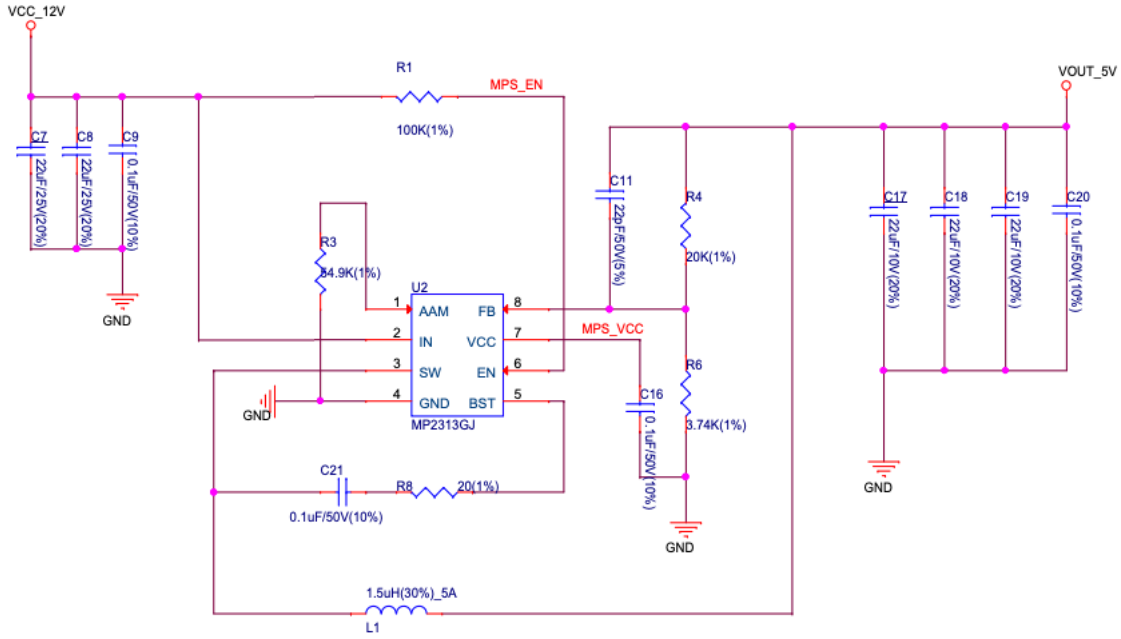


Fig. 52: 12 V to 5 V Buck Power Circuit

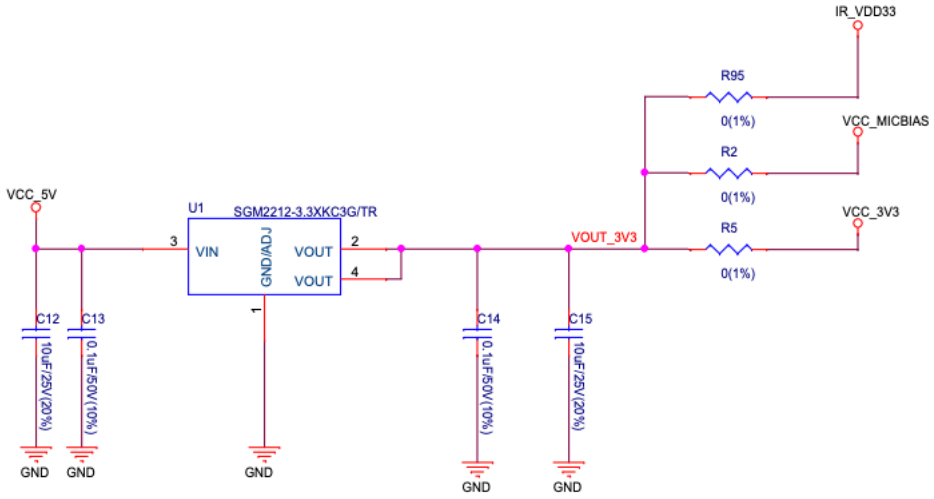


Fig. 53: System 3.3 V Power Circuit

Addressable LED Strip Connector (JP2)

No.	Signal Name	ESP32-C3 Pin
1	VCC_12V_5V	–
2	DIN	GPIO7
3	DIN	GPIO7
4	GND	–

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-C3. Note that some signals are shared. See Section *GPIO Allocation* for details.

UART Header (JP12)

No.	Signal Name	ESP32-C3 Pin
1	VCC_3V3	–
2	ESP_EN	EN
3	ESP_BOOT	GPIO9
4	ESP_UART_RXD	U0RXD
5	ESP_UART_TXD	U0TXD
6	GND	–

I2C Header (JP8)

No.	Signal Name	ESP32-C3 Pin
1	VCC_3V3	–
2	I2C_CLK	GPIO8
3	I2C_DATA	GPIO9
4	GND	–

Hardware Revision Details

No previous revisions.

Ordering

If you order a few samples, each board comes in an individual package.

For retail orders, please go to the official website https://www.espressif.com/en/company/contact/buy-a-sample, or directly order in our Taobao shop https://world.taobao.com/item/677273363812.htm?spm=a21wu.12321156-tw.recommend-tpp.4.19a61924ZMaqpf.

For wholesale orders, please go to the official website https://www.espressif.com/en/contact-us/sales-questions.

Related Documents

- Datasheet
  - ESP32-C3 Series Datasheet (PDF)
  - ESP32-C3-WROOM-02 & ESP32-C3-WROOM-02U Datasheet (PDF)
- Schematic
  - ESP32-C3-Lyra Schematic (PDF)
- PCB Layout
  - ESP32-C3-Lyra PCB Layout (PDF)

For further design documentation for the board, please contact us at sales@espressif.com.

3.5 Audio Samples

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

3.5.1 Formats

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

Long Samples

The audio track duration in this section is 3 minutes and 7 seconds.

Two Channel Audio

No	Format	Audio File	Size [kB]
1	aac	ff-16b-2c-44100hz.aac	2,995
2	ac3	ff-16b-2c-44100hz.ac3	2,994
3	aiff	ff-16b-2c-44100hz.aiff	33,002
4	flac	ff-16b-2c-44100hz.flac	22,406
5	m4a	ff-16b-2c-44100hz.m4a	3,028
6	mp3	ff-16b-2c-44100hz.mp3	2,994
7	mp4	ff-16b-2c-44100hz.mp4	3,079
8	ogg	ff-16b-2c-44100hz.ogg	2,612
9	opus	ff-16b-2c-44100hz.opus	2,598
10	ts	ff-16b-2c-44100hz.ts	5,510
11	wav	ff-16b-2c-44100hz.wav	32,229
12	wma	ff-16b-2c-44100hz.wma	3,227

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

Single Channel Audio

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

Playlist containing all above files: ff-16b-1c-playlist.m3u

Short Samples

If you need shorter audio files for testing, this section provides 16 seconds audio tracks.

Two Channel Audio

No	Format	Audio File	Size [kB]
1	aac	gs-16b-2c-44100hz.aac	241
2	ac3	gs-16b-2c-44100hz.ac3	380
3	aiff	gs-16b-2c-44100hz.aiff	2,792
4	flac	gs-16b-2c-44100hz.flac	1,336
5	m4a	gs-16b-2c-44100hz.m4a	258
6	mp3	gs-16b-2c-44100hz.mp3	254
7	mp4	gs-16b-2c-44100hz.mp4	259
8	ogg	gs-16b-2c-44100hz.ogg	229
9	opus	gs-16b-2c-44100hz.opus	219
10	ts	gs-16b-2c-44100hz.ts	286
11	wav	gs-16b-2c-44100hz.wav	2,792
12	wma	gs-16b-2c-44100hz.wma	276

Playlist containing all above files: gs-16b-2c-playlist.m3u

Single Channel Audio

No	Format	Audio File	Size [kB]
1	amr	gs-16b-1c-8000hz.amr	25
2	aac	gs-16b-1c-44100hz.aac	137
3	ac3	gs-16b-1c-44100hz.ac3	190
4	aiff	gs-16b-1c-44100hz.aiff	1,397
5	flac	gs-16b-1c-44100hz.flac	645
6	m4a	gs-16b-1c-44100hz.m4a	258
7	mp3	gs-16b-1c-44100hz.mp3	127
8	ogg	gs-16b-1c-44100hz.ogg	144
9	opus	gs-16b-1c-44100hz.opus	132
10	wav	gs-16b-1c-44100hz.wav	1,497
11	wma	gs-16b-1c-44100hz.wma	276

Playlist containing all above files: gs-16b-1c-playlist.m3u

3.5.2 Sample Rates

The files in this section have been prepared by converting a single audio file into different sampling rates defined in MPEG Layer III specification. Both mono and stereo versions of files are provided. The bit depth of files is 16 bits.

	Sample Rate	MPEG III	Channels	Bit Rate	Size
Audio File	[Hz]	ver		[kbit/s]	[kB]
ff-16b-1c-8000hz.mp3	8000	2.5	mono	8	183
ff-16b-1c-11025hz.mp3	11025	2.5	mono	16	366
ff-16b-1c-12000hz.mp3	12000	2.5	mono	16	366
ff-16b-1c-16000hz.mp3	16000	2	mono	24	548
ff-16b-1c-22050hz.mp3	22050	2	mono	32	731
ff-16b-1c-24000hz.mp3	24000	2	mono	32	731
ff-16b-1c-32000hz.mp3	32000	1	mono	48	1,097
ff-16b-1c-44100hz.mp3	44100	1	mono	64	1,462
ff-16b-2c-8000hz.mp3	8000	2.5	joint stereo	24	549
ff-16b-2c-11025hz.mp3	11025	2.5	joint stereo	32	731
ff-16b-2c-12000hz.mp3	12000	2.5	joint stereo	32	731
ff-16b-2c-16000hz.mp3	16000	2	joint stereo	48	1,097
ff-16b-2c-22050hz.mp3	22050	2	joint stereo	64	1,462
ff-16b-2c-24000hz.mp3	24000	2	joint stereo	64	1,462
ff-16b-2c-32000hz.mp3	32000	1	joint stereo	96	2,194
ff-16b-2c-44100hz.mp3	44100	1	joint stereo	128	2,924

Playlist containing all above files: ff-16b-mp3-playlist.m3u

Original music files: “Furious Freak” and “Galway”, Kevin MacLeod (incompetech.com), Licensed under Creative Commons: By Attribution 3.0, http://creativecommons.org/licenses/by/3.0/

CHAPTER

FOUR

RESOURCES

- Third party frameworks and libraries to develop audio applications with Espressif chips:
  - The JOSH operating system supports the ESP32 and can be used in scenarios such as intelligent voice interaction, smart home appliances, and smart gateways.
- Third party audio development modules and boards that work with ESP-ADF:
  - ESP32-A1S Audio Module equipped CodeC audio decoding chip that supports music playback and recording, and 4MB PSRAM. The module application schematic is available in datasheet.
- The esp32.com forum is a place to ask questions and find community resources. The forum has a section dedicated to ESP-ADF.
- This ESP Audio Development Framework inherits from ESP IoT Development Framework and you can learn about it in ESP-IDF Programming Guide.
- Check the Issues section on GitHub if you find a bug or have a feature request. Please check existing Issues before opening a new one.
- If you're interested in contributing to ESP Audio Development Framework, please check the Contributions Guide.
- Several books have been written about ESP32 and they are listed on Espressif web site.
- For additional ESP32 product related information, please refer to documentation Section of Espressif site.
- To buy audio development boards, check list of distributors under Get Samples on Espressif web site.

CHAPTER

FIVE

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:

- mp3 library is Copyright (c) 2005-2008, The Android Open Source Project, and is licensed under the Apache License Version 2.0.
- aac library is Copyright (c) 2005-2008, The Android Open Source Project, and is licensed under the Apache License Version 2.0.
- amr library is Copyright (C) 2009 Martin Storsjo and is licensed under the Apache License Version 2.0.
- opus library, Copyright 2001-2011 Xiph.Org, Skype Limited, Octasic,Jean-Marc Valin, Timothy B. Terriberry, CSIRO, Gregory Maxwell, Mark Borgerding, Erik de Castro Lopo, is licensed under 3-clause BSD license.
- flac library, Copyright (C) 2011-2016 Xiph.Org Foundation, is licensed under Xiph.Org's BSD-like license.
- vorbis library, Copyright (c) 2002-2020 Xiph.org Foundation, is licensed under the 3-Clause BSD license.
- aac-enc library is Copyright 2003-2010, VisualOn, and is licensed under the Apache License Version 2.0.
- adpcm library is Copyright (C) 2020 aikiriao <samuraiaiki@gmail.com>.
- jpeg-dec library is Copyright (C) 2019, ChaN.
- alac library is Copyright (C) 2004, developed by Apple, 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

SIX

ENGLISH-CHINESE GLOSSARY

This document lists terms that are used in Espressif Audio Development Framework Guide and other audio related documentation. Each term is followed by its Chinese equivalents and some have definitions.

AAC Chinese equivalent: AAC

Abbreviation for Advanced Audio Coding, an industry-standard audio compression format.

acoustic Chinese equivalent: □□

acoustic calibrator Chinese equivalent: □□□□

Also known as sound level calibrator.

acoustic echo cancellation Chinese equivalent: □□□□□□

Spelled-out form of AEC.

AEC Chinese equivalent: AEC

Abbreviation for acoustic echo cancellation.

Advanced Audio Distribution Profile Chinese equivalent: □□□□□□□□□

Spelled-out form of A2DP.

A2DP Chinese equivalent: A2DP.

Abbreviation for Advanced Audio Distribution Profile.

A2DP sink Chinese equivalent: A2DP sink

A2DP source Chinese equivalent: A2DP source

AFE Chinese equivalent: AFE

Abbreviation for audio front end. Espressif audio front-end algorithm framework is developed by Espressif AI Lab to provide high-quality and stable audio data to the host.

AirKiss Chinese equivalent: AirKiss

AirKiss is a quick-connection technique provided by Weixin device platform for Wi-Fi devices to configure network connection.

AMR Chinese equivalent: AMR

Abbreviation for Adaptive Multi-Rate, an audio compression format optimized for speech coding.

AMR-NB Chinese equivalent: AMR-NB

Abbreviation for Adaptive Multi-Rate Narrowband, a narrowband speech audio coding standard developed based on Adaptive Multi-Rate encoding.

AMR-WB Chinese equivalent: AMR-WB

Abbreviation for Adaptive Multi-Rate Wideband, a wideband speech audio coding standard developed based on Adaptive Multi-Rate encoding.

analog-to-digital converter Chinese equivalent: □□□□□□□□

Spelled-out form of ADC.

ADC Chinese equivalent: ADC

Abbreviation for analog-to-digital converter.

audio codec Chinese equivalent: □□□□□□□□

audio forge Chinese equivalent: □□□□

A combination of several audio backend processing techniques, including resample, downmix, automatic level control, equalizer and sonic. Users can enable or disable certain techniques as needed.

audio gate Chinese equivalent: □□□□

Spelled-out form of AG.

AG Chinese equivalent: AG

Abbreviation for audio gate.

audio front end Chinese equivalent: □□□□

Spelled-out form of AFE.

audio passthru Chinese equivalent: □□□□

Also known as pipeline passthru. It is an audio technique that allows audio files to pass through a pipeline unaltered.

audio pipeline Chinese equivalent: □□□□

Often used as "pipeline". It is a chain of audio processing elements arranged in a particular order so that the output of each element is the input of the next.

Audio Video Remote Control Profile Chinese equivalent: □□□□□□□□□□

Spelled-out form of AVRCP.

AVRCP Chinese equivalent: AVRCP

Abbreviation for Audio Video Remote Control Profile.

automatic gain control Chinese equivalent: □□□□□□

Spelled-out form of AGC.

AGC Chinese equivalent: AGC

Abbreviation for automatic gain control.

automatic level control Chinese equivalent: □□□□□□

Spelled-out form of ALC.

ALC Chinese equivalent: ALC

Abbreviation for automatic level control.

automatic speech recognition Chinese equivalent: □□□□□□

Spelled-out form of ASR.

aux cable Chinese equivalent: aux □□□

Also known as auxiliary cable.

ASR Chinese equivalent: ASR

Abbreviation for automatic speech recognition.

bandwidth Chinese equivalent: □□

Bass Frequency Chinese equivalent: □□

BCLK Chinese equivalent: BCLK

Abbreviation for base clock.

BluFi Chinese equivalent: BluFi

A Wi-Fi network configuration function via Bluetooth channel. See ESP-IDF Programming Guide for more information.

cavity Chinese equivalent: □□

command word Chinese equivalent: □□□

core dump Chinese equivalent: □□□□

cutoff frequency Chinese equivalent: □□□□

decoder Chinese equivalent: □□□

digital media renderer Chinese equivalent: □□□□□□□

Spelled-out form of DMR.

digital signal processor Chinese equivalent: □□□□□□□

Spelled-out form of DSP.

DSP Chinese equivalent: DSP

Abbreviation for digital signal processor or digital signal processing.

digital-to-analog converter Chinese equivalent: □□□□□□□

Spelled-out form of DAC.

dispatcher Chinese equivalent: □□□

distortion Chinese equivalent: □□

DAC Chinese equivalent: DAC

Abbreviation for digital-to-analog converter.

Digital Living Network Alliance Chinese equivalent: □□□□□□□□

Spelled-out form of DLNA.

DLNA Chinese equivalent: DLNA

Abbreviation for Digital Living Network Alliance.

DMR Chinese equivalent: DMR

Abbreviation for digital media renderer.

downmix Chinese equivalent: □□□□

An audio processing technique that mixes more audio streams to less output audio streams.

DuerOS Chinese equivalent: DuerOS

DuerOS is a conversational AI system developed by Baidu.

echo Chinese equivalent: □□

A reflection of sound that arrives at the listener with a delay after the direct sound.

echo reference signal Chinese equivalent: □□□□□□

electret condenser microphone Chinese equivalent: □□□□□□

Spelled-out form of ECM.

ECM Chinese equivalent: ECM

Abbreviation for electret condenser microphone.

element Chinese equivalent: □□

Also known as audio element. It is the basic building block for the application programmer developing with ADF. Every decoder, encoder, filter, input stream, or output stream is in fact an audio element.

encoder Chinese equivalent: □□□

equalizer Chinese equivalent: □□□

ESP VoIP Chinese equivalent: ESP VoIP

ESP VoIP is a telephone client based on the standard SIP protocol, which can be used in some P2P or audio conference scenarios.

fast Fourier transform Chinese equivalent: □□□□□□□

Spelled-out form of FFT.

FFT Chinese equivalent: FFT

Abbreviation for fast Fourier transform.

FatFs Chinese equivalent: FatFs

FatFs stream Chinese equivalent: FatFs □

FLAC Chinese equivalent: FLAC

Abbreviation for Free Lossless Audio Codec, an audio coding format for lossless compression of digital audio.

flexible pipeline Chinese equivalent: □□□□

FPS Chinese equivalent: FPS

Abbreviation for frames per second.

frames per second Chinese equivalent: □□□□□□

Spelled-out form of FPS.

frequency response Chinese equivalent: □□□□

full band Chinese equivalent: □□□

Spelled-out form of FB.

FB Chinese equivalent: FB

Abbreviation for full band.

Hands-Free Chinese equivalent: □□

Spelled-out form of HF.

HF Chinese equivalent: HF

Abbreviation for Hands-Free.

Hands-Free Audio Gateway Chinese equivalent: □□□□□□

Spelled-out form of HFP-AG.

Hands-Free Profile Chinese equivalent: □□□□

Hands-Free Unit Chinese equivalent: □□□□

HFP Chinese equivalent: HFP

Abbreviation for Hands-Free Profile.

hardware abstraction layer Chinese equivalent: □□□□□

Spelled-out form of HAL.

HAL Chinese equivalent: HAL

Abbreviation for hardware abstraction layer.

headset Chinese equivalent: □□

HFP-AG Chinese equivalent: HFP-AG

Abbreviation for Hands-Free Audio Gateway.

Hi-Fi speaker Chinese equivalent: □□□□□

Also known as high-fidelity speaker.

High Frequency Chinese equivalent: □□

high-fidelity microphone Chinese equivalent: □□□□□□

HLS Chinese equivalent: HLS

Abbreviation for HTTP Live Streaming.

HTTP stream Chinese equivalent: HTTP □

HTTP Live Streaming Chinese equivalent: HTTP □□□

Spelled-out form of HLS.

I2S stream Chinese equivalent: I2S □

insertion loss Chinese equivalent: □□□□

Internet radio Chinese equivalent: □□□□

Internet of Things Chinese equivalent: □□□

IoT Chinese equivalent: IoT

Abbreviation for Internet of Things.

JPEG Chinese equivalent: JPEG

A commonly used method of lossy compression for digital images. Same as JPG.

JPG Chinese equivalent: JPG

A commonly used method of lossy compression for digital images. Same as JPEG.

Light and Versatile Graphics Library Chinese equivalent: □□□□□□□□□□

Spelled-out form of LVGL

low-pass filter Chinese equivalent: □□□□□

LVGL Chinese equivalent: LVGL

Abbreviation for Light and Versatile Graphics Library.

M3U8 Chinese equivalent: M3U8

The Unicode version of M3U is M3U8, which uses UTF-8-encoded characters.

M4A Chinese equivalent: M4A

An audio encoding format for lossless compression of digital audio.

mass production Chinese equivalent: □□

maximum output power Chinese equivalent: □□□□□□

MCLK Chinese equivalent: MCLK

Abbreviation for master clock.

mel-frequency cepstral coefficients Chinese equivalent: □□□□□□□□□

Spelled-out form of MFCC.

MFCC Chinese equivalent: MFCC

Abbreviation for mel-frequency cepstral coefficients.

microphone Chinese equivalent: □□□

mic Chinese equivalent: □□□

Informal form for microphone.

micro-electro-mechanical systems microphone Chinese equivalent: □□□□□□□□□□

Spelled-out form of MEMS mic.

MEMS mic Chinese equivalent: MEMS □□□

Abbreviation for micro-electro-mechanical systems microphone.

microphone gain Chinese equivalent: □□□□□

microphone hole Chinese equivalent: □□□

microSD card Chinese equivalent: microSD □

MP3 Chinese equivalent: MP3

MP4 Chinese equivalent: MP4

MultiNet Chinese equivalent: MultiNet

MultiNet is a lightweight model specially designed based on CRNN and CTC for the implementation of multi-command recognition.

multi-room Chinese equivalent: □□□

Multi-Room Music Chinese equivalent: Multi-Room Music

ESP Multi-Room Music is a Wi-Fi-based communication protocol to share music across multiple interconnected speakers. Under this protocol, those connected speakers form a Group. They can play music synchronously and are controlled together, which can easily achieve a theater-grade stereo surround sound system.

narrowband Chinese equivalent: □□

Spelled-out form of NB.

NB Chinese equivalent: NB

Abbreviation for narrowband.

NimBLE Chinese equivalent: NimBLE

An open-source Bluetooth Low Energy or Bluetooth Smart stack.

noise criteria curve Chinese equivalent: □□□□□□

Also known as NC curve.

noise rating curve Chinese equivalent: □□□□□□

Also known as NR curve.

noise floor Chinese equivalent: □□□□

noise suppression Chinese equivalent: □□□□

Spelled-out form of NS.

NS Chinese equivalent: NS

Abbreviation for noise suppression.

non-volatile storage Chinese equivalent: □□□□□□

Spelled-out form of NVS.

NVS Chinese equivalent: NVS

Abbreviation for non-volatile storage.

OGG Chinese equivalent: OGG

An audio compression format.

OPUS Chinese equivalent: OPUS

A lossy audio coding format.

PCM Chinese equivalent: PCM

Abbreviation for pulse-code modulation.

pixel Chinese equivalent: □□

playback Chinese equivalent: □□

It is a noun. The verb is “play back”.

programmable gain amplifier Chinese equivalent: □□□□□□□□

Spelled-out form of PGA.

PGA Chinese equivalent: PGA

Abbreviation for programmable gain amplifier.

protractor Chinese equivalent: □□□

pulse-code modulation Chinese equivalent: □□□□□□

Spelled-out form of PCM.

raw stream Chinese equivalent: □□□

resample Chinese equivalent: □□□

resample filter Chinese equivalent: □□□□□□

resonant frequency Chinese equivalent: □□□□

reverberation Chinese equivalent: □□

RGB Chinese equivalent: RGB

The RGB color model is an additive color model in which the red, green, and blue primary colors of light are added together in various ways to reproduce a broad array of colors.

ring buffer Chinese equivalent: □□□□□

SBC Chinese equivalent: SBC

Abbreviation for subband codec.

SD card Chinese equivalent: SD □

Session Initiation Protocol Chinese equivalent: □□□□□□

Spelled-out form of SIP.

signal-to-echo ratio Chinese equivalent: □□□

signal-to-noise ratio Chinese equivalent: □□□

Spelled-out form of SNR.

SIP Chinese equivalent: SIP

Abbreviation for Session Initiation Protocol.

SNR Chinese equivalent: SNR

Abbreviation for signal-to-noise ratio.

SmartConfig Chinese equivalent: SmartConfig

The SmartConfig ™ is a provisioning technology developed by TI to connect a new Wi-Fi device to a Wi-Fi network. It uses a mobile app to broadcast the network credentials from a smartphone, or a tablet, to an unprovisioned Wi-Fi device.

sonic Chinese equivalent: □□

An audio processing technique that modifies sound frequency and speed.

sound card Chinese equivalent: □□

Also known as audio card.

sound level meter Chinese equivalent: □□□

Also known as sound pressure level meter.

sound pickup hole Chinese equivalent: □□□

sound pickup tube Chinese equivalent: □□□□

sound transmission loss Chinese equivalent: □□□□

Spelled-out form of STL.

speech Chinese equivalent: □□

speech recognition Chinese equivalent: □□□□

Spelled-out form of SR.

SR Chinese equivalent: SR

Abbreviation for speech recognition.

SPI Flash File System Chinese equivalent: SPI □□□□□□

Spelled-out form of SPIFFS.

SPIFFS Chinese equivalent: SPIFFS

Abbreviation for SPI Flash File System.

SPIFFS stream Chinese equivalent: SPIFFS □

subband codec Chinese equivalent: □□□□□

Spelled-out form of SBC.

STL Chinese equivalent: STL

Abbreviation for sound transmission loss.

super wide band Chinese equivalent: □□□□

Spelled-out form of SWB.

SWB Chinese equivalent: SWB

Abbreviation for super wide band.

tape measure Chinese equivalent: □□

text-to-speech Chinese equivalent: □□□□

Spelled-out form of TTS.

TTS Chinese equivalent: TTS

Abbreviation for text-to-speech.

tolerance Chinese equivalent: □□

tone Chinese equivalent: □□□

total harmonic distortion Chinese equivalent: □□□□□

Spelled-out form of THD.

THD Chinese equivalent: THD

Abbreviation for total harmonic distortion.

voice activity detection Chinese equivalent: □□□□□□

Spelled-out form of VAD.

VAD Chinese equivalent: VAD

Abbreviation for voice activity detection.

VoIP Chinese equivalent: VoIP

Abbreviation for Voice over Internet Protocol.

wake word Chinese equivalent: □□□

wake word engine Chinese equivalent: □□□□□

Spelled-out form of WWE.

WakeNet Chinese equivalent: WakeNet

WakeNet is a wake word engine built upon neural network for low-power embedded MCUs.

wake-up Chinese equivalent: □□

It is a noun.

wideband Chinese equivalent: □□

Spelled-out form of WB.

WB Chinese equivalent: WB

Abbreviation for wideband.

WWE Chinese equivalent: WWE

Abbreviation for wake word engine.

YUV Chinese equivalent: YUV

A color model typically used as part of a color image pipeline.

CHAPTER

SEVEN

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**.

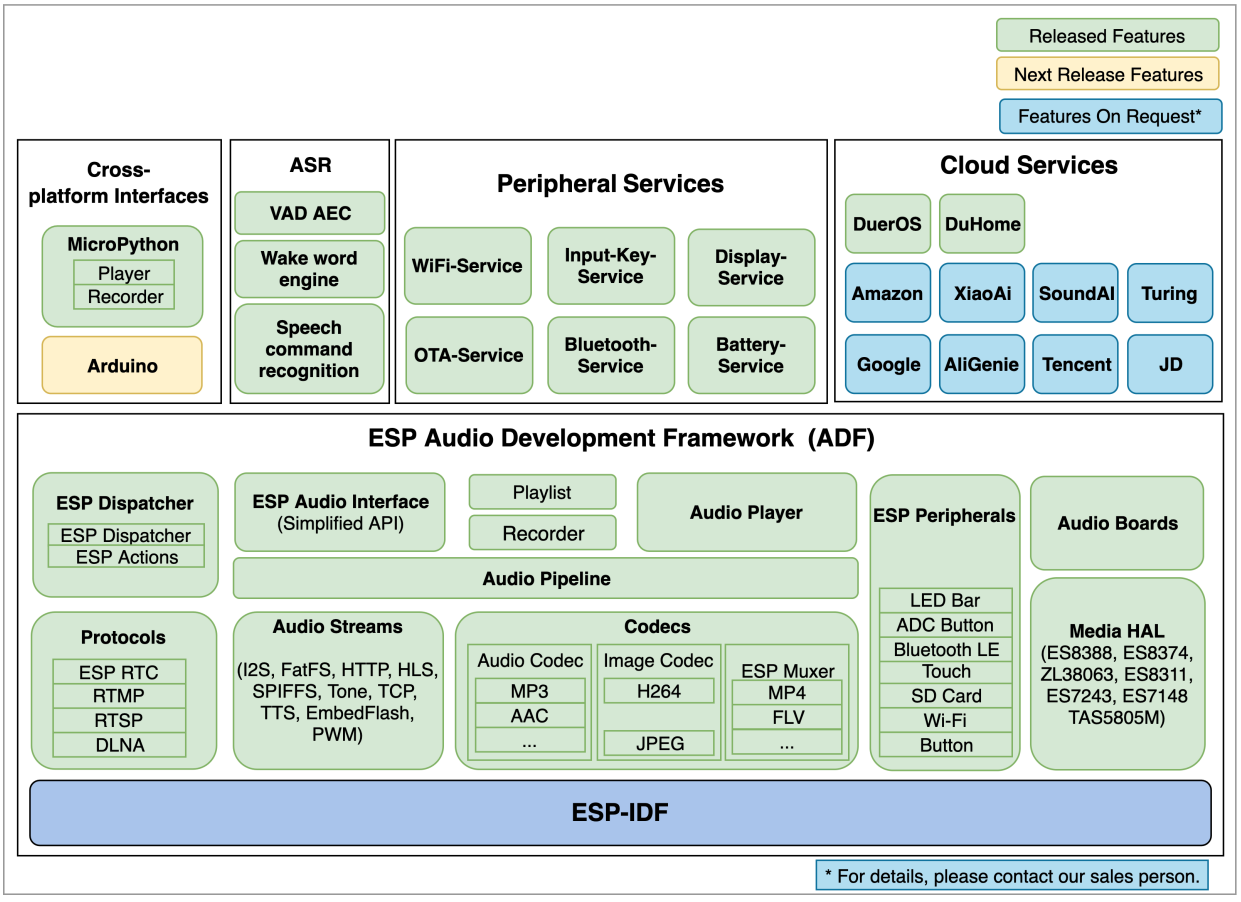


Fig. 1: Espressif Audio Development Framework

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

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

V

W

Y

Z